

# Automatic Modeling and Localization for Object Recognition

Mark Damon Wheeler

October 25, 1996

CMU-CS-96-188

School of Computer Science  
Carnegie Mellon University  
Pittsburgh, PA 15213

*Submitted in partial fulfillment of the requirements  
for the degree of Doctor of Philosophy.*

Thesis Committee:

Katsushi Ikeuchi, Chair  
Martial Hebert  
Steven Shafer, CMU/Microsoft  
Eric Grimson, MIT

© 1996 Mark D. Wheeler

This research has been supported in part by the Advanced Research Projects Agency under the Department of the Army, Army Research Office grant number DAAH04-94-G-0006, and in part by the Department of the Navy, Office of Naval Research grant numbers N00014-95-1-0591 and N00014-93-1-1220. Views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing official policies or endorsements, either expressed or implied, of the Department of the Army, the Department of the Navy or the United States Government.

**Keywords:** localization, pose refinement, automatic object modeling, recognition, tracking, robust statistics, consensus, M-estimation, surface merging

## Abstract

Being able to accurately estimate an object's pose (location) in an image is important for practical implementations and applications of object recognition. Recognition algorithms often trade off accuracy of the pose estimate for efficiency—usually resulting in brittle and inaccurate recognition. One solution is object localization—a local search for the object's true pose given a rough initial estimate of the pose. Localization is made difficult by the unfavorable characteristics (for example, noise, clutter, occlusion and missing data) of real images.

In this thesis, we present novel algorithms for localizing 3D objects in 3D range-image data (3D-3D localization) and for localizing 3D objects in 2D intensity-image data (3D-2D localization). Our localization algorithms utilize robust statistical techniques to reduce the sensitivity of the algorithms to the noise, clutter, missing data, and occlusion which are common in real images. Our localization results demonstrate that our algorithms can accurately determine the pose in noisy, cluttered images despite significant errors in the initial pose estimate.

Acquiring accurate object models that facilitate localization is also of great practical importance for object recognition. In the past, models for recognition and localization were typically created by hand using computer-aided design (CAD) tools. Manual modeling suffers from expense and accuracy limitations. In this thesis, we present novel algorithms to automatically construct object-localization models from many images of the object. We present a consensus-search approach to determine which parts of the image justifiably constitute inclusion in the model. Using this approach, our modeling algorithms are relatively insensitive to the imperfections and noise typical of real image data. Our results demonstrate that our modeling algorithms can construct very accurate geometric models from rather noisy input data.

Our robust algorithms for modeling and localization in many ways unify the treatment of these problems in the range image and intensity image domains. The modeling and localization framework presented in this thesis provides a sound basis for building reliable object-recognition systems.

We have analyzed the performance of our modeling and localization algorithms on a wide variety of objects. Our results demonstrate that that our algorithms improve upon previous approaches in terms of accuracy and reduced sensitivity to the typical imperfections of real image data.



## Acknowledgements

I would first like to thank Katsushi Ikeuchi, my advisor and friend for much of my 7 year stay at Carnegie Mellon. I thank him for giving me the freedom to explore a wide variety of ideas and for having an open door and open mind whenever I needed it. I also thank my thesis committee members Martial Hebert, Steve Shafer, and Eric Grimson for their careful reading of this thesis and their valuable feedback regarding this research.

Takeo Kanade deserves special thanks for creating such a wonderful group and environment for studying computer vision and robotics. The VASC facilities gurus Jim Moody, Bill Ross and Kate Fissell deserve much of the credit for keeping everything going in the VASC group; their frequent help and patience was greatly appreciated.

I thank the faculty at Tulane University, especially Johnette Hassell and Mark Benard, for providing me with a solid undergraduate education and encouraging me to pursue a research career.

I was fortunate to have many great people to work with during my stay at CMU. My fellow grad students contributed many ideas and insights which helped me to develop and implement the experimental systems in this thesis: thanks to Yoichi Sato, Harry Shum, Sing Bing Kang, Fredric Solomon, George Paul, Prem Janardhan, David Simon, and Andrew Johnson for their friendship and help over the years. I also benefited from the help of visiting scientists including Takeshi Shakunaga and Yunde Jiar. I have been fortunate to have a number of excellent officemates during my stay here including Kevin Lynch, Tammy Abel, Greg Morrisett, Andrzej Filinski, Lars Birkedal, Margaret Reid-Miller, and John Ockerbloom.

Grad school would have been much more difficult without the many of the friends I have made here. I am most grateful for having shared an office and house with Kevin Lynch. It was important to have a friend who shared so much in common and could always counted on. Kevin will always be like a brother to me. My friends in the CS PhD program were very special as well. I especially cherished our coop dinner group, including Rich Goodwin, Scott and LeAnn Neal Reilly, Dave Tarditi, Greg and Jack Morrisett, Susan Hinrichs, Alan Carroll, Mei Wang, Kevin Lynch, and Tammy Abel, which shared many fun times together. I also cherish the friendship of John Mount, Nina Zumel, Mark and Heather Leone and Bob Doorenbos (croquet anyone?).

Some of my favorite times were spent playing intramural sports at CMU, with teams such as the infamous Rude Dogs, NP Completions, Viking Death Rats, Base Registers and Bucket Brigade. I will greatly miss all these people, most especially “Super” Bill Niehaus, Will Marrero, John Cheng, John Greiner, Jerry Burch, Todd Jochem, Dave Kosbie, Fabio Cozman, Rich Voyles, David Steere,

Jay Kistler, John Pane, Mike Young, Bunsen Wong, Harry Shum and the many others too numerous to mention.

Lastly, I would like to thank the most important people in my life, my family, for always being there for me and supporting me unconditionally: my siblings Brandon, Douglas, and Dana who can crack me up at any time and make me proud to be related to them; my precious niece Aerial who makes me smile every time I think of her; and, most of all, my Mom and Dad for their love, advice, patience, and help for these many years and for the years to come.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Notation, Terminology, and Assumptions . . . . .	3
1.2	Philosophical Background . . . . .	6
1.3	Thesis Overview . . . . .	7
<b>2</b>	<b>Object Modeling for 3D-3D Localization</b>	<b>13</b>
2.1	Approach . . . . .	13
2.2	Surface Acquisition . . . . .	15
2.3	View Alignment . . . . .	17
2.4	Data Merging . . . . .	20
2.5	3D Object Modeling Results . . . . .	34
2.6	3D Modeling: Summary . . . . .	46
<b>3</b>	<b>3D-3D Object Localization</b>	<b>53</b>
3.1	Approach . . . . .	53
3.2	Point Visibility . . . . .	55
3.3	3D-3D Correspondence . . . . .	65
3.4	3D-3D Pose Optimization . . . . .	70
3.5	Putting It Together: 3D-3D Localization . . . . .	84
3.6	3D-3D Localization: Experimental Results . . . . .	87
3.7	3D-3D Object Localization: Summary . . . . .	100
<b>4</b>	<b>Object Modeling for 3D-2D Localization</b>	<b>102</b>
4.1	Problems and Approach . . . . .	102
4.2	View Acquisition . . . . .	108
4.3	2D to 3D: View Alignment . . . . .	110
4.4	View Merging: the Consensus-Edgel Algorithm . . . . .	116
4.5	Modeling Occluding Contours for 3D-2D Localization . . . . .	119
4.6	Object Modeling for 3D-2D Localization: Results . . . . .	125
4.7	Object Modeling for 3D-2D Localization: Summary . . . . .	127

<b>5</b>	<b>3D-2D Object Localization</b>	<b>136</b>
5.1	Problems and Approach . . . . .	136
5.2	Edgel Visibility . . . . .	139
5.3	Edgel Correspondence . . . . .	145
5.4	Pose Optimization . . . . .	150
5.5	Putting It Together: 3D-2D Localization . . . . .	155
5.6	Multi-Image Localization . . . . .	156
5.7	3D-2D Localization Results . . . . .	157
5.8	3D-2D Localization: Summary . . . . .	171
<b>6</b>	<b>Related Work</b>	<b>175</b>
6.1	3D Object Modeling . . . . .	175
6.2	3D-3D Object Localization . . . . .	181
6.3	Object Modeling for 3D-2D Localization . . . . .	184
6.4	3D-2D Object Localization . . . . .	190
6.5	Object Recognition . . . . .	193
<b>7</b>	<b>Conclusions</b>	<b>196</b>
7.1	Contributions . . . . .	198
7.2	Future Work and Discussion . . . . .	198
<b>A</b>	<b>Camera and Robot Calibration</b>	<b>203</b>
<b>Appendices</b>		
<b>B</b>	<b>Quaternions and Rotation</b>	<b>207</b>
B.1	Why Quaternions? . . . . .	207
B.2	Quaternions . . . . .	208
B.3	The Jacobian of Rotation With Respect to Quaternions . . . . .	209
<b>C</b>	<b>Principal Curvatures from Triangulated Surfaces</b>	<b>213</b>
	Bibliography	



# List of Figures

1.1	An example of (a) 2D image edgels and (b) 3D edgels or edgel generators. . . . .	5
1.2	An example of the 3D object modeling process. . . . .	9
1.3	An example of 3D-3D localization for a simple sphere object. . . . .	10
1.4	An example of the 3D edgel modeling process. . . . .	11
1.5	An example of 3D-2D localization for a simple bulls-eye object. . . . .	12
2.1	Organization of this chapter. . . . .	15
2.2	Connecting range image points to form a surface. . . . .	16
2.3	Examples of triangulation errors. . . . .	17
2.4	An example of zero-crossing interpolation from the grid sampling of an implicit surface. . . . .	21
2.5	In practice, real surface samples are noisy and slightly misaligned and it is difficult to determine where the actual surface might lie. . . . .	24
2.6	Some effects of noise and misalignment on our naive algorithm. . . . .	24
2.7	Graphical illustration of the error in zero-crossing interpolation using Algorithm <i>ClosestSignedDistance</i> with two noisy observations. . . . .	26
2.8	An example of inferring the incorrect sign of a voxel's signed-distance function. . . . .	27
2.9	The extension of the implicit surface in unobserved regions of the grid. . . . .	30
2.10	A demonstration of gradient testing to eliminate false surfaces at zero crossings of the implicit surface. . . . .	31
2.11	A 2D slice of an octree representation of a simple surface illustrates the adaptive resolution which is high around the surface and low elsewhere. . . . .	33
2.12	The two degrees of freedom, $\theta_x$ and $\theta_y$ of the Puma used to vary of object position with respect to the camera's coordinate system. . . . .	35
2.13	Results from modeling the boxcar. . . . .	37
2.14	Three views of the resulting triangulated model of the boxcar. . . . .	38
2.15	Results from modeling the fruit. . . . .	39
2.16	Three views of the resulting triangulated model of the fruit. . . . .	40
2.17	Results from modeling the rubber duck. . . . .	41
2.18	Three views of the resulting triangulated model of the duck. . . . .	42

