Structure from Visibility: Finding low-textured and reflective occluders in outdoor scenes using visibility and occlusion

Student: Martijn Frederik Wouter van der Veen
Supervisor: Gabriel J. Brostow
MSc Computer Graphics, Vision, and Imaging
September 2012

This report is submitted as part requirement for the MSc Degree in Computer Graphics, Vision & Imaging at University College London. It is substantially the result of my own work except where explicitly indicated in the text.

Department of Computer Science
University College London

Abstract

Geometry reconstruction has been an important but challenging field of research within computer vision. Many approaches rely on extracting certain features (silhouettes, interest points, similar coloured pixels) from images, and are inherently dependent on the ability to find reliable features on all objects. Objects not rich in features or objects with non-Lambertian properties result in poor reconstructions. However, detection failures can provide useful clues as well. In this thesis, we propose an approach based on both visibility of detected features *and* occlusion of features detected in other images. The approach aims at finding and reconstructing objects *indirectly* by noticing the absence of expected features, without relying on detection of any cues on the objects theirselves.

The suggested approach is tested on new datasets containing footage of low-textured and reflective objects. Given enough view points and texture on background objects, the developed and implemented method is able to reconstruct both low-textured and reflective objects. Results are compared to a state-of-the-art dense stereo reconstruction and found to perform better on reflective and big low-textured objects.

Acknowledgements

I would like to thank Gabriel Brostow for the supervision and super vision during interesting discussions.

Neill Campbell helped me getting started with the maxflow code, thanks for that.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

The understanding of physical scenes by machines has been of great interest to researchers over the last couple of decades. Being able to construct the three dimensional shape and location of objects in a scene can be very helpful in this task. Based on the observation that, for humans, visual clues seem useful for understanding the structure of their environment, it is not surprising that researchers in the field of computer vision have put great effort in creating methods that reconstruct 3D models out of images. Being able to reconstruct geometry allows for new applications. On a practical level, for example, it allows for easy creation of 3D models for usage in modelling design, computer games and even 3D printing. In addition, geometry reconstruction as an intermediate step opens the door to further processing in advanced street scene understanding, advanced geometry-aware rendering of new content or editing of existing footage (such as changing lighting or virtually moving objects in the scene), navigational tasks in robotic systems, and augmented reality applications[1]. Increasing computer power, cheap and ubiquitous cameras, and recent developments in the field of computer vision and machine learning all help making these exciting applications a reality.

There has been published a great amount of research on geometry reconstruction. A variety of approaches has been proposed, each with their own advantages and disadvantages. Most, if not all, of the approaches seem to be based on one or more visual clues that are relatively easy to find automatically, such as silhouettes, corner points or similarly coloured pixels in images. Different visual clues call for various requirements in the input images, resulting in successes and failures specific for the chosen approach. Current approaches often rely on the ability to detect features such as silhouettes or distinctive corner points on *all* geometry, resulting in bad results for objects where the specific features are less present.

However, detected features are not the only piece of information that can be used for geometry reconstruction; failure of detection can provide useful information too. Detection failures can be detected by using information from features detected in other parts of the data (*e.g.,* other images). In particular, the absence of features in images where they would have been expected - due to detection in images taken from surrounding camera poses - may indicate the existence of other objects. Although this observation might seem obvious, it has not got a lot of attention in the computer vision literature. Bad matches or the

[1] One could even state that geometry reconstruction could be seen as an intermediate feature in advanced computer vision, much in the same way as edge detection is used extensively as a feature in more advanced image processing algorithms.

absence of features have mostly been treated as outliers or as a binary vote for not using certain data in a particular step in the reconstruction; seldomly has it been used actively for reconstructing geometry[2]. Often camera poses are thrown away after triangulating the feature locations, continuing with the positive information (features) only. Intuitively, this seems like throwing away useful information. It is this intuition that motivates the approach taken in this thesis.

In this thesis, a framework is proposed that uses both *visibility* (detected features) and *occlusion* (undetected but expected features) in order to find, and reconstruct, objects not rich in reliable features. The proposed method uses features detected in other parts of the scene to find objects difficult to detect directly. Based on the visibility and occlusion information of features from given observation locations, space is 'carved' in a probabilistic way. The method has been tested on a variety of outdoor scenes and, given enough detected features on objects in the background, gives high probabilities for regions containing objects difficult to detect with widely-used alternative methods. While space-carving based on silhouettes has difficulties with high-textured and complex outdoor scenes, and multi-view stereo is challenged by low-textured and reflective objects, the suggested method targets at scenes containing both and aims at combining strong properties of both methods.

In the course of the project that resulted in this report, most of the time has been used for the practical implementation and evaluation, in addition to a comprehensive literature review and documentation. Various software libraries have been tested and used to develop not only the method described here, but also several visualisation tools helpful for the visual thinking process essential for developing geometry algorithms. Since our results are inherently three-dimensional and this report is not, the reader is encouraged to view the supplemented digital material[3] - either using the provided tools or pre-made videos - alongside reading the report.

(a) Image taken from http://www.ikea.com (b) VisualSfM output of image sequence

Figure 1.1: Reconstructed sparse point cloud of stuffed toy

[2]It is almost as if researchers are afraid of negative information.
[3]The material is supplemented on CD or available online at http://code.google.com/p/martijn-msc-thesis/

Chapter 2

Background and Related Work

Geometry reconstruction has drawn a lot of attention in the computer vision community (Hartley and Zisserman [10], Seitz et al. [27], and Prince [25]). The goal is to reconstruct the three dimensional structure of a particular scene given the data, usually multiple images (Multi-View Stereo). The images can come from different cameras (unordered set), or from the same camera (sequence) that potentially has been moved around. For many algorithms the relative locations from the cameras are required. They are either obtained by calibration between the different cameras or images (*e.g.,* by some identifiable pattern visible in all the images, or measured using external sensors), or estimated based on features naturally occurring in the images. When the positions are known, all the pixels in an image are known to be a projection from a particular point somewhere on a line through the camera centre and the pixel on the image plane, with the location of the point on the line being the only ambiguity. The next task now is to estimate the geometry of the scene pictured in those images, given this ambiguous information. Different approaches exist to tackle this problem, of which we will discuss four categories: layers (Section 2.1), space carving (Section 2.2), structure from motion (Section 2.3), and Multi-View Stereo using Depth Maps (Section 2.4). Note that often multiple approaches are combined and overlap exists. Not all the techniques being discussed are directly relevant to the proposed algorithm, but they give a sense of the variety of approaches being published and try to give a general overview of the geometry reconstruction research literature.

2.1 Layers in Video

One of the earlier, and in some aspects simpler, attempts to represent geometry in images is the use of layers. Images are assumed to be a composition of multiple, possibly overlapping, surfaces that can move relative to one another from frame to frame in a sequence. Objects pictured in images can be at different distances to the camera, resulting in overlapping surfaces when projected onto an image (*i.e.,* captured). Successfully identifying coherent surfaces, including their mutual occlusions and movements from one frame to another, allows for a scene representation consisting of *layers*. Such a representation usually consists of one background layer plus one or more foreground layers representing objects moving in front of the background layer.

As one of the first to suggest representing sequences of images with layers, Wang and Adelson [30] decomposed images into 'motion layers' by analysing the motion of extracted segments in the subsequent images using optical flow. Segments with similar motions are combined, and grown and depth-ordered by tracking them over the sequence, resulting in more extensive layers. The layers can then be used in combination with estimated simple motion models to reconstruct the sequence in a memory-efficient way. Multiple extensions on this scheme exist. For example, Jojic and Frey [15] subdivided layers into flexible sprites that can vary their shapes according to a probabilistic model. The sprites and their probabilistic models are learned using the Expectation Maximisation (EM) algorithm on a representation consisting of one appearance and one mask pixel vector per sprite. Theoretically, the unsupervised EM learning approach even allows for finding sprites in related but unordered photo collections. Another approach, suggested by Smith et al. [28], is to extract and track edges over the sequence in order. Motion models are then fit to the edges using the EM algorithm. The edges, which form a better criterion for segment borders than the locally-smoothed optical flow vector field, are used to find a good segmentation of the frames and segments with similar motion models are again merged, thereby resulting in layers.

Although layered representations do have the notion of depth-ordering - and thus occlusion - they do not aim at recovering the exact depth of the layers, and inherently do not reconstruct three-dimensional geometry. In the next section, we move on to three alternative approaches that aim at true geometry recovery.

Layers are relevant for the current work because initially the intuition was raised that footage of an outdoor scene can be segmented into layers by tracking feature points and noticing when occlusions happen. However, for these feature points we can estimate the three-dimensional location by using Structure from Motion techniques (Section 2.3), allowing to reconstruct more than relative order. Hence, the step from layers to real three-dimensional reconstruction was made and the proposed method as described in Section 3 originated.

2.2 Space carving

While layers in videos try to represent geometry by their projections onto the image planes, thereby ignoring the original three-dimensional geometrical shapes, space carving aims at estimating the actual surface of objects in a scene. To reconstruct 3D from flat projections, space carving assumes the camera poses are known, *i.e.,* we have calibrated cameras. The poses are either known due to explicit calibration (using a calibration object or external sensor), or estimated based on the natural content of the images.

Usually, space carving is based on finding silhouettes [10, 20], thus segmenting each of the images into a binary background and foreground 'layer' or mask. These techniques are also known as Structure from Silhouettes. The pixels in the images are known to be projections of points somewhere on lines from the camera centres passing through the pixels, but the locations on the lines (*i.e.,* depths) are unknown. Space carving based on silhouettes[1] does not estimate this depth; instead, it uses the background-foreground masks to categorise each line as either intersecting with an object, or not. The foreground mask therefore represents a cone of space in which the projected object must lie, at unknown distance.

For reconstruction, usually a 3D volume is chosen that lies in between the camera poses, and the volume is discretised and represented as a *voxel grid* - although polygonal representations have been proposed too. Space carving starts with all the voxels marked as 'object'. For each camera pose, all the voxels are projected onto the image plane. Voxels projecting to pixels not set as foreground (*i.e.,* onto the background mask) are set to 'non-object', finally resulting in the intersection of cones in which the object must lie (see Figure 2.1). This intersection is called *visual hull* or *convex hull*. It can be used as the final reconstruction, or used as a useful prior for further processing as the maximum (convex) boundary in which search can continue.

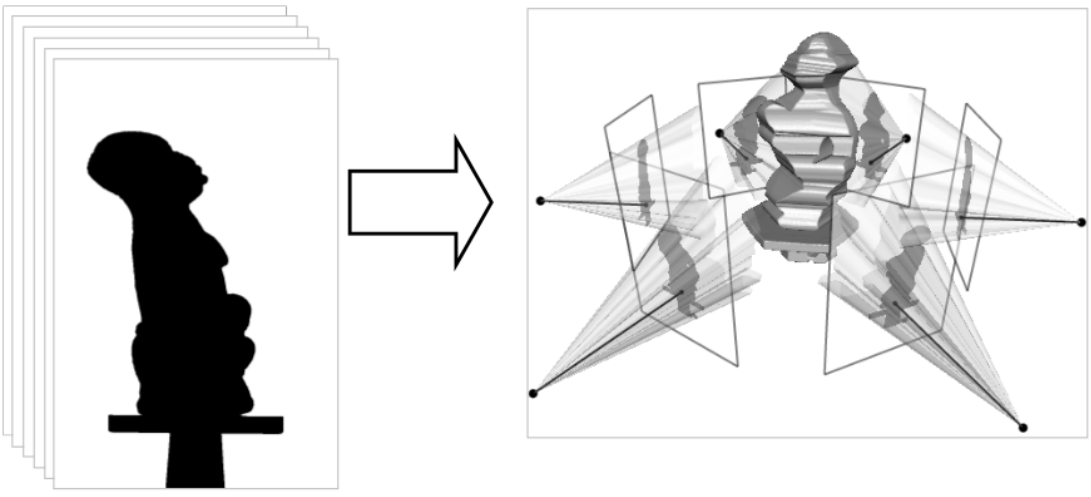


Figure 2.1: Carving using silhouettes. Image taken from 'How to prepare and deliver a presentation' [ppt] by R. Cipolla, who adopted it from unknown source

Segmenting images in foreground and background masks can be accomplished by a simple thresh-

[1] Part of the literature reserves the term Space Carving for space carving based on photo-consistency as described later on; here, it is used for both photo-consistency and silhouette-based space carving iteratively refining a model represented by a voxel grid.

old, or by learning more advanced colour models for both. This can be user-guided or automated. For example, Campbell et al. [3] automatically learned foreground colour models using pixels at the centre of the images (requiring the user making the footage to fixate at the object of interest) and background colour models using pixels near the image corners. The initial colour models are used together with the calibrated camera poses to make an initial binary segmentation in a voxel grid. The segmentation is then used to refine the colour models, and the process is repeated until the solution converges. As is common in algorithms involving voxel grid representations, Campbell et al. [3] use a Markov Random Field (MRF) prior for the segmentation step, promoting local smoothness for obtaining a global solution. A min-cut/max-flow graph-cut algorithm like the one presented by Boykov and Kolmogorov [1] is used to find the solution. This process is also known as regularisation.

Using more pictures in the space carving approach will give better approximations. However, concavities are not visible from silhouettes, so they will not turn up in the final reconstruction. The acquired visual hull therefore is only a broad approximation of the object's surface. Space carving is very suitable to do on a turn-table, for which the background model is known and also the relative camera poses are known due to controlled turn-table rotation. However, a turn-table is only feasible for objects small enough to put on the device. Placing a green screen behind objects is also a possibility, resulting in the same easy silhouette segmentation. Green screens have their limits too, making it an unsuitable remedy for outdoor scenes, where the background often is not controllable. Furthermore, space carving is not suitable for dynamic scenes. As a last disadvantage, multiple objects are not easy recognisable in a single silhouette and will result in ghosting objects [9].

Extensions have been proposed to overcome the limited applicability of carving based on silhouettes. For example, Guan et al. [9] added time-dependent Bayesian probability to the voxel variables, allowing space carving to be used in dynamic scenes. In their representation, voxels represent probabilities on occlusions before and after the voxel on a line passing through the centres of a set of calibrated cameras, filming from fixed locations. Using time-dependencies, their algorithm is able to follow dynamic objects over time, *e.g.,* a human walking through a temple. To find silhouettes, a per-pixel Gaussian background colour model is made for every camera before the dynamic object enters the scene. The estimates of the silhouettes are used as a prior for the next time frame, allowing to spot the object moving behind static occluders. When the object moves extensively through the scene, static occluders will turn up when they occlude the object. Even concavities in static occluders can be detected if an object moves, in fact allowing to carve away all human reachable locations by walking through the scene. In a later publication, Guan et al. [18] extended their algorithm to detect and label multiple dynamic objects. Each time a new, unknown silhouette (that differs enough from the colour models learned so far) enters the scene, a new appearance colour model is learned. Although the algorithm only works with fixed cameras and dynamic objects, it does have notion of visibility and occlusion.

An alternative to space carving with silhouettes is space carving based on photo-consistency, as introduced by Kutulakos and Seitz [17] . Hereby, voxels are projected onto all image planes that are able to see the voxel according to the current voxel grid. The set of pixels a voxel projects to are then

checked for photo-consistency in compliance with a reflectance model (BRDF). The photo-consistency check returns the probability of this set of pixel colours originating from the same surface point. For a low photo-consistency score, it is unlikely that this voxel is part of a solid object, and so it is carved away. The process continues until all the remaining uncarved voxels have a high photo-consistency. Although results look promising, the algorithm has some problems with low-textured (*i.e.,* uniformly coloured) objects and objects with repetitive textures. Furthermore, the method does not scale very well to bigger environments with lots of same coloured objects, and assumes Lambertian surfaces for all objects. In practice, pixels usually have no unique colour and reflective objects are not uncommon.

As a last example in the space carving literature, we discuss the work by Hernández et al. [11], who further explored the concept of visibility using the photo-consistency criterion. The algorithm introduced by Kutulakos and Seitz [17] is extended to include a probabilistic voxel grid: voxels are not carved one by one, but assigned the photo-consistency probability for being part of the surface of an object [2]. A second voxel grid represents the visibility evidence, containing probabilities of voxels being visible by at least one camera. This will give a penalty to voxels that are presumed not to be visible from any camera, and thus not part of the surface of an object. As a final step, graph-cut is used to combine the photo-consistency and visibility voxel grids, obtaining the assumed surface. Note that occlusion is being used, but only to exclude voxels inside objects, not for refining the shape itself.

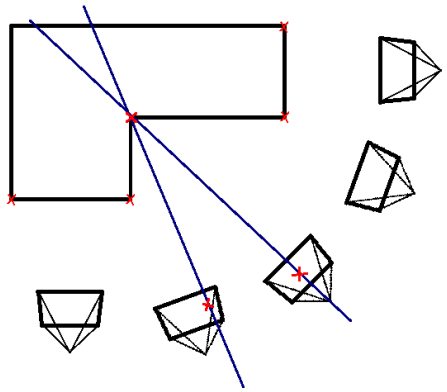
2.3 Structure from Motion

Geometry reconstruction usually involves measurements in the form of images, that is, flat projections of the three-dimensional world at specific locations (and times). If these locations are known, the process of reconstructing the original scene becomes easier. Therefore, much effort has been put in reliable pose estimation. Structure from Motion, as described extensively by Hartley and Zisserman [10], tries to accomplish this. Structure from Motion (SfM) uses natural features in images in order to estimate the relative translation and rotation between the camera poses of different images. It is based on the intuition that moving your head (motion) and noticing differences of the different views allows for a rough estimation of objects and distances (structure) in a scene.

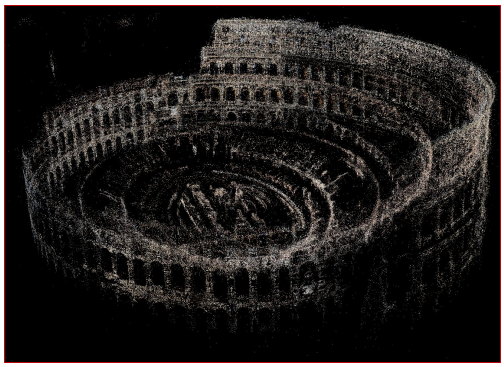
For SfM techniques, the camera poses are unknown a priori and need to be estimated based for a set of given images. As a first stage, features such as interest (corner) points are found in all images. One of the most popular features is SIFT, as introduced by Lowe [19]. SIFT features are found by searching for maxima in scale-space, constructed with Derivative of Gaussian filters. For the detected locations, a unique SIFT descriptor is constructed based on a histogram of gradient directions of points nearby. SIFT features are scale and rotation invariant and partially intensity and contrast change invariant. Alternative features can be used too, as long as they can be matched uniquely between images. In the second step, the interest points are matched between images using their descriptor. Using the matching pairs between images allows for estimating the relative transformation from one camera to another by satisfying projective geometry constraints (Prince [25], Ch15-16, Hartley and Zisserman [10], Ch10-12). For every

[2] In fact, a fast depth-map (Section 2.4) computation is done and photo-consistency is calculated only for locations with 3D points nearby

image pair with estimated transformation, 3D locations of matched interest points can now be estimated by triangulation (Fig. 2.2(a)). The depth ambiguity for *some points* is now eliminated in each image. The image pair point clouds - possibly containing feature points seen in multiple image pairs - can be combined using Bundle Adjustment (*e.g.,* , Wu et al. [31], Hartley and Zisserman [10], Ch18), which minimises noise and estimation errors in point and camera pose locations and obtains a global solution. The result is a *sparse point cloud* (Fig. 2.2(b)), camera models (including pose), plus visibility lists with camera-points pairs. As an example of using alternative features, Chandraker et al. [4] developed a Structure from Motion system based on lines, arguing that indoor environments and scenes containing much human-made objects usually lack distinctive feature points, but a rich of straight lines. Line geometry constraints are used to estimate camera translation and rotation in a similar way as in the case points are used.



(a) 3D point location estimation by triangulating two matched feature points in a pair of images



(b) Sparse point cloud. Image taken from http://grail.cs.washington.edu/rome/dense.html (1st of March 2012)

Figure 2.2: Structure from Motion

After reconstructing both the sparse point cloud and camera poses, usually one of the two is thrown away and the other is used in a subsequent step. Given the sparse point cloud, one can attempt to reconstruct geometry based on the sparse, but reliable points. Different techniques have been used [27]. One simple way of doing this is fitting a polygon mesh to the point cloud. More advanced smooth surface fitting is also possible, as has been done for example by Izadi et al. [14], but it eliminates strong edges and flat surfaces which are overrepresented in real scenes. Another method is fitting primitive shapes like planes and lines to the point cloud, which works well in human-made environments, but not so well in outdoor scenes. In all cases, assumptions are made on possible interpolation between points close to each other, and lack of texture results in gaps and sometimes missing objects in the reconstruction. In practice, sparse point clouds are rarely used for surface reconstruction. More often, the camera poses are kept and used in combination with the original images using a subsequent algorithm (*e.g.,* space carving (Section 2.2) or depth maps (Section 2.4)), obtaining a more dense representation before surface reconstruction is attempted.

2.3.1 Visibility and Occlusion

An important observation that did not got a lot of attention in the literature is the fact that the visibility of the points over time (*i.e.,* over the image sequence) can not only help selecting and fine-tuning points in either sparse or dense point clouds (as noted in the survey of Seitz et al. [27] and used in [22, 11]), but also directly used for surface reconstruction. In other words: occlusion rarely has an essential role in geometry reconstruction. However, visibility tracks over time can help both finding empty space (because we can see points at a certain location from a certain location) and finding occluder objects (because we lost track of a point for a while). Structure from Motion - in addition to pose estimation - therefore does not only deliver sparse points on surfaces, but also gives information about the occupancy state of space between the camera poses and reconstructed sparse points, without the need to find any visual clues at that locations. This observation has an essential role in the proposed approach (Section 3).

Despite the scarcity of occlusion applications, recently there have been some attempts worth a brief discussion. For example, Zach et al. [32]extent the structure from motion pipeline by an outlier check by using 'missing correspondences'. Their method collects triplets of images connected by fair amounts of correspondences. Two of the images are used to triangulate the (sparse) correspondences which are projected into the third image. A probabilistic measure based on the number of found and missing correspondence points in the third image determines whether to discard the third image as a match or not. Pan et al. [23] propose geometry reconstruction (called ProFORMA) of single, well-textured, objects filmed with a fixed-position camera by calculating a sparse point cloud, converting the point cloud into a mesh using Delaunay tetrahedralisation, and carving away tetrahedra who violate visibility constraints. The carving consists of using intersections of rays from camera poses to points to determine - probabilistically - whether to carve away a intersected tetrahedra, finally obtaining a surface mesh. Lovi et al. [5] independently developed an almost identical system (called free-space carving) that can be used on more complicated scenes. Both methods use positive visibility information to carve away tetrahedra; however, negative occlusion information is not being used as tetrahedra are assumed to be occupied by default. Furthermore, the tetrahedralisation is based on triangulated feature points, requiring texture to be omnipresent and determinative for quality and resolution of reconstructed models. Recently Mcllroy et al. [21] (2011) explored the possible use of both visibility and occlusion information with the aim of constructing high-level scene structure. In their approach, every (sparse) point is assigned a discretised 'view sphere' whereby every solid angle bin saves the distance to the furthest camera within the solid angle that detected the point, plus the distance to the closest camera that did not detect the point. Using the view spheres primitives like planes and spheres are fitted to obtain probable locations of scene structure. However, results are shown for simple and well-textured scenes only, modelled by simple planes. Furthermore, since camera poses are used for the fitting, one needs to move extensively through the scene to provide the algorithm with enough camera poses. In contrary, the proposed method (Section 3) focusses not on the specific camera and point locations but on the rays in between, and thence indirectly finds low-textured and reflective objects using both visibility and occlusion information. In

addition, a probabilistic voxel grid is used for discretisation allowing more general shape reconstruction, the implementation is tested on complex real-world scenes, and compared to state-of-the-art Multi-View Stereo results.

2.4 Multi-View Stereo using Depth Maps

As a last category of geometry reconstruction methods, we will discuss reconstruction based on dense depth maps, often called stereoscopic. Usually, two steps are involved. In the first step, pairs or bigger sets of images are combined to form dense depth maps. The second step consists of carefully combining the depth maps in order to reconstruct scene geometry globally, for instance - as with SfM - by using bundle adjustment [31]. Where space carving is mostly defined in scene-space by re-projecting voxel centres onto image planes, multi-view stereo (MVS) using depth maps applies its photo-consistency measure in image-space by comparing pixels [27]. Furthermore, as do structure from motion techniques, MVS using depth maps tries to estimate the depth of pixels explicitly. Depth maps, however, try to estimate this depth for every pixel, giving a dense depth or disparity map. However, estimating the depth for every pixel makes it necessary to match with less confidence than possible with SfM points, which can rely on stable and easy comparable feature points such as SIFT. Finally, combining the dense depth maps results in a *dense point cloud*. We will now look into two different ways of creating the depth maps.

2.4.1 Active lighting

High precision in solving the depth ambiguity can be reached by actively illuminating part of the scene and searching for the illuminated part in images being taken. Depth map techniques recently received increased interest due to affordable active lighting sensors such as Kinect [14] reaching the commercial market, making depth vision suitable for a wider audience, but active lighting systems have been out there for a while. The illumination can consist of some fixed pattern (such as used by the Kinect and exploited by Izadi et al. [14]), a sequence of different patterns (structured light), or a single (laser) point or line that moves over the surface (either for triangulation or time-of-flight). For example, a laser range-finder can be used to accurately map an environment by sending bright laser pulses and measuring travel times. Huang et al. [12] released a database containing range images obtained this way, together with a statistical analysis of natural depths in images. Although pattern-based devices like Kinect are becoming affordable, their active lighting patterns are weak and hence only suitable for indoor scenes. In general, accurate active lighting devices are less affordable and less practical than omnipresent cheap cameras. Therefore, much research effort has been put into passive depth vision based on regular images pairs.

2.4.2 Passive depth vision

If one would know, for every pixel in an image, the particular pixel displaying the same point in scene-space in a second image, the relative displacement allows for calculating the distance between that point in space and the camera ([25], Ch.14). The depth is estimated by triangulation using displacement and relative camera transformation. Stereoscopic methods often attempt to find pixel pairs or matching patches by using a particular photo-consistency measure and exhaustive search. However, pixel colours

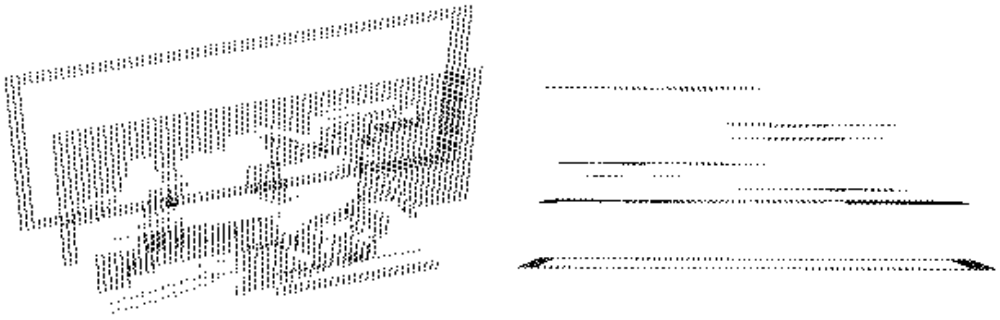


Figure 2.3: Two views of a depth map with few discretised depths, created by graph-cut algorithm. Image taken from own second years project, bachelor artificial intelligence, University of Amsterdam

are less unique than interest point descriptors and also in general not all the points are visible in both images (occlusion), resulting in ambiguous or missing matches, especially for scenes with low-textured surfaces (*e.g.,* painted walls). To simplify the search process, the images are often rectified: morphing them as if they were taken from cameras aligned on a common axis; the search for each pixel now only needs to occur on particular lines of pixels in the second image. Other geometry priors, such as order preservation or minimum and maximum depths, can be used too. Various techniques have been proposed for matching pixels or other primitives in order to obtain smooth depth maps without too many wrong matches or gaps. Hereby, almost all algorithms assume Lambertian surfaces as photo-consistency prior, causing problems for reflective surfaces.

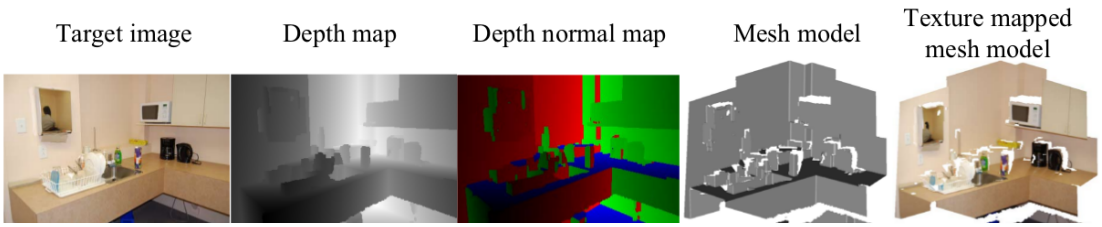


Figure 2.4: Pipeline of Furukawa et al. [7], including a typical depth map. Images taken from [7].

