Simulating the Hitchcock Effect with Kinect

Author(s):
Wirz, Daniel

Publication Date:
2011

Permanent Link:
https://doi.org/10.3929/ethz-a-006900785

Rights / License:
In Copyright - Non-Commercial Use Permitted
Simulating the Hitchcock Effect with Kinect

Semester Thesis

Daniel Wirz
Computer Vision Laboratory

Advisor: Nima Razavi, Alex Mansfield
Supervisor: Prof. Dr. Luc van Gool

September 9, 2011
Abstract

The Hitchcock Effect is a movie effect, which appears when a camera position along the camera axis and its focal length are synchronously changed, such that a focused region stays at a constant size. Because this effect is hard to realize, our aim is to build a system which creates a realistic simulation of this effect. In this work, we focus on small indoor scenes. We work with a consumer RGB-D camera to get a 3D representation of a scene. The RGB-D camera captures an RGB image as well as an aligned depth map which contains a 3D coordinate for each pixel. With the depth map and the color image we are able to compute new renderings of the scene with different camera parameters such as the focal length and the camera position.

Since the RGB-D camera works with structured-light scanning in infra red range, the resulting 3D model contains a large amount of regions, whose depths cannot be measured (holes). Therefore, we built a depth completion system to fill in these holes. The depth completion is based on a piece-wise planar assumption as well as the assumption, that higher order statistics of RGB images and corresponding depth maps correlate. We perform an over-segmentation of the RGB image into homogenous regions and greedily complete the missing depth data for each region by estimating the corresponding plane for each region. In our experiments with the Microsoft Kinect, we show that our approach completes depth maps accurately even for large holes. Our method is even able to extrapolate depth values to large distances. The completed depth maps then get rendered using a simple splatting approach with several optimizations to improve the resulting image quality. To create the Hitchcock Effect, the rendering uses different camera parameters, calculated automatically, such that a focused region remains at a constant size.
Acknowledgements

I would like to thank Nima Razavi and Alex Mansfield for their support and supervision during the project. They always gave me interesting inputs and new ideas for the realization of my work.
Contents

1 Introduction .................................................. 1
   1.1 Focus of this Work ........................................ 3
   1.2 Thesis Organization ....................................... 4

2 Related Work .................................................. 5

3 Materials and Methods ........................................ 9
   3.1 Data Acquisition ........................................... 9
   3.2 Depth Completion ........................................... 9
      3.2.1 Main Idea / Sketch of the Method .................. 12
      3.2.2 Pre-processing ......................................... 12
      3.2.3 Depth Completion ...................................... 12
      3.2.4 Optimizations for Depth Completion ................. 15
   3.3 Rendering .................................................. 15
      3.3.1 Optimizations for Rendering ......................... 17
   3.4 Creating the Hitchcock Effect ............................. 19

4 Results .......................................................... 21
   4.1 Depth Completion .......................................... 21
   4.2 Rendering .................................................. 25
   4.3 Simulation of Hitchcock Effect ............................ 26

5 Discussion ..................................................... 27
   5.1 Discussion .................................................. 27

6 Conclusion ..................................................... 29
   6.1 Conclusion .................................................. 29
   6.2 Outlook ..................................................... 30

A Appendix ......................................................... 31
   A.1 Results for different Datasets ............................. 31
      A.1.1 Dataset 1 - low complexity, small range ............. 32
      A.1.2 Dataset 2 - low complexity, middle range .......... 33
      A.1.3 Dataset 3 - low complexity, large range ............ 34
      A.1.4 Dataset 4 - middle complexity, small range ......... 35
      A.1.5 Dataset 5 - middle complexity, middle range ...... 36
A.1.6 Dataset 6 - middle complexity, large range ........................................... 37
A.1.7 Dataset 7 - high complexity, small range ............................................. 38
A.1.8 Dataset 8 - high complexity, middle range ......................................... 39
A.1.9 Dataset 9 - high complexity, large range ............................................. 40
A.1.10 Dataset 10 - non-planar model ......................................................... 41
A.2 Program Code ....................................................................................... 42
A.2.1 Depth completion ................................................................................. 42
A.2.2 Rendering .......................................................................................... 52
A.2.3 Simulation of the Hitchcock Effect ..................................................... 55
List of Figures

1.1 Three movie frames taken from Goodfellas, where the Hitchcock Effect was used. As we can see, the size of the background changes while the foreground keeps a constant size. 1

1.2 Illustration of changing the camera parameters. The left column shows the setting of the scene including the camera position and the focal length (illustrated as the field of view). In a) we see the camera at the original position, in b) the camera is moved towards an object and in c) the focal length is increased. As we can see, the different settings lead to different images where the proportions between the objects change. 2

3.1 An original data set from Kinect. (a) Shows the RGB image and (b) shows the corresponding, aligned depth map. The black regions are holes. 10

3.2 Linear interpolation. As can be seen, it fails for scenes with a large amount of holes. 11

3.3 Results of the pre-processing step. (a) Shows the clustered planes normals and (b) shows the superpixel segmentation. (Best viewed in color.) 13

3.4 Example for depth completion of a superpixel at iteration $i$. (a) Shows the superpixel to complete and its neighbors. The boundaries of the superpixels are shown in white. (b) Shows a schematic view of the segments. For the neighboring superpixels, the estimated planes are shown. The plane for the empty (black) superpixel is estimated based on the neighboring planes. As can be seen in (c), our method correctly estimates the plane orthogonal to the neighboring planes and completes the missing depth data. 14

3.5 The low resolution of the underlying depth map leads to black holes in the resulting image. 16

3.6 Visualization of rendering a splat. $a_{ij}$: weights of the square, based on gaussian weighting. 16

3.7 Example for iterating over splat sizes. (a) Shows a rendered image using a 9x9 splat. (b) Shows the resulting image rendered with a 7x7 splat and using the image (a) as initial image. (e) Shows the resulting image after all iterations. 18

3.8 Sketch of the software pipeline. The light gray boxes mark steps which only need to be performed when using a separate foreground image while the light blue boxes mark steps where user annotations are required. 19

4.1 Results of the linear interpolation. Small holes can be completet accurately, for large regions the method fails. 21

4.2 The relative errors of depth completion compared to our baseline [27]. In order to evaluate the depth at far distances, where we don’t have Kinect data, refer to the qualitative results 4.3 22
4.3 Qualitative evaluation of depth completion. The complexity of the scenes increases from top to bottom. In all examples, it can be seen that our baseline approach results in underestimations of the depths for far regions. Our method models the depths more accurately, even for the complex scenes in rows three and four. In row five, a failure case of our approach is shown. The frontal facing wall is not detected and therefore, the depth at those regions is overestimated. In the last row, we show an example for a non-planar surface. As can be seen, our planar model seems to not be a strong limitation due to the small size of the superpixels.

4.4 Illustration of the effect of changing the superpixel size. As can be seen, the modelling of the scene’s geometry is more accurate, when we use a large number of small superpixels.

4.5 Rendering of two scenes from three different camera positions: $P_0$ is the original camera position, $[0, 0, 0]$mm. $P_1 = [500, 500, 1000]$mm, $P_2 = [-500, 500, 1000]$mm. The $x$-axis belongs to the horizontal axis, the $y$-axis to the vertical one and the $z$-axis is the camera axis.

4.6 A set of movie frames to get a visualization of the effect. We show six frames, each rendered from a different camera position ($(0,0,0)$mm, $(0,0,200)$mm, $(0,0,400)$mm, $(0,0,600)$mm, $(0,0,800)$mm and $(0,0,1000)$mm) using focal lengths, such that the foreground object stays at a constant size.

A.1 (d) Shows the depth map of our baseline method [27], (f) shows the depth, completed by our approach. The model is completed accurately using our method, as well as using the baseline method.

A.2 (d) Shows the depth map of our baseline method [27], (f) shows the depth, completed by our approach. Both methods, the baseline and ours complete the depth accurately.

A.3 (d) Shows the depth map of our baseline method [27], (f) shows the depth, completed by our approach. The baseline method under-estimates the depths while our approach estimates the geometry of the scene plausibly.

A.4 (d) Shows the depth map of our baseline method [27], (f) shows the depth, completed by our approach. Both approaches estimate the scene geometry plausibly.

A.5 (d) Shows the depth map of our baseline method [27], (f) shows the depth, completed by our approach. The baseline method under-estimates the depth while our approach estimates the depth more plausibly.

A.6 (d) Shows the depth map of our baseline method [27], (f) shows the depth, completed by our approach. The baseline method under-estimates the depths while our approach over-estimates the depths because the frontal facing wall is not detected.

A.7 (d) Shows the depth map of our baseline method [27], (f) shows the depth, completed by our approach. The baseline method over-smoothes the depths while our approach estimates the geometry plausibly.

A.8 (d) Shows the depth map of our baseline method [27], (f) shows the depth, completed by our approach. The baseline method over-smoothes the depths while our approach estimates the geometry plausibly.

A.9 (d) Shows the depth map of our baseline method [27], (f) shows the depth, completed by our approach. Our method is able to estimate the geometry plausibly even for complex scenes.
A.10 (d) Shows the depth map of our baseline method [27], (f) shows the depth, completed by our approach. It can be seen that the planar assumption does not limit the completion strongly. . 41
Chapter 1

Introduction

In the year 1958, the movie Vertigo, directed by Alfred Hitchcock, was released. The main character of this movie suffers from vertigo. In order to visualize the effect of vertigo, Hitchcock introduced the so-called Hitchcock (or Vertigo)-Effect. This effect is based on a synchronous change of a camera’s position (dolly in or out) and its focal length (zoom out or in), such that a focused object stays at a constant size. An exact and time-synchronous realization of this effect leads the viewer to recognize a strange warping of a whole scene. Figure 1.1 illustrates this effect with three movie frames taken from Goodfellas (USA, 1990).