2.19	Results from modeling the ceramic mug. . . . .	43
2.20	Three views of the resulting triangulated model of the mug. . . . .	44
2.21	Results from modeling the toy car. . . . .	45
2.22	Three views of the resulting triangulated model of the car. . . . .	46
2.23	The result of the naive algorithm, Algorithm <i>ClosestSignedDistance</i> , on the duck image. . . . .	47
2.24	A cross section of the final model of the boxcar (thick black line) and the original range-image data (thin black lines) used to construct it. . . . .	48
2.25	A cross section of the final model of the fruit (thick black line) and the original range-image data (thin black lines) used to construct it. . . . .	49
2.26	A cross section of the final model of the rubber duck (thick black line) and the original range-image data (thin black lines) used to construct it. . . . .	50
2.27	A cross section of the final model of the ceramic mug (thick black line) and the original range-image data (thin black lines) used to construct it. . . . .	51
2.28	A cross section of the final model of the toy car (thick black line) and the original range-image data (thin black lines) used to construct it. . . . .	52
3.1	Organization of this chapter. . . . .	56
3.2	An example of the set of visible viewing directions (the unshaded portion of the hemisphere) for the point $x$ on the L-shape. . . . .	58
3.3	Orthographic projection of viewing directions onto a simple LUT array. . . . .	60
3.4	Latitudinal/Longitudinal discretization of a hemisphere. . . . .	61
3.5	Stereographic projection of viewing directions onto a simple LUT array. . . . .	62
3.6	The mapping of viewing directions to visibility values via a two-level lookup table. . . . .	62
3.7	K-d tree subdivision of 2D points. . . . .	67
3.8	An example of nearest-neighbor correspondence (left) versus nearest neighbor+normal correspondence (right). . . . .	67
3.9	Plot of the 3D distance between two unit vectors with respect to the angle between the two vectors. . . . .	69
3.10	Examples of local versus global minima and basins of attraction. . . . .	74
3.11	Plots of $w(z)$ for each of the M-estimators listed in Table 3.1 . . . . .	78
3.12	Plots of $P(z)$ for each of the M-estimators listed in Table 3.1 . . . . .	78
3.13	Example of the fluctuation of $E$ due to changes in the pose and the set of visible points. . . . .	82
3.14	The minimum distance function with respect to the dynamic correspondences. . . . .	82
3.15	Correspondences which compare normals can create discontinuities in the distance function. . . . .	83
3.16	Three views of the mug object generated using our surface point visibility approximation. . . . .	89

3.17	The variation of $E$ along a line of translation in pose space. . . . .	91
3.18	The variation of $E$ along a line of pure rotation in pose space. . . . .	91
3.19	The variation of $E$ along a line of translation and rotation in pose space. . . . .	92
3.20	A close-up of a piece of the graphs of the function $E$ for the Lorentzian, Gaussian and Threshold along a line of translation in pose space. . . . .	92
3.21	The Sharpei dog convergence test data. . . . .	95
3.22	The ceramic mug convergence test data. . . . .	96
3.23	The toy car convergence test data. . . . .	97
3.24	The rubber duck convergence test data. . . . .	98
3.25	Depiction of three iterations of the randomized localization search. . . . .	99
4.1	Organization of this chapter. . . . .	109
4.2	An example of raw and smooth edgel chains. . . . .	110
4.3	An example of mapping a 2D edgel onto a 3D triangulated surface. . . . .	111
4.4	A diagram representing the relationships between the various coordinate frames used in the analysis in this chapter. . . . .	112
4.5	An occluding boundary edgel which does not project onto the model surface. . . . .	114
4.6	The shortest distance between a ray and the surface triangles when they do not intersect. . . . .	114
4.7	An occluding contour mapped to the object. . . . .	115
4.8	The stages of a consensus-edgel search. . . . .	118
4.9	A circle and its occluding contour points. . . . .	121
4.10	Example of normal sections with (a) positive normal curvature and (b) negative normal curvature. . . . .	123
4.11	Classes of surfaces based on curvature. . . . .	125
4.12	Results from modeling the 3D edgel generators of the stop sign. . . . .	128
4.13	Results from modeling the 3D edgel generators of the T-sign. . . . .	129
4.14	Results from modeling the 3D edgel generators of the bulls-eye. . . . .	130
4.15	Results from modeling the 3D edgel generators of the boxcar. . . . .	131
4.16	Results from modeling the 3D edgel generators of the mug. . . . .	132
4.17	Results from modeling the 3D edgel generators of the car. . . . .	133
4.18	Results from modeling the 3D edgel generators of the duck . . . . .	134
5.1	Organization of this chapter. . . . .	138
5.2	Examples of three types of edgel generators: (a) rigid surface marking, (b) convex edges, (c) occluding contours. . . . .	140
5.3	A normal section of a surface. . . . .	143
5.4	Using the tangent direction of an edgel to locate pixels to sample the left and right intensities, $I_L$ and $I_R$ , for computing the reflectance ratio. . . . .	148
5.5	Example of correspondence search using reflectance ratios. . . . .	149

5.6	Gradient-descent minimization path to align a 3d point $x$ with the line of sight $\hat{v}$ (a) infinitesimal steps, (b) discrete steps, (c) shortest path. . . . .	153
5.7	An example of the 3D error vector between the 3D point and the line of sight of edgel. . . . .	154
5.8	The stereo correspondence problem: several points are not visible in both views. . . . .	156
5.9	The variation of $E$ along a line of translation in pose space for the Gaussian and Lorentzian weight functions. . . . .	159
5.10	The variation of $E$ along a line of pure rotation in pose space for the Gaussian and Lorentzian weight functions. . . . .	159
5.11	The variation of $E$ along a line of translation and rotation in pose space for the Gaussian and Lorentzian weight functions. . . . .	160
5.12	The bulls-eye convergence test data. . . . .	162
5.13	The stop sign convergence test data. . . . .	163
5.14	The T-sign convergence test data. . . . .	164
5.15	The boxcar convergence test data. . . . .	165
5.16	The duck convergence test data. . . . .	166
5.17	The car convergence test data. . . . .	167
5.18	The mug convergence test data. . . . .	168
5.19	Tracking the car through 5 images at 10 degree intervals. . . . .	170
5.20	Multi-image localization of the stop sign. . . . .	172
5.21	Multi-image localization of the paint can. . . . .	173
6.1	Illustration of the result of Curless and Levoy's signed distance integration in the neighborhood of thin object parts. . . . .	177
6.2	Illustration of estimating the distance to the surface along rays passing through a voxel. . . . .	178
6.3	Illustration of estimating the distance to the surface along rays passing through a voxel. . . . .	178
A.1	An image of the experimental setup including the CCD camera, light-striping projector, and Puma robot. . . . .	204
A.2	The calibration cube used for range sensor and camera calibration. . . . .	205

# List of Tables

2.1	Statistics of the modeling experiments for each object. . . . .	40
3.1	Table of weight functions for M-estimation. . . . .	79
3.2	Results of the convergence experiments with variations of our 3D-3D localization algorithm and ICP. . . . .	95
3.3	Results of the convergence experiments with variations of ICP on the duck image. . . . .	98
3.4	Results of the randomization convergence experiments. . . . .	100
4.1	Statistics of the edgel modeling experiments for each object. . . . .	127
5.1	Results of the convergence experiments for our 3D-2D localization algorithm using the Lorentzian and Gaussian weight functions. . . . .	161
5.2	Results of the single versus multi-image convergence tests. . . . .	171

# Chapter 1

## Introduction

Object recognition remains one of the central problems in the fields of artificial intelligence and computer vision. The standard definition of the object-recognition problem is to *identify* and *locate* instances of known objects in an image [4, 23, 109]. That is, we do not simply desire to know that an object exists in the image, but that we also need to know where it is in the image. This location or positional information is commonly referred to as the object's *pose*.

Knowledge of an object's pose has many important applications; it enables us: to reason about the image scene, to interact with objects in the scene, to analyze geometric relationships between objects in the scene, and to describe the scene for later synthetic or physical reproduction.

Pose is also fundamental for practical object-recognition algorithms—impacting both the accuracy and efficiency of these algorithms. Pose allows us to improve recognition by enabling us: to accurately verify an object hypotheses using, for example, a project and test strategy; to efficiently recognize time-varying image sequences via object tracking; and to detect and reason about partial occlusion.

When we say that knowledge of the pose is important for applications in object recognition, we should qualify that by saying *accurate* knowledge of the pose is important. Object verification [50] is a very difficult problem for recognition algorithms when pose is perfectly known. If the pose is inaccurate, verification becomes a highly unreliable operation and recognition efficiency and accuracy will be affected. Likewise, applications that rely on knowing where the object is will also fail when the pose error is too large. Thus, we claim that determining accurate pose is a fundamental requirement for object recognition.

Accurate knowledge of pose is also crucial for object tracking. Tracking is a simplification of the object-recognition problem. In the case of tracking, we know roughly the position of the object in the image using a prediction from previous image observations. Thus, an efficient local search can be used to find the new object position and verify it. Tracking ability has great implications for practical recognition in the case of image sequences. If we can reliably track objects, we must perform full scale recognition only on parts of images that contain something new.