Most algorithms are based on matching pixels based on a photo-consistency measure [27]. After rectification, only pixels on a particular line need to be tried. Those algorithms often use a prior promoting local smoothness, such as a Markov Random Field (MRF), while using a graph-cut algorithm to find a solution. The number of possible depths can be kept low, lowering the number of computations and increasing smoothness while allowing jumps in depth (*e.g.,* Figure 2.3). Effectively, the image is segmented in layers for which depth is estimated. Plane-sweeping algorithms iterate between assigning pixels to planes, and refining the plane equations (which do not necessarily need to be parallel to the image plane). One fast but inaccurate plane-sweeping stereo algorithm is used by Merrell et al. [22] to quickly construct rough depth maps for further processing. Although the depth maps are imprecise, images can be processed very fast and depth maps are merged, obtaining reliable high-resolution dense point clouds. An example using an even stronger planar prior is the work by Furukawa et al. [7], who search for points with strong confidence depths and normals, and find three dominant axes that are more

or less perpendicular. Hypothesis planes are proposed based on the points with strong confidence, and a MRF is used to assign pixels to hypothesis planes, obtaining a planar (Manhattan) world consisting of the most confident planes. Example images from their pipeline are shown in Fig. 2.4.

Although planes are a useful prior in indoor scenes and scenes with a lot of human-made architecture, the smoothness prior is more useful in natural environments (*i.e.,* general outdoor scenes). The approach of Hernández et al. [11] uses photo-consistency in scene-space, whereby projection lines of pixels (*i.e.,* a set of possible depth locations) are projected onto camera planes nearby, and the depth candidates giving the best overall photo-consistency are picked. The estimated depths are then used in their carving algorithm using 3D graph-cut on a voxel grid.

Larger areas can be reconstructed using video and reliable location estimation (for example, using SLAM techniques). Pollefeys et al. [24] developed a system for real-time urban 3D reconstruction. It uses GPS and inertia sensors in addition to image-based pose estimation for reliable localisation, and uses a plane-sweeping dense stereo algorithm with a prior preferring locations containing points from the SfM sparse point cloud. Again, depth maps are created at a high rate and then merged into more accurate models. A similar large-scale reconstruction method is described by Frahm et al. [6]. Building on the success of the Photo Tourism system [29] they collect publicly available images on the internet; however, large amounts of images make extensive matching of image pairs intractable, therefore their system finds clusters of images with use of their short image descriptor before structure from motion is applied on each cluster individually. For each cluster, dense stereo is used and clusters are merged based on Geo-tag location and inter-connected clusters, finally obtaining a dense reconstruction for a big part of a city, within a day on a powerful PC.

Chapter 3

Method

3.1 Overview

While space caving based on silhouettes (Structure from Silhouettes) has difficulties with high-textured environments and multi-view stereo techniques using depth maps have difficulties with low-textured or reflective environments, our method targets scenes containing both. Specifically, it aims to reconstruct low-textured and reflective objects without relying on the ability to find *any* features on the objects themselves - provided enough texture-rich surrounding objects are present. To get statistically reliable reconstructions, enough data needs to be provided; therefore, the proposed method works best with video footage shot while moving through the scene. The method is based on space carving and iteratively improves a model represented by a voxel grid. Carving is accomplished by shooting finite rays between camera poses and locations of triangulated features, affecting voxels hit by the ray segments only. This is different from photo-consistent space carving, which affects single voxels each step, and differs from silhouette-based space carving in the sense that carving stops at a specific point, allowing concavities to be reconstructed. Furthermore, the voxel grid is probabilistic, allowing confidence estimation for reconstructed three-dimensional structures depending on the amount of data used for each voxel.

Two algorithms based on visibility and occlusion information are proposed. The intuition that the ability to see a feature with known location means it is unlikely that there exists an object in between the observer and the feature is central to our approach, and not seeing an earlier seen feature means there could well be such an occluding object. As input, the algorithms require calibrated cameras, a reliable feature cloud, and visibility lists describing which cameras successfully detecting which points. The input can be provided by standard Structure from Motion techniques (Sect. 2.3). Output of the algorithms is a probabilistic (or thresholded) voxel grid representing occupancy probability, and can be used for further surface reconstruction if desired.

The first algorithm, called **Visibility Space Carving**, assumes all space is occupied by default, unless proven otherwise. Space between camera poses and corresponding detected features is then carved, that is, marked as free. In case the features are points without clear size (such as those obtained by standard SfM), we carve tubes the size of voxels in practice. Output are those voxels which do re-project in at least one image plane, but for which could not be proven to not occlude anything, either due to an actual occluder object or due to absence of information.

The second algorithm, called **Visibility-Occlusion Space Carving**, starts with unknown occupancy state voxels. The occupancy probability is then altered by both visibility and occlusion information. Occupancy probability of voxels in space between camera poses and detected features is increased, and space between camera poses and undetected features is made more likely to be occupied. A slightly altered version is proposed for cases where visibility information is much more reliable than occlusion information, which can be caused by bad matches, as is often the case for publicly available SfM systems.

Collectively, the proposed algorithms could arguably be called **Structure from Visibility**[1]. More detailed descriptions follow in Sections 3.2 and 3.3. For clarity reasons, a high-level overview of the complete pipeline from footage to reconstructed model is given in the next section.

3.1.1 Reconstruction pipeline

1. **Shoot footage and extract images**

2. **Structure from Motion** (Section 2.3) - Stable features (*i.e.,* SIFT) are found, matched over the sequence of images, and relative camera transformations are estimated. The features are triangulated and the resulting 3D feature clouds are combined (*i.e.,* Bundle Adjustment) to obtain global camera poses, features poses and visibility information of the features for all camera poses. Existing tools can be used for this step, which we review in Sect. 4.

3. Optional: **Extend visibility lists** (Section 3.4) - Many SfM systems are conservative with claiming visibility of features. It may help to review the visibility lists and attempting to extend them.

4. **Space carving** (Sections 3.2-3.3) - Space is partitioned into a voxel grid. Using the camera poses, feature locations, and visibility information, the voxel grid is carved, obtaining occupancy probabilities for all voxels.

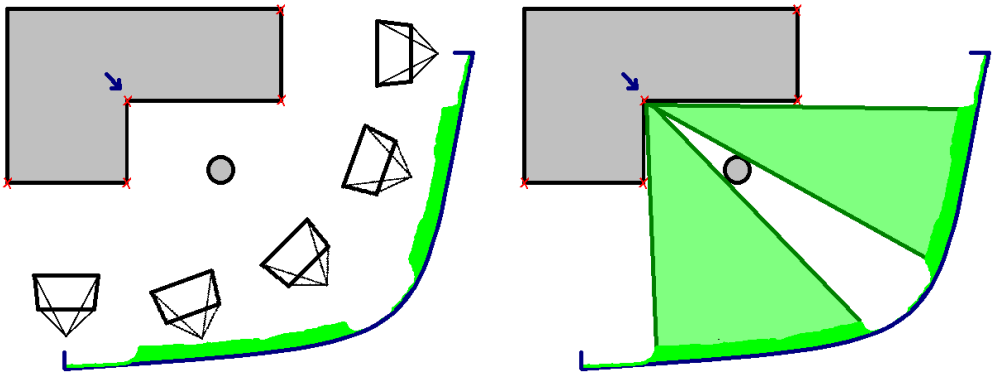
5. **Regularisation** (Section 3.5) - The probabilistic voxel grid - containing independent voxels - is feed to a 3D graph cut algorithm to obtain a global solution where local smoothness is promoted.

6. Optional: **Surface reconstruction** - Raw voxel grids can be processed further to obtain alternative surface representations such as a polygon mesh (*i.e.,* using Marching Cubes) or fitted surface (*i.e.,* Poisson Surface), followed by texture mapping. Surface reconstruction is well-established and will not be covered in this thesis. Our final result will be a voxel grid.

7. **Visualisation** (Section 5) - Depending on the final representation, results can be displayed in a appropriate way. They can be rendered from original camera poses, or new positions. They can be rendered as 3D voxel grid, depth map, annotated on original images (original poses only) or texture-mapped (useful for new poses). Being able to interact with a 3D model usually greatly improves the ability to estimate geometry.

[1] Structure from Motion uses motion to estimate structure, Structure from Silhouettes uses silhouettes, and the proposed algorithms use visibility to estimate structure.

3.2 Visibility Space Carving



(a) Probabilistic visibility track / list (curved blue plot) over image sequence for one interest point (blue arrow)

(b) Carving away space based on one visibility track

Figure 3.1: Graphical example of Visibility Space Carving. Cameras in (a) represent a dense sequence of camera poses, hence the 'interpolation' in between.

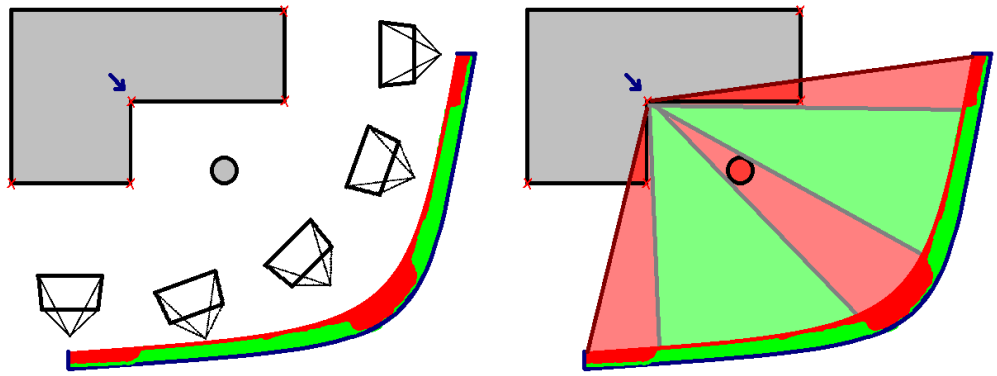
Algorithm 3.1 Visibility Space Carving

function VISIBILITYSPACECARVING(c, f, v, r) ▷ Arguments: cameras c, features f, visibility lists v, voxel grid resolution r
  $G \leftarrow G_{'unknown'}(r)$ ▷ Initialise voxel grid G with label 'unknown' and finite size
  for all Feature f_i in f **do**
    for all Camera c_j in v_{f_i} **do** ▷ Get all camera poses that detected this feature
      $X \leftarrow \text{getVoxelsBetween}(G, c_j, f_i)$ ▷ Get voxels in between camera and feature
      for all Voxel x_i in X **do**
        $x_i \leftarrow$ free ▷ Mark voxel as 'free'
      end for
    end for
  end for
  $H \leftarrow H_{'free'}(r)$ ▷ Initialise voxel grid H with label 'free' and same indexes as G
  for all Voxel G_i in G **do**
    $visible \leftarrow$ false
    for all Camera c_j in c **do**
      if $\text{reprojectsInsideImage}(G_i, c_j)$ **then**
        $visible \leftarrow$ true
      end if
    end for
    if $visible$ **then**
      $H_i \leftarrow G_i$ ▷ Only export the state of those voxels that could have been visible
    end if
  end for
  return H
end function

The Visibility Space Carving algorithm has its origins in Space Carving based on Silhouettes. It starts with a fully occluded world-view and carves away space that is not occluded. It does not carve space based on silhouettes until infinity, but carves until visible triangulated features (*i.e.,* SIFT interest points). Voxels between camera poses and features are found (*i.e.,* by shooting rays) and set to free. Since this is a binary process, the algorithm uses two labels instead of probabilities. The process for

one interest point is shown in Figure 3.1. Note that Structure from Motion can give an indication of the chance a feature is present (*i.e.,* how good the match is), but here we use thresholded, binary visibility lists and standard SfM tools often export only these. Also note that the example is shown in 2D and one interest point will only carve away one slice of space. If enough features are used, space will be carved away in a substantial part of the free space in the scene. After the carving step, a subspace of the grid is exported as output. Since voxels outside the camera views will always be occluded, we project the voxels onto the camera planes and export only those that project on at least one camera window. The algorithm is described more formally in Alg. 3.1.

3.3 Visibility-Occlusion Space Carving



(a) Probabilistic visibility track / list (curved blue plot) over image sequence for one interest point (blue arrow)

(b) Increasing (green) or decreasing (red) occupancy probability based on one visibility track. Space that is likely to get high occupancy probabilities after processing all data are coloured darker red.

Figure 3.2: Graphical example of Visibility-Occlusion Space Carving.

Assuming the world is occupied by default means that information is needed for every free part of space. In practice, footage often turns out to contain insufficient detected features to recover whole the scene. During initial tests with an implementation of Alg. 3.1 it became clear that reconstructed scenes typically contain a few regions which were free but for which there was simply insufficient positive information (*e.g.,* detected features on the background). Here, we present an algorithm that tries to overcome this problem. Regions for which no information is available are marked unknown, while negative information (*i.e.,* missing matches) needed to increase the probability of occluded regions. Two versions are presented: the general algorithm, and a slightly altered version for practical reasons.

The presented algorithm starts with an unknown world, with some prior probability value for occupancy. Similar to Alg. 3.1, voxels representing space between camera poses and features. However, both space towards detected and undetected features are affected. To this end, all features are projected onto all image planes. For the features that project inside the window borders of a particular camera (*i.e.,* all features that would have been visible if no occluders existed and matching was perfect), we check if they were detected or not. Features that were indeed detected contribute positive information, and so the occupancy probability of space between the particular camera and detected features is *decreased* by

a pre-defined constant factor Pr_{decr}. Features that did re-project inside the window borders, but were *not* detected, contribute negative information, and so the occupancy probability of space between the camera and undetected features is *increased* by a pre-defined (although potentially different) constant factor Pr_{incr}. The process for one interest point is shown in Figure 3.2. The algorithm returns the final probabilistic voxel grid. An occupancy threshold Thr_{occ} can be used to extract binary space partitioning where only voxels with high occupancy probability are set to 'occupied'. If desired, a second 'free-space' threshold Thr_{free} can be used for visualising voxels that are highly probable to be free. The general version of the algorithm is formalised in Alg. 3.2.

In practice, occlusion information is often less reliable than (positive) visibility information. Publicly available Structure from Motion solutions often act conservative while creating the visibility lists, and no probabilities are exported (Sect. 4). In the end, claimed visibility of features is very reliable, but claimed invisibility can well be a weak, but correct, match. To solve this balance equality, one could set the positive visibility information factor Pr_{decr} to a high value and set the occlusion information factor Pr_{incr} relatively low. Instead, we propose a modified version of the general algorithm whereby visible features cause a *veto* or absolute vote against occupancy of the space between camera pose and feature. Voxels within this space are set to Thr_{free}, which also prevents further increasing of occupancy probability. The modified version is shown in Alg. 3.3.

Note that all three algorithms require camera models c, features f, and (binary) visibility lists v, which are all provided by standard Structure from Motion techniques. The resolution of the voxel grid r needs to be set too. The two Visibility-Occlusion algorithms also need occupancy prior Pr_{unknown} and probability increasing constant Pr_{incr}. The general version needs probability decreasing constant Pr_{decr} too, where the veto version needs the threshold for free space Thr_{free}. For visualisation purposes, an occupancy threshold Thr_{occ} can be used too. Appropriate values for these probability and threshold parameters need to be chosen. Parameters Pr_{unknown}, Thr_{free} and Thr_{occ} can be kept constant in most cases, and indeed that is the case in the rest of this thesis. However, Pr_{incr} and, if necessary, Pr_{decr}, are dependent on the data provided. Relative displacement between subsequent frames, amount of texture in the scene, stability of the features in the dataset, and voxel grid resolution r are all factors that can influence the choice of these parameters. In practice, often a few parameter values can be tried and results can be inspected visually.

3.4 Extending visibility lists

Since visibility lists provided by standard SfM tools are often conservative, it is worth exploring possibilities to extend those lists. This is especially true for the proposed Visibility-Occlusion Space Carving algorithms, which rely on missing entries in visibility lists. Early experiments using Alg. 3.2 and Alg. 3.3 indeed showed small uncarved regions caused by incorrectly missing matches. Therefore, a short excursion in extending the visibility lists has been made.

Missing matches are either caused by features being truly occluded, or bad similarity scores by

Algorithm 3.2 Visibility-Occlusion Space Carving - General formulation

function VISIBILITYOCCLUSIONSPACECARVING(c, f, v, r, Pr_{unknown}, Pr_{decr}, Pr_{incr})
  $G \leftarrow G_{Pr_{\text{unknown}}}(r)$ ▷ Initialise voxel grid G with prior of 'unknown' space
  for all Camera c_j in v_{f_i} **do**
    for all Feature f_i in f **do**
      if $\text{reprojectsInsideImage}(f_i, c_j)$ **then** ▷ Get features projecting inside image borders
        $X \leftarrow \text{getVoxelsBetween}(G, c_j, f_i)$ ▷ Get voxels in between camera and feature
        if $c_j \in v_{f_i}$ **then** ▷ Feature is visible
          for all Voxel x_i in X **do**
            $x_i \leftarrow x_i - Pr_{\text{decr}}$ ▷ Thus, decrease occupancy probability
          end for
        else ▷ Feature was not detected
          for all Voxel x_i in X **do**
            $x_i \leftarrow x_i + Pr_{\text{incr}}$ ▷ Thus, increase occupancy probability
          end for
        end if
      end if
    end for
  end for
  return H
end function

Algorithm 3.3 Visibility-Occlusion Space Carving - Veto version

function VISIBILITYOCCLUSIONSPACECARVINGVETO(c, f, v, r, Pr_{unknown}, Pr_{incr}, Thr_{free})
  $G \leftarrow G_{Pr_{\text{unknown}}}(r)$ ▷ Initialise voxel grid G with prior of 'unknown' space
  for all Camera c_j in v_{f_i} **do**
    for all Feature f_i in f **do**
      if $\text{reprojectsInsideImage}(f_i, c_j)$ **then** ▷ Get features projecting inside image borders
        $X \leftarrow \text{getVoxelsBetween}(G, c_j, f_i)$ ▷ Get voxels in between camera and feature
        if $c_j \in v_{f_i}$ **then** ▷ Feature is visible
          for all Voxel x_i in X **do**
            $x_i \leftarrow Thr_{\text{free}}$ ▷ Thus, use veto and set to free
          end for
        else ▷ Feature was not detected
          for all Voxel x_i in X **do**
            if $x_i >= Thr_{\text{free}}$ **then** ▷ Only if veto has not been used ...
              $x_i \leftarrow x_i + Pr_{\text{incr}}$ ▷ ... increase occupancy probability
            end if
          end for
        end if
      end if
    end for
  end for
  return H
end function

other reasons than invisibility. To discriminate between the two, we project the features, in our case SIFT interest points, into the cameras that did not detect them. Descriptions of the regions around the projected features are extracted, and compared to descriptions of the same points projected onto image planes of the nearest camera that did detect the particular feature. For simplicity, we use patch matching where patches are centred around the projected points, but other descriptors such as SIFT descriptors can be used too. The patches are projections of pieces of geometry with unknown orientation; however, since sequences are used and nearest cameras (in absolute distance) are searched for, it is reasonable to assume

that relative orientation with respect to the cameras would not differ too much, and thus projections would not show many differences, if indeed the same geometry is projected. Since we use SIFT features - which do not have a clear size when SfM tools do not export their scale - and we therefore space carve tubes the size of voxels, we chose the size of the patches to be equal to the size of the projected voxel containing the SIFT feature. The descriptors (*i.e.,* patches) are compared and thresholded, obtaining possibly new matches. New matches are added to the visibility lists. After extending visibility lists, we can apply one of the Visibility-Occupancy Space Carve algorithms described before. Since the assumption of conservative, thus reliable, visibility lists is now lost and some of the new matches will be incorrect, the veto version is less suited and the general formulation (Alg. 3.2) is preferred.

3.5 Regularisation

The voxel grids returned by the algorithms described all contain voxels that are individual solutions: their occupancy probability or state is independent of their neighbouring voxels' probabilities. To obtain a better global solution we can introduce a prior promoting local smoothness. Together with the proposed algorithms, they form joint voxel-carving algorithms. The voxel grid can be converted to a 3D graph which can be solved by standard min-cut/max-flow graph-cut methods [1]. Voxels are thereby represented as nodes and nodes for neighbouring voxels are connected. Nodes have unary costs and pairwise costs. The unary costs U_i for connection with the Sink node are set equal to occupancy probabilities (after truncating to the range $0.0 - 1.0$), and connection with the Source is set to $1 - U_i$. Pairwise costs could have been based on some photo-consistency comparing appearances of pairs of re-projected voxels, but this would introduce dependencies on appearances again, which were tried to be dropped in the first place. Therefore, pairwise costs are based on differences in occupancy probability weighted with a constant factor, as $P_{i,j} = \gamma * (1 - |U_i - U_j|)$.

Chapter 4

Implementation

In this chapter, details on implementation of the proposed method are discussed. Code is listed in Appendix D, supplied on DVD, and made available online[1]; this chapter will complement the code and will serve as a general introduction, therefore no code is listed in this chapter. The pipeline given in Section 3.1.1 will be used as a guideline, allowing the reader to follow the process in execution order. This order, though, is not necessarily the order of implementation; in fact, most applications developed for visualisation (last step in the pipeline) were written in an early stage, allowing for following progress and viewing intermediate steps. Developing geometry reconstruction algorithms greatly benefits from the ability to quickly shoot footage, run suggested and implemented algorithms, *visualise* intermediate steps, and suggesting modified algorithms. Details on collecting datasets (first step in the pipeline) are discussed in the next chapter; implementation details of the remaining steps in the pipeline are discussed after an overview of the libraries that are being used.

4.1 Libraries

All tools are written using platform-independent and open-source libraries, and all code should compile and run on all major platforms (although extensive tests have only been done under Linux, Ubuntu 12.04). The same holds for all tested external tools (SfM step), except Voodoo which is only free for research-purposes. All tools developed for this thesis are written in C++ and a cmake config file is provided.

The well-established computer vision library **OpenCV**[2] Bradski [2] is used for general image and video i/o, and 2D image and feature visualisation. Very recently, a sister project called **Point Cloud Library**[3] (PCL) was announced by Rusu and Cousins [26], providing common data structures, algorithms, and tools useful for 3D computer vision. Although PCL still lacks extensive documentation and has a rapidly changing API, it does provide useful data structures and tools for handling point clouds, including advanced filter, segmentation and surface algorithms. Regrettably, it has (yet) no data structures for visibility information nor a commonly-agreed camera representation, reflecting the rare interest in visibility and occlusion (Sect. 2.3), so they have been developed for this thesis. In addition, the

[1] Code available on Google Code: http://code.google.com/p/martijn-msc-thesis
[2] OpenCV homepage: http://opencv.willowgarage.com
[3] PCL homepage: http://pointclouds.org

provided voxel grid representations are mostly useful for space partitioning and searching. Instead, we used the more suitable and feature-rich library **OctoMap**[4], recently released by Wurm et al. [16]. OctoMap was chosen because it makes use of probabilistic nodes, implements a memory-efficient octree instead of same-size voxels, and allows for ray shooting. It also includes an octree viewer. Lastly, a set of open-source C++ files released by Boykov and Kolmogorov [1] has been used for efficient graph-cut regularisation. Needless to say, writing methods to convert between representations of these four libraries was inevitable.

4.2 Pipeline implementation

4.2.1 Structure from Motion

Camera models and a feature point cloud can be obtained by standard Structure from Motion algorithms (Sect. 2.3). Structure from Motion has been implemented oftentimes already. Nowadays, reliable software tools are available for processing image sequences and outputting camera poses and point clouds. Both commercial and open-source tools exist. Since the proposed algorithms use SfM output as is and do not innovate on the SfM process itself, we experimented with three freely available Structure from Motion tools instead of building our own. The tools tested are Bundler, Voodoo, and VisualSfM. All tools are used with default settings.

Originating as part of the Photo Tourism work [29] and released under the GPL open-source license, **Bundler**[5] became an authority and is still one of the most cited SfM systems. Originally developed for unordered photo collections taken from the internet, it claims to work on normal sequences too. The command line tool takes a directory with images as input, extracts and matches SIFT features and descriptors by default, and optimises estimated parameters incrementally using sparse bundle adjustment. Bundler outputs an ASCII file containing camera models, triangulated points with colours and visibility lists. The latest binary version (0.3) was tested.

Voodoo[6] was developed as non-commercial alternative to software tools such as Boujou and is free to use for research purposes. Finding and matching SIFT descriptors is possible, although by default Harris edge detection is used and points are tracked over the sequence. Voodoo is developed for sequences only for which the small displacement assumption is reasonable. The GUI allows for manually fine-tuning of estimated feature tracks (*e.g.,* linking a lost feature point to its rediscovery a few frames later) and includes simple modelling tools for improving the results, but those are not used during testing. Voodoo outputs various ASCII file formats, but unfortunately none of them includes visibility information. The latest version (1.2.0 beta) was tested.

Recently, a new graphical tool called **VisualSfM**[7] by Wu et al. [31] was released that constitutes a graphical shell around a GPU implementation of both SIFT (SiftGPU) and Bundle Adjustment by Wu et al. [31] for SfM, and the CMVS/PMVS tools by Furukawa and Ponce [8] for patch-based dense multi-view stereo. VisualSfM and its components are all released under the GPL license. VisualSfM

[4] OctoMap homepage: http://octomap.sourceforge.net
[5] Bundler homepage: http://phototour.cs.washington.edu/bundler
[6] Voodoo homepage: http://www.digilab.uni-hannover.de/docs/manual.html
[7] VisualSfM homepage: http://www.cs.washington.edu/homes/ccwu/vsfm/

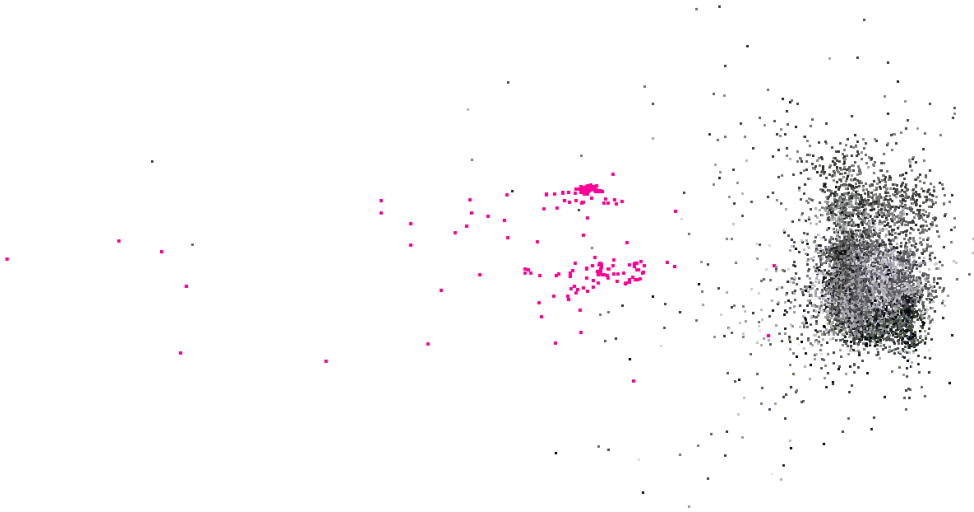
outputs an ASCII file similar to Bundler's format, and a binary file after the optional dense reconstruction step. The interface offers visual feedback during incremental bundle adjustment, and provides quite a few other useful visualisations. The latest version (0.5.17) was tested.

Example outputs are shown for two datasets; one example frame for each is pictured in Fig. 4.1). The chess dataset gives typical results and are shown in Figure 4.2; good results are obtained for the houses1 dataset, shown in Figure 4.3. More details on all datasets used in this thesis are listed in Appendix C.

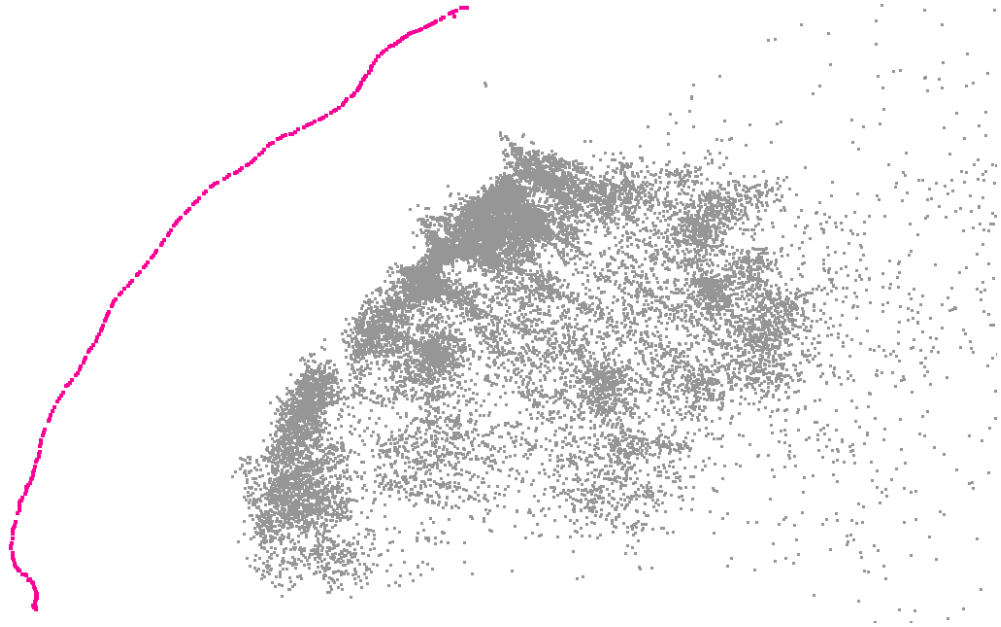
In general, Bundler performs poorly on the supplied datasets. Output usually consists of a point cloud without identifiable structures. An explanation for the bad results can lie in the fact that Bundler is developed for unordered sequences and therefore uses no prior for camera displacement nor for possible identical camera models. Indeed, estimated camera locations often have a variety of distances (and intrinsics) from the point cloud, where the other two tools place cameras in a string. Voodoo often exports reasonable results with, indeed, trustworthy looking camera paths and point clouds. It does not export colours and does not automatically rediscover points lost during tracking earlier on in the sequence, and no bundle adjustment is used to improve results. Using SIFT matching does not improve results, and matching only occurs between nearby frames. Voodoo results often contain a fair amount of points triangulated very far away from the scene. More importantly, none of the variety of export formats contains visibility lists. VisualSfM gives good results on almost all tested sequences. Due to its GPU implementation, it is faster than Bundler (and Voodoo in SIFT matching mode): two hours on sequences of 500 frames against half a night. Visual feedback during reconstruction and bundle adjustment is useful for monitoring progress. Due to Bundler's bad results, Voodoo's lack of visibility lists, and VisualSfM's good performance over all, VisualSfM is used for all further experiments.