As we know, moving a camera to an object and simply zooming by changing the focal length does not have the same effect on the resulting image. If so, the two actions would neutralize each other when they are performed at the same speed, and no changes would be noticeable. But changing the camera’s focal length and its position does actually have different effects. Changing the focal length simply causes a resizing of an image without having an effect on the proportions of the image. Figure 1.2 shows the difference between changing the focal length and the camera position. If we now combine these two actions - moving away and zooming in - and perform them simultaneously, such that the size of the object image remains constant, we’ll recognize a strange effect: The background seems to come closer to us, while the focused object seems to move away, although it appears at a constant size. Since the human perception works with the evaluation of sizes as well as the ratios of distances, our brain gets confused because it is not used to changes of perspectives without changes in size. Therefore, this effect gives the observer a feeling of dizziness which can be used to visualize changes in a character’s psyche [45].
Figure 1.2: Illustration of changing the camera parameters. The left column shows the setting of the scene including the camera position and the focal length (illustrated as the field of view). In a) we see the camera at the original position, in b) the camera is moved towards an object and in c) the focal length is increased. As we can see, the different settings lead to different images where the proportions between the objects change.
1.1 Focus of this Work

Since we focus on small indoor scenes, we will use a consumer RGB-D camera to get a 3D model. The camera chosen is Microsoft’s Kinect which is widely available for consumers at a low price [35]. Using several software packages ([37], [38] and [39]), Kinect provides an RGB image and a calibrated depth map. Since Kinect’s depth map works with structured light scanning in infra-red (IR) range [9], the depth map contains regions which can’t be measured, otherwise known as holes. There are several reasons for the existence of holes: Regions at far distances cannot be scanned accurately because the components of Kinect’s depth camera (IR transmitter and scanner) are designed to capture data in very small environments, e.g. living rooms. The range is limited to about ten meters. Furthermore, glossy or transparent surfaces cannot be scanned either, because of reflections or interferences with other sources of light (e.g. lamps or the sun). Because depth maps are very important for many applications in cartography, computer vision, robotics and human-computer interaction, a lot of different depth acquisition approaches exist, eg. laser-scanning [5], time-of-flight measuring [7], or multi-view reconstruction [8]. But all these approaches suffer from not being able to construct fully complete depth maps without any holes. For our work, a complete depth map of a scene is required for calculating renderings without holes. Therefore, we need a system which takes an incomplete depth map as an input and returns the completed depth map. In our case, a depth map consists of the $y$, $x$ and $z$ coordinates of a point which corresponds to a pixel of an image. As complete depth representations of scenes are of general interest, a lot of research has been done on this subject, e.g. [27], [11], [18], [13]. The aim of these approaches is to get a complete depth map of a scene by filling in depth data for the holes. An other problem which is not addressed in these works and is not addressed in our work either, is the existence of holes caused by occlusion [18].

Some of these approaches mentioned above are based on the assumption, that the higher order statistics of an RGB image and the underlying depth map are correlated [21]. These methods perform a smooth filling of missing regions and work well for small holes. However, they do not consider any extrapolation and thus are not able to model the geometry of a scene plausibly. This leads to inaccurate estimations of depths at far distances and to problems completing large holes.

Inspired by [12], [13], [17] or [20], we propose to model the scene as a set of piece-wise planar regions. These algorithms all complete the missing depth value segment-wise with planar patches. As in [15] and [25], we also assume that intensity changes in the RGB images are correlated to changes in the depth map. We therefore do an over-segmentation of the RGB image into homogenous regions (superpixels) and greedily fill these regions by planar patches. Due to this assumptions, our method is able to extrapolate depths to far distances and estimates the geometry of a scene plausibly (see fig. 4.3).

After completing the depth map, renderings need to be done, using different camera parameters. We tried out several approaches based on simple camera projections [29] as well as on ray-tracing [31]. We then ended up using a forward splatting approach [33] including several optimizations, for example an approximation of $EWA - Splatting$ [34]. Since we use a low-quality consumer RGB-D camera, the resulting image quality is limited by the poor quality of input data. A simple rendering engine is implemented to create images of sufficient quality.
1.2 Thesis Organization

The thesis is organized as follows: In chapter 2 we review the related work. The following section will show the setup and the methods used to solve the problem. Section 4 includes results from applying the developed algorithms to several datasets. We will discuss the results in sec. 5 and make a conclusion and an outlook in sec. 6.
Chapter 2

Related Work

In the following paragraphs, important research is presented. The studies are related to the technical challenges this project poses. 3D scanning is important for the acquisition of depth data. We take a look at several different techniques. Further we have a look at some video and image processing tools in the second paragraph to get an impression of the approaches used to create similar effects as the Hitchcock Effect. The main part of this project is to get a complete depth map from recorded data which includes holes. Therefore we will review related work for depth completion. Another important issue is to render the completed depth map. We will discuss several rendering techniques in paragraph 4.

1. Scanning of 3D surfaces
   There are several different approaches to get a 3D model of a scene. As described in [5], laser scanning is based on reflections and interferences. Laser scanning allows to capture even large regions using Airborne laser scanning, described in [6]. A similar concept of depth measuring is used in time-of-flight cameras, described in [7]. Another approach to get a 3D representation of a scene is given by using multi-view stereo images. An evaluation of several multi-view stereo reconstruction algorithms is done in [8]. The hardware used for this work creates a depth map of a scene using structured light scanning. This technique is based on projecting patterns onto a scene and estimating the depths by analyzing scans of the scene [9].

2. Image and video processing
   In [2], a framework is presented for depth-based video editing. It includes a semi-automatic depth estimation without explicitly reconstructing 3D geometry models. [1] Describes a system which enables editing from different viewpoints, extracting and grouping of image-based objects and other functions. The editing is based on a user-assisted plane estimation of a single image. An algorithm for hole completion in videos is shown in [4]. A framework for user-annotated image-warping is presented in [3]. This enables the user to create visually plausible changes of perspective.
3. Depth completion  The aim of a depth completion algorithm basically is to fill in missing data in a depth map. In [10], missing depth data is completed by volumetric diffusion to achieve a water-tight manifold. The algorithm uses two volumes, the diffusion volume and the source volume. The diffusion volume is increased iteratively analogously to the heat equation until it overlaps with another volume. Two overlapping volumes then are composited to a closed surface. In order to prevent the resulting model from over-smoothing, [11] fills the holes based on the characteristics of the neighboring surfaces. Both approaches show good results for small holes but only have limited applications for our work, because we have to complete large holes with occluding surfaces. Therefore, invalid surface cells are filled with content of a valid surface cell that matches the surface approximation in and around the empty cell.

Missing depth data is also a problem in stereo reconstruction, where regions might be visible in only one of the views. In [18], the problem of occlusion is solved by segmenting images into a set of patches. The correspondence between segments is found to calculate the disparities assuming that an occluding edge represents a discontinuity in the depth map. For stereo algorithms, [25] provides completion for both depth and color images. For the depth completion, a segment-based filling algorithm is used. Similar to our work, the color image is segmented and for each segment a disparity plane is determined, using RANSAC [43] for the calculation of the plane parameters. [26] Proposes a way to synthesize range images based on interpolation of available range data using statistical relations between intensity changes in RGB images and corresponding depth variations from a video sequence. These relationships are then captured by the neighborhood system of a Markov Random Field to complete the holes. Thus, edge information is taken into account and the results are smoothed.

Similar to our work, other research has been done for completing depth based on local planar models. [12] describes a system to estimate a depth map for a single image, based on a local planar model. As in our work, a color image is segmented into a set of superpixels. For each superpixel, a set of features is computed to determine relationships between neighboring superpixels as well as boundaries in the 3D model. Based on these computations, the corresponding plane for each superpixel is calculated. [13] models a scene based on a set of planes and their disparities with stereo images. For fitting a plane into a segment, a robust solution is introduced, which applies a decomposition method to solve each parameter separately. Planar regions are also matched across multiple views to create planar models in [14]. Here, the Canny edge detection algorithm [46] is used to detect lines in several different views of the same scene. Based on the detected lines, half planes are computed to create a locally planar 3D model. An improvement of depth reconstruction is done by Gallup et al. [20]. In this work, an image was divided into planar and non-planar regions, based on multi-view images. The division is driven by multi-view photoconsistency as well as the result of a color and texture-based classifier, learned from hand-labeled planar and non-planar image regions. Using RANSAC and an initial depth map created from the multi-view images, the major planes as well as some spurious planes which fit well to non-planar regions, are found and fitted to the regions. An estimation of surface normals has been done by Tola et al [23]. The surface normals are estimated from the multi-view color images by analyzing the local textures and changes in spatial frequencies that orientation changes produce.
[17] introduces a system that also models a scene as a piecewise planar model for completing large 3D point clouds. Here, a set of geometric primitives is constructed which approximates input data, then adequately assembles them into a well-behaved surface. This system, however, does not consider extrapolation of missing regions. As in our method, [15] also uses RGB images to improve depth data. The focus of this work is set on increasing the resolution of the depth data. The main idea of this approach is to use color similarity as an indication of depth similarity.

However, these approaches all need high quality images for depth completion, and filling large holes or extrapolating depth is not considered in them. Our work, on the other hand, is able to complete the depth even for large holes and for images of poor quality.

4. Rendering Rendering is based on a simple camera projection as described in [29] and in [30]. The projection is just a multiplication of the coordinates of a 3D point and the camera’s projection matrix. The result then determines onto which image point the 3D point is projected. A more complex approach is provided by ray-tracing [31]. There, we follow a ray of light from the camera into a 3D model and check where it intersects the model. The intersection then determines which point we actually see. Ray tracing gives very good results and even allows us to add artificial light and shadows in the model. But since we have to follow each ray shot by the camera, the whole method works very slow [32]. A faster approach for rendering 3D volumes, known as splatting, is described in [33]. It is based on the projection approach which makes it faster than ray-tracing. But in contrast to simple projection rendering, points of a point cloud are projected to a spot instead of a single point. These spots may have shapes of, for example, squares or circles - Principled methods use Gaussians. An advanced implementation of a splatting approach, known as EWA Splatting, is described in [34]. This technique also includes use of point distances to create renderings, which consider the visibility of points.
Chapter 3

Materials and Methods

3.1 Data Acquisition

Since our software framework runs on MATLAB, Kinect has to be connected to a computer. To record data to a computer the OpenNI framework is used [36]. Together with the PrimeSense sensor Kit for Kinect [37] the framework is able to show and store Kinect data. The installation of the framework and the required drivers is described in an online tutorial [38]. To proceed Kinect data in MATLAB, we finally need to use the Kinect Matlab Tool provided on the Mathworks Website [39]. This software is a wrapper for OpenNI which allows us to read Kinect’s RGB images as well as a calibrated real-world-coordinate representation of its depth data into MATLAB.

We built a MATLAB script (getKinectData.m) to read Kinect data and store the RGB image and the 3D model into a .mat file. Data can either be read directly from Kinect or from a pre-recorded .oni file provided by the OpenNI framework. The script allows us to store two datasets of the same scene, one including an object in the foreground, the other just showing the background.

3.2 Depth Completion

Figure 3.1 shows an example of an original data set as Kinect outputs it. Subfigure (b) shows a depth map, calibrated to the RGB image shown in (a). For each pixel in the image, the depth map shows the corresponding Z-coordinate for the real-world-coordinate system which represents the distance to the camera in direction of the camera axis. As described in the introduction (sec. 3.1), the depth map contains a large amount of unknown depth data (marked black), otherwise known as holes. In order to fill these holes, different approaches have been tested (as described in sec. 3.2.1, 3.2.2 and 3.2.3) until we were able to formulate our own approach.
Figure 3.1: An original data set from Kinect. (a) Shows the RGB image and (b) shows the corresponding, aligned depth map. The black regions are holes.
**Linear interpolation approach** The first step, which was done to reconstruct the 3D model, was linear interpolation. For each missing data point the four next neighbours in each direction with valid data are searched. A simple interpolation then returns the depth value for the corresponding point. However, this works for very small holes only. Larger holes and missing data on the borders of the image can’t be - as expected - reconstructed accurately. For simple scenes with more or less complete depth data, this approach helped out to develop the rendering engine. As the scenes got more complex and the amount of missing depth data increased, it became useless. This is shown in fig. 3.2.