Unfortunately, much work on object recognition neglects the problem of accurately computing the object's pose. There are a wide variety of techniques for recognition that give identity of objects and their rough pose (see Chapter 6 for a discussion of these). Because of the highly unfavorable combinatorics of recognition, recognition algorithms must often sacrifice pose accuracy (and, hence, recognition accuracy) for efficiency. The best example of this is the alignment algorithm [70]. The alignment algorithm searches for minimal sets of matches between image and object features to align a 3D object to a 2D image. Grimson and Huttenlocher [50] have shown that the resulting uncertainty in the pose (when using a small number of matches) is enough to make verification prone to frequent failure. For example, under realistic uncertainty in the location of features in an image, an algorithm which estimates the position of the object using 3 or 4 point or line segment matches may be off by 20 or more degrees in rotation. Thus, the predicted positions of other features of the object will be significantly affected—resulting in a very low probability to correctly verify the presence of the object in the image. The solution is to search for more feature correspondences (create an over-determined system); however, this results in combinatorial explosion which is what we were trying to avoid in the first place.

The solution to this tradeoff problem is *localization*<sup>1</sup>—a local search for the true pose given the rough pose of an object in an image. If localization can be done efficiently and reliably, recognition algorithms like the alignment algorithm and others will eventually become practical in terms of accuracy and efficiency. In addition, a solution to localization is immediately useful as a solution to the tracking problem. Localization is fundamentally equivalent to the object-tracking problem. In typical tracking systems, the object's estimated velocity and acceleration is used to better predict the object's current position. From that predicted position, a localization search is performed. Localization is a bit more general than object tracking in that no information other than the object's identity and pose is assumed to be known.

This thesis presents solutions to the localization problem in the two most prevalent sensor modalities in the field of computer vision: the range image (3D) and intensity image (2D) modalities. Specifically, we present novel techniques for localizing 3D objects in 3D range-image data (3D-3D localization) and for localizing 3D objects in 2D intensity-image data (3D-2D localization). The solutions presented here in many ways unify the treatment of the two domains.

This thesis also addresses another fundamental problem of recognition, tracking and localization: model acquisition. In the past, models for recognition and tracking have typically been manually created using computer-aided design (CAD) tools. Manual modeling suffers from expense and accuracy limitations. The modeling solutions presented in this thesis automatically construct object-localization models from real images of the object. These modeling solutions are insensitive to the noisy data typical of real image data (for both range and intensity images). Robust model acquisition is achieved by searching

---

<sup>1</sup>The term localization has previously been used in [52] to denote the complete object-recognition task (from image to object identity and pose). Here we use the localization as a synonym for pose refinement and distinguish it from the task of object indexing and hypothesis generation.

for consensus information among the views to determine which image features justifiably constitute an element in the model. Together, the modeling and localization framework presented in this thesis provide a sound basis for building reliable recognition and tracking systems.

This thesis makes the following contributions:

- Robust 3D surface model construction with the consensus-surface algorithm.
- Robust 3D-3D and 3D-2D object localization by performing M-estimation using dynamic correspondences.
- Robust 3D edgel model construction with the consensus-edgel algorithm.
- Treatment of occluding contours in 3D-2D localization.
- 3D-2D localization performed in three dimensions, unifying the localization search for the 3D-3D and 3D-2D domains.

## 1.1 Notation, Terminology, and Assumptions

Before jumping into the technical portion of this thesis, it is prudent to specify the notation, terminology, and assumptions which will be used throughout this thesis. Much of this will be familiar to computer-vision and object-recognition researchers; however, some of the terminology may have slightly different meanings than used elsewhere.

### 1.1.1 Mathematical Notation

First, we list a few of our mathematical notations and notes:

- Vectors are in boldface type:  $\mathbf{x}$  is a vector,  $x$  is a scalar.
- Unit vectors have the hat symbol:  $\hat{\mathbf{x}}$  is a unit vector,  $\mathbf{x}$  may not be.
- Matrices are capitalized and in boldface type:  $\mathbf{M}$  is a matrix,  $\mathbf{I}$  is the identity matrix.
- $\mathbf{x}$  will be used to denote model points.
- $\mathbf{u}$  will be used to denote 3D image coordinates (before projection to two dimensions).
- $\mathbf{U}$  will be used to denote 2D image coordinates (note, this is the only vector that will be capitalized).
- $\mathbf{y}$  will be used to denote image points.
- Vectors should be assumed to be three dimensional unless otherwise noted.



- Pairs or tuples will be denoted using the angle brackets  $\langle \cdot \rangle$  (e.g.,  $\langle x, y \rangle$  denotes the pair of vectors  $x$  and  $y$ ).
- $p$  will be used to denote the object's pose.
- The 3D rigid transformation defined by the pose  $p$  may be specified by one of the following notations:
  - simply by  $p$
  - by a rotation and translation pair  $\langle \mathbf{R}, \mathbf{t} \rangle$  where  $\mathbf{R}$  is a  $3 \times 3$  rotation matrix and  $\mathbf{t}$  is a 3D translation vector.
  - by the equivalent homogeneous transformation  $\mathbf{R}$  where  $\mathbf{R}$  is the equivalent  $4 \times 4$  homogeneous transform matrix and the point being transformed is assumed to be extended to 4D homogeneous coordinates (i.e., by placing a 1 at the end of the 3D vector)<sup>2</sup>.

### 1.1.2 Terminology

We now define some of our terminology.

We will often refer to *range images* (sometimes known as depth images). Here, a range image is assumed to be a 2D view of a scene in which each image pixel contains the 3D coordinate of the scene in some world or camera coordinate system. The coordinate system should be assumed to be Euclidean.

We will often discuss intensity edges extracted from intensity images. An *edgel* is an individual point or edge element along an edge chain. Typically, an edgel will be represented by a pair  $\langle \mathbf{U}, \hat{\mathbf{t}} \rangle$  where  $\mathbf{U}$  are image coordinates of the edgel and  $\hat{\mathbf{t}}$  is its tangent direction, again in image coordinates. The edgel chain is usually extracted from intensity images using one of many edge operators (e.g., Canny [16] or Deriche [30]). For an example of an edgel, see Figure 1.1(a).

A *3D edgel* denotes the extension of a 2D edgel to an oriented point in 3D space. It is denoted  $\langle \mathbf{x}, \hat{\mathbf{t}} \rangle$  where  $\mathbf{x}$  are the 3D coordinates of the edgel and  $\hat{\mathbf{t}}$  is its (3D) tangent direction. Figure 1.1(b) shows an example of this. An *edgel generator* denotes the point on an object which produces the image of the (2D) edgel in the intensity image.

For intensity edge images, Breuel [14] used the terms *attached* and *non-attached* to distinguish between two types of edgels. *Attached* edgels refer to those detected edgels which are attached to the surface of an object (e.g., a surface marking or corner). *Non-attached* edgels refer to those detected edgels which belong to occluding contours of the object, and appear to float across the surface of the object as it is rotated.

As mentioned previously, the object's six location coordinates are collectively referred to as its *pose*. Again, *localization*—also known as *pose refinement*<sup>3</sup>—is a local search for

<sup>2</sup>The homogeneous notation will be used in place of  $\langle \mathbf{R}, \mathbf{t} \rangle$  for brevity when discussing a transformation as a single entity.

<sup>3</sup>The terms *pose refinement* and *localization* will be used interchangeably throughout this thesis.

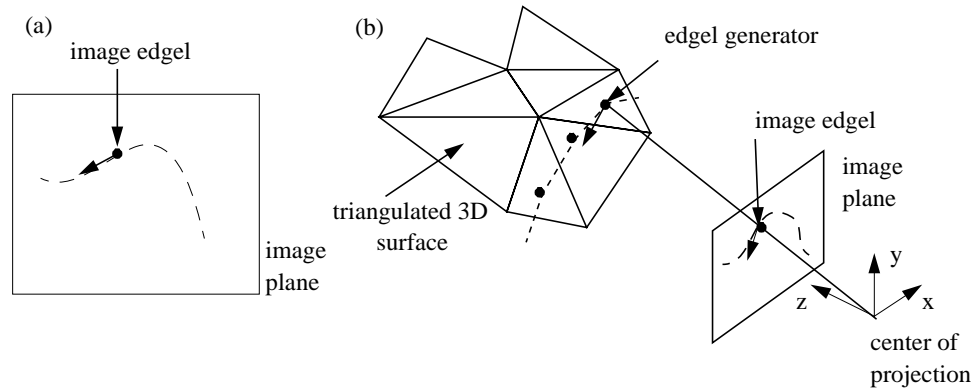


Figure 1.1: An example of (a) 2D image edgels and (b) 3D edgels or edgel generators.

the true pose given the rough pose of an object in an image. *3D-3D localization* refers to the process of localizing 3D objects in 3D range-image data. *3D-2D localization* refers to the process of localizing 3D objects in 2D intensity-image data.

### 1.1.3 Scope and Assumptions

We now briefly list some of the assumptions we make for the algorithms and experiments described in this work. Our main assumptions are practical limitations of the type of objects that may be used and a requirement for calibrated cameras and range sensors.

Our first assumption is that we only address the problems of modeling and localizing rigid, 3D objects. This means we consider objects with six degrees of freedom, three rotational and three translational. This thesis does not address issues related to deformable, non-rigid or generic objects. While not totally general, rigid objects do account for quite a large segment of everyday objects. Rigid object modeling and localization is still a very important area of current research with many applications. We view the rigid object case as an important first step which must be solved before considering completely general classes of objects. We also assume the objects are opaque and do not contain significantly large portions of high frequency texture with respect to our sensors.

Lastly, we assume that we are working with cameras and range sensors which are calibrated. The intrinsic sensor parameters must be known for us to be able to predict an object's appearance, using a 3D model of the object, in the sensor's images.

Our modeling work assumes that we have a calibrated object positioning system (details described in Chapter 2 and Appendix A), which limits the size of objects we may model to those that are mountable on a robot's end effector. We believe that this limitation will soon be obviated with more work on view alignment (as will be discussed in Chapter 7) and improvements in the area of structure from motion.

## 1.2 Philosophical Background

We take a few paragraphs to describe some of the high-level issues and ideas which drive this work.

### 1.2.1 Consensus

One of the key ideas which is embodied by our approach to object modeling and localization is the idea of *consensus*. For modeling from real data, consensus is the key to robustness. Because of the typical data from our sensors, no single measurement can be trusted. However, in sufficient numbers, several independent measurements of the same feature (e.g., a surface point or edgel) can be considered quite reliable.