(a) Frame 70 of chess dataset (b) Frame 11 of houses1 dataset

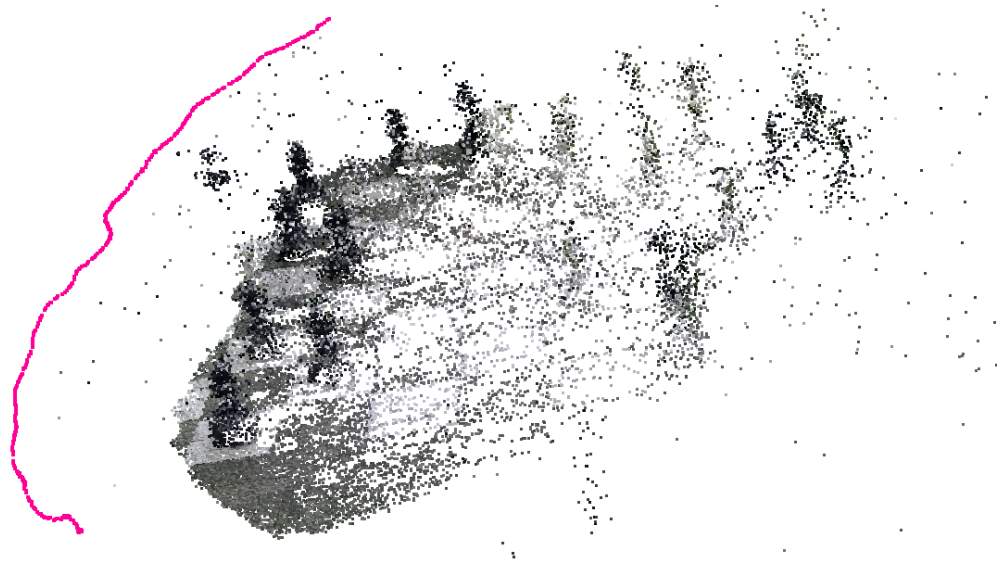
Figure 4.1: Example frames for chess and houses1 dataset (used for SfM tools comparison).



(a) Bundler output



(b) Voodoo output

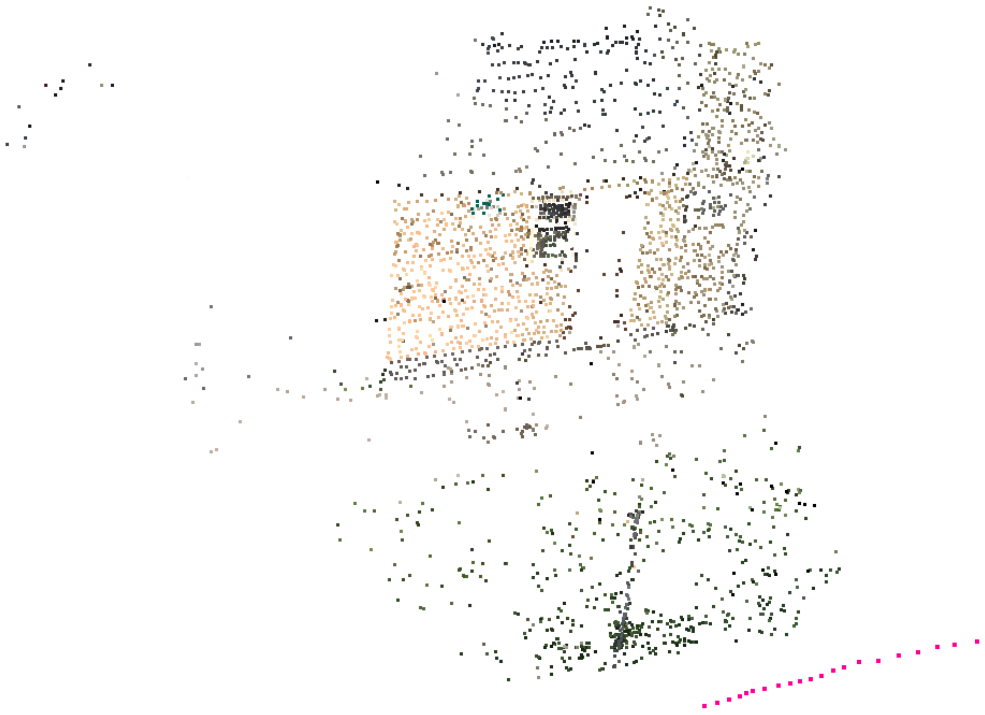


(c) VisualSfM output

Figure 4.2: Comparison of Structure from Motion tools Bundler, Voodoo and VisualSfM; typical example (chess dataset). Points shown in estimated colour (or grey if not given), camera poses displayed in pink. Visualisations by `sfmviewer`.

(a) Bundler output

(b) Voodoo output



(c) VisualSfM output

Figure 4.3: Comparison of Structure from Motion tools Bundler, Voodoo and VisualSfM. Good example (houses1 dataset). Points shown in estimated colour (or grey if not given), camera poses displayed in pink. Visualisations by `sfmviewer`.

4.2.1.1 Reading SfM output

A common interface class `sfm_reader` has been written for reading the various different formats used by different SfM tools. Furthermore, data structures from the C++ standard library are used to represent visibility information. Another reason to discuss this particular class is that it also provides useful methods for the carving algorithm concerning visibility and occlusion.

Four file formats parsers are written. The ply format (Stanford) is shared among SfM tools; however, it only contains points or polygon meshes. Ply files consist of a table (therefore, no variable length lists can be represented) and data can be binary or ASCII. The binary variant can be read by the PCL library, but for the ASCII data variant a parser has been written. Bundler and VisualSfM can both output the ASCII ply format, and VisualSfM outputs the binary ply format after dense reconstruction with CMVS/PMVS. Voodoo can output scripts for various programs (such as Blender). Its native output format has the common extension .txt and contains a list of points and a list of camera models. Bundler's native output format has extension .out and contains camera models, points, and a visibility list for each point. VisualSfM's native output format has extension .nvm and contains the same information as Bundler's, although it contains a different camera model using quaternions instead of R, t matrices, and distortion has one parameter instead of two. For all three native formats a parser have been written.

Points are saved in a `PointCloud` object as provided by PCL. It uses `PointXYZRGB` points which contain a location and an RGB colour. Cameras are saved in two separate structures: camera poses are saved in another `PointCloud` for easy visualisation - no pyramid-like camera visualisation wrappers are provided by PCL - as well as in a separate list of `camera` structures. The user-defined `camera` structure contains intrinsics R and t, focal length, and two radial distortion parameters. All camera models from the different file formats are converted to this structure (*e.g.,* quaternion to R matrix). Each visibility list is saved in a `std::map` object mapping every frame index for which a point was detected to a `visibility` structure. This user-defined structure contains the index of the feature for the particular frame, and the x and y coordinate of detection in the frame. Every feature point now has a global (x, y, z) position in the `PointCloud` object, and a list of local (x, y) camera coordinates for those cameras that detected the feature point. The map object allows for fast checks on visibility given a frame and feature point number, even for larger sets of frames.

Class `sfm_reader` provides re-projection methods for given feature point and camera, including re-distortion. The method `reprojectsInsideImage` returns a boolean indicating whether a given point re-projects inside a camera window or not. This method is also used by `selectPointsForCamera`, which re-projects all points inside a given camera and makes an occlusion list of those points that could have been visible, but have not been detected (*i.e.,* the camera is not present in the visibility list). A list of currently visible points is made as well.

4.2.2 Extending visibility lists

The optional step of extending visibility lists checks all point-camera pairs that are not represented yet in the vector of maps of visibility information. In practise, this is implemented by looping through the points and, for each point, creating a visibility and invisibility list with help of the re-projection

functions. Then, the whole sequence of a particular point is followed and all invisibility entries (re-projects inside image but not detected) are checked. The check consists of finding the nearest camera in absolute distance that did detect the feature point, re-projecting the point into that image and the current image, and comparing patches around the points. The size of the patches is determined by translating the feature point half the size of a voxel (given by the octree) in a direction perpendicular to the vector between the point and camera pose, re-projecting the translated point into one of the images, and calculating the distance to the projection of the original point. This distance is an approximation of the size of a projected voxel near the feature point, and is used to extract rectangular patches from the two images. They are compared by calculating the mean squared distance. A value below the threshold causes the camera index to be added to the visibility list of the feature point. By manually inspecting patch pairs and mean squared distance values, the threshold was set to 0.2 for all further experiments.

4.2.3 Carving

The proposed carving algorithms are implemented using the probabilistic octree of the OctoMap library [16]. The octree is initialised with a single parameter specifying the smallest node size possible. By default, all ray shooting operations are executed on nodes at this smallest scale, that is, the leaf nodes in the tree representation. Various node types are provided, the one containing probabilities is used. Probabilities are stored logarithmically, but conversion to and from normal probability values is straightforward. Two thresholds are used to allow nodes to be labelled with either 'free' ($val <= Thr_{free}$) or 'occupied' ($val >= Thr_{occ}$), or no label (unknown). Those thresholds are $Thr_{free} = 0.2$ and $Thr_{occ} = 0.7$ by default, and are kept at these values for all experiments. For memory efficiency, eight (leaf) nodes can be merged recursively if they have the same label, thereby averaging and thus losing individual occupancy probabilities. By default leaf nodes are not initialised and depth of the tree is kept to a minimum.

For the implemented space carving algorithms, the octree is initialised with smallest node size equal to the largest axis variation in the feature point `PointCloud` divided by a given resolution parameter. OctoMap provides two ray shooting methods: `computeRayKeys` takes as input two points and returns a list of node keys that were hit by the ray between the two points; `insertRay` does the same, but sets the last node to Thr_{occ} and all the others to Thr_{free} instead of returning the list. The latter is used for the binary Visibility Space Carving algorithm (Alg. 3.1), and the former is used for both probabilistic Visibility-Occlusion Space Carving algorithms (Alg. 3.2 and 3.3), there where `getVoxelsBetween` is mentioned. Note that since we carve tubes of space in this implementation, the resolution parameter potentially has a big influence on the result. In practise, there is a range of possible resolution values that give good results, and the range is laid down by the dataset provided.

4.2.4 Regularisation

Regularisation is implemented used the code provided by Boykov and Kolmogorov [1]. For simplicity, the octree is converted to a voxel grid with voxels the size of the smallest possible octree node. First an empty voxel grid is initialised using the occupancy prior for unknown space, which we set to $Thr_{unknown} = 0.2$. Since conversion splits existing bigger nodes and adds nodes where nothing was

initialised before, this increases memory usage quite a lot. This means that only lower resolution settings can be used in the regularisation process. After applying the graph-cut algorithm the voxel grid is converted back to the memory-efficient octree representation.

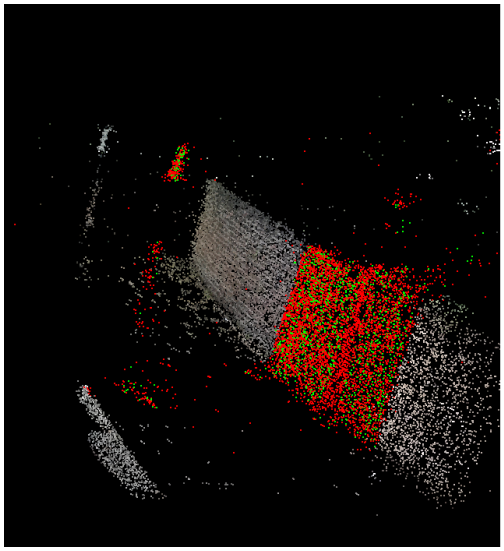
4.2.5 Visualisation

During the pipeline, a few intermediate and final results can be visualised. Three visualisation tools are developed and one provided by the OctoMap library is used. We will discuss each briefly. The first two tools are used for visualising features before carving, one 2D image annotator and one interactive 3D tool. The two other tools are meant for visualising voxel grids after carving, again one 2D image annotator and one interactive 3D tool.

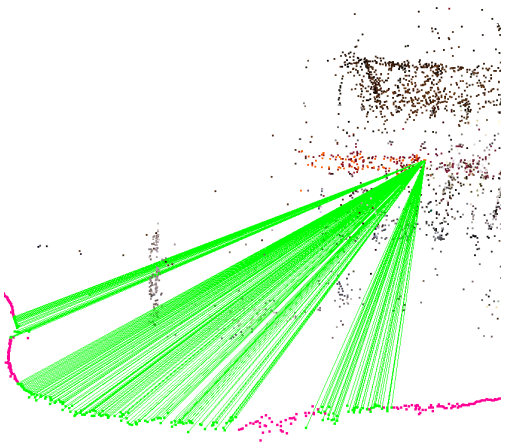
Although external structure from motion tools are being used, a tool has been written for experiments with and visualisations of various feature detectors. It can be used to check the amount of interest points discoverable in a dataset, and the uniqueness of their descriptors during tracking. It is not used in the final pipeline, but is discussed for completeness. The visualiser, `featureviewer`, takes as input either a video, directory with images, or webcam stream, and annotates the images with a given type of interest point (*e.g.,* SIFT, SURF, ORB, FAST, HARRIS) including their size, plus edges (Canny edge-detector). One can interactively click on a feature point, which will be matched over the sequence by a simple ratio test: if the ratio of the two best matches in the next or previous frame (*i.e.,* the L1 distance) is big, the interest point is quite unique and the best match is taken. If the ratio is close to 1, we keep the last known good descriptor and continue to the next frame. Note that this simple heuristic does not use any location prior. We can define the visibility confidence as one minus that ratio, and plot it for the sequence, similar to the visibility tracks shown in Fig. 3.1 and 3.2. One example image processed with `featureviewer` is shown in Figure 4.4(a).

Visualisation of triangulated SfM feature points or dense point clouds can be done with `sfmviewer`. The SfM tools outputs (Fig. 4.2 and 4.3) were made this way. It uses class `sfm_reader` for opening SfM output files and representing their contents. The PCL library provides wrappers around the VTK visualisation toolkit. It is used for easy point cloud visualisation, both the feature point cloud and camera poses. Interaction is implemented using the default PCL visualiser for mouse navigation, plus VTK call-back functions for custom shortcuts, and selection of points and cameras. Selecting a point colours the cameras that detected the point (using the visibility list of the point). Selecting a camera calls `selectPointsForCamera` as described earlier, thereby projecting all the points into the camera. Detected points are coloured green, while undetected points that do re-project within the image borders but are coloured red. To be able to recover the original point colours during the next selection, `sfm_reader` keeps a shadow copy of the original point cloud. Another implemented feature is the ability to visualise those visible (green) and invisible (red) points onto the original corresponding frame. The camera-feature pairs can be visualised even more explicitly by drawing green lines between the camera and visible points or between the point and detecting cameras. Two example visualisation is shown in Fig. 4.4(b) and 4.4(c).

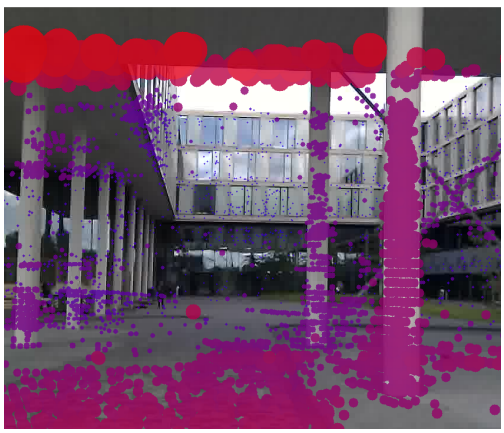
(a) `featureviewer` for frame in chess, including SIFT features and visibility track for the green interest point



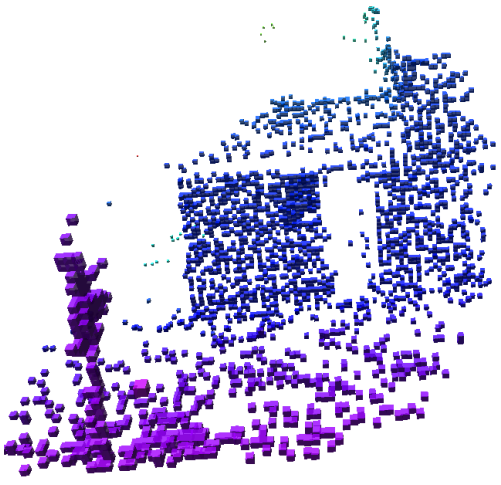
(b) `sfmviewer` for selected camera of lamp-posts_on_wall1; one lamppost causes a red area to emerge on the wall, but the conservative visibility information of VisualSfM is noticeable too in the 'green' areas



(c) `sfmviewer` for selected point of sainsburys1; two reflective poles become 'visible' due to absence of lines



(d) `carveviewer` for frame in sciencepark2 (with incorrect nearby voxels at the top and only few detected features)



(e) `octovis` view for houses1; a partly reconstructed lamppost is visible on the left (purple)

Figure 4.4: Visualisations by three developed tools and one provided viewer

After space carving has been done, a probabilistic octree is the result. For visualisation purposes, we use the occupancy threshold Thr_{occ} to make a binary distinction between occupied and not occupied. The last developed visualisation tool, `carveviewer`, annotates the original images with the carved octree. To prevent axis dependent results, all nodes are imagined as spheres and projected on top of the images. As with the code for extending visibility lists, both the node centres and centres with offsets the size of the nodes are projected into the images, and the projected points are used to estimate the radii, resulting in a list of x, y, r entries. Then, rays are casted from the camera in the direction of the projected nodes, and all rays hitting another, occluding node before the projected one, are removed from the list. The remaining nodes are visible from the given camera and are drawn as semi-transparent circles on top of the image. The circles are drawn with the estimated radii and coloured according to the log distance from the camera (nearby voxels are red, voxels far away blue). One example rendering is shown in Figure 4.4(d). For more dense voxel grids, results remind of depth maps on top of their original images.

An interactive octree viewer is provided by the OctoMap library. It has been used without modification. Nodes are rendered as semi-transparent cubes which are either light blue for occupied and dark blue for very high occupancy probability (threshold undefined), or coloured according to the value of one of the axes. Optionally, free space (occupancy probability below Thr_{free}) can be visualised too as green cubes. One example view is shown in Figure 4.4(e).

The visualisation tools will be used extensively in the next chapter.

Chapter 5

Results and Evaluation

5.1 Data capturing

The proposed method was designed for scenes containing objects occluding other, rich-textured objects. While alternative methods work best for either low-textured objects (silhouette space carving) or high-textured objects (MVS with depth maps), the proposed Structure from Visibility alternative is expected to reconstruct scenes containing both. Specifically, the proposed method has no requirements with regard to the appearance of occluder objects in scenes containing other sufficiently textured objects. Therefore, it should work on complex coloured scenes containing low-textured or even reflective objects, while existing techniques will have difficulties reconstructing those kind of objects. Unfortunately, datasets containing those kind of scenes are rare, and none are known by the authors that contain ground truth models. For example, the Middlebury Multi-View Stereo dataset[1] [27] does provide ground truths, but only contains statuettes made of approximately Lambertian objects, and background-foreground segmentation is easy. Therefore, we collected our own data and evaluate the proposed and implemented system visually by comparison with state-of-the-art.

Outdoor scenes containing low-textured and reflective objects were captured using two different cameras: a Google Nexus One phone-camera and a Sanyo HD camcorder. In filming mode, the phone-camera gives surprisingly low-quality looking 720x1280 resolution images; the HD camcorder was set to 1080x1920 and all other settings were set manually to obtain clear footage for every scene. Datasets are listed in Appendix C; datasets and results are supplied on DVD and made available online[2] in various native output formats and as screen-casts. Here, we show two typical results (Figures 5.2 and 5.4) and two good results (Figures 5.6 and 5.8. As mentioned in Section 1, the interested reader is encouraged to view the supplied three-dimensional results in addition to the 2D snapshots shown here.

Each result consists of one example frame, the sparse point cloud as reconstructed by VisualSfM (`sfmviewer`), the same sparse point cloud discretised as octree (for comparison), result of Visibility Space Carving (Alg. 3.1) and Visibility-Occlusion Space Carving Veto version (Alg. 3.3) as 3D model (`octovis`) and occasionally as annotated image too (`carveviewer`), and the result of the patch-match based dense reconstruction algorithm CMVS/PMVS by Furukawa and Ponce [8] from 2010

[1] http://vision.middlebury.edu/mview/

[2] Link to datasets available at: http://code.google.com/p/martijn-msc-thesis

(`sfmviewer`). For the Visibility-Occlusion algorithm, a few values for Pr_{incr} are tried and the best result is shown. Note that our space carving pipeline currently does not include texture mapping; the results will be judged based on shape rather than photo-realistic rendering.

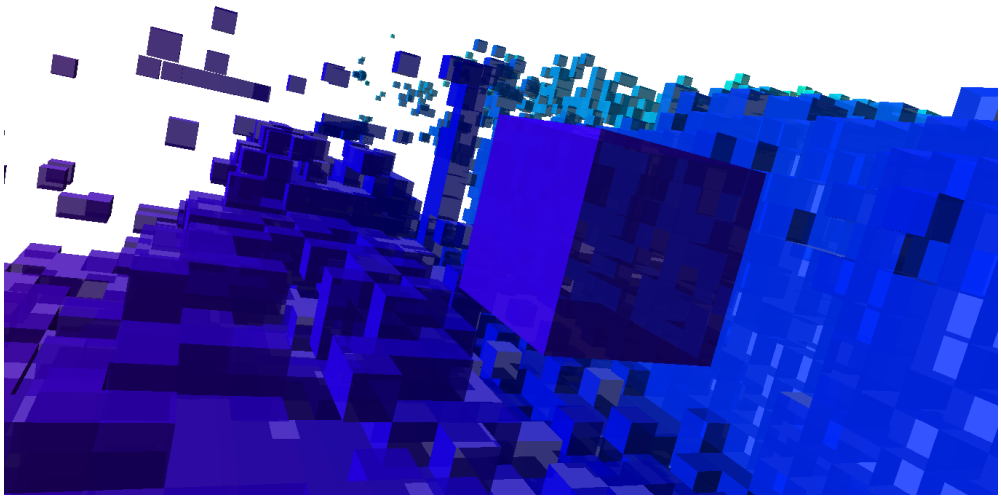
Before showing the four complete examples, we note that regularisation and the extension of visibility lists (and thus Alg. 3.2) are not applied for the shown examples. However, results of both are shown in figure 5.1 in comparison with the result of Alg. 3.3 at low resolution ($r = 250$). In general, regularisation removes some clutter and fills holes in reconstructions, but sometimes also removes occupied space at the surface of real objects. Results usually look a bit smoother, but do not change the result drastically. As mentioned before, regularisation results can only be obtained for low resolution settings and thus regularisation is omitted for the remaining of the results. Extending visibility lists followed by the general Visibility-Occlusion Space Carving algorithm requires an extra parameter (Thr_{decr}) and relies on the extension method (now simple patch-matching). In general, we obtained better results using the conservative visibility lists in combination with the veto version of the algorithm. We did not explore other extension methods (such as feature descriptor matchers), nor did we elaborate the influence of parameters. Further results obtained using extended visibility lists and Alg. 3.2 are also omitted.

Recently, Autodesk announced plans for releasing a new product aiming at geometry reconstruction[3] called Photofly / 123D Catch and featuring promising results in an online sharing community. Indeed, our (easy) houses1 dataset returns a good model similar to the result of CMVS/PMVS in 5.3. Details on the underlying algorithm are not given, but a beta version can be tried that sents datasets to Autodesk's cloud computing backbone. We submitted a few datasets that we believed to be more challenging, but after eight days only one result has been returned. It is shown in Figure **??**, next to our result and the result of CMVS/PMVS for the same dataset (sainsburys3) for comparison.

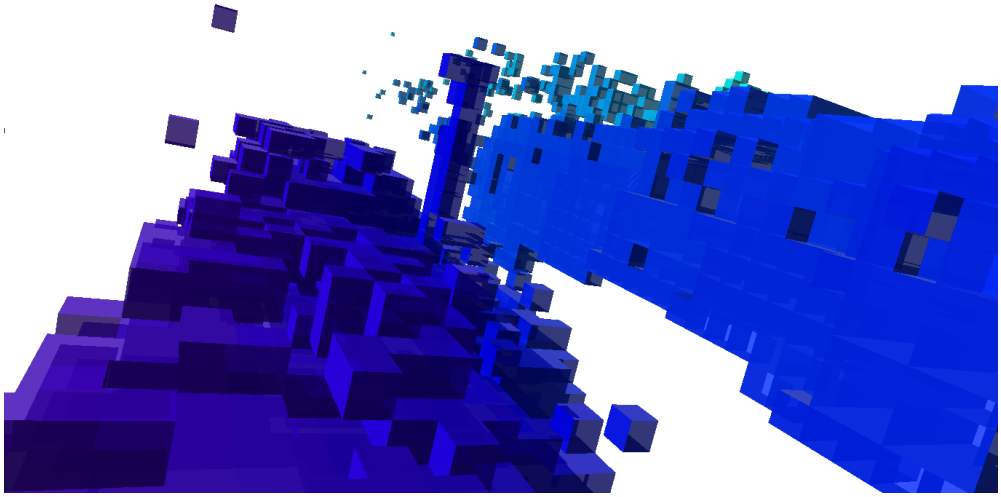
Typical run times for our own pipeline (on an AMD Phenom X4 quad-core 3.2 GHz) for sequences of around 400 images are 1.5 hours for structure from motion (VisualSfM), an additional 1.5 hours for dense reconstruction, or 10 minutes for carving with $r = 2500$ (1 minute with $r = 250$, plus 5 seconds for regularisation).