![Figure 3.2: Linear interpolation. As can be seen, it fails for scenes with a large amount of holes.](image)

**Manual reconstruction of planes** After discarding the linear interpolation approach we decided to reconstruct the model plane-wise, similarly to the approaches given e.g. in [12], [13], [14] or [20]. The first approach was a manual one which requires user interaction. The user can mark an arbitrary number of planes by drawing polygons onto the image. After marking 6 planes, the corresponding plane parameters are calculated using a least squares fit [41]. So, each polygon should include at least 3 valid points. If this isn’t possible (e.g. the wall on the end of the corridor in fig. 3.1), the regions have to be drawn such that they have overlapping regions. The polygons are proceeded in the same order as they are drawn. Therefore, regions without any depth data should be marked at last, overlapping with neighboring regions.

**Plane detection approach** To continue on the planar approach, we next implemented an automatic plane detection. How this plane detection works is described in section 3.2.2. In order to estimate each pixel’s depth we performed ray-tracing and intersect with the planes. The nearest intersection point then determined the depth value for the corresponding pixel. Although the resulting depth map is inaccurate, this approach constitutes an important part of the final completion algorithm which we will describe in the following section.
3.2.1 Main Idea / Sketch of the Method

For the depth completion we make a planar assumption ([12], [13], [14], [20]) and make use of the correlation between the given RGB image and the corresponding depth map ([25], [26], [15] or [21]). Therefore we segment the RGB image into a set of homogenous regions (so-called superpixels) and assume that the points within a superpixel all correspond to the same plane. Furthermore, we assume that neighboring planar regions either belong to the same plane or belong to intersecting planes. So, the boundaries of neighboring regions have to lie on the same plane. Based on these assumptions and the measured depth data for each region we finally determine the underlying plane and fill the holes. We assume all superpixels to be continuous and to consist of a single plane.

3.2.2 Pre-processing

Determination of existing planes  The raw data from the Kinect is noisy. For our approach it is important to have good estimations of the model’s main planes (e.g. ground plane, walls, ceiling). In order to get these planes, we first calculate the surface normals for each pixel with a known depth value [23]. This is done by fitting a plane to a small neighborhood around each pixel. The size of this neighborhood has an impact on the resulting surface normals: The larger we set the patch size, the smoother our model gets. A square 7x7 patch turned out to give good results. Based on the measured 3D points in the masked neighborhood, we calculate the corresponding surface normal using a least squares fit [41]. So, for each pixel whose depth is known, we also know the normal of the corresponding surface. Next, we cluster all these surface normals using k-means clustering [42]. We cluster into 10 clusters, using random initialization. For each cluster we calculate the corresponding plane parameters, using a least squares fit, again [41]. After these steps, we then have a set of 10 planes which occur in the model. The planes are given by their parameters \(a\), \(b\), \(c\) and \(d\). Note that we are clustering the plane normals. So, parallel planes are assigned to the same cluster.

Segmentation of RGB Image  The next step is to perform a segmentation of the RGB image ([26], [15]). The segmentation is done by Quickshift [24]. This returns a color-based segmentation of the RGB image, represented as a set of labelled superpixels. Note that the number and size of the superpixels have a strong influence on the completion. Using a large number of small superpixels for the image segmentation lets us create much more accurate depth completions. We will show that sec. 4.

The results of the pre-processing step are shown in fig. 3.3

3.2.3 Depth Completion

We propose an iterative greedy algorithm for depth completion. At each iteration, we pick a superpixel and complete its missing depth data by assigning a plane to this superpixel. There are superpixels for which we know all the depth data, for other superpixels we know the depth data only partly and there are also superpixels for which we don’t have any depth data at all. For our greedy optimization the order of superpixels is very important and has an impact on the results. We propose sorting the superpixels by their fraction of known depth data. At each step the remaining superpixel with the largest fraction will be
completed. The fraction $F$ of the measured depth data $D$ to the area of a superpixel is given by

$$F = \frac{|D|}{|\mathcal{R}|}, \quad D = \text{depth data in region } \mathcal{R}. \quad (3.1)$$

This ordering is a heuristic and offers no optimality guarantees. But, since we always complete depth of a superpixel and its boundaries, it intuitively can be seen that this ordering propagates information from segments with more information to those with less. Not just completing the superpixel’s depth, but also its boundaries is done in order to be able to estimate planes which have no depth data at all and to ensure consistency between segments.

For each superpixel picked, based on the known depth data, we decide whether:

1. The region belongs to one of the clustered planes, or
2. The region belongs to a plane which lies parallel to one of the calculated planes, or
3. The region belongs to another plane, which has to be calculated using a least squares fit.

For each proposal we recalculate the plane from the known data and get the squared error $E$.

$$E = \lambda \sum_{x \in \mathcal{R}} ||x - \tilde{x}||^2_2 \quad (3.2)$$

Where $\lambda$: Constant prior for weighting the squared error; $x$: original coordinates of the points in region $\mathcal{R}$; $\tilde{x}$: recalculation of $x$
For each proposal a different number of degrees of freedom (DOF) has to be estimated: One DOF for proposal a), two DOFs for b) and three for c). In order to regularize against over-fitting, we set a uniform prior $\lambda$ over the types of proposals by multiplying the calculated error by a constant for proposals with more degrees of freedom. We set $\lambda = 1.3$ for proposal 2 and $\lambda = 1.7$ for proposal 3. We chose the proposal which creates the smallest weighted error $E$. Based on this decision we get the parameters of the plane which the superpixel belongs to. By means of the estimated plane we are able to construct the missing depth data for the considered superpixel. Note that in this procedure, we can still estimate a plane for the region even if the pre-processed planes were inaccurate. Figure 3.4 shows an example completion of the depth for a superpixel. For the estimation of the plane in proposal 3, we also tried RANSAC [43]. But the least squares estimation was more reliable, because of the local discontinuities caused by the noise of the data. The least squares estimation calculates the plane with respect to all data in a region, while RANSAC only uses the inliers to calculate the plane.

Figure 3.4: Example for depth completion of a superpixel at iteration $i$. (a) Shows the superpixel to complete and its neighbors. The boundaries of the superpixels are shown in white. (b) Shows a schematic view of the segments. For the neighboring superpixels, the estimated planes are shown. The plane for the empty (black) superpixel is estimated based on the neighboring planes. As can be seen in (c), our method correctly estimates the plane orthogonal to the neighboring planes and completes the missing depth data.
3.2.4 Optimizations for Depth Completion

In order to improve the results, we did some optimizations on the algorithm:

- **Recalculation of superpixel order**: Each iteration affects the number of known depth data and thus, the fraction of known depth data per superpixel. So, we make use of this and recalculate the order after iteration.

- **Recalculate depth map for a number of iterations**: After all the superpixels have been completed, we restore the original depth data from Kinect and repeat the completion algorithm for a number of iterations (4 in our experiments). This smoothes the resulting depth map and prevents us from having strong discontinuities in the 3D data.

3.3 Rendering

The first rendering engine implementation was based on the simple projection approach, described in [29] and [30]: A position vector $x$ of a 3D point is multiplied by a projection matrix:

$$
\rho \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = K R^t (x - c) = \begin{bmatrix} f \cdot k_x & f \cdot s & x_0 \\ 0 & f \cdot k_y & y_0 \\ 0 & 0 & 1 \end{bmatrix} R^t (x - c)
$$

(3.3)

Where $k_x, k_y$: width and height of a pixel; $x_0, y_0$: the pixel coordinates of the principal point; $f$: the camera’s focal length; $s$: skew coefficient; $R$ rotation matrix of the camera; $c$: camera position.

The image’s nearest point to $x$ and $y$ then gets colored with the corresponding RGB value. Because of the low resolution of the underlying depth map, the resulting images often contain holes when changing the camera parameters drastically. This issue is shown in fig. 3.5. In order to avoid these holes, a rendering engine based on ray tracing [31] was implemented. But the point cloud representation of our 3D model leaded to problems using such a rendering approach. One of the problems was the visibility of occluded points. The main issue, however, was the performance - the rendering was too slow.

In order to avoid these problems, we implemented a forward splatting algorithm [33]. Using forward splatting, each 3D point is projected to a splat instead of a single point. In our case, this blob is a square including gaussian weighting (see fig. 3.6). So, the point to be projected is multiplied by the projection matrix, as described in (3.3). The resulting 2D point $[x, y]$ determines the middle of the gaussian weighted square with respect to the image coordinates. The coloring of the pixels within the square is done by the gaussian-weighting-based averaging of the point’s color and the already existing color of the pixels. We use a black image as an initial image to project the splats to. This approach is straight forward and, very important, it is fast. In order to improve the results, we did some optimizations.
Figure 3.5: The low resolution of the underlying depth map leads to black holes in the resulting image.

\[
\text{Color(new)} = (1 - a_{ij}) \text{Color(old)} + a_{ij} \text{Color(point)}
\]

Figure 3.6: Visualization of rendering a splat. $a_{ij}$: weights of the square, based on gaussian weighting.
3.3.1 Optimizations for Rendering

- **Depth-depended order of projection**: First, we did a sorting of the points based on their depths. So, points which have a larger depth and thus lie further away are projected first. The nearest points are projected last.

- **Approximation of EWA splatting [34]**: EWA (elliptical weighted average) splatting is a complex splatting technique for point-based surface data. It contains a method for dealing with pixel visibilities which we included in our rendering engine. It works as follows: For each rendered pixel we store the points distance into a so-called z-buffer. When the splat of a rendered 3D point is projected to a pixel rendered from another 3D point, we compare the distance of the 3D point with the value stored in the z-buffer. If both values are very close to each other, we merge the splats by averaging their RGB values. The value in the z-buffer is replaced by the average of the point distance and the stored value. Otherwise, if the new splat comes from a point much closer, we replace the RGB and z-buffer values by the corresponding values from the new splat.