Consensus is also subtly important for localization. The key for model based recognition and localization is the over-constrained nature of the problem (i.e., more constraints than free variables). Localization works because the consensus of the image data usually guides the local search to the true object pose.

### 1.2.2 3D vs View-Based Model Representations

An important decision is whether to use view-based (2D) models or full fledged 3D models. For recognition and localization in 3D image data, 3D models are obviously desirable. For recognition and localization in 2D images, there exists an ongoing debate between using many sample 2D views as the model or constructing a monolithic 3D model. One advantage of basic view-based methods is that model acquisition is trivial—take an image and add it to the collection. The problem is that an arbitrarily large number of images may be necessary to build a complete model of appearance. The model is only as good as the sampling resolution. For rigid, 3D objects, we will usually have six degrees of freedom over which to sample views. If we then consider lighting variations and camera model variations, the number of required samples can quickly become unmanageable.

Some view-based methods go a little further and try to alleviate the sampling problem by interpolating between views. This requires a complete set of correspondences between two views—this is not possible in general (see Chapter 6 for a discussion of this problem). Foreground/background separation becomes an important issue for modeling from real images; manual segmentation (i.e., selection of object features from the images) may be required.

3D models have the advantage that one model can be used to predict appearances with respect to all possible variations of pose, camera models, and illumination (if known a priori). Verification, localization and tracking can be accomplished with a 3D model as the change in appearance can be smoothly predicted with respect to changes in pose. The main detraction of monolithic 3D models is that they are difficult to construct. Automatically or semi-automatically constructing accurate 3D models from real images is a non-trivial problem. As we are already considering the problem of building 3D models for 3D-3D

localization, the acquisition of 3D models for localization in 2D images can be made practical as we shall show in this thesis.

### 1.2.3 Point-Based Models

Another philosophical issue is the type of models to use for these localization tasks. Our philosophy is to keep the model representation as simple as possible. Our choice of model representation is to consider each object as a collection of points.

For 3D-3D localization we represent an object as a set of points on the object's surface. This representation can be acquired from a densely triangulated surface model by sampling points from the triangles.

We also use points to represent the object for 3D-2D localization. Instead of simply using just any surface point, we use points on the model which generate edgels in intensity images. We refer to these points on the model as *3D edgels* or *edgel generators*. Such points comprise surface markings, occluding contours and convex/concave geometric edges of the object.

Not only are these representations simple, which makes them efficient to implement, but they are also very general. We can model a large subset of the possible rigid, 3D objects using these representations.

### 1.2.4 Data Driven Optimization

Our philosophy for localization derives much from the ideas of active contours introduced by Witkin, Terzopoulos and Kass [75, 134]. The active contour paradigm allowed the user to initialize the location of a contour and let the forces defined by image features act on the contour to find its optimal state. Besl and McKay's [6] iterative closest point (ICP) algorithm for 3D registration is based on this principle: correspondences are computed, a new estimate of the pose is computed and the process is repeated until the pose estimate converges. Our localization algorithm is designed in much the same way. We simply start it off in an initial position and find the local minimum of the energy function defined by the object model and the nearby image data. This is in contrast to other localization methods such as Lowe's [84], which performs a local interpretation-tree search that matches high level image tokens to the model. Our methods are data driven and rely on the lowest-level data available from our sensor (e.g., 3D points from range images and 2D edgels from intensity images).

## 1.3 Thesis Overview

There are four main technical contributions in this thesis which correspond to the breakdown of the following chapters:

- object modeling for 3D-3D localization

- 3D-3D object localization
- object modeling for 3D-2D localization
- 3D-2D object localization

We will now briefly discuss our approach to each of these problems and give a simple example to give the reader a hint of what follows.

### 1.3.1 Preview of Object Modeling For 3D-3D Localization

For the problem of object modeling for 3D-3D localization, we are interested in acquiring a 3D triangulated surface model of our object from real range data. From the triangulated surface model, we extract the points for the model for 3D-3D localization.

Our basic approach is to acquire several range-image views of the object, align the image data, merge the image data using the aid of a volumetric representation, and then extract a triangle mesh from the volumetric representation of the merged data. Our main contribution is a new algorithm for computing the volumetric representation from the sets of image data. Our algorithm, the consensus-surface algorithm, eliminates many of the troublesome effects of noise and extraneous surface observations in our data.

Figure 1.2 shows a simple example of the steps of the 3D object modeling process. We begin with by acquiring range images of the object from various views, the views are then aligned to the same coordinate system, and the views are merged into a volumetric implicit-surface representation which is then used to generate a triangulated model of the surface.

### 1.3.2 Preview of 3D-3D Object Localization

For the problem of 3D-3D object localization, we are interested in computing the precise pose of a 3D object in a (3D) range image given a rough estimate of the object's pose in the image.

Our localization algorithm iteratively refines the pose by optimizing an objective function defined over the image data, model data and the object's pose. The main contribution of our algorithm is the use of an objective function which is specified to reduce the effect of noise and outliers which are prevalent in real image data and a method for minimizing this function in practice. The objective function is effectively minimized by dynamically recomputing correspondences as the pose improves.

Figure 1.3 shows a simple example of our iterative 3D-3D localization algorithm. We begin with a range image of the object and an initial pose estimate. The correspondences between the model and image are a dynamic function of pose and allow the search to follow the proper path to the true pose of the object.

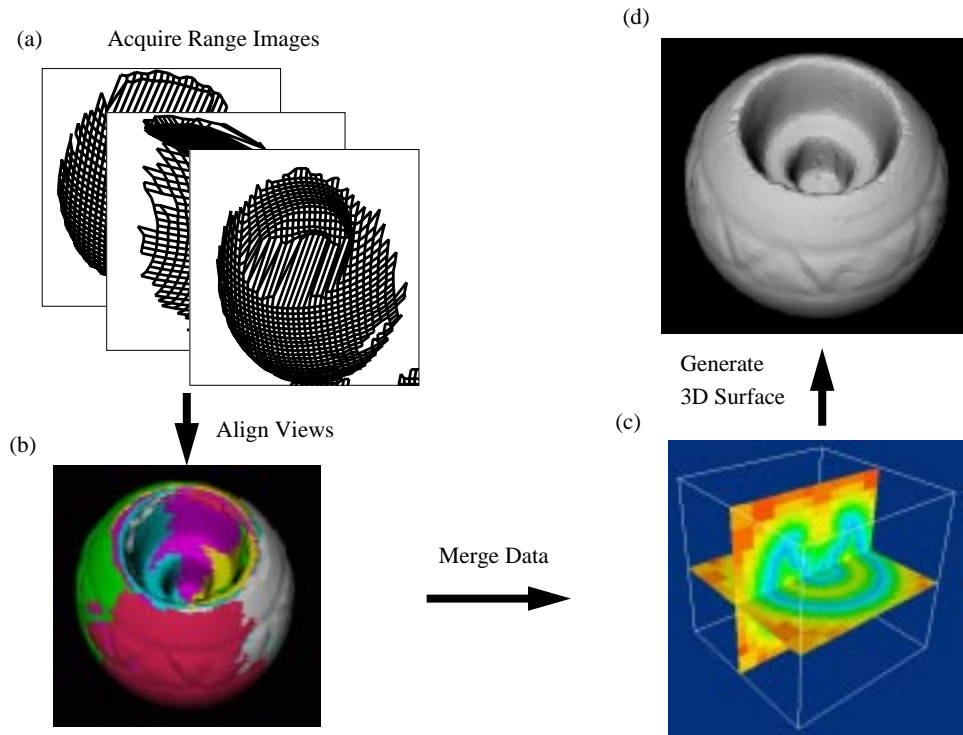


Figure 1.2: An example of the 3D object modeling process: (a) range-image acquisition, (b) view alignment, (c) the merged volumetric representation, (d) resulting triangulated model.

### 1.3.3 Preview of Object Modeling For 3D-2D Localization

For the problem of object modeling for 3D-2D localization, we are interested in acquiring a 3D model of the edgel generators of the object. Our approach is similar to our approach to 3D surface modeling

We also collect a set of intensity-image views of the object and extract the edgels from them using a standard edge operator. The edgels are then projected and aligned in the object's 3D coordinate system using a 3D surface model of the object (built using the 3D surface modeling approach alluded to previously). The aligned data is then merged to produce a set of rigid edgels. To account for occluding-contour edgels, we use curvature analysis of the points on our 3D surface model to predict which surface points are contour edgel generators. The main contributions of our 3D edgel modeling work is the consensus-edgel algorithm for extracting rigid edgel generators and the framework for representing occluding contours as edgel generator points. The consensus-edgel algorithm is able to reliably extract the significant edgel generators from large sets of rather noisy input data. Edgel generators can be used to efficiently and accurately predict the appearance of occluding contours in 2D

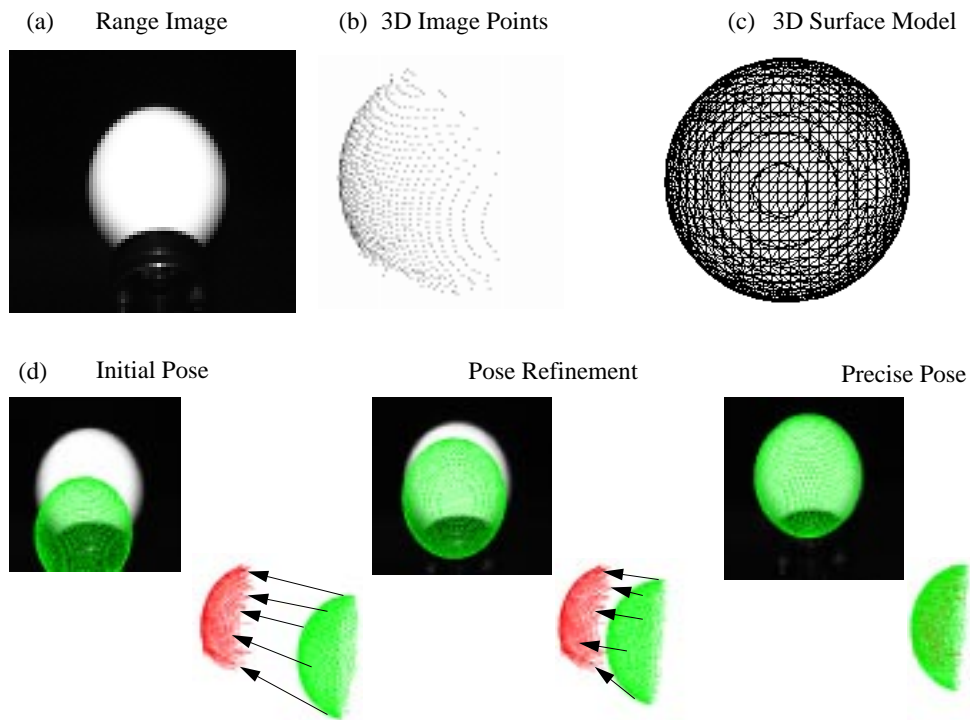


Figure 1.3: An example of 3D-3D localization for a simple sphere object: (a) range image of the sphere, (b) the 3D range data corresponding to the image, (c) the sphere object model, (d) three iterations of the localization search with arrows indicating the correspondences.

images.