[3] http://www.123dapp.com/catch

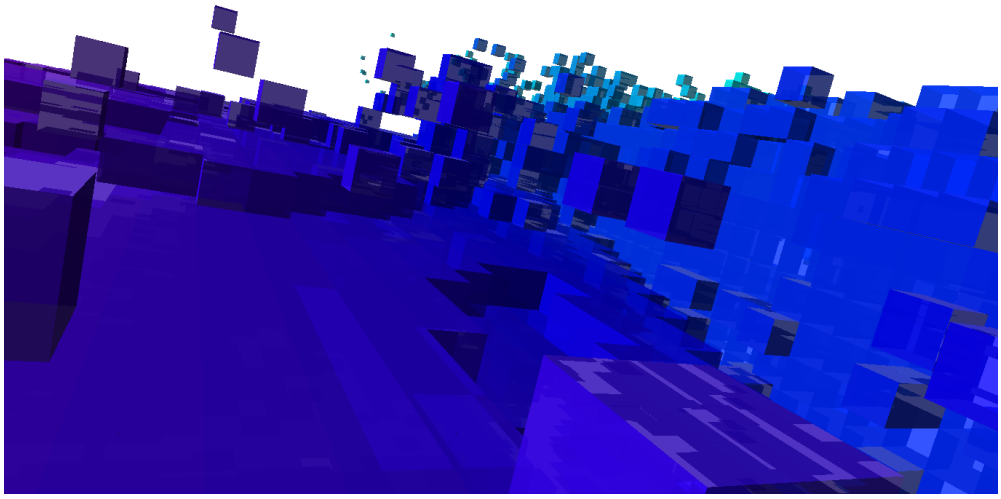
5.2 Visual results



(a) Result of our Visibility-Occlusion Space Carving Veto (Alg. 3.3) with $r = 250$, $Pr_{incr} = 0.005$

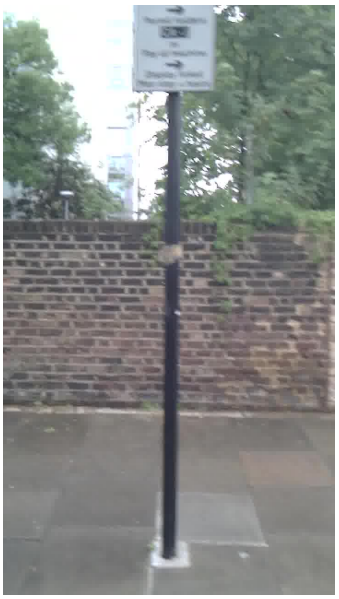


(b) Result after regularisation (graph-cut) with $\gamma = 1.8$

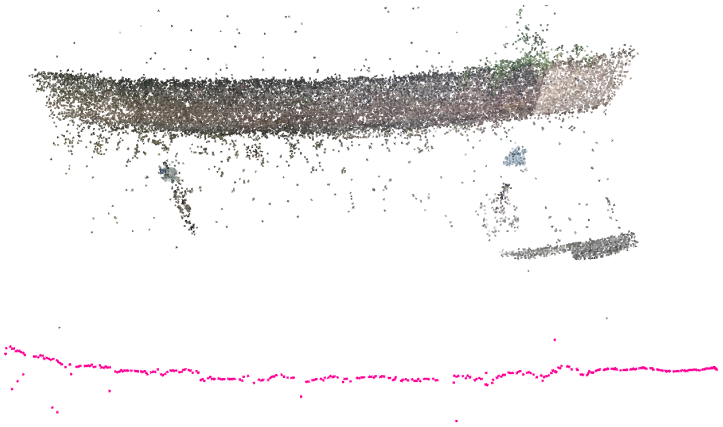


(c) Result of our Visibility-Occlusion Space Carving General (Alg. 3.2) with $r = 250$, $Pr_{incr} = Pr_{decr} = 0.001$

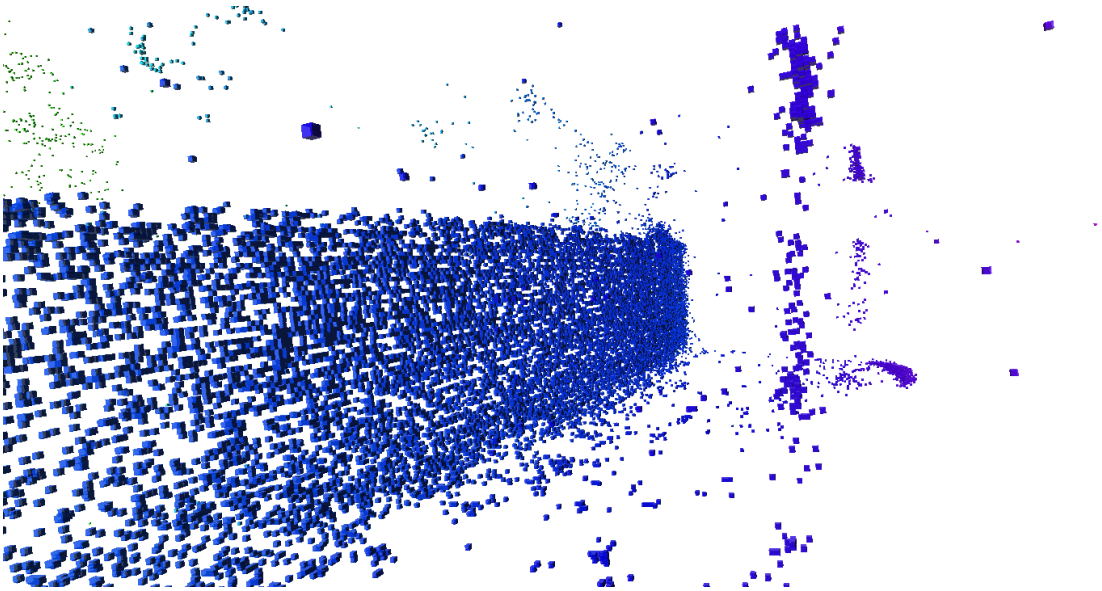
Figure 5.1: Regularisation and extended visibility list examples car_and_wall1 dataset, picturing two cars and one lamppost (more details visible in Figures 5.4 and 5.5)



(a) Example frame; notice the poor image quality

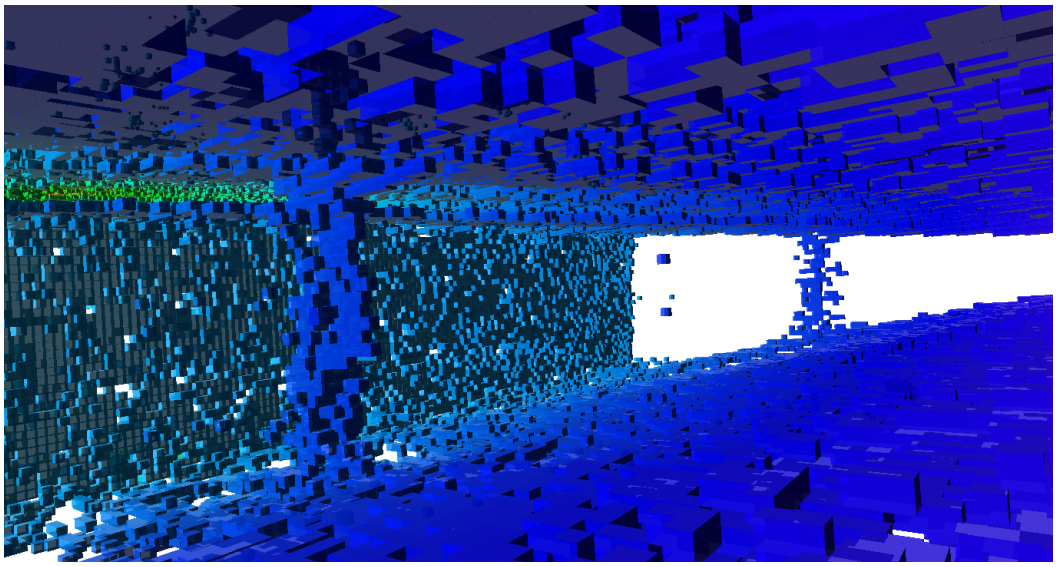


(b) Sparse point cloud and camera poses

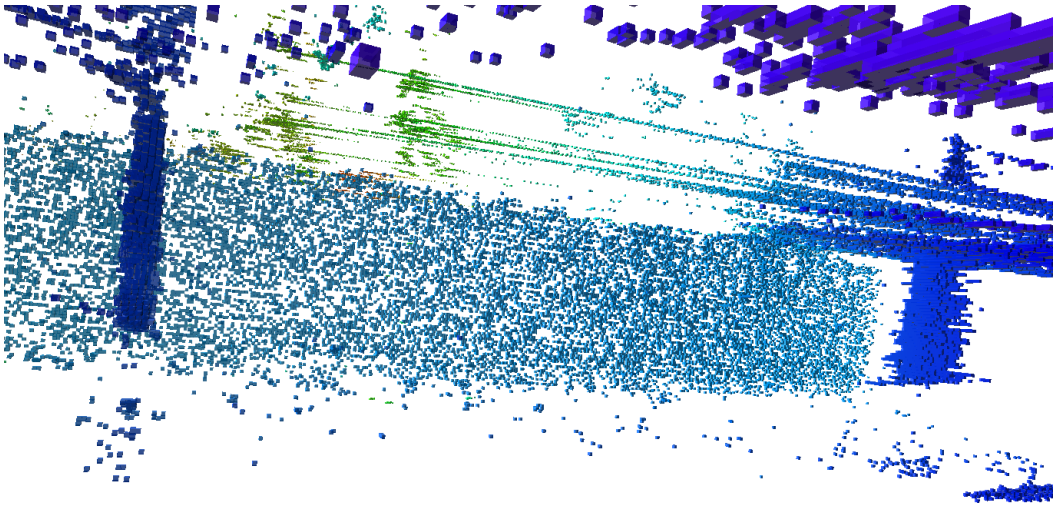


(c) Discretised sparse point cloud ($r = 1000$)

Figure 5.2: Typical result 2: lampposts_on_wall1 dataset



(a) Result of our Visibility Space Carving (Alg. 3.1) with $r = 500$; Notice the occupied-labelled space above and underneath the carved space, caused by an insufficient amount of feature points in the trees above the wall and on the ground.

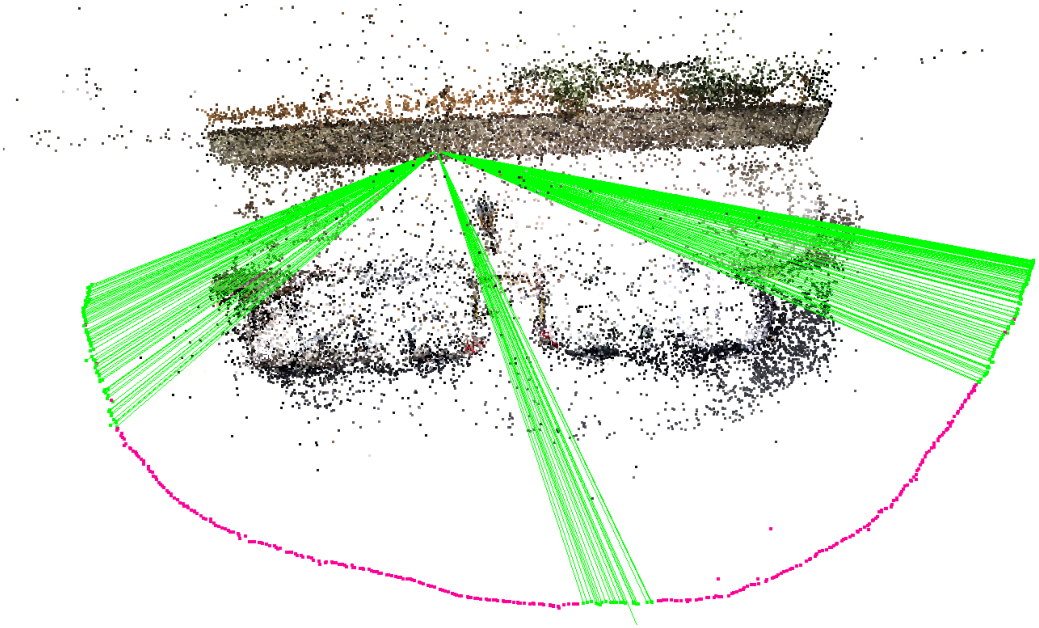


(b) Result of our Visibility-Occlusion Space Carving Veto (Alg. 3.3) with $r = 1000$, $Pr_{incr} = 0.005$

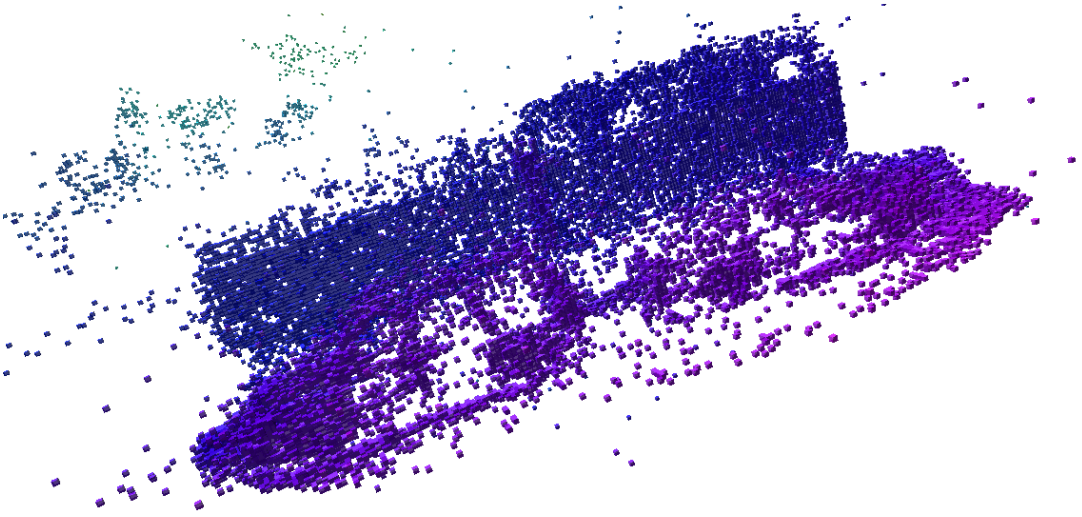
(c) Result of CMVS/PMVS [8]

Figure 5.3: Typical result 2: lampposts_on_wall1 dataset (cont.)

(a) Example frame

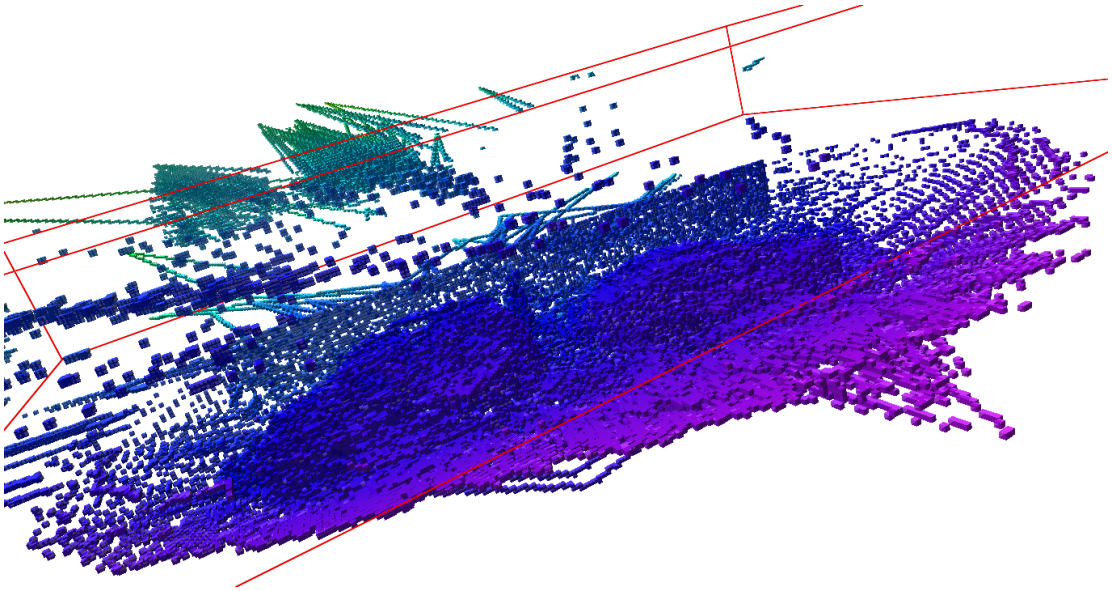


(b) Sparse point cloud and camera poses, annotated with visibility lines from one point (hinting at possible occluding object locations)

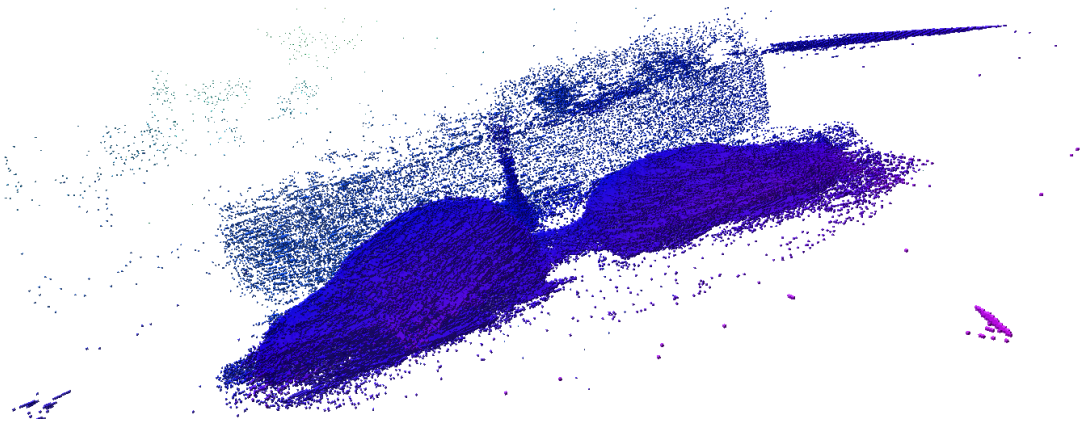


(c) Discretised sparse point cloud ($r = 1000$)

Figure 5.4: Typical result 2: car_and_wall1 dataset



(a) Result of our Visibility Space Carving (Alg. 3.1) with $r = 1000$; Incorrectly occupied-labelled space inside the red 'box', caused by an insufficient amount of feature points above the wall (*e.g.,* on trees), was removed for visualisation purposes.

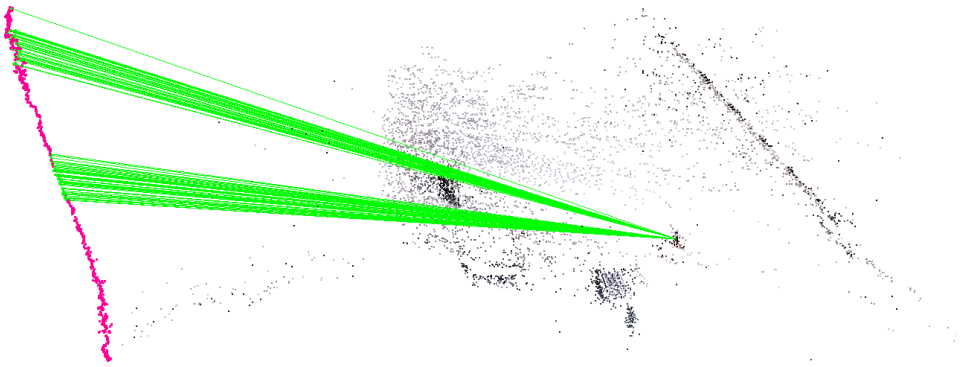


(b) Result of our Visibility-Occlusion Space Carving Veto (Alg. 3.3) with $r = 2500$, $Pr_{incr} = 0.005$

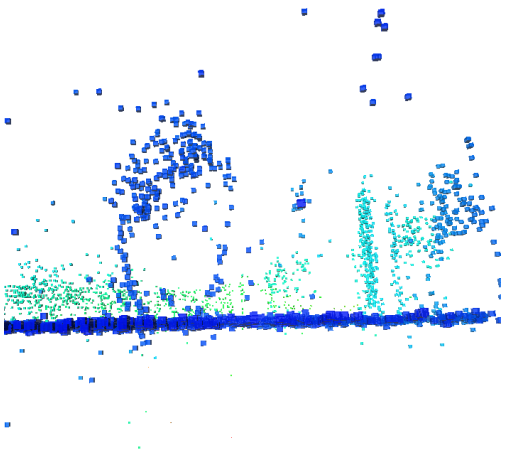
(c) Result of CMVS/PMVS [8]

Figure 5.5: Typical result 2: car_and_wall1 dataset (cont.)

(a) Example frame



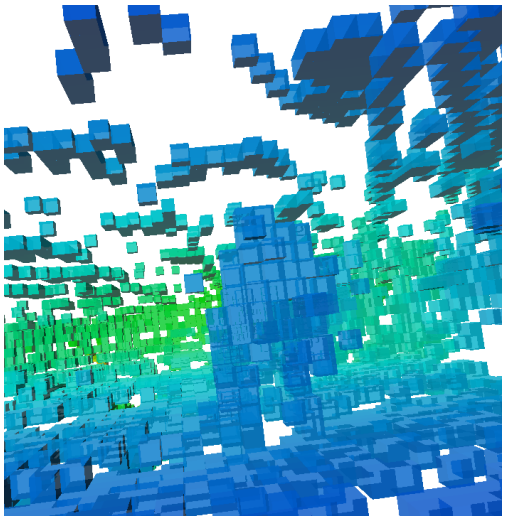
(b) Sparse point cloud and camera poses, annotated with visibility lines from one point



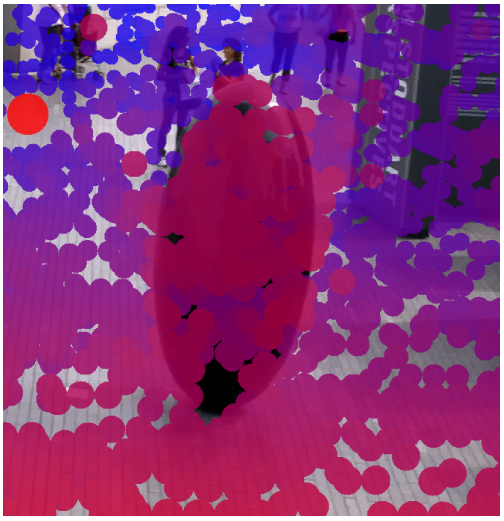
(c) Discretised sparse point cloud ($r = 1000$)

(d) Discretised sparse point cloud ($r = 1000$), carveviewer; notice the incorrectly visible ground voxels (blue-ish) on the lower half of the sculpture

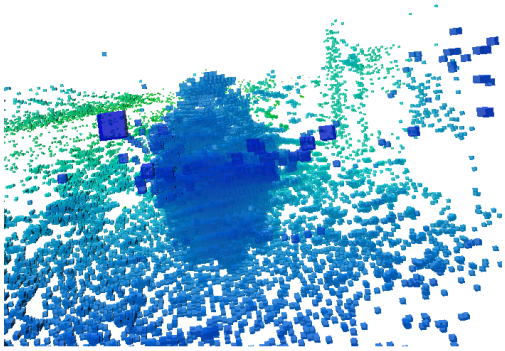
Figure 5.6: Good result 2: memorial dataset



(a) Result of our Visibility Space Carving (Alg. 3.1) with $r = 250$



(b) Result of our Visibility Space Carving (Alg. 3.1) with $r = 250$, carveviewer



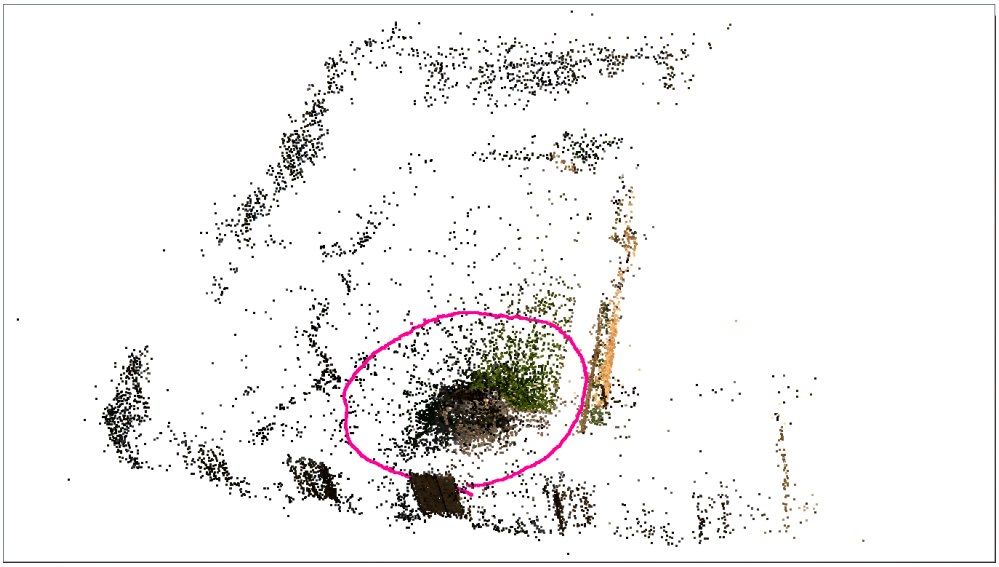
(c) Result of our Visibility-Occlusion Space Carving Veto (Alg. 3.3) with $r = 1000$, $Pr_{incr} = 0.001$

(d) Result of our Visibility-Occlusion Space Carving Veto (Alg. 3.3) with $r = 1000$, $Pr_{incr} = 0.001$, carveviewer

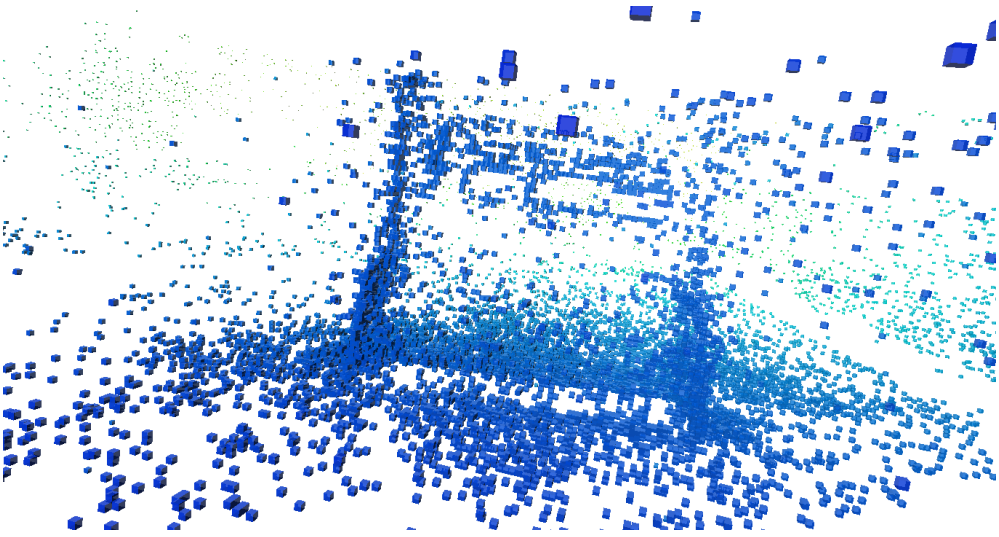
(e) Result of CMVS/PMVS [8]. Notice the similarity with the sparse point cloud as seen from this angle, shown in 5.6(c); lack of texture clearly influences both sparse and dense point cloud reconstructions.

Figure 5.7: Good result 1: sculpture1 dataset (cont.)

(a) Example frame

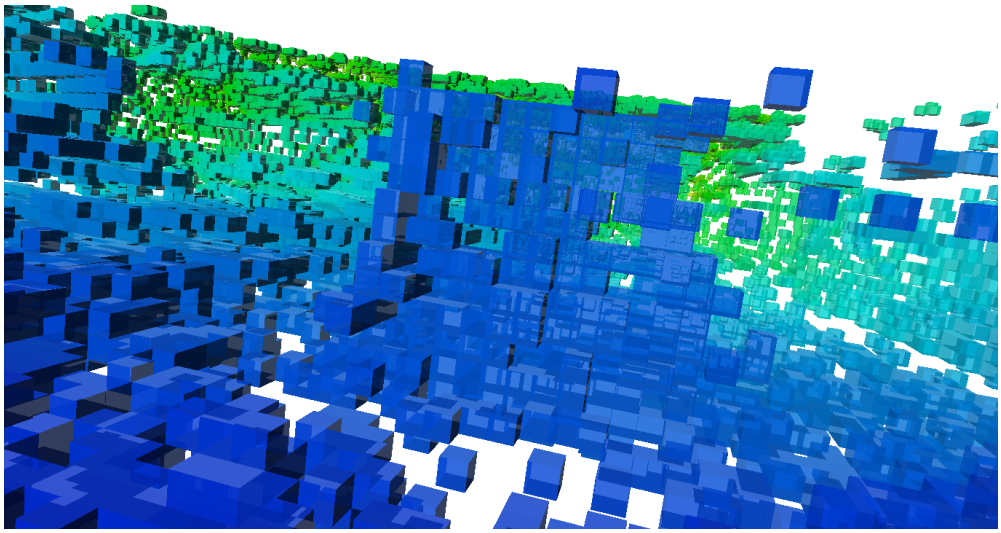


(b) Sparse point cloud and camera poses

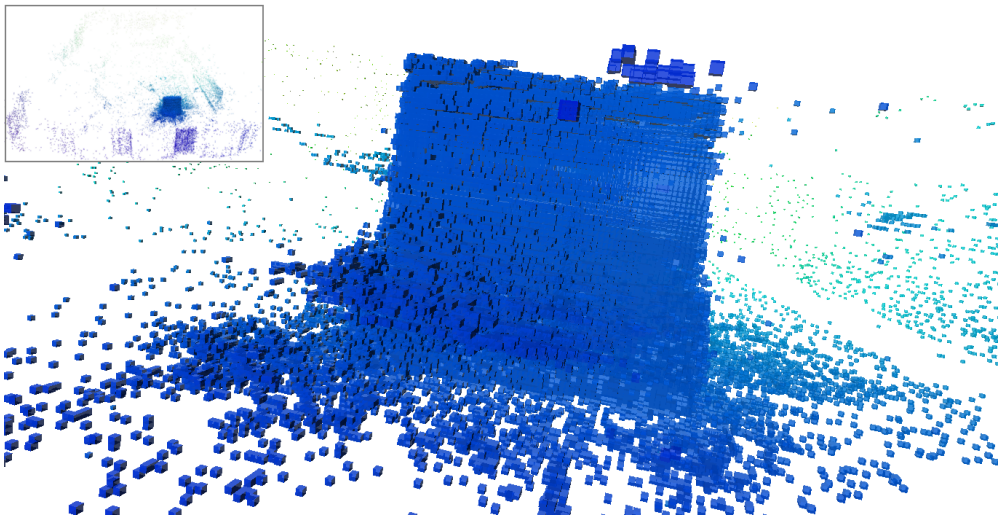


(c) Discretised sparse point cloud ($r = 2500$)

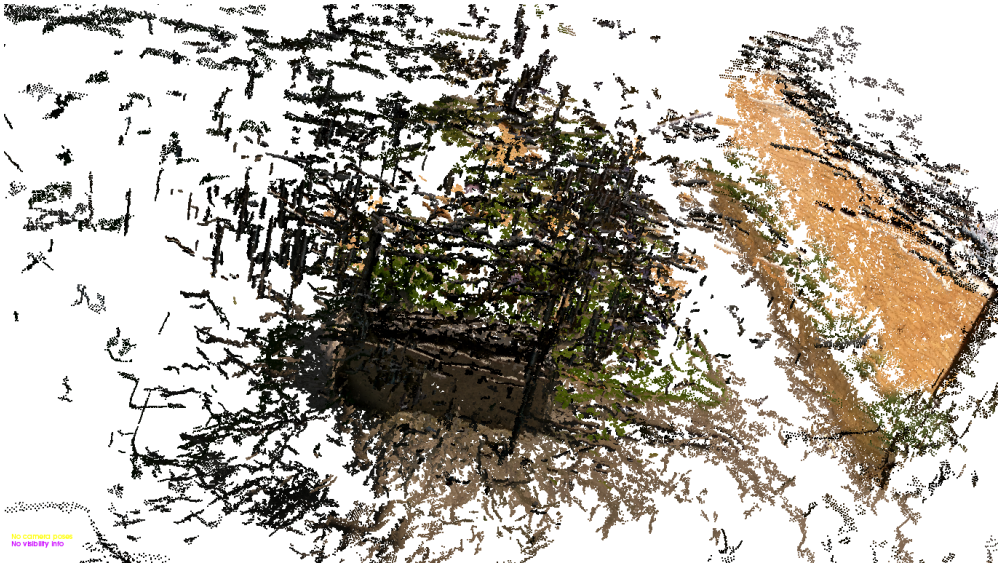
Figure 5.8: Good result 2: memorial dataset



(a) Result of our Visibility Space Carving (Alg. 3.1) with $r = 500$



(b) Result of our Visibility-Occlusion Space Carving Veto (Alg. 3.3) with $r = 2500, Pr_{incr} = 0.001$; Top-left: scene overview

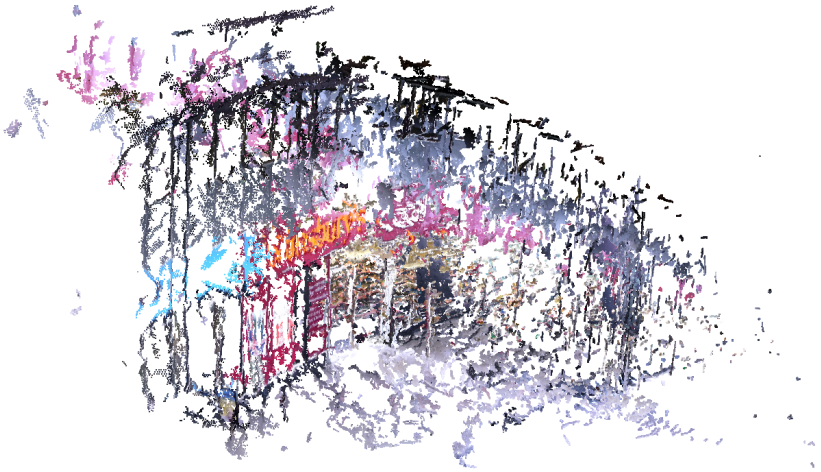


(c) Result of CMVS/PMVS [8]

Figure 5.9: Good result 2: memorial dataset (cont.)