- **Iteration over splat sizes**: We have a trade-off when using different splat sizes: When we use small splats, we get sharp images with a high grade of details. But, depending on our projection parameters, the image may contain large holes. On the other hand, using large splats results in unsharp, diffuse images with smaller holes. In order to get the best result for this issue, we use several different splat sizes. This means, that we first use a large splat to project the 3D model. Next, we decrease the splat size and perform another projection using the resulting image from the projection using the large splat as an intial image. These steps are repeated until we reach a splat size of just one single pixel. Using this technique, we achieve sharp images with small holes. Figure 3.7 shows an example for this.
Figure 3.7: Example for iterating over splat sizes. (a) Shows a rendered image using a 9x9 splat. (b) Shows the resulting image rendered with a 7x7 splat and using the image (a) as initial image. (c) Shows the resulting image after all iterations.
CHAPTER 3. MATERIALS AND METHODS

3.4 Creating the Hitchcock Effect

With the tools described in the previous section, we now have everything we need to do a simulation of the Hitchcock Effect. The effect can be simulated using two different settings: One including a foreground object and one without any foreground objects. A foreground can for example be a person. If we choose the version with a foreground object, we need two images of the scene, one showing the scene without the object (basically the background) and the other including the object (foreground).

The steps taken are illustrated as a software pipeline in fig. 3.8.

1. If no foreground model has to be proceeded, mark the region to stay at constant size. This has to be done manually by the user. Otherwise, if there is a foreground model, the object’s size and distance can be calculated automatically, without requiring user input. This is done by a foreground-background-segmentation, based on background subtraction. We, therefore, subtract the (uncompleted) background depths $Z_{BG}$ (only the z-coordinates) from the foreground depths $Z_{FG}$. The absolute values of these differences, which exceed a threshold $T$, are then defined as foreground regions $R_{FG}$:

$$R_{FG} = |Z_{FG} - Z_{BG}| \geq T$$

(3.4)

Since the data is noisy, we perform a region analysis and define the largest coherent region as the foreground object. This is done by MATLAB’s built-in function `bwconncomp` [40]. The user is able to verify the automatical selection.

2. Set camera movement. This is, again, done by the user. The user can set the start, the end and the duration (in frames) of the move.

3. Based on (1), determine the distance and the real-world-size of the marked region.

4. When a foreground and background image is used, compute a histogram equation of both images. This is necessary because Kinect automatically sets the brightness of an image.

Figure 3.8: Sketch of the software pipeline. The light gray boxes mark steps which only need to be performed when using a separate foreground image while the light blue boxes mark steps where user annotations are required.
5. For each frame perform these steps:

- Read camera position based on user input from (2).
- Calculate the camera’s focal length such that the projection of the marked region (1) stays at a constant size.
- Render background image.
- If needed, render foreground image, using the background image as initialization image.

After performing these five steps, we get a low quality video file (MPEG) and a set of single images for each movie frame, in order to create a high quality video.
Chapter 4

Results

4.1 Depth Completion

The first depth completion method we used is based on simple linear interpolation. Obviously this method does not yield satisfying results, as can be seen in fig 4.1. It can be seen that small holes can be completed accurately. But for large holes, the method fails and the method is not able to extrapolate missing depth data.

Figure 4.1: Results of the linear interpolation. Small holes can be completet accurately, for large regions the method fails.
For an evaluation of our depth completion algorithm, we recorded the RGB data and the calibrated depth maps for several scenes. The datasets have all been taken in indoor environments and consist of various complexities. In 4.3, a selection of these scenes is shown together with their depth data provided by Kinect (columns (a) and (b)).

To get a quantitative evaluation of our algorithm for small holes, we use a similar technique as described in [26]: We remove square patches from the original data and apply the completion on these. We then compare the resulting 3D model with the original one and calculate the relative errors for the chosen patches. The patches were chosen randomly and we tried out various edge lengths: 5, 11, 27, 69 and 111. As a baseline for the evaluation we used the method described in [27], which interpolates the missing depth using Delaunay triangulation. As we can see in fig. 4.2, the errors of our methods are similar to those of the baseline method. Note that this evaluation method only shows the relative errors for regions that can be measured by Kinect. It does not evaluate the results at far, transparent or reflective surfaces. We therefore show the resulting depth maps for the selection in column (e) of fig. 4.3. Have a look at the appendix for the full set of results (sec. A.1). Additionally, the underlying segmentation can be seen for each set.

As described in sec. 3.2.2, the size and the number of the superpixels have a strong impact on the quality of the depth completion. Figure 4.4 shows an example of this issue, where we can see that the scene model is modelled more accurately, using a large number of small superpixels.
Figure 4.3: Qualitative evaluation of depth completion. The complexity of the scenes increases from top to bottom. In all examples, it can be seen that our baseline approach results in underestimations of the depths for far regions. Our method models the depths more accurately, even for the complex scenes in rows three and four. In row five, a failure case of our approach is shown. The frontal facing wall is not detected and therefore, the depth at those regions is overestimated. In the last row, we show an example for a non-planar surface. As can be seen, our planar model seems to not be a strong limitation due to the small size of the superpixels.
Figure 4.4: Illustration of the effect of changing the superpixel size. As can be seen, the modelling of the scene’s geometry is more accurate, when we use a large number of small superpixels.
4.2 Rendering

In order to evaluate the rendering engine, we show a small set of renderings from two scenes, each having different projection parameters. Basically we change the camera position. As an initial image we use a black image. For each scene we use the same parameters. As can be seen in fig. 4.5, there are some holes in the images. In order to achieve a better visualization of the effect, we added a movie clip to the supplementary material.

Figure 4.5: Rendering of two scenes from three different camera positions: $P_0$ is the original camera position, $[0, 0, 0]$mm. $P_1 = [500, 500, 1000]$mm, $P_2 = [-500, 500, 1000]$mm. The $x$-axis belongs to the horizontal axis, the $y$-axis to the vertical one and the $z$-axis is the camera axis.
4.3 Simulation of Hitchcock Effect

Analogously to sec. 4.2, we show a set of rendered images to visualize the simulation. Note that there are black regions on the top border of the images. In order to achieve a better visualization of the effect, we added a movie clip to the supplementary material.

![Image of movie frames](image)

Figure 4.6: A set of movie frames to get a visualization of the effect. We show six frames, each rendered from a different camera position \((0, 0, 0)\)mm, \((0, 0, 200)\)mm, \((0, 0, 400)\)mm, \((0, 0, 600)\)mm, \((0, 0, 800)\)mm and \((0, 0, 1000)\)mm) using focal lengths, such that the foreground object stays at a constant size.
Chapter 5

Discussion

5.1 Discussion

Depth Completion  Similarly to the methods described in [12], [13], [14], [17] or [20], we also work with a piecewise planar assumption. As in [23], we also do an estimation of local surface models to determine the existing planes in the given model. Based on these estimated planes and the given depth data, we complete the model region-wise, where we on the one hand assume that neighboring regions relate to each other [26] and, on the other hand, that the RGB image correlates to the underlying 3D model [15], [25]. It can be seen that the assumptions made do not limit the model strongly (fig. 4.3, column six) and even spherical surfaces may be completed in a plausible way, due to the large number of small superpixels.

As in [10] and [11] our approach is able to fill small regions of missing depth data accurately. For those regions we got similar errors to those occured using our baseline method (fig. 4.2), which is based on linear interpolation using Delaunay triangulation [27]. But our approach is also able to complete larger holes. In contrast to the introduced depth completion algorithms, our approach is even able to extrapolate missing 3D data to create a more plausible estimation of a scene’s geometry. Especially for simple scenes, which do not include objects lying in the foreground, the resulting 3D models are accurate, in the sense that existing planes are detected and completed correctly. Columns three and four in fig. 4.3 show that even complex scenes can be completed in a more or less plausible way, while the baseline approach always underestimates the depths. Note that our method works in a fully automated way and even for inputs of very poor quality.

For models which only contain a small amount of given 3D data, our method fails and might overestimate the depths (fig. 4.2, column five). Another factor, which may lead our approach to failure, is the existence of objects in the model. Such objects might be, for example, chairs, tables or persons. Our method is built to avoid strong discontinuities between neighboring superpixels by fitting planar patches based on the regions boundaries. However, if the real-world model actually does contain discontinuities between neighboring regions, our approach results in an over-smoothing and, therefore, estimates the depth values wrongly. In order to avoid this, a classifier might be implemented in future work to classify each superpixel. According to this classification, relationships between neighboring superpixels can be found and used to improve the results.
CHAPTER 5. DISCUSSION

Rendering  As we can see in sec. 4.2, our renderings often contain black holes. Furthermore the quality of the rendered images is quite low. The main cause for these issues is the poor quality of the input data. The depth data recorded by Kinect is very noisy. Since the depth acquisition is based on structured light scanning [9], shadows may occur around objects. Our completion approach performs a smoothing of such regions, and therefore, an object does not necessarily consist of sharp boundaries. Because of this, there are often noisy borders in our renderings.

Another issue, which has an impact on the quality of the rendered images, is given by the point-cloud representation of a 3D model. The 3D data is recorded from one single camera position. Therefore, the scene may include regions, which are not visible because they are covered by other objects, e.g. tables, chairs, columns. When the camera position is changed for rendering the scene from a different viewpoint, such covered regions then might become visible. This causes the rendered picture to contain holes. Another reason why there are holes in the renderings is the low resolution of depth data.

Under these circumstances, an implementation of a more extravagant rendering approach, e.g. EWA Splatting [34] would just be an overkill. In order to increase the resulting image quality, the hardware should be changed.
Chapter 6

Conclusion

6.1 Conclusion

**Depth Completion** We provide a simple, fully automated method for a depth completion of RGB-D images taken by consumer cameras. Our approach is based on a planar assumption, so the image can be segmented into homogenous planar regions. The regions (i.e. superpixels) get filled greedily one after the other, where the order of filling has an important effect on the results. With our method it is possible to complete simple scenes accurately. The planar assumptions we made do not limit the method strongly, so even the depths of non-planar regions are estimated plausibly.

Our approach fails for scenes for which we just have a very small amount of given depth data. In this kind of scenes, our approach leads to an overestimation of the depth data. Since our method is built to avoid strong discontinuities, the completion is limited for scenes which actually do contain discontinuities, such as several objects which are occluding each other.

Our algorithm is relatively slow. The calculation and clustering of the plane normals as well as the over-segmentation of the image and the superpixel-wise estimation of the depth data are very costly, especially in our current MATLAB implementation.

**Rendering** Under the given circumstances of the low input quality, our simple rendering approach creates images with a sufficient quality. The hardware used has a stronger effect on the resulting image quality than the rendering engine used. The basic limitations of the rendering are given by the representation of the depth map as a point-cloud as well as the low quality of the RGB image and the low resolution of the depth data.

**Simulation of the Hitchcock Effect** With the provided depth completion and rendering techniques, the Hitchcock Effect can be simulated trivially. Especially in case of simple scenes, the optical effect looks very realistic. The main limitations we have are based on the low quality and resolution of Kinect cameras and on the point-cloud representation of the 3D model.
6.2 Outlook

Depth completion Although our heuristic works well, for future work it would be interesting to formulate a global model for our problem. Further improvements might be done by restricting the superpixel boundaries to lines, which should improve the results on Manhattan scenes [14]. A general improvement might be reached by implementing a comparison for neighboring superpixels. This way, superpixels are classified according to different features, such as for example the texture or spatial frequency. With these classifications, realationships between superpixels can be analyzed and it might be determined whether two neighboring superpixels are lying on the same plane or not. This would decrease the effect of over-smoothing between neighboring regions which do not lie on the same plane.