Figure 1.4 shows a simple example of the steps of the 3D rigid edgel modeling process. We begin by acquiring intensity edges from various intensity-image views of our object, the edgels are then projected and aligned onto the object surface in the object's 3D coordinate system, and 3D edgel data is merged to form a 3D rigid edgel model.

### 1.3.4 Preview of 3D-2D Object Localization

For the problem of 3D-2D object localization, we are interested in computing the precise pose of a 3D object in a 2D intensity image given a rough estimate of the object's pose in the image.

Our localization algorithm iteratively refines the pose by optimizing an objective function defined over the image data, model data and the object's pose. The main contribution of our algorithm is the use of an objective function which is specified to reduce the effect of noise and outliers which are prevalent in real image data. The objective function is effectively minimized by dynamically recomputing correspondences as the pose improves.

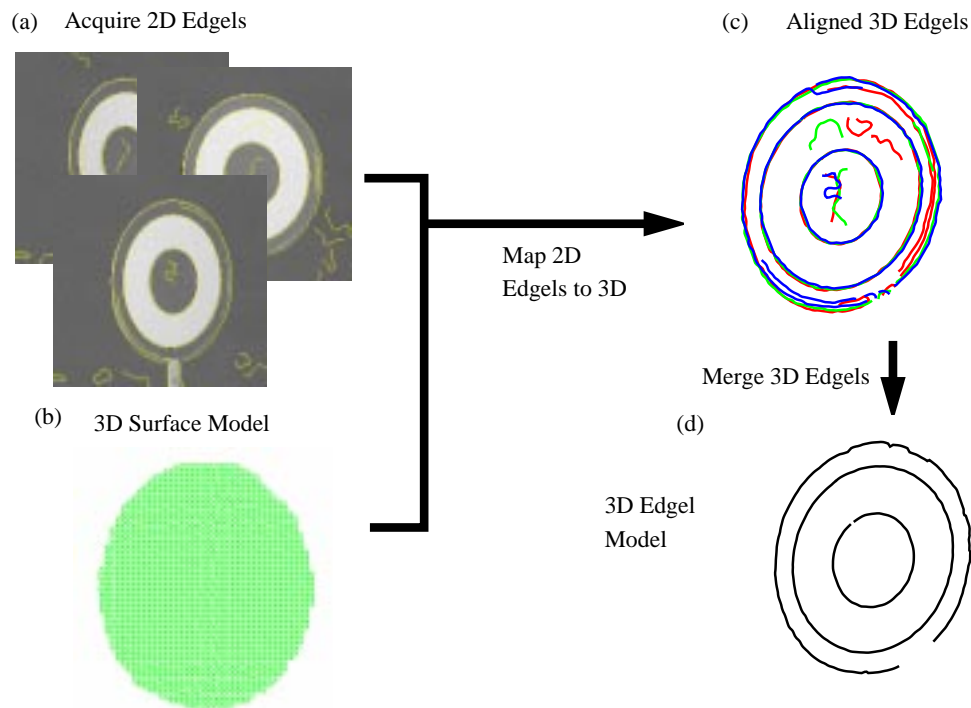


Figure 1.4: An example of the 3D edgel modeling process: (a) acquired intensity edgels (overlaid on the intensity images) (b) the object’s 3D surface model, (c) the edgels from various views mapped onto the object’s coordinate system, and (d) the rigid edgels extracted from the sets of mapped edgels.

Useful correspondences are efficiently found—despite significant pose errors and high densities of edgels in the intensity image—by extending the nearest-neighbor-search concept to include edgel attributes such as edgel normals and reflectance ratios. We show that the pose can be refined using much the same minimization algorithm as Algorithm 3D-3D Localization of Chapter 3.

Figure 1.5 shows a simple example of our iterative 3D-2D localization algorithm. We begin with an intensity image, its edge image, the 3D edgel model, and an initial pose estimate. The pose is refined iteratively as in the 3D-3D localization algorithm, the correspondences are computed dynamically as the pose search proceeds.

We now begin the technical discussion of object modeling for 3D-3D localization in Chapter 2, followed by discussion of 3D-3D object localization, object modeling for 3D-2D localization, and 3D-2D object localization in Chapters 3, 4 and 5, respectively. We then discuss the related research which influenced much of this thesis. We end by offering some conclusions, the contributions of this thesis, and a discussion of future research directions.



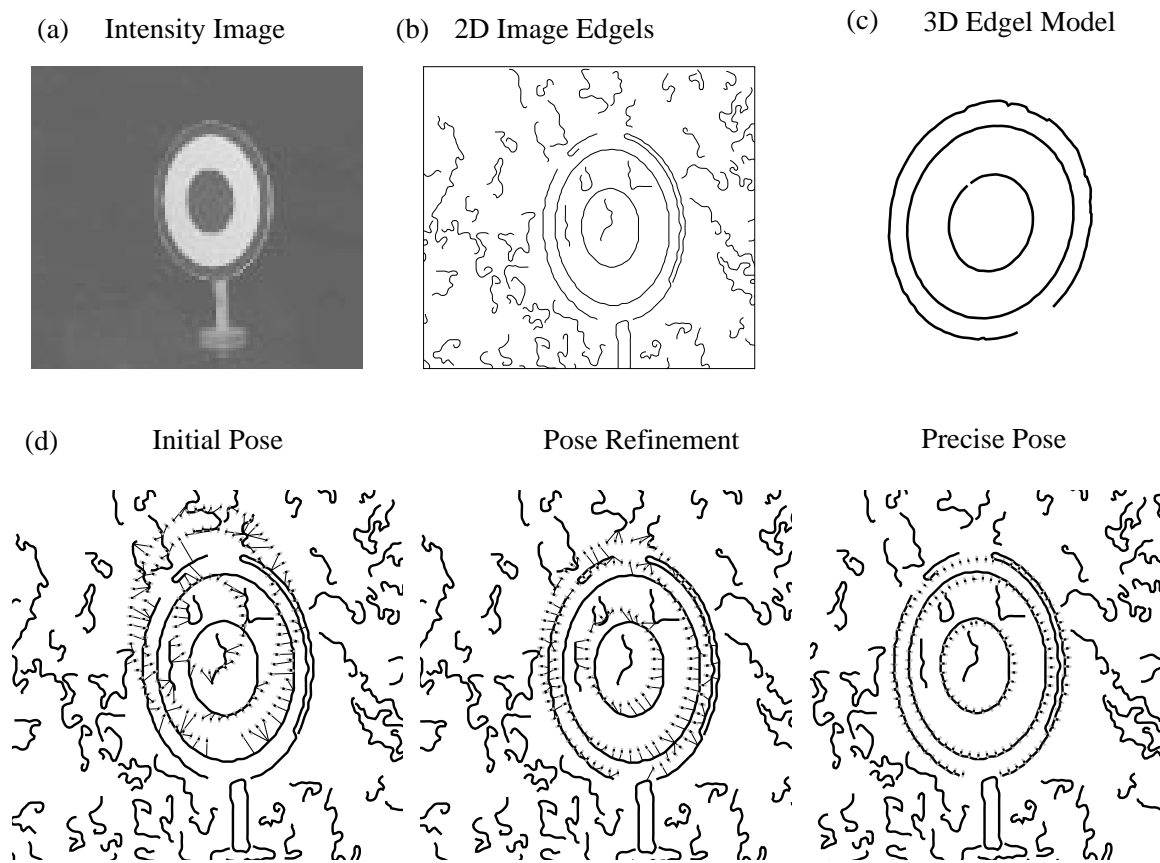


Figure 1.5: An example of 3D-2D localization for a simple bulls-eye object: (a) the input intensity image, (b) the intensity edges of the image, (c) the object model, and (d) three iterations of the localization search with lines drawn between the image to model edgel correspondences.

# Chapter 2

## Object Modeling for 3D-3D Localization

The goal of this thesis is to develop solutions for localizing known objects in images. The first localization problem that we will address is the 3D-3D localization problem—localizing 3D objects in 3D range-image data. This thesis also focuses on the problem of acquiring models for localization, in addition to the localization problems themselves. For 3D-3D localization, a good starting point for an object model is a 3D surface model.

In this chapter, we present a novel approach for building a 3D surface model from many range images of an object. The goal of this work is to use real images of an object to automatically create a model which is:

- Geometrically accurate: depicts the correct dimensions of the object and captures small details of the object geometry
- Clean: eliminates noise and errors in the views
- Complete: models the surface as much as is observable from the sample views

Efficiency is desirable, but is not a main concern, since model creation will be done off-line. The following section overviews the specific problems we face and our general approach for solving these problems.

### 2.1 Approach

The problem we are tackling in this chapter is to build a 3D model from a number of range images of an object. In other words, we will take  $N$  range images of an object from various views and use them to compute a unified surface representation of the object. We can simply stick all of the image data together; this sounds easy enough. Well, this is almost correct, but to do so, we must address several serious problems.