(a) Example frame

(b) Result of our Visibility-Occlusion Space Carving Veto (Alg. 3.3) with $r = 250$, $Pr_{incr} = 0.001$



(c) Result of CMVS/PMVS [8]

(d) Result of Photofly / 123D Catch

Figure 5.10: Comparison between our result, CMVS/PMVS [8] result and Autodesk's closed-source Photofly / 123D Catch service for the sainsburys3 dataset (featuring a pillar of similar material as the memorial stone in Fig. 5.8)

5.3 Evaluation

We will now discuss the examples individually, which allows us to state general observations on the tested methods. Since presenting quantitative results is difficult without ground truth, our evaluation will mostly consists of a discussion on the visual results.

Result 1 (Fig. 5.2) consists of a sequence with poor image quality picturing two low-textured lampposts in front of a wall; nonetheless, some sparse points are reconstructed on the lampposts, probably on the edges between foreground and background. The lampposts are visible for both proposed methods and for CMVS/PMVS dense reconstruction. Visibility Space Carving causes uncarved space above and beneath the space between camera poses and the wall; although those voxels re-project inside the footage, it clearly does not have enough features on those places; however, it does show more solid structures in front of the wall. We typically found similar results for such scenes, especially when shot with a bad camera or bad lighting conditions. Usually this also means less features on feature-rich objects, decreasing the maximum resolution for which good results are obtained. The resolution settings are less important for the Visibility-Occlusion Space Carving algorithm, which marks voxels for which no information is available as unknown (thus not occupied). There, we obtain even more solid structures, but also some 'clutter' caused by unstable features behind the wall, also very typical for low-quality footage with less stable background structures (*e.g.,* trees). Note that camera movement is only parallel to the wall: the maximum viewing angle causes elongated lampposts to emerge, since not enough information is present to carve space in front and behind the posts. Another key observation therefore is that enough view points needs to be provided for the algorithm if high-quality results are expected. Small low-textured objects like this are usually reasonably recovered by CMVS/PMVS.

Result 2 pictures a similar scene containing two cars and one lamppost in front of al wall. It is more challenging due to the reflective (dark) windows and sparse texture on the metal of the cars and lamppost. Indeed, sparse reconstruction does recover some but not too many points on the front objects. Visibility Space Carving again does not carve space above the wall and we had to remove some for visualisation purposes (see caption). A semi-circle has been walked, providing more views useful for finer object reconstruction. Visibility-Occlusion carving provides more solid models on higher resolution, without the need to remove clutter. CMVS/PMVS on the other hand also recovers part of the car surfaces, although holes are visible, and back and top are not recovered since they are not directly observed in enough frames. Our algorithm does recover solid models, although it may over-estimate the shape given less view points.

The third result pictures an even more challenging object with nearly no texture and some weak reflections, and people were walking in the scene while filming. Overall, the background contains less texture than a brick wall. Compared to the distance, only a short walk has been made. For Visibility Space Carving therefore low resolution has been used. Still some unwanted occupied-labelled space is floating in the air, but a solid sculpture model emerges. Visibility-Occlusion Space Carving reconstructs quite a solid structure, although some space in front of the object is labelled occupied and the object is elongated at the back (not pictured) due to views from one side only. Also compare with the

`carveviewer` renderings, which should roughly correspond to what a depth map would look like (*i.e.,* single objects should have uniform colour); in the second carving result it shows a small overestimation of the shape and pictures a spike probably caused by unstable feature points at that hight. On contrary, dense reconstruction misses most of the object, and 'reconstructs' quite some dirt in the air. The dirt can be caused by reflections, while the poor reconstruction of the object is likely due to low-texture.

The fourth and last full example pictures a reflective memorial stone, with little texture on it. Footage was made by walking around the object completely, obtaining many view points. Notice the good structure from motion results, even for footage of more than 800 frames; however, also observe the few wrongly triangulated feature points floating in the air around the memorial (Fig. 5.8(c)). Only part of the background, which consists of a typical urban environment, contains dense textures. Visibility Space Carving reconstructs the memorial, but leaves some holes in the model. Visibility-Occlusion Space Carving results are quite neat and can be set to high resolutions. On contrary, CMVS/PMVS performs badly on the memorial. The stone is almost invisible apart from the edges and base underneath, and the reflections cause the reconstructed scene to be filled with wrongly triangulated, non-existing small structures. Notice that the reflections are likely to cause problems on all current methods aiming at direct reconstruction (that is, methods dependent on some photo-consistency, or clean silhouettes).

Finally, the comparison in Figure 5.10 shows that not only CMVS/PMVS but also Autodesk's commercial geometry reconstruction system fails to reconstruct the featured reflective pillar, whereby our method does show part of the structure. It is likely that Photofly / 123D Catch will also fail on similar datasets, although no other results were returned at the time of writing.

In general, low-quality footage gives reasonable results for finding small and low-textured objects, while good results can be obtained with better footage provided enough view points are captured. Therefore, the quality of the footage is somewhat relevant. Furthermore, background objects need to be present, preferably with a decent amount of texture. Unstable features, caused by moving or ambiguous objects, or bad footage, can also cause clutter to emerge. Space carving using visibility information only (Alg. 3.1) gives reasonable results for lower resolutions. Part of the scene may remain occupied when few features are present. Visibility-Occlusion space carving, veto version (Alg. 3.3) generally makes cleaner reconstructions at a higher resolution with less clutter. This is true even if objects have little texture or reflective surfaces, which cause difficulties for many existing algorithms. On the other hand, for the tested sequences CMVS/PMVS outputs high amounts of clutter for reflective objects, and surfaces of bigger low-textured objects contain holes. However, it is possible that dense stereo algorithms perform better than the proposed algorithm on general scenes containing mostly decently textured objects with Lambertian-like properties.

Chapter 6

Conclusion

The problem of geometry reconstruction is challenging and the diversity in scenes asks for solid solutions. The literature on geometry reconstruction is comprehensive and many approaches have been tried, with varying degrees of success. Most approaches are based on some form of photo-consistency, that is, they rely on *directly* observing all objects that are being reconstructed. However, this casts constraints on the surface properties of the objects. Surfaces often need to be either uniformly coloured (low-textured) and well differentiable from the background, or they need to have clear and rich textures and Lambertian-like properties. In practice, these constraints are not always met. Therefore, many algorithms fail on certain kinds of objects.

We proposed a new approach based on *indirectly* observing objects using visibility information. Based on the intuition that detection of features from certain view points and absence of detection in others can provide clues on where occluder objects may exist, we developed two algorithms; one based on visibility information alone, and one using both visibility and occlusion (*i.e.,* absence of visibility). An occupancy grid is altered iteratively based on this information. Feature estimation is obtained by using reliable Structure from Motion tools and no pixel pair-wise guesswork needs to be done.

The proposed method has been shown to provide reasonable results for scenes containing low-textured or reflective objects. For good results, enough view points need to be provided (*e.g.,* by walking around an object) and background objects containing a decent amount of features need to be present. Visibility information helps carving much of the empty space for which reliable features on background objects are extracted, but leaves space with less visibility rays partly labelled as occupied. Using both visibility and occlusion cues (and default world view 'unknown') improves results, even though visibility information exported by standard structure from motion tools is often very conservative, making the majority of the occlusion claims untrue. Furthermore, the veto version of the Visibility-Occlusion algorithm (Alg. 3.3) outperforms the recent dense reconstruction algorithm CMVS/PMVS on the tested reflective objects. Visibility and occlusion therefore appear promising cues for further research on geometry reconstruction.

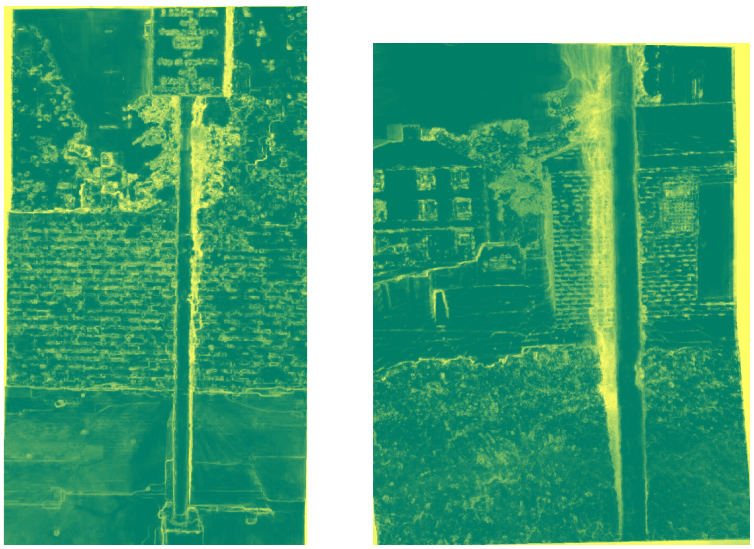
While sparse point clouds reconstructed by structure from motion are reliable, points do not have a clear size or shape and the exact shape of the affected space between cameras and features is therefore undefined. The taken approach of ray shooting and altering the tube of voxels hit by the ray makes

the algorithm dependent on the amount of strong feature points and chosen resolution. In other words, the amount of features determines the maximum resolution. Future work could focus on relaxing those dependencies - allowing arbitrary voxel grid resolutions - by using features with better defined sizes. One possibility is to create features based on the provided sparse point cloud. For example, nearby and planar feature points can be combined to form local patches with known size and normal. They could be texture-mapped, projected into, and matched over the sequence. Potentially, small patches could also be combined using a graph-like connection map, and the changing graph over the sequence could be used for carving. The local patches could be seen as acting like small local 'green screens' in the scene carving away part of space while the camera moves around.

Other features could be used as well. Edges could provide useful cues on object borders (*e.g.,* feature points disappearing after 'hitting' an edge). As a last possibility, one could consider using full resolution occlusion images, for example using occlusion posterior predictions as given by the system of Humayun et al. [13] (for example outputs on our sequences, see Fig. 6.1). Hereby full-resolution probabilistic *carve images* will be used instead of only a few sparse interest points. A solution needs to be found for the lack of depth, *i.e.,* where to stop carving.

Furthermore, texture mapping would make results more appealing and increases photo-realism, even for new (unseen) camera poses.

Lastly, we like to remind the reader of the observation that all current approaches - including the one proposed in this thesis - have both success stories and failure cases, but not necessarily the same failures. Therefore, the proposed approach may be used in conjunction with other approaches. For example, Visibility-Occlusion Space Carving can provide a fast and rough geometry reconstruction, which can be used as a prior for possible depths in patch-based or plane-sweeping multi-view dense reconstruction approaches.



(a) Frame from lampposts_on_wall1 (b) Frame from houses1

Figure 6.1: Example output of the system developed by Humayun et al. [13], showing cyan for low and yellow for high probabilities of pixels getting occluded in the next frame

Bibliography

[1] Y. Boykov and V. Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *Pattern Analysis and Machine Intelligence*, 26(9):1124–1137, 2004.

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

[3] N. Campbell, G. Vogiatzis, C. Hern, and R. Cipolla. Automatic 3D Object Segmentation in Multiple Views using Volumetric Graph-Cuts. *Proc. 18th British Machine Vision Conference*, 28(1):14–25, 2007.

[4] M. Chandraker. Moving in Stereo : Efficient Structure and Motion Using Lines. In *Computer Vision, 2009 IEEE*, pages 1741–1748, 2009.

[5] M. J. D. Lovi, N. Birkbeck, D. Cobzas. Incremental free-space carving for real-time 3d reconstruction. In *Proc. of 3DPVT 2010*, 2010.

[6] J. Frahm, P. Fite-Georgel, and D. Gallup. Building Rome on a cloudless day. In *Computer VisionECCV 2010*, volume 6314/2010, pages 368–381, 2010.

[7] Y. Furukawa, B. Curless, S. Seitz, and R. Szeliski. Manhattan-world stereo. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1422–1429. Ieee, June 2009.

[8] Y. Furukawa and J. Ponce. Accurate, dense, and robust multiview stereopsis. *IEEE transactions on pattern analysis and machine intelligence*, 32(8):1362–76, Aug. 2010.

[9] L. Guan, M. Pollefeys, and U. N. C. C. Hill. 3D Occlusion Inference from Silhouette Cues. In *Computer Vision and Pattern Recognition, 2007. CVPR '07. IEEE Conference*, pages 1–8, 2007.

[10] R. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, 2003.

[11] R. Hernandez, C. and Vogiatzis, G. and Cipolla. Probabilistic visibility for multi-view stereo. In *IEEE Conference on Computer Vision and Pattern Recognition (2007)*, pages 1–8, 2007.

[12] D. Huang, J. and Lee, A.B. and Mumford. Statistics of range images. In *Proc. IEEE Conference on Computer Vision and Pattern Recognition. CVPR 2000*, volume 1, pages 324–331. IEEE Comput. Soc, 2000.

[13] A. Humayun, O. Mac Aodha, and G. J. Brostow. Learning to find occlusion regions. In *Cvpr 2011*, pages 2161–2168. Ieee, June 2011.

[14] S. Izadi, D. Kim, O. Hilliges, D. Molyneaux, R. Newcombe, P. Kohli, J. Shotton, S. Hodges, D. Freeman, A. Davison, and A. Fitzgibbon. KinectFusion : Real-time 3D Reconstruction and Interaction Using a Moving Depth Camera. In *Proc. of the 24th annual ACM symposium on User interface software and technology (UIST '11)*, pages 559–568, 2011.

[15] N. Jojic and B. Frey. Learning flexible sprites in video layers. In *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, volume 1, pages I–199–I–206. IEEE Comput. Soc, 2001.

[16] W. B. Kai M. Wurm , Armin Hornung , Maren Bennewitz , Cyrill Stachniss. OctoMap: A probabilistic, flexible, and compact 3D map representation for robotic systems. In *Proc. of the ICRA 2010 workshop*, 2010.

[17] S. M. Kutulakos, Kiriakos N. and Seitz. A Theory of Shape by Space Carving. *International Journal of Computer Vision*, 38(3):199–218, 2000.

[18] M. Li Guan and Franco, J.-S. and Pollefeys. Multi-Object Shape Estimation and Tracking from Silhouette Cues. In *IEEE Conference on Computer Vision and Pattern Recognition (2008)*, pages 1–8, 2008.

[19] D. G. Lowe. Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision*, 60(2):91–110, Nov. 2004.

[20] W. Matusik, C. Buehler, and R. Raskar. Image-based visual hulls. In *Proceedings of the 27th . . .*, pages 369–374, 2000.

[21] P. McIlroy, R. Cipolla, and E. Rosten. High-level scene structure using visibility and occlusion. In *Proc. of the British Machine Vision Conference*, pages 1–11, 2011.

[22] P. Merrell, A. Akbarzadeh, L. Wang, P. Mordohai, J.-m. Frahm, R. Yang, and D. Nist. Real-Time Visibility-Based Fusion of Depth Maps. In *IEEE Conference on Computer Vision (2007)*, pages 1–8, 2007.

[23] T. Pan, Q. and Reitmayr, G. and Drummond. ProFORMA: Probabilistic feature-based on-line rapid model acquisition. In *Proc. 20th British Machine Vision Conference (BMVC)*, pages 1–11, 2009.

[24] M. Pollefeys, D. Nistér, J.-M. Frahm, A. Akbarzadeh, P. Mordohai, B. Clipp, C. Engels, D. Gallup, S.-J. Kim, P. Merrell, C. Salmi, S. Sinha, B. Talton, L. Wang, Q. Yang, H. Stewénius, R. Yang, G. Welch, and H. Towles. Detailed Real-Time Urban 3D Reconstruction from Video. *International Journal of Computer Vision*, 78(2):143–167, Oct. 2008.

[25] S. J. D. Prince. *Computer vision : models, learning and inference*. Cambridge University Press, 2012.

[26] R. B. Rusu and S. Cousins. 3D is here: Point Cloud Library (PCL). *Robotics and Automation (ICRA), 2011 IEEE International Conference*, pages 1–4, May 2011.

[27] R. Seitz, S.M. and Curless, B. and Diebel, J. and Scharstein, D. and Szeliski. A Comparison and Evaluation of Multi-View Stereo Reconstruction Algorithms. *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, 1:519 – 528, 2006.

[28] P. Smith, T. Drummond, and R. Cipolla. Layered motion segmentation and depth ordering by tracking edges. *IEEE transactions on pattern analysis and machine intelligence*, 26(4):479–94, Apr. 2004.

[29] N. Snavely, S. Seitz, and R. Szeliski. Photo tourism: exploring photo collections in 3D. *ACM Transactions on Graphics (TOG)*, 25(3):835 – 846, 2006.

[30] E. Wang, J.Y.A. and Adelson. Representing Moving Images with Layers. *Image Processing, IEEE Transactions on*, 3(5):625–638, 1994.

[31] C. Wu, S. Agarwal, B. Curless, and S. M. Seitz. Multicore bundle adjustment. In *Cvpr 2011*, pages 3057–3064. Ieee, June 2011.

[32] C. Zach, A. Irschara, and H. Bischof. What can missing correspondences tell us about 3D structure and motion? In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, volume 1, pages 1–8, 2008.

Appendix A

Installation manual

Installation instructions are provided for Linux (Ubuntu/Debian) and Windows. All paths are relative to the corresponding root of unpacked archive files. The Structure from Motion tools are available online, or provided as Windows/Linux binaries in sfm/. Libraries are also available online, or provided in the mentioned versions as source code in lib/.

1. Install OpenCV (I used 2.4.1) - http://opencv.willowgarage.com

   Linux Ubuntu, older version (2.3):

   ```
   $ sudo add-apt-repository ppa:gijzelaar/cuda
   $ sudo add-apt-repository ppa:gijzelaar/opencv2.3
   $ sudo apt-get update
   $ sudo apt-get install libcv-dev
   ```

   Linux Ubuntu, newest version from source:

   ```
   $ sudo apt-get install cmake pkg-config libavformat-dev libswscale-dev \
     libavcodec-dev libavfilter-dev libpython2.7 python-dev python2.7-dev \
     python-numpy libgstreamer0.10-0-dbg libgstreamer0.10-0 \
     libgstreamer0.10-dev libgstreamer-plugins-base0.10-dev libdc1394-22-dev \
     libdc1394-22 libdc1394-utils libavformat-dev libxine-dev \
     libgstreamer0.10-dev libgstreamer-plugins-base0.10-dev libv4l-dev \
     libtbb-dev libqt4-dev libgtk2.0-dev openni-dev sphinx-common
   $ mkdir release && cd release
   $ cmake -D CMAKE_BUILD_TYPE=RELEASE -D CMAKE_INSTALL_PREFIX=/usr/local -D \
     BUILD_PYTHON_SUPPORT=ON -D WITH_QT=ON -D WITH_XINE=ON -D WITH_V4L=ON -D \
     WITH_OPENGL=ON -D WITH_OPENNI=ON -D WITH_TBB=ON -D BUILD_DOCUMENTATION=ON \
     -D BUILD_EXAMPLES=ON release ..
   $ make && sudo make install
   ```

Windows:

Run binary executable from http://sourceforge.net/projects/opencvlibrary/files/opencv-win/2.4.0/

2. Install Point Cloud Library (I used 1.6) - http://pointclouds.org

   Linux Ubuntu:

   ```
   $ sudo add-apt-repository ppa:v-launchpad-jochen-sprickerhof-de/pcl
   $ sudo apt-get update;
   $ sudo apt-get install libpcl-all
   ```

   Others (including Windows):

   Run binary executable or compile source from http://pointclouds.org/downloads/

3. Install OctoMap (I used 1.4.2) - http://octomap.sourceforge.net

   Linux Ubuntu:

   ```
   $ sudo apt-get install cmake doxygen libqt4-dev libqt4-opengl-dev
   $ cd build && cmake .. && make && sudo make install
   ```

   Windows (Cygwin):

   ```
   $ cd build && cmake .. && make && make install
   ```

4. Install one or more of the following Structure from Motion estimators. VisualSFM seems to give the best results, so I recommend going for that one. Bundler and Voodoo output files are partly supported, but currently not everything will work.

   Website: http://www.cs.washington.edu/homes/ccwu/vsfm/

   - VisualSFM (I used 0.5.17)

     Linux (tested on Ubuntu):
     Need to install SiftGPU, which needs some hacks in the source code (!) Need to install PBA, which needs some hacks in the source code as well (include sys/types.h, stdio.h, stdlib.h) For GPU acceleration, need to install CUDA as well. Otherwise, download Lowe's sift binary (extract binary only to vsfm/bin/): http://www.cs.ubc.ca/ lowe/keypoints/

     Windows:
     Executable seems to work fine after installing the CUDA toolbox. You may get errors concerning some CUDA dll file; search and download it from The Internet and put it in the same directory as the VisualSfM executable.

     Optionally, one could additionally install CMVS/PMVS for dense reconstruction (for comparison only; the dense output is not used for carving).

- Voodoo (I used 1.2.0 beta)

  Download binary from: http://www.digilab.uni-hannover.de/download.html

  Note that Voodoo does not export visibility information for feature points. Therefore, important parts of the program (i.e. carving) will not work using Voodoo's output. The viewers will work fine.

- Bundler (I used 0.4)

  Windows:

  Download and install binary from: http://phototour.cs.washington.edu/bundler/

  Linux (Ubuntu/Debian):

```
$ sudo apt-get install gfortran libgfortran3 liblapack-dev libblas-dev \
                       libblacs-mpi-dev libminpack1 libf2c2-dev libann-dev
Compile from source (http://phototour.cs.washington.edu/bundler/)
Need to make some patches as suggested by gcc.
$ sudo cp lib/libANN_char.so /usr/lib/
```

Appendix B

User manual

1. Structure from Motion

   VisualSfM:

   ```
   1. Load images:               File -> Open+ Multiple Images
   2. Calc+Match SIFT features:  SfM  -> Pairwise Matching -> Compute Missing Match
   3. Reconstruct (Bundle Adj):  SfM  -> Reconstruct 3D
   4. Save sparse point cloud:   SfM  -> Save NV Match (.nvm)
   5. Dense (optional):          SfM  -> Run CMVS/PMVS
   ```

   Voodoo:

   ```
   1. Load images:               File -> Open -> Sequence
   2. Calc+Track features:       Track (bottom panel)
   3. Save:                      File -> Save -> Textfile (.txt)
   ```

   Bundler:

   ```
   1. Run provided script from within direct of images (Cygwin)
   2. Result is saved in bundle/bundle.out
   ```

2. Visualisation of SfM results

   ```
   ./sfmviewer <sfm> [<images>]
   where sfm is the path to an nvm, ply, out or txt file,
   and images is the path to the corresponding directory with images.
   ```

   Select points or cameras with Shift + Click.

   Enable/disable line drawing with 'd'.

   More shortcuts listed with 'h' (PCL) and '?' (custom).

3. Space Carving

```
./sfmcarver <sfm> <imgs> <save> [<method> [<resolution> [<param1>]]
  sfm:    path to an nvm, ply, out or txt file
  imgs:   path to the directory containing the images
  save:   filename for saving the octree (.ot, or .bt for binary)
  method: carve method (0-3); default: 2;
          0=discretised, 1=vis, 2=vis+occ veto, 3=vis+occ+extend vis lists
  resol.: voxelgrid sizes (determines smallest octree node size); default: 250
  param1: first parameter of given method; default: 0.1
```

4. Visualisation of Space Carve results

```
octovis <carve>
  where carve is an octree file (.ot or .bt)
```

or:

```
./carveviewer <sfm> <imgs> <carve>
  where sfm is the path to an nvm, ply, out or txt file
  imgs is the path to the directory containing the images
  and carve is an octree file (.ot or .bt)
```

5. Optional: regularisation

```
./octreegraphcut <octree in> <octree out> [<gamma> [<unknown>]]
  octree in/out: filename of octree file (.ot, or .bt for binary)
  gamma:         weight of voxel prob difference for pairwise cost; default: 1
  unknown:       occupancy probability of unknown (un-initialised) voxels;
                 default: 0.2
```

6. Optional: feature visualisation

```
./featureviewer <path> [<pointDetector> [<pointDescriptor> [<edgeDetector>]]]
  where path is the path to an AVI file, directory containing
  a sequence of images, or webcam device number;
  pointDetector: {SIFT, SURF, ORB, FAST, STAR, MSER, GFTT, HARRIS,
                  Dense, SimpleBlob};
  pointDescriptor: {SIFT, SURF, ORB, BRIEF};
  edgeDetector: {CANNY, HARRIS}
```

Appendix C

Datasets

Listed are the datasets for which VisualSfM results are available[1]. For most datasets, space carve results are provided too.