The current MATLAB implementantion of the algorithm is quite slow, because of the image segmentation and the iterative filling procedure. In future work, a faster implementation of the method on GPU should be realized, in order to make the system work in real-time. Having a real-time implementation would also allow us to not just include one single image for the completion, but a whole RGB-D video stream, which would improve the results.

Rendering In order to improve the resulting images, other cameras could be used. Having better cameras, it would be worth improving the rendering engine. This could be done by a full implementation of the EWA splatting algorithm [34] or by an implementation of a ray-tracing-based rendering approach [31], combined with a triangulation of the depth map. In future work, the implementation of the rendering should also be optimized with respect to the speed, such that a real-time rendering would be possible.

Simulation of Hitchcock Effect In order to create a more realistic simulation of the Hitchcock Effect, the described improvements for the depth completion and rendering enginges should be done. Since the main limitation is given by the hardware, a better RGB camera could be used in further works. The high quality RGB image then could be aligned to Kinect’s depth data. For a better alignment, the 3D models need to be upsamled to the RGB image size, which can be done using the approach given in [15].
Appendix A

Appendix

A.1 Results for different Datasets

On the following pages we show the results of ten sets of experiments. The scenes of the experiments are ordered with respect to their ranges and their complexities.
A.1.1 Dataset 1 - low complexity, small range

Figure A.1: (d) Shows the depth map of our baseline method [27], (f) shows the depth, completed by our approach. The model is completed accurately using our method, as well as using the baseline method.
A.1.2 Dataset 2 - low complexity, middle range

(a) RGB image

(b) Original depth map

(c) Clustering of the estimated planes

(d) Smoothed depth

(e) Color segmentation (1122 superpixels)

(f) Completed depth

Figure A.2: (d) Shows the depth map of our baseline method [27], (f) shows the depth, completed by our approach. Both methods, the baseline and ours complete the depth accurately.
A.1.3 Dataset 3 - low complexity, large range

Figure A.3: (d) Shows the depth map of our baseline method [27], (f) shows the depth, completed by our approach. The baseline method under-estimates the depths while our approach estimates the geometry of the scene plausibly.
A.1.4 Dataset 4 - middle complexity, small range

Figure A.4: (d) Shows the depth map of our baseline method [27], (f) shows the depth, completed by our approach. Both approaches estimate the scene geometry plausibly.
A.1.5 Dataset 5 - middle complexity, middle range

Figure A.5: (d) Shows the depth map of our baseline method [27], (f) shows the depth, completed by our approach. The baseline method under-estimates the depth while our approach estimates the depth more plausibly.
A.1.6 Dataset 6 - middle complexity, large range

Figure A.6: (d) Shows the depth map of our baseline method [27], (f) shows the depth, completed by our approach. The baseline method under-estimates the depths while our approach over-estimates the depths because the frontal facing wall is not detected.
A.1.7 Dataset 7 - high complexity, small range

Figure A.7: (d) Shows the depth map of our baseline method [27], (f) shows the depth, completed by our approach. The baseline method over-smoothes the depths while our approach estimates the geometry plausibly.
A.1.8 Dataset 8 - high complexity, middle range

Figure A.8: (d) Shows the depth map of our baseline method [27], (f) shows the depth, completed by our approach. The baseline method over-smoothes the depths while our approach estimates the geometry plausibly.
A.1.9 Dataset 9 - high complexity, large range

Figure A.9: (d) Shows the depth map of our baseline method [27], (f) shows the depth, completed by our approach. Our method is able to estimate the geometry plausibly even for complex scenes.
A.1.10 Dataset 10 - non-planar model

Figure A.10: (d) Shows the depth map of our baseline method [27], (f) shows the depth, completed by our approach. It can be seen that the planar assumption does not limit the completion strongly.
APPENDIX A. APPENDIX

A.2 Program Code

A.2.1 Depth completion

*completeModel.m*

```matlab
function [coords3DOut,
    numOfLS,
    numOfNormals,
    numOfParams,
    numOfSP,
    allocationOfModes,
    origDataRecon] = completeModel(coords3D,
                                  imgRGB,
                                  KK0,
                                  superpixelSize,
                                  numOfIterations,
                                  costLS,
                                  costNormals,
                                  costParams)

% ---------------------------------------------------------------------
% getRelativeErrors.m Complete a depth map
% % Input
% % coords3D Input depth map
% % imgRGB Align RGB image
% % KK0 Camera intristics
% % superpixelSize Used for quikshift segmentation. Determines size
% and number of superpixels
% % numOfIterations Number of recalculation of the depth
% % costLS, Costs for each of the proposals
% % costNormals
% % costParams
% %
% % Output
% % coords3DOut Completed depth map
% % numOfLS, How many times each proposal was used
% % numOfNormals
% % numOfParams
% % numOfSP Number of Superpixels
% % allocationOfModes Which superpixel uses which proposal
```

42
% origDataRecon Recalculation of the original depth map
% ---------------------------------------------------------------------

% Setup
coordsX = coords3D(:,:,1);
coordsY = coords3D(:,:,2);
coordsZ = coords3D(:,:,3);
coordsX(coordsZ==0)=nan;
coordsY(coordsZ==0)=nan;
coordsZ(coordsZ==0)=nan;

sizeOfModel = size(coordsX);
width = sizeOfModel(2);
height = sizeOfModel(1);

% Create image coordinates
[yIP xIP] = ind2sub([480 640], 1:width*height);

numOfClusters = 10;
segmentBorder = 3;

numOfLS = 0;
numOfNormals = 0;
numOfParams = 0;

% Create surface normals
sizeOfPatch = 7; % odd number
planeNorms = NaN(480, 640, 3);
for j=find(~isnan(coordsZ))'
    xx = ceil(j / 480);
    yy = mod(j,480);
    xPatch = xx-floor(sizeOfPatch/2):xx+floor(sizeOfPatch/2);
    yPatch = yy-floor(sizeOfPatch/2):yy+floor(sizeOfPatch/2);
    xPatch = xPatch(xPatch>0 & xPatch<width);
    yPatch = yPatch(yPatch>0 & yPatch<height);
    A = [reshape(coordsX(yPatch, xPatch), length(xPatch)*length(yPatch), 1),
         reshape(coordsY(yPatch, xPatch), length(xPatch)*length(yPatch), 1),
         reshape(coordsZ(yPatch, xPatch), length(xPatch)*length(yPatch), 1)];
    A = A(~isnan(A(:,1)),:);
    if min(size(A))≥3
        [U S V]=svd(A);
        S_inv = S;
        S_inv(S_inv#0) = S_inv(S_inv#0).^(-1);
        S_inv=S_inv';
        coeffs = V*S_inv*U'*ones(length(A),1)*1000;
        planeNorms(yy,xx,:) = coeffs./norm(coeffs);
    end
end

% Cluster surface normals
A
PPENDIX

% Get superpixels
maxdist = superpixelSize;
ratio = 0.5;
kernelsize = 2;
[ISEG LABELS MAPS GAPS E] = vl_quickseg(imgRGB, ratio, kernelsize, maxdist);
umOfSP = max(max(LABELS));

% Sort LABELS by their numbers of valid co"ds compared with their sizes
validCoordRatio = zeros(max(max(LABELS)), 1);
validCoordRatio2 = validCoordRatio;
for i=1:max(max(LABELS))
    segment = (LABELS == i);
    % Include borders of segment, too,
    segmentWithBorder = imdilate(segment,strel('square',segmentBorder));
    validCoordRatio(i) = length(find(coordsZ(segmentWithBorder)>0))
    /length(find(segmentWithBorder));
end
[validCoordRatio, segmentQueue] = sort(validCoordRatio, 'descend');
segmentQueueInit = segmentQueue;
segmentQueueIterations = [];

% For each cluster, calculate the corresponding plane equation
planeParams = [];
planeParams2 = [];
for i = 1:max(max(planeNormsClustered))
    segment = (planeNormsClustered == i);
    segmentValid = (coordsZ.*segment)>0;
    validPointsInSegment = [coordsX(segmentValid)';
                            coordsY(segmentValid)';
                            coordsZ(segmentValid)'];
    try
        [x0, N, d, normd] = lsplane(validPointsInSegment);
        N(4) = - (x0(1)*N(1) + x0(2)*N(2) + x0(3)*N(3));
        N = N*sqrt(N(1)^2 + N(2)^2 + N(3)^2);
        planeParams(i,:) = N';
    catch exception
        exception(i,:) = N';
    end
end
% Get mean of normals of cluster
a = N(1);
b = N(2);
c = N(3);

planeParams2(i,:) = [a, b, c];
end

cordsReconX = coordsX;
cordsReconY = coordsY;
cordsReconZ = coordsZ;

counter = 0;
allocationOfModesThisIteration = zeros(height, width);
origDataReconX = zeros(height, width);
origDataReconY = zeros(height, width);
origDataReconZ = zeros(height, width);
while length(segmentQueue)>0
  % Create segment
  counter = counter+1;
  segment = (LABELS == segmentQueue(1));
  % Include borders of segment, too
  segmentWithBorder = imdilate(segment,strel('square',segmentBorder));

  segmentValid = (coordsReconZ.*segmentWithBorder)>0;
  segmentInvalid = (segment & isnan(coordsReconZ));

  validPointsInSegment = [coordsReconX(segmentValid)';
                          coordsReconY(segmentValid)';
                          coordsReconZ(segmentValid)'];
  % Fill Segment using LS fit
  try
    [x0, N, d, normd] = lsplane(validPointsInSegment);
    d;
    normd;
    N(4) = - (x0(1).*N(1) + x0(2).*N(2) + x0(3).*N(3));
    N = N*sqrt(N(1)^2 + N(2)^2 + N(3)^2);
  catch
    % Fall-back code
    d = 0;
    normd = 0;
    N(4) = 0;
    N = N*sqrt(N(1)^2 + N(2)^2 + N(3)^2);
  end
  %...
APPENDIX A. APPENDIX

\[ a_{LSFit} = N(1); \]
\[ b_{LSFit} = N(2); \]
\[ c_{LSFit} = N(3); \]
\[ d_{LSFit} = N(4); \]

\[
t = - d_{LSFit} \times \text{ones}(\text{length(find(segmentValid))),1} ./ \left( a_{LSFit} \times \text{Pvalid(:,1)} + b_{LSFit} \times \text{Pvalid(:,2)} + c_{LSFit} \times \text{Pvalid(:,3)} \right);
\]