With a little bit of wishful thinking, let us assume that we can obtain 3D surfaces from various views and that we are able to align these views into a single object coordinate system. The first problem is how to combine the surfaces from all views into a single

surface representation—a data-merging problem. It is a problem of topology: how are all these surfaces connected?

Our solution makes use of a volumetric representation to avoid difficulties associated with topology. We will show how the volumetric representation simplifies our data-merging problem—virtually eliminating the topology issue. The volumetric representation can be conveniently converted into a triangulated mesh representation with little loss of geometric accuracy. The merging problem is then a matter of converting our input surface data to the volumetric representation.

Conversion from surfaces to the volumetric representation is simple if we are given perfect input surface data devoid of noise or extraneous data (image data which is nearby the object but not belonging to the object). The conversion problem is exacerbated by the fact that input surface data from real sensors (e.g., range sensors or stereo) is noisy and, in fact, will contain surfaces that are not part of the object we are interested in modeling. Our method for merging the surfaces into a volumetric representation takes full consideration of these facts to best take advantage of the multiple observations to smooth out the noise and eliminate undesired surfaces from the final model.

Unfortunately, the volumetric representation is not the answer to all that plagues us. We must now step back and determine how to get the image data aligned in the first place. Several strategies are possible, involving varying degrees of human interaction. Our approach is to make alignment fully automatic by taking the images using a calibrated robotic positioner.

Finally, we must consider the input from our sensor. Unfortunately, current sensors provide us with point samples of the surface—not the surface itself. Range-image sensors do not provide us with information on how the points in the image are connected. So even from a single view, we cannot guarantee that we know the topology of the viewed surface. This is a depressing state of affairs. Fortunately, we can make a good guess to get started. Also, we are fortunate that our data merging algorithm is designed to robustly handle errors such as the mistakes that we might make when converting our range data to surface data.

Another important issue which we do not address in this thesis is how to select views in order to best cover the surface. The sensor planning problem is very difficult and is the subject of ongoing research [128]. In this work, we do not try to optimize the number of views (i.e., taking the smallest number of views that cover the surface). Rather, we take a large set of views with the hope that they cover the surface. The human operator in fact determines the number of views and the object orientation for each view.

To summarize, to build a 3D surface model from multiple range images, we face the following problems:

- Input data: Surfaces are desired but the sensor provides points.
- View alignment: To merge the data, it must be in the same coordinate system.
- Data merging: We need to merge all the image data while eliminating or greatly reducing the effects of noise and extraneous data.

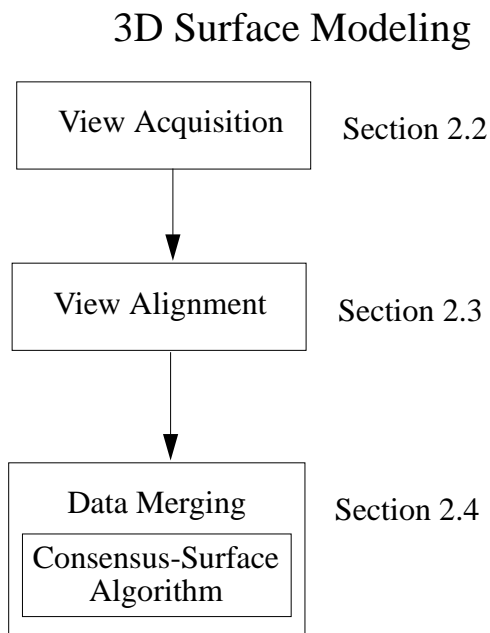


Figure 2.1: Organization of this chapter.

The rest of this chapter provides the details of our solutions to these problems which combine to form a practical method for building 3D surface models from range images of an object. Figure 2.1 shows a diagram of the technical sections of the thesis. We begin by discussing acquisition and alignment of surface views and then follow with a discussion of our surface merging algorithm, the consensus-surface algorithm, which is the main contribution of this chapter.

## 2.2 Surface Acquisition

Our first problem is that 3D sensors such as range finders produce images of 3D points; however, for many purposes including ours, it is necessary to sense surfaces. Unfortunately, such a sensor is not currently available. The missing information is whether the scene surface is well approximated by connecting two neighboring surface samples. With a little work, however, we can transform the 3D points from the range image into a set of triangular surfaces.

We can begin by joining pairs of neighboring range-image points based on our belief that the two points are connected by a locally smooth surface. When joining two points, there is very little basis for our decision. The limitations of the sensor prevents us from knowing the answer. However, we can make this decision based on our experience—understanding that

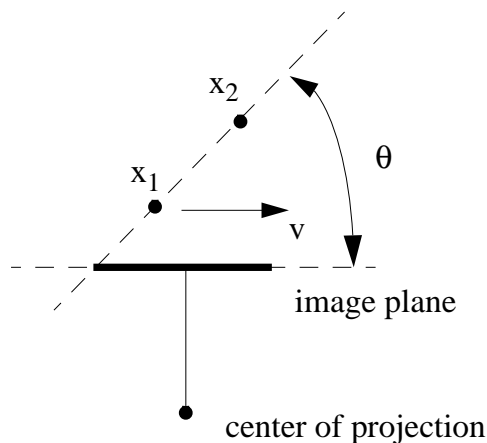


Figure 2.2: The orientation,  $\theta$ , of the line connecting two 3D surface points  $x_1$  and  $x_2$  with respect to the image plane of the camera is used to determine whether the two points are connected.

we will often make mistakes. Our experience tells us that if two (pixel-wise) neighboring range-image points have similar 3D coordinates, then they are likely to be connected by a locally smooth surface; if 3D coordinates are far apart, it is very unlikely that they are connected. This is using accomplished by using a threshold to determine whether the two points are close enough in three dimensions according to our experience. We adopt a thresholding scheme used by [64]. Two range data points,  $x_1$  and  $x_2$  are labeled as connected if

$$\frac{(x_2 - x_1)}{\|x_2 - x_1\|} \cdot \hat{v} = \cos \theta \geq \cos \theta_0$$

where  $\hat{v}$  is the unit direction vector of the difference in image (pixel) coordinates<sup>1</sup> of the two points,  $\theta$  is surface angle of the two connected points with respect to the camera, and  $\theta_0$  is the largest acceptable angle (typically 80 degrees or so). Figure 2.2 geometrically depicts the test performed here. This threshold scheme has the benefit that it does not depend on a specific scale of data or on specific camera parameters (e.g., aspect ratio). After all pairs of neighboring points are examined, we can create surface triangles by accumulating all triples of mutually connected points.

This, like most thresholds, is an *ad hoc* assumption and will often result in mistakes: surfaces will be created where there should be none, and some existing surfaces will be missed. Figure 2.3 which shows an example of such mistakes. These errors are not significantly different from other errors that our model building algorithm must confront. As will be shown in Section 2.5, our model building algorithm compensates for errors such

<sup>1</sup>The direction vectors of pixel differences in the image plane can be converted to world coordinates using the camera parameters, which are acquired via calibration as described in Chapter 1.

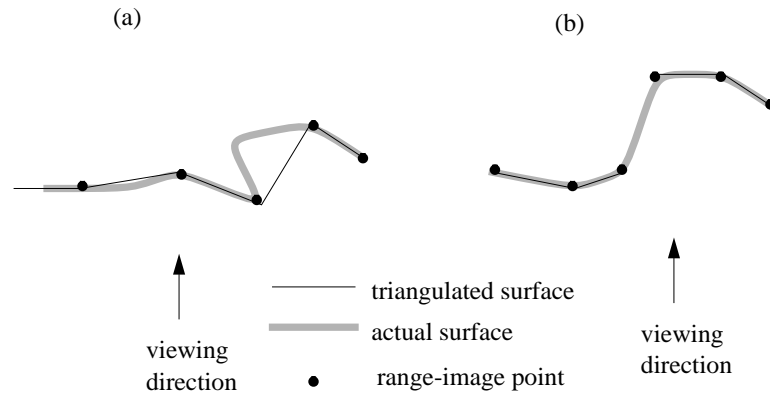


Figure 2.3: Examples of triangulation errors shown via 2D slices: (a) an incorrectly instantiated surface between two points and (b) a surface that is missed.

as these to produce correct surface models after merging.

## 2.3 View Alignment

After taking several range images of an object and converting them to surfaces, we need to eventually merge all these surfaces into a single model. The problem is that each view and, hence, the surface data is taken from a different coordinate system with respect to the object. In order to compare or match the data from different views, we have to be able to transform all the data into the same coordinate system with respect to the object.<sup>2</sup>

To do this implies that we need to determine the rigid body transformation (motion) between each view and some fixed object coordinate system. This rigid body motion comprises rotation and translation in 3D space—six degrees of freedom, three in translation and three in rotation. We can denote a rigid body motion by a linear transform

$$\mathbf{x}_1 = \mathbf{R}\mathbf{x}_0$$

where  $\mathbf{R}$  is a  $4 \times 4$  homogeneous matrix<sup>3</sup> denoting the rigid transform and the points  $\mathbf{x}_0$  and  $\mathbf{x}_1$  are in homogeneous coordinates. For clarity, let us denote  $\mathbf{R}$  by  $\mathbf{R}_{1 \leftarrow 0}$  which indicates

<sup>2</sup>It is conceivable that we could merge all the data from different views without computing rigid motion but by determining correspondences among all the data between views [54]. If correct correspondences can be made, view alignment is certainly achievable [135, 106].

<sup>3</sup>We will sometimes find it useful to denote a rotation and translation using either homogeneous transformations (i.e.,  $\mathbf{R}\mathbf{x}$  where  $\mathbf{R}$  is a  $4 \times 4$  matrix and  $\mathbf{x}$  is understood to be extended to 4 dimensions by appending a 1 to the 3D vector) or rotation followed by translation (i.e.,  $\mathbf{R}\mathbf{x} + \mathbf{t}$  where  $\mathbf{t}$  is a translation vector and  $\mathbf{R}$  is a  $3 \times 3$  rotation matrix). Both are mathematically equivalent, however, the homogeneous form is used when we desire to be concise in our notation.

that the motion takes a point in coordinate system 0 and transforms it to coordinate system 1.