Dataset	# images	Camera	Resolution	Comments
bikestand	530	Sanyo HD	1920x1080	Walk around partly; pavement at background
car_and_wall1	515	Nexus One	1280x720	Walk around, lots of features on wall, shiny cars + lamppost
car_and_wall2	394	Nexus One	1280x720	Walk around, lots of features on wall, shiny car (bad SfM cloud)
chess	387	Nexus One	1280x720	Human-size chess board; walk around partly; low height
houses1	21	Nexus One	1920x2592	Small set of hi-res photos
lampposts_on_wall1	324	Nexus One	720x1280	Sliding parallel to wall; two lampposts in front
memorial	896	Sanyo HD	1920x1080	Walk around completely, lots of features in background, shiny and black
pole3	502	Nexus One	1080x1920	Walk around partly; lots of unique features at background
sainsburys1	545	Sanyo HD	1080x1920	Sliding parallel; sunny day; shiny poles
sainsburys2	626	Sanyo HD	1080x1920	Sliding parallel; sunny day; dark poles
sainsburys3	653	Sanyo HD	1080x1920	Walk around partly while moving up and down; again shiny and black
sciencepark2	410	Nexus One	1280x720	Walk around partly; lots of poles underneath building
sculpture1	809	Sanyo HD	1920x1080	Walk around partly; pavement at background

[1] Datasets and results available at: http://code.google.com/p/martijn-msc-thesis

Appendix D

Code listing

D.1 Code of Applications

../code/apps/sfmcarver.cpp

```
#include <iostream>

#include "../objects/occupancy_grid.hpp"

using namespace std;


void usage(char* name)
{
  cout << "Usage:" << endl;
  cout << " " << name << " <sfm> <imgs> <save> [<method> [<resolution> [<
      param1>]]" << endl;
  cout << " sfm:    path to an nvm, ply, out or txt file" << endl;
  cout << " imgs:   path to the directory containing the images" << endl;
  cout << " save:   filename for saving the octree (.ot, or .bt for binary)"
      << endl;
  cout << " method: carve method (0-3); default: 2" << endl;
  cout << " resol.: voxelgrid sizes (determines smallest octree node size);
      default: 250" << endl;
  cout << " param1: first parameter of given method; default: 0.1" << endl;
  exit(1);
}


/* main */
int main (int argc, char** argv)
{

  if (argc < 4)
    usage(argv[0]);

  // settings
  int method = 2;
  int resolution = 250;
  double param1 = 0.01;
  bool extent = false;
  bool graphcut = false;

  // command line arguments
  if (argc >= 5)
    method = atoi(argv[4]);
  if (argc >= 6)
    resolution = atoi(argv[5]);
  if (argc >= 7)
    param1 = atof(argv[6]);

  // carve
  OccupancyGrid occgrid(argv[1], argv[2], resolution);
  if (extent)
    occgrid.extentVisibilityLists(0.2);
  occgrid.carve(method, param1);
  if (graphcut)
    occgrid.graphcut(0.5);

  // save; use extension (.bt, .ot) to determine file format
  bool binary = (string(argv[3]).find_last_of(".bt") > -1);
  occgrid.save(argv[3], binary);
  cout << "Result saved as " << argv[3];
  if (binary)
    cout << " (binary)";
  cout << endl;

  return 0;
}
```

../code/apps/sfmviewer.cpp

```
#include <iostream>
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <pcl/common/common.h>
#include <pcl/point_cloud.h>
#include <pcl/point_types.h>
#include <pcl/visualization/pcl_visualizer.h>

#include "../io/sfm_reader.hpp"
#include "../io/sequence_capture.hpp"

using namespace std;
using namespace pcl;
using namespace cv;



void usage(char* name)
{
  cout << "Usage:" << endl;
  cout << " " << name << " <sfm> [<images>]" << endl;
  cout << " where sfm is the path to an nvm, ply, out or txt file," << endl;
  cout << " and images is the path to the corresponding directory with images.
      " << endl;
  exit(1);
}

typedef struct {
  SfMReader* sfm;
  visualization::PCLVisualizer* viewer;
  SequenceCapture* sc;
  bool updated;
  int camID;
  int pointID;
  bool lineDraw;
  bool cameraWindow;
} callbackObject;



/* drawing functions */

void drawLines(visualization::PCLVisualizer& viewer, SfMReader* sfm, int
    maxLines = 250)
{
  // faster drawing: http://stackoverflow.com/questions/2140796/draw-a-
      multiple-lines-set-with-vtk

  viewer.removeAllShapes();
  stringstream name("");
  if (sfm->line_ends.size() > maxLines) {
    cout << "Too many lines (" << sfm->line_ends.size() << ") to draw! ";
    cout << "Only drawing first " << maxLines << " lines." << endl;
  }
  for (int i=0; i<min((int)sfm->line_ends.size(), maxLines); i++) {
    name.str("line");
    name << i;
    viewer.addLine(*sfm->line_start, *sfm->line_ends.at(i), 0,1,0, name.str())
        ;
  }
}

void showCameraView(callbackObject* options, string window_name)
{
  if (options->cameraWindow) {
    // camera window enabled
    if (options->camID >= 0) {
      // camera selected
      Mat frame;
```

```
64
    // get camera image
    if (options->sfm->ext == "nvm") {
      /* VisualSfM (nvm) does not save the cameras in the order
       * in which the images appear in a sorted directory listing.
69     * Therefore we need to do some tricks to get the right
       * image. I am very sorry for the messy code this requires
       * here and in the used classes.
       */
      string img_filename = options->sfm->image_filenames.at(options->camID)
          ;
74    options->sc->setPosition(img_filename);
    } else {
      // all other formats are fine
      options->sc->setPosition(options->camID);
    }
79  options->sc->read(frame);

    // draw keypoints
    Size s = frame.size();
    double mx, my, r, focal;
84  for (vector<visibility*>::iterator it = options->sfm->
        curr_visible_keypoints.begin();
         it != options->sfm->curr_visible_keypoints.end(); it++) {
      mx = (*it)->x;
      my = (*it)->y;
      /* Optionally, one could undistort the the measurements
89     * (instead of redistorting the reprojected points) with:
       * focal = options->sfm->cameras.at(options->camID).focal;
       * r = (mx*mx + my*my) / (focal*focal) * options->sfm->cameras.at(
           options->camID).radial[0];
       * mx = (1 + r) * (*it)->x;
       * my = (1 + r) * (*it)->y;
94     */
      mx += s.width/2.0;
      my += s.height/2.0;
      circle(frame, Point(mx, my), 2, Scalar(255,100,0), 2);
    }
99

    // draw reprojected keypoints
    PointXYZRGB projected;
    int visible_in = 0, visible_out = 0;
    int invisible_in = 0, invisible_out = 0;
    double x, y, gamma;
104 for (int i=0; i<options->sfm->points.size(); i++) {
      options->sfm->reproject(&options->sfm->points.at(i),
                              &options->sfm->cameras.at(options->camID),
                              &projected);
      // redo distortion for reprojected keypoints
109   gamma = options->sfm->cameras.at(options->camID).focal;
      options->sfm->distortPointR1(&projected, &options->sfm->cameras.at(
          options->camID));
      x = projected.x + s.width/2.0;
      y = projected.y + s.height/2.0;
114

      if (x > 0 && x < s.width && y > 0 && y < s.height) {
        if (options->sfm->points_curr_visible.at(i)) {
          circle(frame, Point(x, y), 3, Scalar(0,255,0), 2);
          visible_in++;
        } else {
119       circle(frame, Point(x, y), 3, Scalar(0,0,255), 1);
          invisible_in++;
        }
      } else {
        if (options->sfm->points_curr_visible.at(i)) {
124       visible_out++;
        } else {
          invisible_out++;
```

```
            }
          }
        }
        cout << "Showing " << visible_in << "/" << (visible_in + visible_out);
        cout << " (" << (100.0*visible_in/(visible_in + visible_out)) << "%)";
        cout << " visible and " << invisible_in << "/" << (invisible_in +
            invisible_out);
        cout << " (" << (100.0*invisible_in/(invisible_in + invisible_out)) << "
            %)";
        cout << " invisible points." << endl;

        // show result
        imshow(window_name, frame);
      }
    }
  }


/* callback functions */

void pp_callback(const visualization::PointPickingEvent& event, void*
    options_ptr)
{
  callbackObject* options = (callbackObject*) options_ptr;

  int id = event.getPointIndex();
  if (id == -1)
    return;

  float x, y, z;
  event.getPoint (x, y, z);

  if (options->sfm->poses.size() > id &&
      options->sfm->poses.at(id).x == x &&
      options->sfm->poses.at(id).y == y &&
      options->sfm->poses.at(id).z == z) {
    // update camera view
    cout << ">> Clicked on camera at (" << x << ", " << y << ", " << z << ")!"
         << endl;
    options->sfm->selectPointsForCamera(id);
    options->camID = id;
    options->pointID = -1;
    showCameraView(options, "camera view");
  } else {
    cout << ">> Clicked on point at (" << x << ", " << y << ", " << z << ")!"
        << endl;
    options->sfm->selectCamerasForPoint(id);
    options->camID = -1;
    options->pointID = id;
  }

  options->updated = true;
}

void kb_callback(const visualization::KeyboardEvent& event, void* options_ptr)
{
  callbackObject* options = (callbackObject*) options_ptr;

  if (!event.keyDown())
    return;

  if (event.getKeyCode() == '?') {
    cout << "Keyboard shortcuts: (press h for PCL shortcuts)" << endl;
    cout << " b/w:   set background colour to black/white" << endl;
    cout << " d:     line draw on/off" << endl;
    cout << " n/p:   select next/prev camera or point" << endl;
    cout << " f/l:   select first/last camera or point" << endl;
```

```
    cout << " i:     camera window enable/disable (shows image for current
        camera)" << endl;
  } else if (event.getKeyCode() == 'q') {
    exit(0);
  /** visuals **/
  } else if (event.getKeyCode() == 'b') {
    options->viewer->setBackgroundColor(0,0,0);
  } else if (event.getKeyCode() == 'w') {
    options->viewer->setBackgroundColor(1,1,1);
  } else if (event.getKeyCode() == 'd') {
    options->lineDraw = !options->lineDraw;
    options->updated = true;
  } else if (event.getKeyCode() == 'n') { // select next camera/point
    if (options->camID >= 0) {
      // select next camera
      options->camID = min(options->camID + 1,
                           (int)options->sfm->poses.size() - 1);
      options->sfm->selectPointsForCamera(options->camID);
      showCameraView(options, "camera view");
    } else if (options->pointID >= 0) {
      // select next point
      options->pointID = min(options->pointID + 1,
                             (int)options->sfm->points.size() - 1);
      options->sfm->selectCamerasForPoint(options->pointID);
    }
    options->updated = true;
  } else if (event.getKeyCode() == 'p') { // select prev camera/point
    if (options->camID >= 0) {
      // select prev camera
      options->camID = max(options->camID - 1, 0);
      options->sfm->selectPointsForCamera(options->camID);
      showCameraView(options, "camera view");
    } else if (options->pointID >= 0) {
      // select prev point
      options->pointID = max(options->pointID - 1, 0);
      options->sfm->selectCamerasForPoint(options->pointID);
    }
    options->updated = true;
  } else if (event.getKeyCode() == 'f') { // select first camera/point (begin)
    if (options->camID >= 0) {
      // select first camera
      options->camID = 0;
      options->sfm->selectPointsForCamera(options->camID);
      showCameraView(options, "camera view");
    } else if (options->pointID >= 0) {
      // select first point
      options->pointID = 0;
      options->sfm->selectCamerasForPoint(options->pointID);
    }
    options->updated = true;
  } else if (event.getKeyCode() == 'l') { // select last camera/point (end)
    if (options->camID >= 0) {
      // select last camera
      options->camID = options->sfm->poses.size() - 1;
      options->sfm->selectPointsForCamera(options->camID);
      showCameraView(options, "camera view");
    } else if (options->pointID >= 0) {
      // select last point
      options->pointID = options->sfm->points.size() - 1;
      options->sfm->selectCamerasForPoint(options->pointID);
    }
    options->updated = true;
  /** extra windows **/
  } else if (event.getKeyCode() == 'i') { // camera window
    options->cameraWindow = !options->cameraWindow;
    if (options->sc == NULL) {
      cout << "[!!] Warning: no image path specified" << endl;
      options->cameraWindow = false;
```

```
    } else if (options->sfm->poses.size() == 0) {
      cout << "[!!] Warning: no camera poses known (not in file)" << endl;
    } else if (options->cameraWindow) {
      namedWindow("camera view", CV_WINDOW_NORMAL);
      showCameraView(options, "camera view");
    } else {
      destroyWindow("camera view");
    }
  }


}



/* main */
int main (int argc, char** argv)
{

  if (argc==1)
    usage(argv[0]);

  SfMReader* sfm;
  if (argc==2)
    sfm = new SfMReader(argv[1]);
  else
    sfm = new SfMReader(argv[1], argv[2]);
  visualization::PCLVisualizer viewer("Cloud viewer");

  // set options
  callbackObject options;
  options.sfm = sfm;
  options.viewer = &viewer;
  options.updated = false;
  options.camID = -1;
  options.pointID = -1;
  options.lineDraw = false;
  options.cameraWindow = false;
  options.sc = (argc>=3) ? new SequenceCapture(argv[2]) : NULL;

  PointCloud<PointXYZRGB>::Ptr points(&sfm->points);
  PointCloud<PointXYZRGB>::Ptr poses(&sfm->poses);

  viewer.registerPointPickingCallback(pp_callback, &options);
  viewer.registerKeyboardCallback(kb_callback, &options);

  viewer.addPointCloud(points, "points");
  viewer.addPointCloud(poses, "poses");
  viewer.setBackgroundColor(0,0,0);
  viewer.setPointCloudRenderingProperties(visualization::
      PCL_VISUALIZER_POINT_SIZE, 2, "points");
  viewer.setPointCloudRenderingProperties(visualization::
      PCL_VISUALIZER_POINT_SIZE, 3, "poses");
  //viewer.addCoordinateSystem (1.0);  // draw origin
  viewer.initCameraParameters ();

  int msg_nr = 1;
  if (sfm->visible.size() == 0)
    viewer.addText("No visibility info", 10, 10 + (10 * msg_nr++), 0.8,0,1);
  if (sfm->poses.size() == 0)
    viewer.addText("No camera poses", 10, 10 + (10 * msg_nr++), 1,1,0);


  int opencvKey;
  while (!viewer.wasStopped()) {
    viewer.spinOnce(100);   // needed for pcl visualizer to work
    opencvKey = waitKey(1); // needed for highgui windows to work
```

```
    if (opencvKey != -1)
      // pass opencv keyboard events to pcl keyboard handler
      // (works for user defined shortcuts only)
      kb_callback(visualization::KeyboardEvent(true, "", opencvKey,
                                               false, false, false), &options)
                                                   ;


    if (options.updated) {
      viewer.updatePointCloud(points, "points");
      viewer.updatePointCloud(poses, "poses");
      if (options.lineDraw)
        drawLines(viewer, sfm, 500);
      else
        viewer.removeAllShapes();
      options.updated = false;
    }
  }

  viewer.close();

  // more drawing: use viewer.getRenderWindow() and use VTK

  return 0;
}
```

../code/apps/octreegraphcut.cpp

```
#include <iostream>

#include "../objects/occupancy_grid.hpp"

using namespace std;


void usage(char* name)
{
  cout << "Usage:" << endl;
  cout << " " << name << " <octree in> <octree out> [<gamma> [<unknown>]]" <<
      endl;
  cout << " octree in/out: filename of octree file (.ot, or .bt for binary)"
      << endl;
  cout << " gamma:         weight of voxel prob difference for pairwise cost;
      default: 1" << endl;
  cout << " unknown:       occupancy probability of unknown (un-initialised)
      voxels; default: 0.2" << endl;
  exit(1);
}


/* main */
int main (int argc, char** argv)
{

  if (argc < 3)
    usage(argv[0]);

  // settings
  double gamma =   1; // penalty for discontinuity in space
                      // (weight for probability difference pairwise cost)
                      // > 1 does not seem to make a difference
  double unknown = 0.2; // prior; below 0.5 makes sense

  // command line arguments
  if (argc >= 4)
    gamma = atof(argv[3]);
  if (argc >= 5)
    unknown = atof(argv[4]);

  // open file
```

```
  OccupancyGrid occgrid;
  occgrid.load(argv[1]);
  occgrid.graphcut(gamma, unknown);

  // save; use extension (.bt, .ot) to determine file format
  bool binary = (string(argv[2]).find_last_of(".bt") > -1);
  occgrid.save(argv[2], binary);
  cout << "Result saved as " << argv[2];
  if (binary)
    cout << " (binary)";
  cout << endl;

  return 0;
}
```

../code/apps/carveviewer.cpp

```
#include <iostream>
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>

#include "../objects/occupancy_grid.hpp"

using namespace std;


void usage(char* name)
{
  cout << "Usage:" << endl;
  cout << " " << name << " <sfm> <imgs> <carve>" << endl;
  cout << " where sfm is the path to an nvm, ply, out or txt file" << endl;
  cout << " imgs is the path to the directory containing the images" << endl;
  cout << " and carve is an octree file (.ot or .bt)" << endl;
  exit(1);
}

/* main */
int main (int argc, char** argv)
{

  if (argc<4)
    usage(argv[0]);

  // make octree object
  OccupancyGrid occgrid(argv[1], argv[2], 1);
  occgrid.load(argv[3]);

  /* This is requiring too much memory for larger datasets:
  // calculate annotated images
  vector<Mat> output;
  cout << "Annotating images.." << endl;
  occgrid.visualisePath(&output, "circle", 0.5);
  */

  // visualise
  cout << "Visualise.." << endl;
  int k;
  int i=0, last_i=-1;
  int count = occgrid.sfm->poses.size();
  bool paused = true;
  VideoWriter* recorder = NULL;
  Mat output;
  namedWindow("visualise", CV_WINDOW_NORMAL);
  while (1) {
    if (occgrid.visualisePose(output, i, "circle", 0.2)) {

      imshow("visualise", output);
```

```
    // if recording, save frame
    if (last_i != i && recorder)
      *recorder << output;
    last_i = i;

  } else {
    cout << "Image not used for carving: " << i << endl;
  }

  k = waitKey(100);
  switch (k) {
    case '?':
      cout << "Keyboard shortcuts:" << endl;
      cout << " q/ESC:    quit" << endl;
      cout << " --- navigation ---" << endl;
      cout << " r:        refresh" << endl;
      cout << " space:    pause/resume" << endl;
      cout << " ->/n:     next frame" << endl;
      cout << " <-/p:     prev frame" << endl;
      cout << " PGUP:     +25 frames" << endl;
      cout << " PGDN:     -25 frames" << endl;
      cout << " f/b/HOME: first frame" << endl;
      cout << " l/e/END:  last frame" << endl;
      cout << " g + tty#: goto frame #" << endl;
      cout << " --- various ---" << endl;
      cout << " s         start/stop saving visible frames" << endl;
      break;
    case 'q':
    case 27:  // ESC
      destroyWindow("visualise");
      exit(0);
    case 'r':
      last_i = -1; // refresh
      break;
    case 's':
      if (recorder) {
        // stop recording
        cout << "Stopped recording at frame " << i << endl;
        delete recorder;
        recorder = NULL;
      } else {
        // start recording
        cout << "Starting recording from frame " << i << endl;
        time_t tim; time(&tim); string ctim = (string)ctime(&tim);
        string record_filename = "output " + ctim.substr(0,ctim.size()-1) +
            ".avi";
        recorder = new VideoWriter(record_filename, CV_FOURCC('D','I','V','X
            '),
                                   10, output.size(), true);
        if (paused)
          last_i = -1; // save next frame if paused
      }
      break;
    case ' ':
    case 13:  // return
      paused = !paused;
      break;
    case 'n':
    case 65363: // right arrow
      i++;
      if (i==count)
        i--;
      break;
    case 'p':
    case 65361: // left arrow (same for all keyboards?)
      i--;
      if (i<0)
        i=0;
```

```
      break;
    case 65365: // page up
      i = max(0, i-25);
      break;
    case 65366: // page down
      i = min(count-1, i+25);
      break;
    case 'f':
    case 'b':
    case 65360: // home
      i = 0;
      break;
    case 'l':
    case 'e':
    case 65367: // end
      i = count-1;
      break;
    case 'g': // goto
      cout << "goto: ";
      cin >> i;
      if (i<0)
        i = 0;
      if (i>=count)
        i = count-1;
      break;
    }
    if (!paused) {
      i++;
      if (i==count) {
        i--;
        paused = true;
      }
    }
}
```



```

  return 0;
}
```

../code/apps/featureviewer.cpp

```
#include <iostream>

#include "../io/sequence_capture.hpp"
#include "../features/feature_sequence.hpp"
#include "../features/keypoint_tracker.hpp"

using namespace std;



void usage(char* name)
{
  cout << "Usage:" << endl;
  cout << " " << name << " <path> [<pointDetector> [<pointDescriptor> [<
      edgeDetector>]]]" << endl;
  cout << " where path is the path to an AVI file, directory containing" <<
      endl;
  cout << " a sequence of images, or webcam device number;" << endl;
  cout << " pointDetector: {SIFT, SURF, ORB, FAST, STAR, MSER,";
  cout << " GFTT, HARRIS, Dense, SimpleBlob};" << endl;
  cout << " pointDescriptor: {SIFT, SURF, ORB, BRIEF};" << endl;
  cout << " edgeDetector: {CANNY, HARRIS}" << endl;
  exit(1);
}

/* useful functions */
bool isInteger(char* arg)
{
  while (*arg)
    if (*arg<='9' && *arg>='0')
```

```
      arg++;
    else
      return false;
  return true;
}

/* callback */
typedef struct trackerData {
  Point point;
  bool startTracking;
} trackerData;

void onMouse(int event, int x, int y, int, void* td)
{
  if (event != CV_EVENT_LBUTTONDOWN)
    return;

  ((trackerData*)td)->point = Point(x,y);
  ((trackerData*)td)->startTracking = true;
}


/* main */
int main (int argc, char** argv)
{

  if (argc==1)
    usage(argv[0]);

  string pointDetector = (argc>2) ? argv[2]: "SIFT";
  string pointDescriptor = (argc>3) ? argv[3]: "SIFT";
  string edgeDetector = (argc>4) ? argv[4]: "CANNY";

  SequenceCapture* sc;

  if (isInteger(argv[1]))
    sc = new SequenceCapture(atoi(argv[1]));
  else
    sc = new SequenceCapture(argv[1]);

  FeatureSequence* fs = new FeatureSequence(sc, pointDetector,
                                pointDescriptor, edgeDetector);

  namedWindow("features", CV_WINDOW_NORMAL);

  trackerData td;
  td.startTracking = false;
  setMouseCallback("features", onMouse, &td);
  KeypointTracker* tracker = NULL;
  Mat distinctiveness_plot;

  VideoWriter* recorder = NULL;

  bool paused = false;
  bool richDraw = true, matchDraw = true;
  Mat output;
  int count = sc->getFrameCount();
  int i = 0, last_i = -1;
  while (true) {
    if (count>-1 && i>=count-1 && !paused) {
      i = count - 1; // end of animation: stay at last frame
      paused = true;
    }
    if (!paused || i != last_i) {
      cout << "Calculating features for frame " << i << endl;
      fs->calculateFeaturesOf(i*(count>-1));
      fs->visualiseFeaturesOf(i*(count>-1), output, richDraw);
      if (matchDraw)
        fs->visualiseMatchesOf(i-1, i, output, true, richDraw, false);
```

```
  if (td.startTracking) {
    td.startTracking = false;
    cout << "Starting tracking at location " << td.point << " for frame "
        << i << endl;
    Point2f point(td.point.x, td.point.y);
    int nearest = fs->getNearestKeypoint(point, i);
    if (tracker)
      delete tracker;
    tracker = new KeypointTracker(&fs->kp, &fs->kd, nearest, i);
    namedWindow("distinctiveness", CV_WINDOW_NORMAL);
  }
  // draw tracked interest point
  if (tracker) {
    //tracker->trackUntil(i);
    // calculate all tracking info:
    // (will be done incrementally if feature data not known yet)
    tracker->trackUntil(0);
    tracker->trackUntil(count-1);
    if (tracker->getBestMatch(i) > -1)
      circle(output, fs->kp.at(i)->at(tracker->getBestMatch(i)).pt, 10,
          Scalar(0,255,0), 4);
    tracker->plotDistinctiveness(distinctiveness_plot, i);
    imshow("distinctiveness", distinctiveness_plot);
  }

  // show result
  imshow("features", output);

  // if recording, save frame
  if (recorder)
    *recorder << output;
}
last_i = i;

switch(waitKey(30)) {
 case '?':
  cout << "Keyboard shortcuts:" << endl;
  cout << " q/ESC:    quit" << endl;
  cout << " --- navigation ---" << endl;
  cout << " r:        refresh" << endl;
  cout << " space:    pause/resume" << endl;
  cout << " ->/n:     next frame" << endl;
  cout << " <-/p:     prev frame" << endl;
  cout << " PGUP:     +25 frames" << endl;
  cout << " PGDN:     -25 frames" << endl;
  cout << " f/b/HOME: first frame" << endl;
  cout << " l/e/END:  last frame" << endl;
  cout << " --- visuals ---" << endl;
  cout << " 1:        rich keypoint draw on/off" << endl;
  cout << " 2:        match arrows draw on/off" << endl;
  cout << " --- various ---" << endl;
  cout << " <click>   start tracking interest point" << endl;
  cout << " t         stop tracking" << endl;
  cout << " s         start/stop saving visible frames" << endl;
  break;
 case 'q':
 case 27: // esc
  exit(0);
 case 'r':
  last_i = -2; // refresh
  break;
 case 32: // space
 case 13: // return
  paused = !paused;
  break;
 case 'p':
 case 65361: // left arrow (same for all keyboards?)
  if (i>0 && count>-1)
```

```
      i--;
     break;
    case 'n':
    case 65363: // right arrow
     if (i+1<count || count==-1)
      i++;
     break;
    case 65365: // page up
     if (count>-1)
      i = max(0, i-25);
     break;
    case 65366: // page down
     if (count != -1)
      i = min(count-1, i+25);
     break;
    case 'f':
    case 'b':
    case 65360: // home
     i = (count==-1) ? -1 : 0;
     break;
    case 'l':
    case 'e':
    case 65367: // end
     i = (count==-1) ? -1 : count-1;
     break;
    case '1':
     richDraw = !richDraw;
     last_i = -2; // refresh
     break;
    case '2':
     matchDraw = !matchDraw;
     last_i = -2; // refresh
     break;
    case 't':
     if (tracker) {
      delete tracker;
      tracker = NULL;
      destroyWindow("distinctiveness");
      last_i = -2; // refresh
     }
     break;
    case 's':
     if (recorder) {
      // stop recording
      cout << "Stopped recording at frame " << i << endl;
      delete recorder;
      recorder = NULL;
     } else {
      // start recording
      cout << "Starting recording from frame " << ((!paused) ? i : i+1) <<
          endl;
      time_t tim; time(&tim); string ctim = (string)ctime(&tim);
      string record_filename = "output " + ctim.substr(0,ctim.size()-1) + ".
          avi";
      recorder = new VideoWriter(record_filename, CV_FOURCC('D','I','V','X')
          ,
                                 10, output.size(), true);
      if (paused)
       last_i = -2; // save next frame if paused
     }
    }

    if (!paused)
     i++;
   }

   return (0);
  }
```

D.2 Scripts

../code/scripts/multi_sfmcarver.py

```
#!/usr/bin/env python

#// Useful Python script to run sfmcarver multiple times
#// with different settings

# set variables
datasets    = ['memorial', 'sculpture1', 'car_and_wall1', 'car_and_wall2', '
    phonebox1', 'lampposts_on_wall1', 'sainsburys1', 'sainsburys2', 'houses1',
     'sciencepark2']
methods     = [0, 1, 2, 3]
params1     = [ [0], [0, 0.5], [0.005, 0.01, 0.05, 0.1], [0.001, 0.005, 0.01]
    ] # list of lists (one foreach method)
resolutions = [250, 500, 1000, 2500]

datasets    = ['temple']
methods     = [0,2,1]
params1     = [ [0], [0.0001, 0.0005, 0.001], [0] ]
resolutions = [250, 500, 1000, 2500]
ext         = '.ot'  # .ot or .bt for binary
skip_done   = True

# paths
path_vsfm  = '../../data/vsfm'
path_pict  = '../../data/pictures'
path_carve = '../../data/carve'
path_bin   = '../build/sfmcarver'

# check files and settings before we start
import sys, os
OK = True
for d in range(len(datasets)):
  curr_vsfm = path_vsfm  + '/' + datasets[d] + '.nvm'
  curr_pict = path_pict  + '/' + datasets[d]
  if not os.path.exists(curr_vsfm):
    print '[!!] Error: path does NOT exist: ' + curr_vsfm
    OK = False
  if not os.path.exists(curr_pict):
    print '[!!] Error: path does NOT exist: ' + curr_pict
    OK = False
  for m in range(len(methods)):
    if methods[m] < 0 or methods[m] > 3:
      print '[!!] Error: invalid method: ' + str(methods[m])
      OK = False
    for r in range(len(resolutions)):
      for p1 in range(len(params1[m])):
        curr_out  = path_carve + '/' + str(methods[m]) + '/' \
                  + datasets[d] + '_' + str(resolutions[r])  \
                  + '_' + str(params1[m][p1]) + ext
        if not skip_done and os.path.exists(curr_out):
          print '[!!] Warning: path DOES already exist: ' + curr_out
          OK = False
        if resolutions[r] < 1 or resolutions[r] > 10000:
          print '[!!] Error: invalid resolution: ' + str(resolutions[r])
          OK = False
        if params1[m][p1] < 0 or params1[m][p1] > 1:
          print '[!!] Error: invalid param1: ' + str(params1[m][p1])
          OK = False

if not OK:
  sys.exit()
else:
  print "All file paths and settings OK."
  print

# run
```

```
for d in range(len(datasets)):
  for m in range(len(methods)):
    for r in range(len(resolutions)):
      for p1 in range(len(params1[m])):
        print '++++++++++++++++++++++++++++++++++++++'
        print '  DATASET:    ' + datasets[d]         + ' ('+str( d+1)+'/'+str(
            len(datasets))+')'
        print '  METHOD:     ' + str(methods[m])     + ' ('+str( m+1)+'/'+str(
            len(methods))+')'
        print '  RESOLUTION: ' + str(resolutions[r]) + ' ('+str( r+1)+'/'+str(
            len(resolutions))+')'
        print '  PARAM1:     ' + str(params1[m][p1]) + ' ('+str(p1+1)+'/'+str(
            len(params1[m]))+')'
        print '++++++++++++++++++++++++++++++++++++++'
        curr_vsfm = path_vsfm  + '/' + datasets[d] + '.nvm'
        curr_pict = path_pict  + '/' + datasets[d]
        curr_out  = path_carve + '/' + str(methods[m]) + '/' \
                  + datasets[d] + '_' + str(resolutions[r])  \
                  + '_' + str(params1[m][p1]) + ext
        command = path_bin + ' ' + curr_vsfm + ' ' + curr_pict + ' ' \
                + curr_out + ' ' + str(methods[m]) + ' ' \
                + str(resolutions[r]) + ' ' + str(params1[m][p1])

        if skip_done and os.path.exists(curr_out):
          print ' SKIPPING (already exists): ' + curr_out
          continue

        print '  Running: '
        print '  ' + command
        print '++++++++++++++++++++++++++++++++++++++'
        os.system(command)

print
print 'Finished!'
```