\[ \text{Pvalid}_{\text{recon}} = [\text{Pvalid(:,1)}.\times t, \text{Pvalid(:,2)}.\times t, \text{Pvalid(:,3)}.\times t]; \]

\[ \text{errors}_{\text{LSFit}} = \sum((\text{validPointsInSegment(:,1)} - \text{Pvalid}_{\text{recon}}(:,1)).^2 + (\text{validPointsInSegment(:,2)} - \text{Pvalid}_{\text{recon}}(:,2)).^2 + (\text{validPointsInSegment(:,3)} - \text{Pvalid}_{\text{recon}}(:,3)).^2); \]

\[
\text{catch} \ \text{exception}
\]
\[
\text{errors}_{\text{LSFit}} = \infty;
\]
\[
\text{end}
\]

\[
\text{if} \ \text{length(find(t } \geq 0)) > 0
\]
\[
\text{errors}_{\text{LSFit}} = \infty;
\]
\[
\text{end}
\]

\[
\text{%% Fill segment using plane params}
\]
\[
\text{errors}_{\text{ParamsFit}} = \infty;
\]
\[
\text{for} \ j=1:\text{size(planeParams,1)}
\]
\[
\text{t} = - \text{planeParams}(j,4) \times \text{ones}(\text{length(find(segmentValid))),1} ./ \left( \text{planeParams}(j,1) \times \text{Pvalid(:,1)} + \text{planeParams}(j,2) \times \text{Pvalid(:,2)} + \text{planeParams}(j,3) \times \text{Pvalid(:,3)} \right);
\]

\[ \text{Pvalid}_{\text{recon}} = [\text{Pvalid(:,1)}.\times t, \text{Pvalid(:,2)}.\times t, \text{Pvalid(:,3)}.\times t]; \]

\[ \text{error} = \sum((\text{validPointsInSegment(:,1)} - \text{Pvalid}_{\text{recon}}(:,1)).^2 + (\text{validPointsInSegment(:,2)} - \text{Pvalid}_{\text{recon}}(:,2)).^2 + (\text{validPointsInSegment(:,3)} - \text{Pvalid}_{\text{recon}}(:,3)).^2); \]

\[ \text{if} \ \text{error} < \text{errors}_{\text{ParamsFit}}
\]
\[
\text{Pvalid}_{\text{recon}}_{\text{params}} = \text{Pvalid}_{\text{recon}};
\]
\[ \text{errors}_{\text{ParamsFit}} = \text{error}; \]
\[ a_{\text{ParamsFit}} = \text{planeParams}(j,1); \]
\[ b_{\text{ParamsFit}} = \text{planeParams}(j,2); \]
\[ c_{\text{ParamsFit}} = \text{planeParams}(j,3); \]
\[ d_{\text{ParamsFit}} = \text{planeParams}(j,4); \]
\[ \text{end} \]
\[
\text{%% Fill segment using surface Normals}
\]
\[
\text{errors}_{\text{NormalsFit}} = \infty;
\]
\[
\text{for} \ j=1:\text{size(planeParams2,1)}
\]
\[
\text{d} = \text{mean(}-\left( \text{planeParams2}(j,1) \times \text{validPointsInSegment(:,1)} + \text{planeParams2}(j,2) \times \text{validPointsInSegment(:,2)} + \text{planeParams2}(j,3) \times \text{validPointsInSegment(:,3)} \right));
\]
\[
\text{t} = - \text{d} \times \text{ones}(\text{length(find(segmentValid))),1} ./ \left( \text{planeParams2}(j,1) \times \text{Pvalid(:,1)} + \right.
\]
\[
\left( \text{planeParams2}(j,2) \times \text{Pvalid(:,2)} + \right. \text{planeParams2}(j,3) \times \text{Pvalid(:,3)}); \]
planeParams2(j,2)*Pvalid(:,2) + 
planeParams2(j,3)*Pvalid(:,3)); 

Pvalid_recon = [Pvalid(:,1).*t, Pvalid(:,2).*t, Pvalid(:,3).*t]; 

error = sum((validPointsInSegment(:,1)-Pvalid_recon(:,1)).^2 + 
(validPointsInSegment(:,2)-Pvalid_recon(:,2)).^2 + 
(validPointsInSegment(:,3)-Pvalid_recon(:,3)).^2); 

if error < errorsNormalsFit 
    Pvalid_recon_normals = Pvalid_recon; 
    errorsNormalsFit = error; 
    aNormalsFit = planeParams2(j,1); 
    bNormalsFit = planeParams2(j,2); 
    cNormalsFit = planeParams2(j,3); 
    dNormalsFit = d; 
end 
end 

%% Chose method with least error 
t = 1; 
reconValid = true; 
while length(find(t>0))>0 
    if errorsParamsFit == inf && errorsNormalsFit==inf && errorsLSFit==inf 
        reconValid = false; 
        break; 
    end 
    switch min(find([errorsParamsFit*costParams, 
                      errorsNormalsFit*costNormals, 
                      errorsLSFit*costLS] == min([errorsParamsFit*costParams, 
                                                  errorsNormalsFit*costNormals, 
                                                  errorsLSFit*costLS]))) 
    case 1 
        t = -dParamsFit*ones(length(find(segmentInvalid)),1) ./
            (aParamsFit*Pinv(:,1) + 
             bParamsFit*Pinv(:,2) + 
             cParamsFit*Pinv(:,3)); 
        t2= -dParamsFit*ones(length(find(segmentValid)),1) ./
            (aParamsFit*Pvalid(:,1) + 
             bParamsFit*Pvalid(:,2) + 
             cParamsFit*Pvalid(:,3)); 
        method = 'Params'; 
        errorsParamsFit = inf; 
    case 2 
        t = -dNormalsFit*ones(length(find(segmentInvalid)),1) ./
            (aNormalsFit*Pinv(:,1) + 
             bNormalsFit*Pinv(:,2) + 
             cNormalsFit*Pinv(:,3)); 
        t2= -dNormalsFit*ones(length(find(segmentValid)),1) ./
            (aNormalsFit*Pvalid(:,1) + 
             bNormalsFit*Pvalid(:,2) + 
             cNormalsFit*Pvalid(:,3)); 
        method = 'Normals'; 
        errorsNormalsFit = inf; 
    case 3 
        t = -dLSFit*ones(length(find(segmentInvalid)),1) ./
            (aLSFit*Pinv(:,1) + 
             bLSFit*Pinv(:,2) + 
             cLSFit*Pinv(:,3)); 
        t2= -dLSFit*ones(length(find(segmentValid)),1) ./
            (aLSFit*Pvalid(:,1) + 
             bLSFit*Pvalid(:,2) + 
             cLSFit*Pvalid(:,3)); 
        method = 'LS'; 
        errorsLSFit = inf; 
    end 
end
APPENDIX A. APPENDIX