If we are given  $N$  views, we can start by making one of the views the *central* view. For convenience, the central view is chosen to be view 0. The goal is to compute the rigid motion  $\mathbf{R}_{0 \leftarrow i}$  for all views  $i \neq 0$ . Once that is accomplished, the data in all views can be transformed to a single coordinate system (e.g., view 0), and we can then consider the problem of merging the data into a unified object model.

Notice the problem does not change if the camera is moving relative to the object or vice versa.<sup>4</sup> Determining the motion between two views can be a difficult task and is the subject of a large body of previous and ongoing research [82, 135, 130, 124, 106]. There are several ways we can approach this alignment problem—each requiring varying levels of human interaction. We break these into three levels—manual alignment, semi-automatic alignment, and automatic alignment—which are briefly discussed in what follows.

### 2.3.1 Manual Alignment

The first option, and perhaps the least attractive, is manual alignment of the views. The user could choose a particular view as the object's coordinate system and manipulate each view separately using a graphical interface to align the data of each view to the data of the central view. This is tedious and is made difficult by the limits of visualizing 3D objects with 2D displays. For example, two points may be aligned as viewed from one direction, but when viewed from another direction, the two points may lie at different distances along the original line of sight.

A less painful and more precise option is the use of registration marks on the object. The registration marks are easily identified points on the object that can be seen in multiple views. These points may be painted (e.g., distinguishable white or black dots) onto the object for this purpose.

Another option is for the user to select them via a point-and-click interface. Care must be taken to ensure that the selected points are really the same point in different views.

Regardless of how marks are selected/detected, the user must manually denote the correspondence between marks in each view. Once this is accomplished, the motions between all views and the central view can be computed. For 3D data, three corresponding points between two views are sufficient to estimate the motion between the views. However, if there are any errors or noise in the 3D coordinates of any of these marks then the motion estimate will also be noisy. The accuracy of the motion estimate can be improved by using a larger number of points. The problem of rigid motion estimation, or pose estimation, from corresponding points in three dimensions is discussed further in Section 3.4.

Note that either of these methods will be painful and time consuming for the person doing the modeling; however, depending on the situation, it may be the only option.

---

<sup>4</sup>This is not completely true. The case where the camera is moving creates the problem that the object and background have the same motion which means manual editing will be necessary to separate the object from the background.

### 2.3.2 Semi-Automatic Alignment

With current motion estimation techniques [6, 135, 106], we may be able to automatically compute the motion between each pair of views. This may be possible if the images are taken in a single sequence with small motions between each view. What we would have is a list of transforms between neighboring views

$$\mathbf{R}_{0\leftarrow 1}, \mathbf{R}_{1\leftarrow 2}, \dots, \mathbf{R}_{N-2\leftarrow N-1}.$$

From this it is possible to compute the transform from each view to view 0 by composing transforms, for example,

$$\mathbf{R}_{0\leftarrow 2} = \mathbf{R}_{0\leftarrow 1}\mathbf{R}_{1\leftarrow 2}.$$

While this sounds easy enough, it suffers from a fatal flaw. The flaw is that each estimated transform will have some error associated with it, and that as we compose erroneous transforms, the error accumulates. What one will find is that  $\mathbf{R}_{0\leftarrow N-1}$  will be too inaccurate for practical purposes for even moderate values of  $N$ .

The solution is to revert to manual alignment to finish the job by manually adjusting and improving the motion estimates to align the views and eliminate the accumulated errors. This is much easier than the previous fully manual alignment since the estimated transforms will actually be reasonably close and will only require small corrections.

### 2.3.3 Automatic Alignment

Finally, we can think about ways to achieve alignment without manual intervention. There are basically two ways that may allow us to achieve automatic view alignment: automatic motion estimation and controlled motion with calibration.

Fully automatic motion estimation [6, 135, 106] that is accurate enough for 3D modeling is still on the horizon. Currently, there are solutions to this problem which are becoming mature [135, 130, 124, 106] but are still not quite reliable enough for practical application which means some manual intervention may be required. The problem of error accumulation will still be an issue; however, recent work by Shum, Ikeuchi, and Reddy [127] using a technique called principal components analysis with missing data shows promise to solve this problem in the near future.

Because of the current state of the art, we use the second approach, controlled motion with calibration—the most practical option for an automatic solution. There are some arguments against such an approach:

1. Calibration is difficult.
2. Robots, turntables, and other positioning mechanisms are expensive.
3. Requiring controlled motion limits the applicability.



While arguments (2) and (3) are quite valid, argument (1) is not. Calibration is a mature area in photogrammetry and computer vision and many excellent algorithms exist [136, 121]. The process of acquiring calibration points can be made less tedious with the use of special calibration objects (specially painted boxes or boards) and simple techniques for detecting these points in the calibration images.

In our experimental setup, we calibrate two axes of a Unimation Puma robot with respect to a range-sensor coordinate frame. We can then mount the object on the robot's end effector and acquire images of an object at arbitrary orientations. The details of the camera and robot calibration process are described in Appendix A.

From this point we assume that the views are aligned. Next, we consider the problem of merging all the data from these views into a single model of the object's surface.

## 2.4 Data Merging

We are now faced with the task of taking many triangulated surfaces in 3D space and converting them to a triangle patch surface model. In this section, we assume that the various triangle sets are already aligned in the desired coordinate system.

As discussed in Section 2.1, even if we are given perfect sets of triangulated surfaces from each view which are more or less perfectly aligned, the merging problem is difficult. The problem is that it is difficult to determine how to connect triangles from different surfaces without knowing the surface beforehand. There are innumerable ways to connect two surfaces together, some acceptable and some not acceptable. This problem is exacerbated by noise in the data and errors in the alignment. Not only does the determination of connectedness become more difficult, but now the algorithm must also consider how to eliminate the noise and small alignment errors from the resulting model. Recently, however, several researchers have moved from trying to connect together surface patches from different views to using volumetric methods which hide the topological problems—making the surface-merging problem more tractable. In the next section we discuss the volumetric method which we use to solve the surface-merging problem.

### 2.4.1 Volumetric Modeling and Marching Cubes

When mentioning volumetric modeling, the first thought in most people's minds is the occupancy-grid representation. Occupancy grids are the earliest volumetric representation [95, 22] and, not coincidentally, the conceptually simplest. An occupancy grid is formed by discretizing a volume into many *voxels*<sup>5</sup> and noting which voxels intersect the object. The result is usually a coarse model that appears to be created by sticking sugar cubes together to form the object shape. Of course, if we use small enough cubes, the shape will look fine, but this becomes a problem since the amount of memory required will be  $O(n^3)$  where the volume is discretized into  $n$  slices along each dimension.

---

<sup>5</sup>*Voxel* is a common term for an individual element, rectangular box or cube, of the discretized volume. It is short for *volume element*.

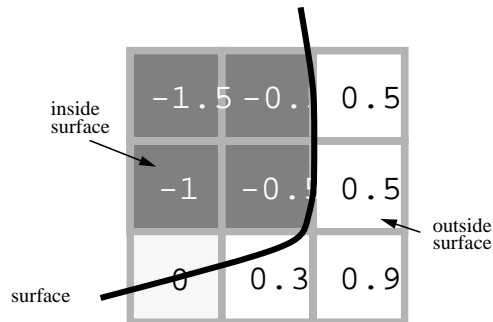


Figure 2.4: An example of zero-crossing interpolation from the grid sampling of an implicit surface.

Recently, however, an algorithm developed by Lorensen and Cline [83] for graphics modeling applications has made volumetric modeling a bit more useful by virtually eliminating the blocky nature of occupancy grids. This algorithm is called the marching-cubes algorithm [83]. The representation is slightly more complicated than the occupancy grid representation. Instead of storing a binary value in each voxel to indicate if the cube is empty or filled, the marching-cubes algorithm requires the data in the volume grid to be samples of an implicit surface. In each voxel, we store the value,  $f(\mathbf{x})$ , of the signed distance from the center point of the voxel,  $\mathbf{x}$ , to the closest point on the object’s surface. The sign indicates whether the point is outside,  $f(\mathbf{x}) > 0$ , or inside,  $f(\mathbf{x}) < 0$ , the object’s surface, while  $f(\mathbf{x}) = 0$  indicates that  $\mathbf{x}$  lies on the surface of the object.

The marching-cubes algorithm constructs a surface mesh by “marching” around the cubes while following the zero crossings of the implicit surface  $f(\mathbf{x}) = 0$ . The signed distance allows the marching-cubes algorithm to interpolate the location of the surface with higher accuracy than the resolution of the volume grid. Figure 2.4 shows an example of the interpolation.

The marching-cubes algorithm and the volumetric implicit-surface representation provide an attractive alternative to other conceivable mesh-merging schemes (see Chapter 6 for more discussion on related 3D-modeling research). First, they eliminate the global topology problem—how are the various surfaces connected—for merging views. The representation can model objects of arbitrary topology as long as the grid sampling is fine enough to capture the topology. Most importantly, the whole problem of creating the volumetric representation can be reduced to a single, simple question:

*What is the signed distance between a given point and the surface?*

The given point is typically the center of a given voxel, but we don’t really care. If we can answer the question for an arbitrary point, then we can use that same question at each voxel in the volume.

Now we may focus on two more easily defined problems:

1. How do we compute  $f(\mathbf{x})$ ?
2. How can we achieve desired resolutions and model accuracy knowing that the volumetric representation requires  $O(n^3)$  storage and computation?

The real problem underlying our simple question is that we do not have a surface; we have many surfaces, and some elements of those surfaces do not belong to the object of interest but rather are artifacts of the image acquisition process or background surfaces. In the next subsection we present an algorithm that answers the question and does so reliably in spite of the existence of noisy and extraneous surfaces in our data.

### 2.4.2 Consensus-Surface Algorithm

In this section, we will answer the question of how to compute the signed distance function  $f(\mathbf{x})$  for arbitrary points  $\mathbf{x}$  when given  $N$  triangulated surface patches from various views of the object surface. We call our algorithm the *consensus-surface algorithm*.