D.3 Code of Helper classes

../code/objects/occupancy_grid.hpp

```
/* Occupancy grid; useful for carving */

/* Implementation:
 * - own
 *    can use PCL VoxelGrid (now used as downsample filter):
 *    http://docs.pointclouds.org/trunk/classpcl_1_1_voxel_grid.html
 * - Octree PCL:        [looks binary; leafnodes are occupancy;
 *                       more general than octomap acc. to forum]
 *    http://docs.pointclouds.org/trunk/
    classpcl_1_1octree_1_1_octree_point_cloud_occupancy.html
 *    http://www.pcl-users.org/Volume-estimation-td2807985.html
 * - OctoMap library:   [occupied, free, unknown; probabilistic possible]
 *    incl. viewer (octovis)
 *    http://octomap.sourceforge.net/
 *    http://octomap.sourceforge.net/doxygen/
 *    \_ carving: insertRay (uses integrateMissOnRay) for freeing
 *                a ray of voxels
 *    \_ visualisation: example code; castRay
 *    http://www.pcl-users.org/OctoMap-and-PCL-Question-td3850053.html
 */

#ifndef OCCUPANCYGRID_H
#define OCCUPANCYGRID_H

#include <octomap/octomap.h>
#include <octomap/OcTree.h>
#include <opencv2/core/core.hpp>
#include "../io/sfm_reader.hpp"
#include "../io/sequence_capture.hpp"
#include "../objects/listhelpers.cpp"
#include "../maxflow/graph.h"

using namespace std;
using namespace pcl;
using namespace octomap;


typedef struct projected_voxel
{
  double x, y;
  double r;
  double distance;

} projected_voxel;


bool projected_voxel_comp(projected_voxel a, projected_voxel b);


class OccupancyGrid
{
  public:
    OcTree* tree;
    SfMReader* sfm;

    OccupancyGrid(string path, string imagespath="", int resolution=250);
    OccupancyGrid();
    ~OccupancyGrid();
    bool load(string path);
    bool carve(int method=0,
               double param1=0.1);
    bool carveBaseline();
    bool carveVisSingleRay(double ignoredBorderSize=0.1,
                           bool exportUnknowns=false,
                           bool exportOccupied=true);
```

```
    bool carveVisOccSingleRayVeto(double occluderProbAddition=0.1,
                                  bool exportOccupied=true);
    bool carveVisOccSingleRayGeneral(double occluderProbAddition=0.1,
                                     double visibleProbAddition=0.1,
                                     double threshold=0.2,
                                     bool exportOccupied=true);
    bool save(string filename, bool binary=false);
    void graphcut(double gamma=1.0, double unknownProb=0.4);
    void extentVisibilityLists(double threshold=0.2);
    double reprojectMatch(Mat* img1, Mat* img2,
                          camera* cam1, camera* cam2,
                          PointXYZRGB point,
                          PointXYZRGB pose,
                          double voxel_size,
                          string method="L2",
                          bool showImgs=false);
    double patchDistance(Mat* patch1, Mat* patch2, string method="L2");
    void projectVoxel(PointXYZRGB* point,
                      double voxel_size, camera* cam,
                      PointXYZRGB* centre, double* r);

    void pcl2octomap(PointXYZRGB pcl, point3d& octomap);
    void octomap2pcl(point3d octomap, PointXYZRGB& pcl);

    void visualise(Mat& output, camera* cam,
                   bool redraw=true,
                   string method="circle", double alpha=0.5,
                   double max_dist=25);
    bool visualisePose(Mat& output, int poseID,
                       string method="circle", double alpha=0.5,
                       double max_dist=25);
    void visualisePath(vector<Mat>* output,
                       string method="circle", double alpha=0.5,
                       double max_dist=25);
    void visualisePath(vector<Mat>* output, vector<camera>* path,
                       string method="circle", double alpha=0.5,
                       double max_dist=25);

};

#endif
```

../code/objects/occupancy_grid.cpp

```
#include "occupancy_grid.hpp"

bool projected_voxel_comp(projected_voxel a, projected_voxel b)
{
  return a.distance < b.distance;
}

void error(const char* msg)
{
  cerr << "[!!] Error: " << msg << endl;
  exit(1);
}


OccupancyGrid::OccupancyGrid(string path, string imagespath, int resolution)
{
  sfm = new SfMReader(path, imagespath);

  // subdivide longest axis into given resolution
  Scalar min, max, axes;
  double max_axis_length;
  sfm->getExtrema(min, max);
  axes = max - min;
  max_axis_length = axes[0];
  if (axes[1] > max_axis_length) max_axis_length = axes[1];
```

```
  if (axes[2] > max_axis_length) max_axis_length = axes[2];

  tree = new OcTree(max_axis_length / resolution);
}

OccupancyGrid::OccupancyGrid()
{
  // default: load nothing
  // (for opening existing octree file and applying graphcut)
  sfm = NULL;
  tree = NULL;
}

OccupancyGrid::~OccupancyGrid()
{
  delete sfm;
}



// Delete current octree and load in another
// Note: still using original sfm file and image directory!
bool OccupancyGrid::load(string path)
{
  if (tree)
    delete tree;
  AbstractOcTree* abstract = AbstractOcTree::read(path);
  tree = dynamic_cast<OcTree*>(abstract);
}

/* General Carving function.
 * If exportUnknowns is set to true, the final result will
 * be altered such that the unknowns are set to occupied.
 * If exportOccupied is set to false, the final result will
 * be altered such that the occupied voxels are set to
 * unknown.
 */
bool OccupancyGrid::carve(int method,
                          double param1)
{
  if (sfm->poses.size() == 0 && method > 0) {
    cerr << "[!!] No camera poses known!" << endl;
    return false;
  }
  if (sfm->visible.size() < sfm->poses.size() && method > 0) {
    cerr << "[!!] No or not enough visibility information!" << endl;
    return false;
  }

  cout << "Carving.." << endl;
  switch (method) {
    case 0:
      return carveBaseline();
    case 1:
      return carveVisSingleRay(param1, true, true);
    case 2:
      return carveVisOccSingleRayVeto(param1, true);
    case 3:
      return carveVisOccSingleRayGeneral(param1, param1, 0.2, true);
    default:
      cerr << "[!!] Error: unknown carving method (" << method << ")" << endl;
      return false;
  }

}


/* Baseline 'carving': set voxels containing a point
 * to occupied
```

```
 */
bool OccupancyGrid::carveBaseline()
{
  point3d centre;
  int frame;

  if (sfm->points.size()==0)
    return false;

  // for every point ..
  for (int p=0; p<sfm->points.size(); p++) {
    // .. set corresponding voxel to occupied
    pcl2octomap(sfm->points.at(p), centre);
    tree->updateNode(centre, true);
  }

  // update up-tree and compress
  tree->updateInnerOccupancy();
  tree->prune();
  return true;
}


/* Visibility Space Carving (using single rays):
 * Carve using rays from camera poses to points
 * (ray sets every hitting voxel to free and point
 * voxel to occupied; the others are unknown and
 * therefore probably occluders.
 * Most simple and fast method (only one ray per
 * camera-point pair, forall points: forall cameras)
 * If exportUnknowns is set, only the unknown voxels
 * that reproject in at least one camera view are set
 * to occupied threshold. To ignore points projected
 * near camera view borders, set ignoredBorderSize != 0.0.
 */
bool OccupancyGrid::carveVisSingleRay(double ignoredBorderSize,
                                      bool exportUnknowns,
                                      bool exportOccupied)
{
  point3d origin, end;
  map<int,visibility>* vismap;
  map<int,visibility>::iterator it;
  int frame;
  cout << " a. carve" << endl;
  // for every point ..
  for (int p=0; p<sfm->points.size(); p++) {
    // .. carve for sequence of camera poses
    vismap = &sfm->visible.at(p);
    for (it = vismap->begin(); it!=vismap->end(); it++) {
      frame = (*it).first;
      pcl2octomap(sfm->poses.at(frame), origin);
      pcl2octomap(sfm->points.at(p), end);
      tree->insertRay(origin, end);
    }
  }

  // change unknowns or occupied voxels for desired output
  cout << " b. visualise" << endl;
  PointXYZRGB centre;
  Size subwindow_size;
  sfm->getWindowSize(subwindow_size);
  subwindow_size.height = (1.0 - ignoredBorderSize) * subwindow_size.height;
  subwindow_size.width  = (1.0 - ignoredBorderSize) * subwindow_size.width;
  if (exportUnknowns || !exportOccupied) {
    // update up-tree and compress
    tree->updateInnerOccupancy();
    tree->prune();
    // iterate through leaf nodes and alter state if necessary
```

```
    for (OcTree::leaf_iterator it = tree->begin_leafs(),
         end=tree->end_leafs(); it!=end; it++) {
      if (exportUnknowns
          && !tree->isNodeAtThreshold(*it)
          && !tree->isNodeOccupied(*it)) {
        /* Only set this unknown state node to occupied if the
         * centre is inside at least one camera view
         * (or don't initialise nodes that are not visible
         * earlier on)
         * Note: this is the bottle neck of the carve0 algorithm
         *       O(n^3) with n = resolution
         */
        // NOTE: this hugely decreases speed (for higher resolution)
        octomap2pcl(it.getCoordinate(), centre);
        for (int c=0; c<sfm->cameras.size(); c++) {
          if (sfm->reprojectsInsideImage(&centre,
                    &sfm->cameras.at(c),
                    subwindow_size))
          {
            it->setValue(tree->getOccupancyThresLog());
            break;
          }
        }
      } else if (!exportOccupied
          && tree->isNodeOccupied(*it)) {
        it->setValue(tree->getOccupancyThresLog()-1); // hack
      }

    }
  }
  // update up-tree and compress
  tree->updateInnerOccupancy();
  tree->prune();
  return true;
}

bool OccupancyGrid::carveVisOccSingleRayVeto(double occluderProbAddition,
                                             bool exportOccupied)
{
  point3d origin, end;
  map<int,visibility>* vismap;
  map<int,visibility>::iterator it;
  int frame;
  KeyRay ray;
  OcTreeNode* node;
  // find window size
  Size window_size;
  sfm->getWindowSize(window_size);

  // for every camera pose ..
  for (int c=0; c<sfm->poses.size(); c++) {
    cout << "\r >> " << c+1 << "/" << sfm->poses.size();
    cout.flush();
    sfm->selectPointsForCamera(c, true);
    // .. check, for each point, if it is either visible
    //    or - if not - at least lies inside the camera window
    for (int p=0; p<sfm->points.size(); p++) {
      if (sfm->points_curr_visible.at(p)) {
        // if it is visible (thus inside camera window):
        // carve ray (binary)
        pcl2octomap(sfm->poses.at(c), origin);
        pcl2octomap(sfm->points.at(p), end);
        if (exportOccupied) {
          // mark end point as occupied
          tree->insertRay(origin, end);
        } else {
          // don't mark end point as occupied
          // unfortunately, this method is protected:
```

```
      //tree->integrateMissOnRay(origin, end);
      // therefore we hack:
      node = tree->search(end);
      double value;
      if (node)
        value = node->getValue();
      tree->insertRay(origin, end);
      if (node)
        node->setValue(value);
    }
  } else if (sfm->points_curr_invisible.at(p)) {
    // if it is not visible but lies inside camera window:
    // increase occluder probability on the ray
    pcl2octomap(sfm->poses.at(c), origin);
    pcl2octomap(sfm->points.at(p), end);
    // cast a ray and get voxels we hit in between
    ray.reset();
    tree->computeRayKeys(origin, end, ray);
    for (KeyRay::const_iterator it = ray.begin();
         it != ray.end(); it++) {
      node = tree->search(*it);
      if (!node) {
        // create node (p = 0.2)
        tree->updateNode(*it, false);
        node = tree->search(*it);
      }

      // increase if not below 'free' threshold
      if (exp(node->getValue()) > tree->getProbMiss()) {
        node->setValue( log( exp(node->getValue()) + occluderProbAddition)
            );
      }
    }
  }
    } // end for ev point
  } // end for ev camera
  cout << endl;

  // update up-tree and compress
  tree->updateInnerOccupancy();
  tree->prune();
  return true;
}

bool OccupancyGrid::carveVisOccSingleRayGeneral(double occluderProbAddition,
    double visibleProbAddition, double threshold, bool exportOccupied)
{
  point3d origin, end;
  map<int,visibility>* vismap;
  map<int,visibility>::iterator it;
  int frame;
  KeyRay ray;
  OcTreeNode* node;
  // find window size
  Size window_size;
  sfm->getWindowSize(window_size);

  // improve matches
  extentVisibilityLists(threshold);

  // for every camera pose ..
  for (int c=0; c<sfm->poses.size(); c++) {
    cout << "\r >> " << c+1 << "/" << sfm->poses.size();
    cout.flush();
    sfm->selectPointsForCamera(c, true);
    // .. check, for each point, if it is either visible
    //    or - if not - at least lies inside the camera window
    for (int p=0; p<sfm->points.size(); p++) {
```

```
    if (sfm->points_curr_visible.at(p)
     || sfm->points_curr_invisible.at(p)) {
      pcl2octomap(sfm->poses.at(c), origin);
      pcl2octomap(sfm->points.at(p), end);
      // cast a ray and get voxels we hit in between
      ray.reset();
      tree->computeRayKeys(origin, end, ray);
      for (KeyRay::const_iterator it = ray.begin();
           it != ray.end(); it++) {
        node = tree->search(*it);
        if (!node) {
          // create node (p = 0.2)
          tree->updateNode(*it, false);
          node = tree->search(*it);
        }

        // if it is visible (thus inside camera window):
        // decrease occluder probability on the ray
        if (sfm->points_curr_visible.at(p))
          node->setValue( log( exp(node->getValue()) - visibleProbAddition)
              );
        // if it is not visible but lies inside camera window:
        // increase occluder probability on the ray;
        else if (sfm->points_curr_invisible.at(p))
          node->setValue( log( exp(node->getValue()) + occluderProbAddition)
               );

      }
      if (exportOccupied)
        // set end to occupied
        tree->updateNode(end, true);
    }
  } // end for ev point
} // end for ev camera
  cout << endl;

  // update up-tree and compress
  tree->updateInnerOccupancy();
  tree->prune();
  return true;
}

bool OccupancyGrid::save(string filename, bool binary)
{
  if (binary)
    return tree->writeBinary(filename);
  else
    return tree->write(filename);
}


void OccupancyGrid::graphcut(double gamma, double unknownProb)
{

  cout << "Graphcut (regularisation).." << endl;

  tree->expand();

  cout << " a. init graph" << endl;

  // get extrema
  point3d min_pt, max_pt;
  OcTreeKey min, max;
  double minX, minY, minZ, maxX, maxY, maxZ;
  int maxI, maxJ, maxK;
  tree->getMetricMin(minX, minY, minZ);
  tree->getMetricMax(maxX, maxY, maxZ);
  min_pt.x() = (float)minX;
```

```
min_pt.y() = (float)minY;
min_pt.z() = (float)minZ;
max_pt.x() = (float)maxX;
max_pt.y() = (float)maxY;
max_pt.z() = (float)maxZ;
tree->genKey(min_pt, min);
tree->genKey(max_pt, max);
maxI = max.k[0] - min.k[0] + 1;
maxJ = max.k[1] - min.k[1] + 1;
maxK = max.k[2] - min.k[2] + 1;

// init graph
// source = occupied, sink = free
int nodeCount = maxI * maxJ * maxK;
typedef float GraphCost;
typedef Graph<GraphCost, GraphCost, GraphCost> GraphType;
GraphType *graph = new GraphType(nodeCount, 3*nodeCount, error);
// We can't get the unary costs from the graph structure,
// but we need them to set the pairwise costs, so we'll keep
// the values in a separate list too:
float* unary = new float[nodeCount];
if (unary == NULL)
  error("not enough memory for unary array");
cout << " nodes: " << nodeCount << endl;

// add nodes
graph->add_node(nodeCount);
// add unary costs (default values)
int id;
for (int k=0; k<maxK; k++) {
  for (int j=0; j<maxJ; j++) {
    for (int i=0; i<maxI; i++) {
      id = k * maxI * maxJ + j * maxI + i;
      graph->add_tweights(id, unknownProb, 1 - unknownProb);
      unary[id] = unknownProb;
    }
  }
}

// set octree leaf nodes unary costs
double prob;
OcTreeKey key;
OcTree::leaf_iterator end;
for (OcTree::leaf_iterator it = tree->begin_leafs(),
     end = tree->end_leafs(); it != end; it++) {
  key = it.getKey();
  prob = exp( it->getValue() ) - 0.2;
  prob = std::max( std::min(prob, 1.0), 0.0 ); // truncate
  //prob = (prob > 0.7);
  //prob = tree->isNodeOccupied(*it);
  id = (key.k[2] - min.k[2]) * maxI * maxJ
     + (key.k[1] - min.k[1]) * maxI
     + (key.k[0] - min.k[0]);
  graph->add_tweights(id, prob, (1 - prob));
  unary[id] = prob;
}

// add edges + pairwise costs
int id2, val;
for (int k=0; k<maxK-1; k++) {
  for (int j=0; j<maxJ-1; j++) {
    for (int i=0; i<maxI-1; i++) {
      id = k * maxI * maxJ + j * maxI + i;
      // connect to right
      id2 = k * maxI * maxJ + j * maxI + (i+1);
      val = gamma * (1 - abs(unary[id] - unary[id2]) );
      //val = gamma * int( (unary[id]<0.7) != (unary[id2]<0.7) );
      graph->add_edge(id, id2, val, val);
```

```
        // connect to bottom
        id2 = k * maxI * maxJ + (j+1) * maxI + i;
        val = gamma * (1 - abs(unary[id] - unary[id2]) );
        //val = gamma * int( (unary[id]<0.7) != (unary[id2]<0.7) );
        graph->add_edge(id, id2, val, val);
        // connect to back
        id2 = (k+1) * maxI * maxJ + j * maxI + i;
        val = gamma * (1 - abs(unary[id] - unary[id2]) );
        //val = gamma * int( (unary[id]<0.7) != (unary[id2]<0.7) );
        graph->add_edge(id, id2, val, val);
      }
    }
  }
  delete unary;

  // run graphcut / maxflow algorithm
  cout << " b. running graphcut" << endl;
  int flow = graph->maxflow();

  // convert back
  cout << " c. converting back to octree" << endl;
  tree->clear();
  bool occupied;
  for (int k=0; k<maxK; k++) {
    for (int j=0; j<maxJ; j++) {
      for (int i=0; i<maxI; i++) {
        id = k * maxI * maxJ + j * maxI + i;
        occupied =
          (graph->what_segment(id, GraphType::SINK) == GraphType::SOURCE);
        if (occupied) {
          key = OcTreeKey(min.k[0]+i, min.k[1]+j, min.k[2]+k);
          tree->updateNode(key, true);
        }
      }
    }
  }

  if (graph)
    delete graph;
  tree->updateInnerOccupancy();
  tree->prune();

}

/* extent the visibility lists in sfmreader using
 * reprojection and distance between patches around
 * the projected pixels
 */
void OccupancyGrid::extentVisibilityLists(double threshold)
{
  cout << "Extending visibility lists.." << endl;
  Size window_size;
  sfm->getWindowSize(window_size);
  double voxel_size = tree->getResolution();

  // load all images (memory intensive!)
  if (sfm->imagespath == "") {
    cerr << "[!!] Error: image path not given" << endl;
    return;
  }
  vector<Mat> images;
  images.push_back(Mat());
  SequenceCapture sc(sfm->imagespath);
  while (sc.read(images.back()))
    try {
      images.push_back(Mat());
    } catch (exception& e) {
      cerr << "[!!] Error loading images: " << e.what() << endl;
```

```
    }
  images.pop_back();
  //cout << " >> " << images.size() << " images in memory" << endl;

  // for each point ..
  ListHelper LH;
  bool visible[sfm->poses.size()];
  bool invisible[sfm->poses.size()];
  for (int p=0; p<sfm->points.size(); p++) {
    cout << "\r >> " << p+1 << "/" << sfm->points.size();
    cout.flush();
    for (int c=0; c<sfm->poses.size(); c++) {
      visible[c] = sfm->visible.at(p).find(c) != sfm->visible.at(p).end();
      invisible[c] = (!visible[c] && sfm->reprojectsInsideImage(p, c,
          window_size));
    }

    // .. follow in sequence
    double distance;
    double dist_cam_bw, dist_cam_fw;
    int item, nearest, nearest_bw, nearest_fw;
    item = LH.bool_find_first(visible, sfm->poses.size());

    for (int i=0; i<sfm->poses.size(); i++) {
      // if point reprojects inside this image, but was marked invisible:
      if (invisible[i]) {
        // 1. find nearest camera in which it was visible ..
        //    (heuristic: search nearest in both sides, take euclidean nearest
            )
        nearest_fw = LH.bool_find_next(visible, sfm->poses.size(), i);
        nearest_bw = LH.bool_find_prev(visible, sfm->poses.size(), i);
        if (nearest_fw < 0 && nearest_bw < 0)
          continue; // point never visible (this should never happen)
        else if (nearest_fw < 0)
          nearest = nearest_bw;
        else if (nearest_bw < 0)
          nearest = nearest_fw;
        else {
          dist_cam_bw = pow(sfm->poses.at(nearest_bw).x - sfm->poses.at(i).x
              ,2)
                  + pow(sfm->poses.at(nearest_bw).y - sfm->poses.at(i).y
                      ,2);
          dist_cam_fw = pow(sfm->poses.at(nearest_fw).x - sfm->poses.at(i).x
              ,2)
                  + pow(sfm->poses.at(nearest_fw).y - sfm->poses.at(i).y
                      ,2);
          nearest = (dist_cam_fw>dist_cam_bw) ? nearest_bw : nearest_fw;
        }
        // 2. calculate reprojection match measure
        distance = reprojectMatch(&images.at(i),
                                  &images.at(nearest),
                                  &sfm->cameras.at(i),
                                  &sfm->cameras.at(nearest),
                                  sfm->points.at(p),
                                  sfm->poses.at(i),
                                  voxel_size,
                                  "L2", false);
        // 3. if the reprojection patches seem to match,
        //    add 'invisible' camera pose to visibility list!
        if (distance < threshold) {
          PointXYZRGB reproj;
          sfm->reproject(&sfm->points.at(p), &sfm->cameras.at(i), &reproj);
          sfm->distortPointR1(&reproj, &sfm->cameras.at(i));
          sfm->visible.at(p).insert(pair<int,visibility>(i,visibility()));
          sfm->visible.at(p).find(i)->second.index = -1; // i don't know
          sfm->visible.at(p).find(i)->second.x = reproj.x;
          sfm->visible.at(p).find(i)->second.y = reproj.y;
        }
```

```
      }
    }

  }
  cout << endl;
}

double OccupancyGrid::reprojectMatch(Mat* img1, Mat* img2, camera* cam1,
    camera* cam2, PointXYZRGB point, PointXYZRGB pose, double voxel_size,
    string method, bool showImgs)
{

  // find patch centres
  PointXYZRGB centr1, centr2;
  double patch_size;
  projectVoxel(&point, voxel_size, cam1, &centr1, &patch_size);
  sfm->reproject(&point, cam2, &centr2);
  sfm->distortPointR1(&centr2, cam2);


  int x1_start = int(centr1.x - patch_size + img1->size().width/2.0);
  int x1_stop  = int(centr1.x + patch_size + img1->size().width/2.0);
  int y1_start = int(centr1.y - patch_size + img1->size().height/2.0);
  int y1_stop  = int(centr1.y + patch_size + img1->size().height/2.0);
  int x2_start = int(centr2.x - patch_size + img2->size().width/2.0);
  int x2_stop  = x2_start + (x1_stop - x1_start);
  int y2_start = int(centr2.y - patch_size + img2->size().height/2.0);
  int y2_stop  = y2_start + (y1_stop - y1_start);

  // check if inside image and not near edge
  if (x1_start <= 0 || x1_stop > img1->size().width
   || y1_start <= 0 || y1_stop > img1->size().height
   || x2_start <= 0 || x2_stop > img2->size().width
   || y2_start <= 0 || y2_stop > img2->size().height)
    return -1;

  // calculate distance between patches
  Mat patch1 = (*img1)(Range(y1_start, y1_stop), Range(x1_start, x1_stop));
  Mat patch2 = (*img2)(Range(y2_start, y2_stop), Range(x2_start, x2_stop));
  double distance = patchDistance(&patch1, &patch2, method);

  if (showImgs && distance >= 0) {
    // /*
    cout << "D = " << distance << endl;

    namedWindow("IMG1", CV_WINDOW_NORMAL);
    namedWindow("IMG2", CV_WINDOW_NORMAL);
    Mat image1 = img1->clone();
    Mat image2 = img2->clone();
    rectangle(image1, Point(x1_start, y1_start), Point(x1_stop, y1_stop),
        Scalar(255,0,0), 4);
    rectangle(image2, Point(x2_start, y2_start), Point(x2_stop, y2_stop),
        Scalar(255,0,0), 4);
    circle(image1,
      Point((x1_start+x1_stop)/2, (y1_start+y1_stop)/2),
      5, Scalar(0,0,255), 4);
    circle(image2,
      Point((x2_start+x2_stop)/2, (y2_start+y2_stop)/2),
      5, Scalar(0,0,255), 4);
    imshow("IMG1", image1);
    imshow("IMG2", image2);
    while (32 != waitKey(0))
      ; // wait for space
    destroyWindow("IMG1");
    destroyWindow("IMG2");
    // */

}
```

```
  return distance;

}

// Normalised average r,g,b, intensity difference
// Note: assuming images are represented by 8bit unsigned integers
double OccupancyGrid::patchDistance(Mat* patch1, Mat* patch2, string method)
{
  double distance = 0;
  double r, g, b;
  for (int x=1; x<=patch1->size().width; x++) {
    for (int y=1; y<=patch1->size().height; y++) {
      r = patch1->at<Vec3b>(x, y)[0] - patch2->at<Vec3b>(x, y)[0];
      g = patch1->at<Vec3b>(x, y)[1] - patch2->at<Vec3b>(x, y)[1];
      b = patch1->at<Vec3b>(x, y)[2] - patch2->at<Vec3b>(x, y)[2];

      distance += pow( double(r) / pow(2,8), 2 );
      distance += pow( double(g) / pow(2,8), 2 );
      distance += pow( double(b) / pow(2,8), 2 );
      //distance += abs( double(img1->at<unsigned int>(x1, y1) - img2->at<
          unsigned int>(x2, y2)) / pow(2,32) );
    }
  }

  return sqrt( distance / (3*patch1->size().width*patch1->size().height) );

}


/* determine patch size (approximation):
 * project point with distance voxelSize/2 from point inside img1
 * and decide patch size based on distance from projected point
 * Note: assuming same patch size for both images for easy comparing
 * (small camera displacement assumption)
 * Outputs: centre (projected point), r (projected voxel size)
 */
void OccupancyGrid::projectVoxel(PointXYZRGB* point, double voxel_size, camera
    * cam, PointXYZRGB* centre, double* r)
{
  PointXYZRGB pose;
  pose.x = cam->t.at<double>(0);
  pose.y = cam->t.at<double>(1);
  pose.z = cam->t.at<double>(2);
  // project point to get centre output
  sfm->reproject(point, cam, centre);
  sfm->distortPointR1(centre, cam);

  PointXYZRGB offset_vect, pointpose;
  // offset_point = point + offset_vect
  //              = point + 0.5*voxel_size * ( (point - pose) X [1 0 0]' )
  pointpose.x = point->x - pose.x;
  pointpose.y = point->y - pose.y;
  pointpose.z = point->z - pose.z;
  double offset_vect_length = sqrt(pow(pointpose.z,2)+pow(pointpose.y,2));
  offset_vect.x = 0;

  offset_vect.y = pointpose.z *  0.5*voxel_size / offset_vect_length;
  offset_vect.z = pointpose.y * -0.5*voxel_size / offset_vect_length;
  PointXYZRGB offset_point;
  offset_point.x = point->x + offset_vect.x;
  offset_point.y = point->y + offset_vect.y;
  offset_point.z = point->z + offset_vect.z;

  PointXYZRGB offset_proj;
  sfm->reproject(&offset_point, cam, &offset_proj);
  sfm->distortPointR1(&offset_proj, cam);
```

```
  // calculate approximated size of projected voxel
  *r = sqrt(pow(centre->x - offset_proj.x, 2)
          + pow(centre->y - offset_proj.y, 2));
}


void OccupancyGrid::visualise(Mat& output, camera* cam, bool redraw, string
    method, double alpha, double max_dist)
{

  Mat annotated = output.clone();
  vector<projected_voxel> voxels;

  // reproject all voxels into image
  PointXYZRGB centre, projected, pose, direction, hit;
  pose.x = cam->t.at<double>(0);
  pose.y = cam->t.at<double>(1);
  pose.z = cam->t.at<double>(2);
  point3d pose_oct, direction_oct, hit_oct;
  pcl2octomap(pose, pose_oct);
  Size window_size;
  sfm->getWindowSize(window_size);
  double voxel_size = tree->getResolution();
  double r, distance;
  bool found;
  for (OcTree::leaf_iterator it = tree->begin_leafs(),
       end=tree->end_leafs(); it!=end; it++) {
    if (tree->isNodeOccupied(*it)) {
      octomap2pcl(it.getCoordinate(), centre);
      // if reprojects into image..
      if (sfm->reprojectsInsideImage(&centre, cam, window_size, &projected)) {
        // .. and isn't occluded by other voxels ..
        direction.x = centre.x - pose.x;
        direction.y = centre.y - pose.y;
        direction.z = centre.z - pose.z;
        pcl2octomap(direction, direction_oct);
        found = tree->castRay(pose_oct, direction_oct, hit_oct, true);
        if (found)
          octomap2pcl(hit_oct, hit);
        //if (found && hit.x == centre.x && hit.y == centre.y && hit.z ==
            centre.z) {
        if (found && sqrt(pow(hit.x-centre.x,2) + pow(hit.y-centre.y,2) + pow(
            hit.z-centre.z,2)) < max_dist*voxel_size) {
          // .. save to the list
          sfm->distortPointR1(&projected, cam);
          projectVoxel(&centre, voxel_size, cam, &projected, &r);
          distance = sqrt(pow(centre.x - pose.x, 2)
                        + pow(centre.y - pose.y, 2)
                        + pow(centre.z - pose.z, 2));
          voxels.push_back(projected_voxel());
          voxels.back().x = projected.x + annotated.size().width/2.0;
          voxels.back().y = projected.y + annotated.size().height/2.0;
          voxels.back().r = r;
          voxels.back().distance = distance;
        }
      }
    }
  }

  // sort list based on distance
  sort(voxels.begin(), voxels.end(), projected_voxel_comp);

  // draw list on annotated image from far to near
  Scalar colour1(255,0,0);
  Scalar colour2(0,0,255);
  Scalar colour;
  double cmin = log(voxels.front().distance);
  double cmax = log(voxels.back().distance);
```

```
  double rate;
  for (int i=voxels.size()-1; i>=0; i--) {
    rate = (log(voxels[i].distance) - cmin) / (cmax - cmin);
    //rate = log(voxels[i].distance) / cmax;
    colour = Scalar(
           rate * colour1[0] + (1-rate) * colour2[0],
           rate * colour1[1] + (1-rate) * colour2[1],
           rate * colour1[2] + (1-rate) * colour2[2]);
    if (method == "circle")
      circle(annotated, Point(voxels[i].x, voxels[i].y), voxels[i].r, colour,
          -1);
    else if (method == "rectangle")
      rectangle(annotated, Point(voxels[i].x - voxels[i].r,
                                 voxels[i].y - voxels[i].r),
                           Point(voxels[i].x + voxels[i].r,
                                 voxels[i].y + voxels[i].r),
                           colour, -1);
    else
      cerr << "[!!] Error: unknown visualisation method: " << method << endl;

  }


  // combine original and annotated images
  addWeighted(output, alpha, annotated, 1-alpha, 0, output);

}


bool OccupancyGrid::visualisePose(Mat& output, int poseID, string method,
    double alpha, double max_dist)
{
  // use order as given by sequence capture (sorted directory listing)
  // (we need to correct the cameraID for this)
  string filename = sfm->sc->images.at(poseID);
  int camID = sfm->getImageID(filename);
  if (poseID < 0) {
    cerr << "[!!] Error: cannot find image file " << filename << endl;
    return false;
  }
  sfm->sc->setPosition(poseID);
  sfm->sc->read(output);
  visualise(output, &sfm->cameras.at(camID), false, method, alpha, max_dist);
  return true;
}



// Visualise (project) voxels onto images in the camera pose path
// (output vector should be empty)
void OccupancyGrid::visualisePath(vector<Mat>* output, string method, double
    alpha, double max_dist)
{
  output->clear();

  for (int i=0; i < sfm->poses.size(); i++) {
    output->push_back(Mat());
    visualisePose(output->at(i), i, method, alpha, max_dist);
  }

}


// Visualise (project) voxels onto images in vector output for given path
void OccupancyGrid::visualisePath(vector<Mat>* output, vector<camera>* path,
    string method, double alpha, double max_dist)
{
  if (path->size() > 0 && path->size() != output->size()) {
    cerr << "[!!] Error: output is supposed to be the same size";
```

```
    cerr << " as path, or empty (OccupancyGrid::visualisePath)" << endl;
    return;
  }

  for (int i=0; i<path->size(); i++) {
    if (output->size() <= i) {
      output->push_back(Mat());
      visualise(output->at(i), &path->at(i), true, method, alpha, max_dist);
    } else {
      visualise(output->at(i), &path->at(i), false, method, alpha, max_dist);
    }
  }

}


void OccupancyGrid::pcl2octomap(PointXYZRGB pcl, point3d& octomap)
{
  octomap.x() = pcl.x;
  octomap.y() = pcl.y;
  octomap.z() = pcl.z;
}


void OccupancyGrid::octomap2pcl(point3d octomap, PointXYZRGB& pcl)
{
  pcl.x = octomap.x();
  pcl.y = octomap.y();
  pcl.z = octomap.z();
  pcl.r = 150;
  pcl.g = 150;
  pcl.b = 150;
}
```