\[
t_2 = - \frac{d_{\text{NormalsFit}} \cdot \text{ones} \left( \text{length} \left( \text{find} \left( \text{segmentValid} \right) \right), 1 \right)}{(a_{\text{NormalsFit}} \cdot \text{Pvalid}(:,1) + b_{\text{NormalsFit}} \cdot \text{Pvalid}(:,2) + c_{\text{NormalsFit}} \cdot \text{Pvalid}(:,3))};
\]

```
t2 = - dLSFit*ones(length(find(segmentValid)),1) ./(aLSFit*Pvalid(:,1) + bLSFit*Pvalid(:,2) + cLSFit*Pvalid(:,3));
```

```
method = 'Normals';
errorsNormalsFit = inf;
case 3
```
```
t = - dLSFit*ones(length(find(segmentInvalid)),1) ./(aLSFit*Pinv(:,1) + bLSFit*Pinv(:,2) + cLSFit*Pinv(:,3));
```
```
t2= - dLSFit*ones(length(find(segmentValid)),1) ./(aLSFit*Pvalid(:,1) + bLSFit*Pvalid(:,2) + cLSFit*Pvalid(:,3));
```
```
method = 'LS';
errorsLSFit = inf;
end
end
if reconValid
```
switch method
```
case 'Params'
```
umOfParams = numOfParams+1;
```
```
mode = 1;
```
case 'Normals'
```
umOfNormals = numOfNormals+1;
```
```
mode = 2;
```
case 'LS'
```
umOfLS = numOfLS+1;
```
```
mode = 3;
```
end
```
P_inv = [Pinv(:,1).*t, Pinv(:,2).*t, Pinv(:,3).*t];
```
```
coordsReconX(segmentInvalid) = P_inv(:,1);
```
```
coordsReconY(segmentInvalid) = P_inv(:,2);
```
```
coordsReconZ(segmentInvalid) = P_inv(:,3);
```
```
allocationOfModesThisIteration(segment) = mode;
```
```
Pvalid_recon = [Pvalid(:,1).*t2, Pvalid(:,2).*t2, Pvalid(:,3).*t2];
```
```
origDataReconX(segmentValid) = Pvalid_recon(:,1);
```
```
origDataReconY(segmentValid) = Pvalid_recon(:,2);
```
```
origDataReconZ(segmentValid) = Pvalid_recon(:,3);
```
end

% Delete segment from queue
```
segmentQueueIterations = [segmentQueue(1);segmentQueueIterations];
```
```
segmentQueue(1) = [];
```
```
validCoordRatio(1) = [];
```
```
% Resort queue
for j=neighbourSegments'
```
segment = (LABELS == j);
% Include borders of segment, too,
segmentWithBorder = imdilate(segment, strel('square', segmentBorder));
validCoordRatio{segmentQueue == j} = length(find(coordsReconZ(segmentWithBorder)>0))/
length(find(segmentWithBorder));
end
[validCoordRatio, segmentQueue2] = sort(validCoordRatio, 'descend');
segmentQueue = segmentQueue(segmentQueue2);
end
allocationOfModes(:,:,1) = allocationOfModesThisIteration;
origDataRecon(:,:,1) = origDataReconX;
origDataRecon(:,:,2) = origDataReconY;
origDataRecon(:,:,3) = origDataReconZ;

%% Do recalculations
for numOfIteration = 1:numOfIterations
  segmentQueueRecalc = segmentQueueIterations;
  allocationOfModesThisIteration = zeros(height, width);
  numOfParams(numOfIteration + 1) = 0;
  numOfLS(numOfIteration + 1) = 0;
  numOfNormals(numOfIteration + 1) = 0;
  while length(segmentQueueRecalc)>0
    % Create segment
    counter = counter+1;
    segment = (LABELS == segmentQueueRecalc(1));
    % Include borders of segment, too
    segmentWithBorder = imdilate(segment, strel('square', segmentBorder));
    segmentValid = (segmentWithBorder & ~segment);
    segmentInvalid = (segment & isnan(coordsZ));
    validPointsInSegment = [coordsReconX(segmentValid)';
        coordsReconY(segmentValid)';
        coordsReconZ(segmentValid)'];
    xxx = KK0(1,3)*2/width * xIP(segmentInvalid) - KK0(1,3);
    yyy = KK0(2,3)*2/height * yIP(segmentInvalid) - KK0(2,3);
    Pinv = [xxx', yyy', -KK0(1)*ones(length(find(segmentInvalid)),1)];
    xxx = KK0(1,3)*2/width * xIP(segmentValid) - KK0(1,3);
    yyy = KK0(2,3)*2/height * yIP(segmentValid) - KK0(2,3);
    Pvalid = [xxx', yyy', -KK0(1)*ones(length(find(segmentValid)),1)];
    try
      [x0, N, d, normd] = lsplane(validPointsInSegment);
      d;
      normd;
  end
  end
  allocationOfModes(:,:,numOfIteration + 1) = allocationOfModesThisIteration;
  origDataRecon(:,:,numOfIteration + 1) = origDataReconX;
  origDataRecon(:,:,numOfIteration + 1) = origDataReconY;
  origDataRecon(:,:,numOfIteration + 1) = origDataReconZ;
end
\( N(4) = -(x0(1) \cdot N(1) + x0(2) \cdot N(2) + x0(3) \cdot N(3)); \)
\( N = N \cdot \sqrt{N(1)^2 + N(2)^2 + N(3)^2}; \)
\( a_{LSFit} = N(1); \)
\( b_{LSFit} = N(2); \)
\( c_{LSFit} = N(3); \)
\( d_{LSFit} = N(4); \)

\[ t = - d_{LSFit} \cdot \text{ones}(\text{length}(\text{find}(\text{segmentValid})),1) \div \left( a_{LSFit} \cdot \text{Pvalid}(:,1) + b_{LSFit} \cdot \text{Pvalid}(:,2) + c_{LSFit} \cdot \text{Pvalid}(:,3) \right); \]
\( \text{Pvalid}_{\text{recon}} = [\text{Pvalid}(:,1) \cdot t, \text{Pvalid}(:,2) \cdot t, \text{Pvalid}(:,3) \cdot t]; \)
\( \text{errors}_{\text{LSFit}} = \text{sum}\left( (\text{validPointsInSegment}(:,1) - \text{Pvalid}_{\text{recon}}(:,1))^2 + (\text{validPointsInSegment}(:,2) - \text{Pvalid}_{\text{recon}}(:,2))^2 + (\text{validPointsInSegment}(:,3) - \text{Pvalid}_{\text{recon}}(:,3))^2 \right); \)

\( \text{catch} \ \text{exception} \)
\( \text{errors}_{\text{LSFit}} = \infty; \)
\( \text{end} \)
\( \text{if} \ \text{length}(\text{find}(t \geq 0)) > 0 \)
\( \text{errors}_{\text{LSFit}} = \infty; \)
\( \text{end} \)

\( \text{%% Fill segment using plane params} \)
\( \text{errors}_{\text{ParamsFit}} = \infty; \)
\( \text{for} \ j=1:\text{size(\text{planeParams},1)} \)
\( \quad t = - \text{planeParams}(j,4) \cdot \text{ones}(\text{length}(\text{find}(\text{segmentValid})),1) \div \left( \text{planeParams}(j,1) \cdot \text{Pvalid}(:,1) + \text{planeParams}(j,2) \cdot \text{Pvalid}(:,2) + \text{planeParams}(j,3) \cdot \text{Pvalid}(:,3) \right); \)
\( \quad \text{Pvalid}_{\text{recon}} = [\text{Pvalid}(:,1) \cdot t, \text{Pvalid}(:,2) \cdot t, \text{Pvalid}(:,3) \cdot t]; \)
\( \quad \text{error} = \text{sum}\left( (\text{validPointsInSegment}(:,1) - \text{Pvalid}_{\text{recon}}(:,1))^2 + (\text{validPointsInSegment}(:,2) - \text{Pvalid}_{\text{recon}}(:,2))^2 + (\text{validPointsInSegment}(:,3) - \text{Pvalid}_{\text{recon}}(:,3))^2 \right); \)
\( \text{if} \ \text{error} < \text{errors}_{\text{ParamsFit}} \)
\( \quad \text{errors}_{\text{ParamsFit}} = \text{error}; \)
\( \quad a_{\text{ParamsFit}} = \text{planeParams}(j,1); \)
\( \quad b_{\text{ParamsFit}} = \text{planeParams}(j,2); \)
\( \quad c_{\text{ParamsFit}} = \text{planeParams}(j,3); \)
\( \quad d_{\text{ParamsFit}} = \text{planeParams}(j,4); \)
\( \text{end} \)
\( \text{end} \)

\( \text{%% Fill segment using surface Normals} \)
\( \text{errors}_{\text{NormalsFit}} = \infty; \)
\( \text{for} \ j=1:\text{size(\text{planeParams2},1)} \)
\( \quad d = \text{mean}(- (\text{planeParams2}(j,1) \cdot \text{validPointsInSegment}(:,1) + \text{planeParams2}(j,2) \cdot \text{validPointsInSegment}(:,2) + \text{planeParams2}(j,3) \cdot \text{validPointsInSegment}(:,3))); \)
\( \quad t = - d \cdot \text{ones}(\text{length}(\text{find}(\text{segmentValid})),1) \div \left( \text{planeParams2}(j,1) \cdot \text{Pvalid}(:,1) + \right. \)
\( \left. \text{planeParams2}(j,2) \cdot \text{Pvalid}(:,2) + \text{planeParams2}(j,3) \cdot \text{Pvalid}(:,3) \right); \)
\( \quad \text{Pvalid}_{\text{recon}} = [\text{Pvalid}(:,1) \cdot t, \text{Pvalid}(:,2) \cdot t, \text{Pvalid}(:,3) \cdot t]; \)
\( \quad \text{error} = \text{sum}\left( (\text{validPointsInSegment}(:,1) - \text{Pvalid}_{\text{recon}}(:,1))^2 + (\text{validPointsInSegment}(:,2) - \text{Pvalid}_{\text{recon}}(:,2))^2 + (\text{validPointsInSegment}(:,3) - \text{Pvalid}_{\text{recon}}(:,3))^2 \right); \)
\( \text{if} \ \text{error} < \text{errors}_{\text{NormalsFit}} \)
\( \quad \text{errors}_{\text{NormalsFit}} = \text{error}; \)
\( \quad a_{\text{NormalsFit}} = \text{planeParams2}(j,1); \)
\( \quad b_{\text{NormalsFit}} = \text{planeParams2}(j,2); \)
\( \quad c_{\text{NormalsFit}} = \text{planeParams2}(j,3); \)
\( \quad d_{\text{NormalsFit}} = \text{planeParams2}(j,4); \)
\( \text{end} \)
\( \text{end} \)
planeParams2(j,2)*Pvalid(:,2) +
planeParams2(j,3)*Pvalid(:,3));

P_valid_recon = [Pvalid(:,1).*t, Pvalid(:,2).*t, Pvalid(:,3).*t];

error = sum((validPointsInSegment(:,1)-P_valid_recon(:,1)).^2 +
(validPointsInSegment(:,2)-P_valid_recon(:,2)).^2 +
(validPointsInSegment(:,3)-P_valid_recon(:,3)).^2);

if error < errorsNormalsFit
    errorsNormalsFit = error;
    aNormalsFit = planeParams2(j,1);
    bNormalsFit = planeParams2(j,2);
    cNormalsFit = planeParams2(j,3);
    dNormalsFit = d;
end
end

%% Chose method with least error
switch min(find([errorsParamsFit*costParams,
                 errorsNormalsFit*costNormals,
                 errorsLSFit*costLS] == min([errorsParamsFit*costParams,
                                             errorsNormalsFit*costNormals,
                                             errorsLSFit*costLS])))
    case 1
        t = - dParamsFit*ones(length(find(segmentInvalid)),1) ./
            (aParamsFit*Pinv(:,1) +
            bParamsFit*Pinv(:,2) +
            cParamsFit*Pinv(:,3));
        method = 'Params';
        mode = 1;
        numOfParams(numOfIteration + 1) = numOfParams(numOfIteration + 1) + 1;
    case 2
        t = - dNormalsFit*ones(length(find(segmentInvalid)),1) ./
            (aNormalsFit*Pinv(:,1) +
            bNormalsFit*Pinv(:,2) +
            cNormalsFit*Pinv(:,3));
        method = 'Normals';
        mode = 2;
        numOfNormals(numOfIteration + 1) = numOfNormals(numOfIteration + 1) + 1;
    case 3
        t = - dLSFit*ones(length(find(segmentInvalid)),1) ./
            d(aLSFit*Pinv(:,1) + bLSFit*Pinv(:,2) + cLSFit*Pinv(:,3));
        method = 'LS';
        mode = 3;
        numOfLS(numOfIteration + 1) = numOfLS(numOfIteration + 1) + 1;
end
P_inv = [Pinv(:,1).*t, Pinv(:,2).*t, Pinv(:,3).*t];
coordsReconX(segmentInvalid) = P_inv(:,1);
coordsReconY(segmentInvalid) = P_inv(:,2);
coordsReconZ(segmentInvalid) = P_inv(:,3);
allocationOfModesThisIteration(segment) = mode;
A.2.2 Rendering

renderImage.m

function [imgRendered] = renderImage(imgOrig, coordsOrig, projectionMatrix, maxPatchSize, imgInit)

% renderImage.m Renders a pointcloud to an image
% Input
% imgOrig, Color values for each 3D point
% coordsOrig, Depth map
% projectionMatrix Projectionmatrix which has to be used for rendering
% maxPatchSize Maximum splat size. This is decreased after each step
% imgInit Initial image
% Output
% imgRendered rendered image

% Setup
sizeOfModel = size(imgOrig);
width = sizeOfModel(2);
height = sizeOfModel(1);

imgOrigR = double(imgOrig(:,:,1));
imgOrigG = double(imgOrig(:,:,2));
imgOrigB = double(imgOrig(:,:,3));
\[
\begin{align*}
X &= \text{coordsOrig}(::,1); \\
Y &= \text{coordsOrig}(::,2); \\
Z &= \text{coordsOrig}(::,3);
\end{align*}
\]

\[
X(Z==0)=\text{NaN}; \\
Y(Z==0)=\text{NaN}; \\
Z(Z==0)=\text{NaN};
\]

%% Get set of holes and its image coordinates
imagePoints = (1:width*height)';
distances = Z(imagePoints);
[distances, imagePoints] = sort(distances,'descend');

%% Project each 3D point to image
imgRenderedR = flipud(fliplr(imgInit(:,:,1)));
imgRenderedG = flipud(fliplr(imgInit(:,:,2)));
imgRenderedB = flipud(fliplr(imgInit(:,:,3)));

%% Iterate over splat sizes
for patchSize = [maxPatchSize:-1:0]
    zBuffer = nan(height, width);
    if patchSize >0
        weightMask = fspecial('gaussian', 2*patchSize+1,patchSize/3);
        weightMask = weightMask/max(max(weightMask));
    else
        weightMask = 1;
    end
    threshold = 10;
    for i=1:length(imagePoints)
        ind = imagePoints(i);
        coordsImg = projectionMatrix*[X(ind);Y(ind);Z(ind);1];
        if coordsImg(3)>0
            coordsImg = coordsImg./coordsImg(3);
        end
        coordsImg = int16(round(coordsImg));
        xImg = coordsImg(1);
        yImg = coordsImg(2);
        try
            for xx = xImg-patchSize : xImg+patchSize
                for yy = yImg-patchSize : yImg+patchSize
                    if isnan(zBuffer(yy,xx))
                        xxMask = xx - (xImg-patchSize)+1;
                        yyMask = yy - (yImg-patchSize)+1;
                        imgRenderedR(yy, xx) = imgOrigR(ind)*weightMask(yyMask, xxMask) + 
                        imgRenderedR(yy, xx) * 
                        (1-weightMask(yyMask, xxMask));
                        imgRenderedG(yy, xx) = imgOrigG(ind)*weightMask(yyMask, xxMask) + 
                        imgRenderedG(yy, xx) * 
                    end
                end
            end
        end
    end
end
APPENDIX A. APPENDIX

\( \text{imgRenderedB}(yy, xx) = \text{imgOrigB}(ind) \times \text{weightMask}(yyMask, xxMask) + \\text{imgRenderedB}(yy, xx) \times (1-\text{weightMask}(yyMask, xxMask)) \)

\( \text{zBuffer}(yy, xx) = \text{distances}(i) \)

\textbf{elseif distances}(i) < \text{zBuffer}(yy, xx) - \text{threshold}

\begin{align*}
\text{xxMask} &= xx - (xImg - \text{patchSize}) + 1; \\
\text{yyMask} &= yy - (yImg - \text{patchSize}) + 1; \\
\text{imgRenderedR}(yy, xx) &= \text{imgOrigR}(ind) \times \text{weightMask}(yyMask, xxMask) + \\
&\quad \text{imgRenderedR}(yy, xx) \times (1-\text{weightMask}(yyMask, xxMask)) \\
\text{imgRenderedG}(yy, xx) &= \text{imgOrigG}(ind) \times \text{weightMask}(yyMask, xxMask) + \\
&\quad \text{imgRenderedG}(yy, xx) \times (1-\text{weightMask}(yyMask, xxMask)) \\
\text{imgRenderedB}(yy, xx) &= \text{imgOrigB}(ind) \times \text{weightMask}(yyMask, xxMask) + \\
&\quad \text{imgRenderedB}(yy, xx) \times (1-\text{weightMask}(yyMask, xxMask)) \\
\text{zBuffer}(yy, xx) &= \text{distances}(i) \\
\end{align*}

\textbf{elseif distances}(i) \geq \text{zBuffer}(yy, xx) - \text{threshold}

\begin{align*}
\&\& \text{distances}(i) < \text{zBuffer}(yy, xx) + \text{threshold}
\text{xxMask} &= xx - (xImg - \text{patchSize}) + 1; \\
\text{yyMask} &= yy - (yImg - \text{patchSize}) + 1; \\
\text{imgRenderedR}(yy, xx) &= (\text{imgOrigR}(ind) + \\text{imgRenderedR}(yy, xx) \times (1-\text{weightMask}(yyMask, xxMask))))/2; \\
\text{imgRenderedG}(yy, xx) &= (\text{imgOrigG}(ind) + \\text{imgRenderedG}(yy, xx) \times (1-\text{weightMask}(yyMask, xxMask))))/2; \\
\text{imgRenderedB}(yy, xx) &= (\text{imgOrigB}(ind) + \\text{imgRenderedB}(yy, xx) \times (1-\text{weightMask}(yyMask, xxMask))))/2; \\
\text{zBuffer}(yy, xx) &= (\text{zBuffer}(yy, xx) + \text{distances}(i))/2; \\
\end{align*}

\textbf{catch exception}

\textbf{end}

\textbf{end}

\textbf{end}

catch exception

end
A.2.3 Simulation of the Hitchcock Effect

*startHitchcock.m*

```matlab
%% Setup
clc
clear all
addpath(genpath('Include'))
disp('Setting up rendering...')

% Get file to proceed.
[filename, pathname] = uigetfile('.mat','Load Kinect Data');
scene = load(fullfile(pathname, filename));

% Load foreground
try
    imgRGB_FG = scene.foreground.RGB;
    coords3D_FG = scene.foreground.Coords;
    hasForeground = true;
catch exception
    hasForeground = false;
end

% Load background
imgRGB_BG = scene.background.RGB;
coords3D_BG = scene.background.Coords;

% Set constants
KK0 = [578.3650 0 320;
      0 573.7689 240;
      0 0 1.0000];

sizeOfMovieClip = size(imgRGB_BG);
width = sizeOfMovieClip(2);
height = sizeOfMovieClip(1);

%% Get completed background model
try
```
% If a completed depth map has been stored -> read it
coords3D_BG = scene.background.CoordsRecon;
catch exception
% Else -> create a new one
[coords3D_BG, numOfLS, numOfNormals,
 numOfParams, numOfSP, allocationOfModes,
 modelReconstructedComplete] = completeModel(coords3D_BG, imgRGB_BG,
                                              KK0, 10, 4, 1.7, 1.3, 1);

foreground = scene.foreground;
background = scene.background;
background.CoordsRecon = coords3D_BG;
save([pathname filename], 'foreground', 'background')
scene.background.CoordsRecon = coords3D_BG;
end

%% Equalize foreground and background color
if hasForeground
    [imgRGB_FG, imgRGB_BG] = equalizeFGandBG(imgRGB_FG,
                                              imgRGB_BG,
                                              coords3D_FG,
                                              scene.background.Coords);
end

%% Get measurements
if hasForeground
    mask = getSegmentationMap(scene.background.Coords,
                               coords3D_FG,
                               scene.background.RGB,
                               scene.foreground.RGB);
    imgGetRegionR = imgRGB_FG(:,:,1);
    imgGetRegionR(mask==1)=255;
    imgGetRegion(:,:,1) = imgGetRegionR;
    imgGetRegion(:,:,2) = imgRGB_FG(:,:,2);
    imgGetRegion(:,:,3) = imgRGB_FG(:,:,3);
else
    imgGetRegion = imgRGB_BG;
    mask=zeros(height,width);
end

maskOK = false;

while ~maskOK
    try
        disp('Set a new region to be focused by clicking on the
             image. Press enter to set the finish the marking.')
        imshow(imgGetRegion)
        hold on
        polyg = [];
        numOfPoints = 0;
        [x,y] = ginput(1);
while length(x>0)
    polyg = [polyg; [x y]];
    plot(x, y, '.r', 'MarkerSize', 6)
    if size(polyg,1) > 1
        line([x; polyg(end-1,1)], [y; polyg(end-1,2)])
    end
    [x,y] = ginput(1);
end

catch exception
end

if size(polyg,1) > 2
    line([polyg(1,1); polyg(end,1)], [polyg(1,2); polyg(end,2)])
    mask = poly2mask(polyg(:,1), polyg(:,2), height, width);
end

if length(find(mask==1))>1
    maskOK = true;
end

if hasForeground
    depth = coords3D_FG(:,:,3);
else
    depth = coords3D_BG(:,:,3);
end

Y0 = mean(depth(mask==1));
% Image coordinates
[yIP xIP] = ind2sub([height width], (1:width*height)');
objSizeImg = abs(min(yIP(find(mask==1)))-max(yIP(find(mask==1))));
objSizeReal = objSizeImg*Y0/KK0(1);
objWidthImg = abs(min(xIP(find(mask==1)))-max(xIP(find(mask==1))));
objWidthReal = objWidthImg*Y0/KK0(2,2);

R = eye(3);

% Setup movie params
numOfFrames = 2;
numOfFrameRepeats = 1;

fig1=figure(1);
winsize = get(fig1,'Position');
winsize(1:2) = [0 0];
numOfMovieFrames = 1;
A=moviein(numOfFrames,fig1,winsize);
set(fig1,'NextPlot','replacechildren')
%% Create Output Dir
dir = ['Data/Export/', strrep(filename, '.mat', '')];
mkdir(dir);

%% Set camera positions
cameraPositions = [];
getMoreMoves = true;
while getMoreMoves
    startPoint = input('Start point of move ');
    endPoint = input('End point of move ');
    duration = input('Duration of move (frames) ');
    disp('---------------------------------------------------')
    if ~isempty(startPoint) && ~isempty(endPoint) && ~isempty(duration)
        cameraPositions = [cameraPositions, linspace(startPoint, endPoint, duration)];
    end
    if ~isempty(cameraPositions) && isempty(startPoint)
        getMoreMoves = false;
    end
end

%% Calculate each frame
for i=1:length(cameraPositions)
    % Change camera center
    P = [0;0;cameraPositions(i)];

    focalLength_1 = (Y0-P(3))*objSizeImg/(objSizeReal);
    focalLength_2 = (Y0-P(3))*objWidthImg/(objWidthReal);

    KK = [focalLength_1 0 320; 0 focalLength_2 240; 0 0 1.0000];

    projectionMatrix = KK*[R', -R'*P];
    disp(['Focal length: ', num2str(KK(1)), ' Position: ', num2str(P(3))])

    % Render foreground and background
    disp('Render foreground and background...')
    imgBG = renderImage(imgRGB_BG, coords3D_BG, projectionMatrix, 4, zeros(480,640,3));
    if hasForeground
        imgRendered = renderImage(imgRGB_FG, coords3D_FG, projectionMatrix, 1, imgBG);
    else
        imgRendered = imgBG;
    end

    % Create final image of background, foreground and MAP
disp('Create image...')
imgRendered = uint8(imgRendered);
imshow(imgRendered)
imwrite(imgRendered,[dir, '/movieframe_',num2str(i),'.jpg'] , 'jpg','Quality', 100);

for j = 1:numOfFrameRepeats
A(:,numOfMovieFrames)=getframe(fig1);
numOfMovieFrames = numOfMovieFrames+1;
end

disp('done...')
disp(['done ', num2str(i), ' frame(s) of ', num2str(length(cameraPositions)), ' frames'])
disp(['---------------------------'])
end

filename = [dir, 'mpg_export'];
disp(['saving movie to ', filename])
try
mpgwrite(A,jet,filename);
catch exception
end

imshow(zeros(height, width, 3));
movie(fig1,A,30,6)
disp('done...')

disp(['---------------------------'])
Bibliography


[41] Schomaker, Waser, Marsh and Bergman *To fit a plane or a line to a set of points by least squares*. In Acta Cryst. 12, pages 600-604. 1959.