../code/io/sfm_reader.hpp

```
/* reader for output of SfM programs; extracts camera poses and points */
/* note: would be nice to split into seperate readers */

#ifndef SFMREADER_H
#define SFMREADER_H

#include <string>
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <map>
#include <dirent.h>
#include <opencv2/core/core.hpp>
#include <pcl/point_types.h>
#include <pcl/point_cloud.h>
#include <pcl/io/ply_io.h>
#include <pcl/visualization/common/common.h>

#include "sequence_capture.hpp"

using namespace std;
using namespace pcl;
using namespace cv;

typedef struct visibility
{
  int index;
  double x;
  double y;
} visibility;

typedef struct camera
```

```
{
  Mat R;
  Mat t;
  double focal;
  double radial[2];
} camera;

class SfMReader
{

  private:

    bool readNVM();
    bool readOUT();
    bool readPLY();
    bool readTXT();
    bool readPCD();
    bool integerLine(string line, int count=1);

  public:
    string path;
    string imagespath;
    Size window_size;
    string ext; // file type
    SequenceCapture* sc;
    PointCloud<PointXYZRGB> points, points_original;
    PointCloud<PointXYZRGB> poses;
    vector<camera> cameras;
    vector<map<int,visibility> > visible;
    PointXYZRGB* line_start;
    vector<PointXYZRGB*> line_ends;
    vector<visibility*> curr_visible_keypoints;
    vector<bool> points_curr_visible;
    vector<bool> points_curr_invisible;
    vector<string> image_filenames; // used for nvm only

/* Possible other camera representations:
 * - visualization::Camera  f, t,    fovy, lookat, up
 *    \_ has nice cvtWindowCoordinates function
 * - io::ply::camera
 * - CameraParams           f, t, R, ratio, ppx, ppy
 */

    SfMReader();
    SfMReader(string path, string imagespath="");
    ~SfMReader();
    bool read();
    bool read(string path);
    void getWindowSize(Size& size);
    int getImageID(string filename);
    bool selectPointsForCamera(int id,
                               bool calcInvisible=true,
                               Scalar colourCamera = Scalar(150,0,255),
                               Scalar colourSelectedCamera = Scalar(0,255,0),
                               Scalar colourVisiblePoint = Scalar(0,255,0),
                               Scalar colourInvisiblePoint = Scalar(0,0,255));
    bool selectCamerasForPoint(int id,
                               Scalar colourCamera = Scalar(150,0,255),
                               Scalar colourSelectedCamera = Scalar(0,255,0),
                               Scalar colourSelectedPoint = Scalar(0,255,0));

    void resetPointColours();
    void getExtrema(Scalar& min, Scalar& max);
    void reproject(PointXYZRGB* point, camera* cam, PointXYZRGB* projected,
        bool* front=NULL);
    bool reprojectsInsideImage(int pointID, int camID, Size size, PointXYZRGB*
         projected=NULL);
```

```
    bool reprojectsInsideImage(PointXYZRGB* point, camera* cam, Size size,
        PointXYZRGB* projected=NULL);
    void distortPointR1(PointXYZRGB* point, camera* cam);
    void quaternion2matrix(Mat& q, Mat& R);

};

#endif
```

../code/io/sfm_reader.cpp

```
#include "sfm_reader.hpp"

/* Outputs:
 *
 *                 cameras  points   visibility     sift in:
 * VisualSFM: NVM     x       x          x          .sift files
 *            PLY     -       x          -
 * Bundler:   OUT     x       x          x          .tar.gz files
 *            PLY     -       x          -
 * Voodoo:    TXT     x       x          -          .key files
 *                                                  (active export)
 * [PCL]:     PCD     -       x          -          -
 *
 *                                                  (might want to use
 *                                                  SIFT descriptors too)
 */

SfMReader::SfMReader()
{
  SfMReader("");
}

SfMReader::SfMReader(string path, string imagespath)
{
  this->path = path;
  this->imagespath = imagespath;
  window_size = Size(0,0);
  if (imagespath != "")
    sc = new SequenceCapture(imagespath);
  else
    sc = NULL;
  read();
}

SfMReader::~SfMReader()
{
  delete sc;
}


bool SfMReader::read()
{
  if (path == "")
    return false;

  return read(path);
}

bool SfMReader::read(string path)
{
  if (path.find_last_of(".") == string::npos)
    return false;

  // save filename if not saved earlier
  if (this->path == "")
    this->path = path;

  // call appropiate function for file extension
```

```
  ext = path.substr(path.find_last_of(".") + 1);

  if (ext == "nvm")
    return readNVM(); // VisualSFM
  else if (ext == "out")
    return readOUT(); // Bundler
  else if (ext == "ply")
    return readPLY(); // VisualSFM/Bundler
  else if (ext == "txt")
    return readTXT(); // Voodoo
  else
    cerr << "[!!] (SfMReader::read) unknown file format: " << ext << endl;

  return true;
}

bool SfMReader::readNVM()
{
  /* output from: VisualSFM
   * contains: cameras, points, visibility
   * note: only reading first model
   * format:
   *  http://www.cs.washington.edu/homes/ccwu/vsfm/doc.html#nvm
   *  0: <header>          (3 lines)
   *  1: <cameras>         (1 line each)
   *  2: <header>          (2 lines)
   *  3: <points pos+col>  (1 line each)  + <visibility ijxy>  <-- many
   *  4: <header>          (9+ lines)
   */

  ifstream file(path.c_str());
  if (!file.is_open())
    return false;

  int camera_count, point_count;
  size_t found;

  int state = 0; // for states see comment above
  string line;
  while (file.good()) {
    getline(file, line);
    stringstream sline(line);
    string filename;
    Mat q(4, 1, CV_64FC1);
    switch (state) {
      case 0: // header
        if (integerLine(line, 1)) {
          sline >> camera_count;
          state = 1;
        }
        break;
      case 1: // cameras
        if (camera_count-- == 0) {
          state = 2;
          break;
        }
        // new camera
        double cx, cy, cz;  // pos: position
        double fx,fy,fz;    // focal: lookat
        double vx,vy,vz;    // view: up vector
        double fovy;        // fovy: FoV in y direction
        double focal, q1,q2,q3,q4, radial;
        sline >> filename >> focal >> q1>>q2>>q3>>q4 >> cx >> cy >> cz >>
            radial;
        found = filename.find_last_of("/\\");
        if (found != string::npos)
          filename = filename.substr(found+1);
```

```
      q.at<double>(0) = q1; q.at<double>(1) = q2;
      q.at<double>(2) = q3; q.at<double>(3) = q4;

      // add new camera to list of cameras
      cameras.push_back(camera());
      cameras.back().t = Mat(3, 1, CV_64FC1);
      // apparently, t = -1 * t in nvm; we need to correct this:
      cameras.back().t.at<double>(0) = cx;
      cameras.back().t.at<double>(1) = cy;
      cameras.back().t.at<double>(2) = cz;
      cameras.back().R = Mat(3, 3, CV_64FC1);
      quaternion2matrix(q, cameras.back().R);
      cameras.back().focal = focal;
      cameras.back().radial[0] = radial;
      cameras.back().radial[1] = 0;
      image_filenames.push_back(filename);
      // also add camera pose to poses point cloud:
      poses.push_back(PointXYZRGB(255,0,150));
      poses.back().x = cx;
      poses.back().y = cy;
      poses.back().z = cz;
      break;
    case 2: // header
      if (integerLine(line, 1)) {
        sline >> point_count;
        state = 3;
      }
      break;
    case 3: // points
      if (point_count-- == 0) {
        state = 4;
        break;
      }
      // add new point to point cloud
      float x,y,z;
      int r,g,b;
      sline >> x >> y >> z;
      sline >> r >> g >> b;
      points.push_back(PointXYZRGB(r,g,b));
      points.back().x = x;
      points.back().y = y;
      points.back().z = z;

      // add visibility to visible
      visible.push_back(map<int,visibility>());
      int vis_count, frame, index;
      sline >> vis_count;
      while (vis_count--) {
        sline >> frame >> index >> x >> y;
        visible.back().insert(pair<int,visibility>(frame,visibility()));
        visible.back().find(frame)->second.index = index;
        visible.back().find(frame)->second.x = x;
        visible.back().find(frame)->second.y = y;
      }

      break;
    case 4: // header
      break;
  }
}
file.close();

cout << poses.size() << " poses read!" << endl;
cout << points.size() << " points read!" << endl;

return true;
}
```

```
bool SfMReader::readOUT()
{
  /* output from: Bundler
   * contains: cameras, points, visibility
   * format:
   *  http://phototour.cs.washington.edu/bundler/bundler-v0.4-manual.html#S6
   *  0: <header>          (2 lines)
   *  1: <cameras>         (5 lines each)
   *  2: <points pos+col>  (3 lines each)  + <visibility ijxy>
   */

  ifstream file(path.c_str());
  if (!file.is_open())
    return false;

  int camera_count, point_count;

  int state = 0; // for states see comment above
  string line;
  while (file.good()) {
    getline(file, line);
    stringstream sline(line);
    string filename;
    switch (state) {
      case 0: // header
        if (integerLine(line, 2)) {
          sline >> camera_count >> point_count;
          state = 1;
        }
        break;
      case 1: // cameras
        if (camera_count-- == 0) {
          state = 2;
          sline.flush();
          sline << line;
          // no break! use this line
        } else {
          // new camera
          double cx, cy, cz;  // pos: position
          double focal, r00,r01,r02,r10,r11,r12,r20,r21,r22;
          double radial1, radial2;
          sline >> focal >> radial1 >> radial2;
          getline(file, line);
          sline.str(line);
          sline >> r00 >> r10 >> r20;
          getline(file, line);
          sline.str(line);
          sline >> r01 >> r11 >> r21;
          getline(file, line);
          sline.str(line);
          sline >> r02 >> r12 >> r22;
          getline(file, line);
          sline.str(line);
          sline >> cx >> cy >> cz;

          // add new camera to list of cameras
          cameras.push_back(camera());
          cameras.back().t = Mat(3, 1, CV_64FC1);
          cameras.back().t.at<double>(0) = cx;
          cameras.back().t.at<double>(1) = cy;
          cameras.back().t.at<double>(2) = cz;
          cameras.back().R = Mat(3, 3, CV_64FC1);
          cameras.back().R.at<double>(0,0) = r00;
          cameras.back().R.at<double>(0,1) = r01;
          cameras.back().R.at<double>(0,2) = r02;
          cameras.back().R.at<double>(1,0) = r10;
          cameras.back().R.at<double>(1,1) = r11;
```

```
        cameras.back().R.at<double>(1,2) = r12;
        cameras.back().R.at<double>(2,0) = r20;
        cameras.back().R.at<double>(2,1) = r21;
        cameras.back().R.at<double>(2,2) = r22;
        cameras.back().focal = focal;
        cameras.back().radial[0] = radial1;
        cameras.back().radial[1] = radial2;
        // also add camera pose to poses point cloud:
        poses.push_back(PointXYZRGB(255,0,150));
        poses.back().x = cx;
        poses.back().y = cy;
        poses.back().z = cz;
        break;
      }
    case 2: // points
      if (point_count-- == 0) {
        state = 3;
        break;
      }

      // add new point to point cloud
      float x,y,z;
      int r,g,b;
      sline >> x >> y >> z;
      getline(file, line);
      sline.str(line);
      sline >> r >> g >> b;

      points.push_back(PointXYZRGB(r,g,b));
      points.back().x = x;
      points.back().y = y;
      points.back().z = z;

      // add visibility to visible
      visible.push_back(map<int,visibility>());
      int vis_count, frame, index;
      getline(file, line);
      sline.str(line);
      sline >> vis_count;
      while (vis_count-- > 0) {
        sline >> frame >> index >> x >> y;
        visible.back().insert(pair<int,visibility>(frame,visibility()));
        visible.back().find(frame)->second.index = index;
        visible.back().find(frame)->second.x = x;
        visible.back().find(frame)->second.y = y;
      }

      break;
    case 3: // end
      break;
    }
  }
  file.close();

  cout << poses.size() << " poses read!" << endl;
  cout << points.size() << " points read!" << endl;

  return true;
}

bool SfMReader::readPLY()
{
  /* output from: VisualSFM, Bundler
   * contains:
   * format:
   *  http://local.wasp.uwa.edu.au/~pbourke/dataformats/ply/
   *  0: <header>          (10+ lines)     list of columns of table
```

```
 *  1: <points pos+col>  (1 line each)   NO visibility (no dynamic list
     length)
 *                                       and NO camera poses
 */

ifstream file(path.c_str());
if (!file.is_open())
  return false;

int state = 0; // for states see comment above
string line;
while (file.good()) {
  getline(file, line);
  stringstream sline(line);
  string filename;
  switch (state) {
    case 0: // header
      if (line.find("format binary") == 0) {
        // binary file; use PCL reader and exit
        // (however, ascii files don't seem to work,
        //  so we parse them manually)
        cout << "Binary PLY file" << endl;
        file.close();
        PLYReader plyreader;
        plyreader.read(path, points);
        state = 2;
        break;
      }
      if (line.find("end_header") == 0) {
        state = 1;
      }
      break;
    case 1: // points
      if (line == "") {
        state = 2;
        break;
      }

      // add new point to point cloud
      float x,y,z;
      int r,g,b;
      // ! assuming X Y Z R G B [foo]
      sline >> x >> y >> z >> r >> g >> b;
      points.push_back(PointXYZRGB(r,g,b));
      points.back().x = x;
      points.back().y = y;
      points.back().z = z;
      break;
    case 2: // end
      break;
  }
}
if (file.is_open())
  file.close();

cout << points.size() << " points read!" << endl;

return true;

}

bool SfMReader::readTXT()
{
  /* output from: voodoo (probably)
   * contains:
   * format:
   *  (documented in the files itself)
   *  0: <header>          (39+ lines)    documentation of file format
```

```
 *  1: <cameras>         (2 lines each)
 *  2: <header>          (3 lines)
 *  3: <points pos>      (1 line each)   NO visibility, NO colour
 *  4: <empty lines>
 */

ifstream file(path.c_str());
if (!file.is_open())
  return false;

int state = 0; // for states see comment above
string line;
while (file.good()) {
  getline(file, line);
  stringstream sline(line);
  string filename;
  switch (state) {
    case 0: // header
      if (line.find("#timeindex") == 0) {
        state = 1;
      }
      break;
    case 1: // cameras
      if (line.find("# 3D") == 0) {
        state = 2;
        break;
      } else if (line.at(0) == '#') {
        break;
      }
      // new camera
      double cx, cy, cz;  // pos: position
      sline >> cx >> cy >> cz;
      // TODO: convert and use other variables
      // (re-projection for voodoo output currently not supported!)


      // add new camera to list of cameras
      /*
      cameras.push_back(visualization::Camera());
      cameras.back().pos[0] = cx;
      cameras.back().pos[1] = cy;
      cameras.back().pos[2] = cz;
      */
      // also add camera pose to poses point cloud:
      poses.push_back(PointXYZRGB(255,0,150));
      poses.back().x = cx;
      poses.back().y = cy;
      poses.back().z = cz;
      break;
    case 2: // header
      if (line.at(0) != '#') {
        state = 3;
      }
      break;
    case 3: // points
      if (line == "") {
        state = 4;
        break;
      }

      // add new point to point cloud
      float x,y,z;
      sline >> x >> y >> z;
      points.push_back(PointXYZRGB(150,150,150));
      points.back().x = x;
      points.back().y = y;
      points.back().z = z;
      break;
    case 4: // header
```

```
            break;
        }
      }
      file.close();

      cout << poses.size() << " poses read!" << endl;
      cout << points.size() << " points read!" << endl;

      return true;
    }

    bool SfMReader::readPCD()
    {
      /* output from: [PCL programs]
       * contains:
       * format:
       *  http://pointclouds.org/documentation/tutorials/pcd_file_format.php
       */

      // TODO: pcl::PCDReader?

      cout << "TODO: readPCD" << endl;
      return false;
    }

    /* this getter is used to prevent SfMReader from using
     * the slow SequenceCapture for calculating the window size
     * only, unless strictly necessary
     */
    void SfMReader::getWindowSize(Size& size)
    {
      if (this->window_size.width == 0) {
        // determine window size
        if (imagespath == "") {
          cerr << "[!!] Error: no image path specified for SfMReader::
              getWindowSize" << endl;
          return;
        }
        Mat frame;
        sc->setPosition(0);
        sc->read(frame);
        this->window_size = frame.size();
      }
      // set window size
      size = this->window_size;
    }

    int SfMReader::getImageID(string filename)
    {
      // remove directory names
      size_t found = filename.find_last_of("/\\");
      if (found != string::npos)
        filename = filename.substr(found+1);

      // search filename
      for (int i=0; i<image_filenames.size(); i++)
        if (image_filenames[i] == filename)
          return i;

      return -1;
    }

    bool SfMReader::integerLine(string line, int count)
    {
      int number = 0;
      int state = 1;
      for (string::iterator c = line.begin(); c != line.end(); c++) {
        switch (state) {
```

```
      case 1: // spaces
        if (isspace(*c)) {
          state = 1;
        } else if (isdigit(*c)) {
          state = 2;
          number++;
        } else {
          return false;
        }
        break;
      case 2: // digit
        if (isspace(*c)) {
          state = 1;
        } else if (isdigit(*c)) {
          state = 2;
        } else {
          return false;
        }
        break;
    }
  }

  if (number == count)
    return true;
  else
    return false;
}



bool SfMReader::selectPointsForCamera(int id,
                                      bool calcInvisible,
                                      Scalar colourCamera,
                                      Scalar colourSelectedCamera,
                                      Scalar colourVisiblePoint,
                                      Scalar colourInvisiblePoint)
{
  if (poses.size() < id)
    return false;

  // colour and save points visible from given camera
  resetPointColours();
  line_ends.clear();
  curr_visible_keypoints.clear();
  points_curr_visible.clear();
  points_curr_invisible.clear();
  Size window_size;
  getWindowSize(window_size);
  for (int p=0; p<visible.size(); p++) {
    points_curr_visible.push_back( visible.at(p).find(id) != visible.at(p).end
        () );
    if (points_curr_visible.back()) {
      points.at(p).r = colourVisiblePoint[2];
      points.at(p).g = colourVisiblePoint[1];
      points.at(p).b = colourVisiblePoint[0];
      line_ends.push_back(&points.at(p));
      curr_visible_keypoints.push_back(&visible.at(p).find(id)->second);
    }
    if (calcInvisible) {
      points_curr_invisible.push_back( !points_curr_visible[p] &&
          reprojectsInsideImage(p, id, window_size));
      if (points_curr_invisible.back()) {
        points.at(p).r = colourInvisiblePoint[2];
        points.at(p).g = colourInvisiblePoint[1];
        points.at(p).b = colourInvisiblePoint[0];
      }
    }
  }
```

```
    // colour all camera poses
    for (int i=0; i<poses.size(); i++) {
      poses.at(i).r = colourCamera[2];
      poses.at(i).g = colourCamera[1];
      poses.at(i).b = colourCamera[0];
    }

    // recolour given camera
    poses.at(id).r = colourSelectedCamera[2];
    poses.at(id).g = colourSelectedCamera[1];
    poses.at(id).b = colourSelectedCamera[0];
    // save pointer to given camera
    line_start = &poses.at(id);

    return true;
  }


bool SfMReader::selectCamerasForPoint(int id,
                                      Scalar colourCamera,
                                      Scalar colourSelectedCamera,
                                      Scalar colourSelectedPoint)
{
  if (points.size() < id)
    return false;

  // colour given point
  resetPointColours();
  points.at(id).r = colourSelectedPoint[2];
  points.at(id).g = colourSelectedPoint[1];
  points.at(id).b = colourSelectedPoint[0];
  // save pointer to given point
  line_start = &points.at(id);

  // colour all camera poses
  for (int i=0; i<poses.size(); i++) {
    poses.at(i).r = colourCamera[2];
    poses.at(i).g = colourCamera[1];
    poses.at(i).b = colourCamera[0];
  }

  // recolour and save poses for given point
  line_ends.clear();
  curr_visible_keypoints.clear();
  points_curr_visible.clear();
  points_curr_invisible.clear();
  map<int,visibility>* vismap = &visible.at(id);
  map<int,visibility>::iterator it;
  int frame;
  for (it = vismap->begin(); it!=vismap->end(); it++) {
    frame = (*it).first;
    poses.at(frame).r = colourSelectedCamera[2];
    poses.at(frame).g = colourSelectedCamera[1];
    poses.at(frame).b = colourSelectedCamera[0];
    line_ends.push_back(&poses.at(frame));
  }

  return true;
}


void SfMReader::resetPointColours()
{
  if (points_original.size() == 0) {
    // first time call: backup point cloud
    for (int i=0; i<points.size(); i++)
      points_original.push_back(points.at(i));
  } else {
```

```
    // reset points, get from point cloud backup
    // (this is, indeed, a lot of work)
    for (int i=0; i<points.size(); i++) {
      points.at(i).r = points_original.at(i).r;
      points.at(i).g = points_original.at(i).g;
      points.at(i).b = points_original.at(i).b;
    }
  }
}


void SfMReader::getExtrema(Scalar& min, Scalar& max)
{
  PointXYZRGB min_pt, max_pt;
  getMinMax3D(points, min_pt, max_pt);
  min[0] = min_pt.x;
  min[1] = min_pt.y;
  min[2] = min_pt.z;
  max[0] = max_pt.x;
  max[1] = max_pt.y;
  max[2] = max_pt.z;
}

/* reprojects point using camera parameters
 * note: this will return the undistorted location
 * (re-distort before displaying in an image)
 */
void SfMReader::reproject(PointXYZRGB* point, camera* cam, PointXYZRGB*
    projected, bool* front)
{
  Mat K = Mat::zeros(3, 3, CV_64FC1);
  K.at<double>(0,0) = cam->focal;
  K.at<double>(1,1) = cam->focal;
  K.at<double>(2,2) = 1;
  Mat v(3, 1, CV_64FC1);
  Mat x(3, 1, CV_64FC1);
  x.at<double>(0) = point->x;
  x.at<double>(1) = point->y;
  x.at<double>(2) = point->z;
  v = K * cam->R * (x - cam->t);
  projected->x = v.at<double>(0)/v.at<double>(2);
  projected->y = v.at<double>(1)/v.at<double>(2);
  projected->z = 0;
  projected->r = point->r;
  projected->g = point->g;
  projected->b = point->b;
  if (front != NULL)
    if (v.at<double>(2) < 0)
      *front = false;
    else
      *front = true;
}

bool SfMReader::reprojectsInsideImage(int pointID, int camID, Size size,
    PointXYZRGB* projected)
{
  if (pointID<0 || pointID>=points.size()
    || camID<0 || camID>=cameras.size()) {
    cerr << "[!!] Error: invalid pointID or camID (in SfMReader::
        reprojectsInsideImage)" << endl;
    return false;
  }

  return reprojectsInsideImage(&points.at(pointID),
                               &cameras.at(camID),
                               size,
                               projected);
}
```

```
bool SfMReader::reprojectsInsideImage(PointXYZRGB* point, camera* cam, Size
    size, PointXYZRGB* projected)
{
  PointXYZRGB projected2;
  if (!projected)
    projected = &projected2; // don't care about results, but have to save
        somewhere

  PointXYZRGB proj;
  bool front;
  reproject(point, cam, &proj, &front);
  bool ret = (proj.x >= size.width/-2.0
           && proj.x <= size.width/2.0
           && proj.y >= size.height/-2.0
           && proj.y <= size.height/2.0
           && front);
  if (ret) {
    projected->x = proj.x;
    projected->y = proj.y;
    projected->z = proj.z;
  }
  return ret;
}

/* Distort point x, y using one distortion parameter.
 * Code for this function is shamelessly copy-pasted from:
 * https://groups.google.com/forum/#!msg/vsfm/IcbdIVv_Uek/Us32SB
 */
void SfMReader::distortPointR1(PointXYZRGB* point, camera* cam) {
  const double k1 = cam->radial[0];

  if (k1 == 0)
    return;
  const double x = point->x / cam->focal;
  const double y = point->y / cam->focal;

  const double t2 = y*y;
  const double t3 = t2*t2*t2;
  const double t4 = x*x;
  const double t7 = k1*(t2+t4);
  if (k1 > 0) {
    const double t8 = 1.0/t7;
    const double t10 = t3/(t7*t7);
    const double t14 = sqrt(t10*(0.25+t8/27.0));
    const double t15 = t2*t8*y*0.5;
    const double t17 = pow(t14+t15,1.0/3.0);
    const double t18 = t17-t2*t8/(t17*3.0);
    point->x = cam->focal * (t18*x/y);
    point->y = cam->focal * t18;
  } else {
    const double t9 = t3/(t7*t7*4.0);
    const double t11 = t3/(t7*t7*t7*27.0);
    const std::complex<double> t12 = t9+t11;
    const std::complex<double> t13 = sqrt(t12);
    const double t14 = t2/t7;
    const double t15 = t14*y*0.5;
    const std::complex<double> t16 = t13+t15;
    const std::complex<double> t17 = pow(t16,1.0/3.0);
    const std::complex<double> t18 = (t17+t14/
(t17*3.0))*std::complex<double>(0.0,sqrt(3.0));
    const std::complex<double> t19 = -0.5*(t17+t18)+t14/(t17*6.0);
    point->x = cam->focal * (t19.real()*x/y);
    point->y = cam->focal * t19.real();
  }
}
```

```
void SfMReader::quaternion2matrix(Mat& q, Mat& R)
{
  double a = q.at<double>(0);
  double b = q.at<double>(1);
  double c = q.at<double>(2);
  double d = q.at<double>(3);
  // conversion based on the description on
  // http://en.wikipedia.org/wiki/Quaternions_and_spatial_rotation#
      Conversion_to_and_from_the_matrix_representation

  R.at<double>(0,0) = a*a + b*b - c*c - d*d;
  R.at<double>(1,0) = 2*b*c + 2*a*d;
  R.at<double>(2,0) = 2*b*d - 2*a*c;
  R.at<double>(0,1) = 2*b*c - 2*a*d;
  R.at<double>(1,1) = a*a - b*b + c*c - d*d;
  R.at<double>(2,1) = 2*c*d + 2*a*b;
  R.at<double>(0,2) = 2*b*d + 2*a*c;
  R.at<double>(1,2) = 2*c*d - 2*a*b;
  R.at<double>(2,2) = a*a - b*b - c*c + d*d;
}
```