As described above, the positive value of  $f(\mathbf{x})$  indicates the point  $\mathbf{x}$  is outside the object surface, a negative value indicates that  $\mathbf{x}$  is inside, and a value of zero indicates that  $\mathbf{x}$  lies on the surface of the object. We can break down the computation of  $f(\mathbf{x})$  into two steps:

- Compute the magnitude: compute the distance,  $|f(\mathbf{x})|$ , to the nearest object surface from  $\mathbf{x}$
- Compute the sign: determine whether the point is inside or outside of the object

We are given  $N$  triangle sets—one set for each range image of our object as described in Section 2.2—which are aligned in the same coordinate system. The triangle sets are denoted by  $T_i$ , where  $i = 0, \dots, N - 1$ . The union of all triangle sets is denoted by  $T = \bigcup_i T_i$ . Each triangle set,  $T_i$ , consists of some number  $n_i$  of triangles which are denoted by  $\tau_{i,j}$ , where  $j = 0, \dots, n_i - 1$ .

If the input data were perfect (i.e., free of any noise or alignment errors in the triangle sets from each view), then we could apply the following *naive algorithm*, Algorithm *ClosestSignedDistance*, to compute  $f(\mathbf{x})$ :

**Algorithm** *ClosestSignedDistance*

**Input:** point  $\mathbf{x}$

**Input:** triangle set  $T$

**Output:** the signed distance  $d$

(\* Naive algorithm for computing  $f(\mathbf{x})$  by searching \*)

(\* for the closest surface from all triangles in  $T$  \*)

1.  $\langle \mathbf{p}, \hat{\mathbf{n}} \rangle \leftarrow \text{ClosestSurface}(\mathbf{x}, T)$
2.  $d \leftarrow \| \mathbf{x} - \mathbf{p} \parallel$
3. **if**  $(\hat{\mathbf{n}} \cdot (\mathbf{x} - \mathbf{p}) < 0)$

4. **then**  $d \leftarrow -d$
5. **return**  $d$

where Algorithm *ClosestSurface* returns the point,  $\mathbf{p}$ , and its normal,  $\hat{\mathbf{n}}$ , such that  $\mathbf{p}$  is the closest point to  $\mathbf{x}$  from all points on triangles in the triangle set  $T$ .

**Algorithm** *ClosestSurface*

**Input:** point  $\mathbf{x}$

**Input:** triangle set  $T$

**Output:** the point and normal vector pair  $\langle \mathbf{p}, \hat{\mathbf{n}} \rangle$

(\* Return the closest point to  $\mathbf{x}$  from all \*)

(\* points on triangles in the set  $T$ , and the normal \*)

(\* of the closest triangle. \*)

1.  $\tau \leftarrow \arg \min_{\tau \in T} \min_{\mathbf{p} \in \tau} \|\mathbf{x} - \mathbf{p}\|$
2.  $\mathbf{p} \leftarrow \arg \min_{\mathbf{p} \in \tau} \|\mathbf{x} - \mathbf{p}\|$
3.  $\hat{\mathbf{n}} \leftarrow$  outward pointing normal of triangle  $\tau$
4. **return**  $\langle \mathbf{p}, \hat{\mathbf{n}} \rangle$

The naive algorithm for  $f(\mathbf{x})$  finds the nearest triangle from all views and uses the distance to that triangle as the magnitude of  $f(\mathbf{x})$ . The normal of the triangle can be used to determine whether  $\mathbf{x}$  is inside or outside the surface. If the normal vector points towards  $\mathbf{x}$ , then  $\mathbf{x}$  must be outside the object surface. This fact can be verified by a simple proof. First, no other surface point lies within the circle of radius  $|f(\mathbf{x})|$  around point  $\mathbf{x}$ . For  $\mathbf{x}$  to be inside the object, it is necessary that every line drawn between  $\mathbf{x}$  and any point outside the object will cross a surface first. If  $\mathbf{y}$  is the closest surface point to  $\mathbf{x}$ , the line from  $\mathbf{y}$  to  $\mathbf{x}$  must cross a surface if  $\mathbf{x}$  is inside the surface. The fact that no closer surface exists excludes the possibility that any such surface exists between  $\mathbf{x}$  and  $\mathbf{y}$ .

Again, the naive algorithm will work for perfect data. However, we must consider what happens when we try this idea on real data. The first artifact of real sensing and small alignment errors is that we no longer have a single surface, but several noisy samples of a surface (see Figure 2.5). We are now faced with choices on how to proceed. Clearly, choosing the nearest triangle (as in Algorithm *ClosestSignedDistance*) will give a result as noisy as the constituent surface data. For example, a single noisy bump from one view can result in a bump on the final model, as shown in Figure 2.6 (a). Inconsistent values for the implicit distances will appear when a voxel center is on or near a surface, since the samples will be randomly scattered about the real surface location. For example, we could see three surfaces form if noise or alignment error produces an inside-outside-inside-outside (+/-/+/-) transition when, in fact, only one real surface was observed (see Figure 2.6 (b)). This is especially a problem if the noise is of similar scale to the voxel size.

With many views, the computed implicit distances from the surface will be biased towards the closest side of the surface and result in inaccurate zero-crossing interpolation during surface-mesh generation. This is a very subtle problem best explained by considering noisy samples of a surface as it crosses a line between two adjacent voxel points (see Figure 2.7). We can show mathematically that the zero-crossing interpolation will generate

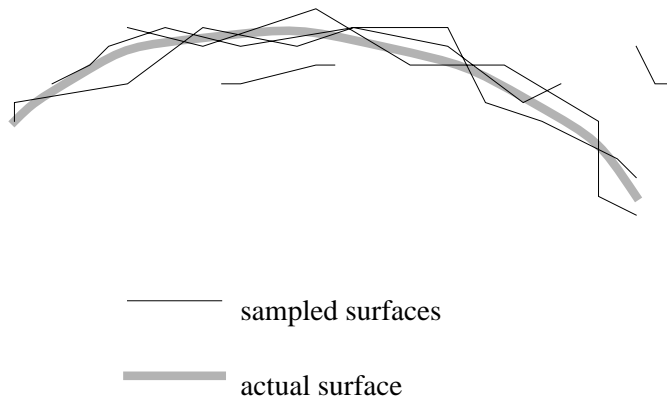


Figure 2.5: In practice, real surface samples are noisy and slightly misaligned and it is difficult to determine where the actual surface might lie.

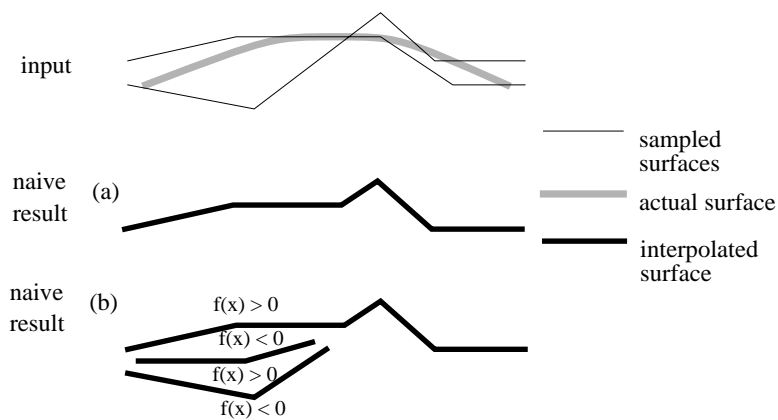


Figure 2.6: Some effects of noise and misalignment on our naive algorithm, Algorithm *ClosestSignedDistance*. In each case two observations of the same surface (actual surface is denoted by the shaded line) are shown, the resulting surface is : (a) the resulting surface is as noisy as the data, (b) three surfaces are detected when only one exists created by an inside-outside-inside-outside transition of  $f(x)$ .

significant errors. Suppose that we are evaluating  $f(x)$  (along the line  $y = z = 0$ ) at points  $x_0 = 0$  and  $x_1 = 1$ , and there is a real surface in between the two points at  $x \in [0, 1]$ . Also, let us assume that two observations are available at  $x + \epsilon$  and  $x - \epsilon$ . Assuming  $x_0$  is outside the surface (i.e.,  $f(x_0) > 0$ ), the signed distances will be

$$f(x_0) = x - \epsilon \quad (2.1)$$

$$f(x_1) = -(1 - (x + \epsilon)). \quad (2.2)$$

Using these values of  $f$ , the zero crossing will be interpolated to give an estimate,  $\hat{x}$ , of  $x$ :

$$\hat{x} = \frac{x - \epsilon}{1 - 2\epsilon}$$

The error of the estimate is

$$e_x = x - \hat{x} \quad (2.3)$$

$$= x - \frac{x - \epsilon}{1 - 2\epsilon} \quad (2.4)$$

$$= \frac{\epsilon(1 - 2x)}{1 - 2\epsilon}. \quad (2.5)$$

Thus, the error,  $e_x$ , will only be zero when  $x = 0.5$  (i.e., the real surface is exactly between points  $x_0$  and  $x_1$ ). The magnitude of the interpolation error will increase as the real surface approaches either of the points. This illustration points out the fragility of zero-crossing estimates based on inaccurate values of  $f(x)$ . If a discrete implicit surface is to be used, the estimates of  $f(x)$  must be as accurate as possible and the values must be locally consistent across surfaces. In the above scenario, a simple estimate of the average of the observations when computing  $f(x_0)$  and  $f(x_1)$  would yield the correct zero-crossing estimate.

A more sinister problem for the naive algorithm applied to real images is the existence of noise and extraneous data. For example, it is not uncommon to see triangles sticking out of a surface or other triangles that do not belong to the object. This can occur due to sensor noise, quantization, specularities and other possibly systematic problems of range imaging. Also, we must consider the fact that other incorrect triangles may be introduced by the range image triangulation process as described in Section 2.2. This makes it very easy to infer the incorrect distance and more critically the incorrect sign, which will result in very undesirable artifacts in the final surface. For example, Figure 2.8 shows how one badly oriented triangle can create an implicit distance with the incorrect sign. This results in a hole rising out of the surface as shown.

Our solution to these problems is to estimate the surface locally by averaging the observations of the same surface. The trick is to specify a method for identifying and collecting all observations of the same surface.

Nearby observations are compared using their location and surface normal. If the location and normal are within a predefined error tolerance (determined empirically), we can consider them to be observations of the same surface. Given a point on one of the observed triangle surfaces, we can search that region of 3D space for other nearby observations from

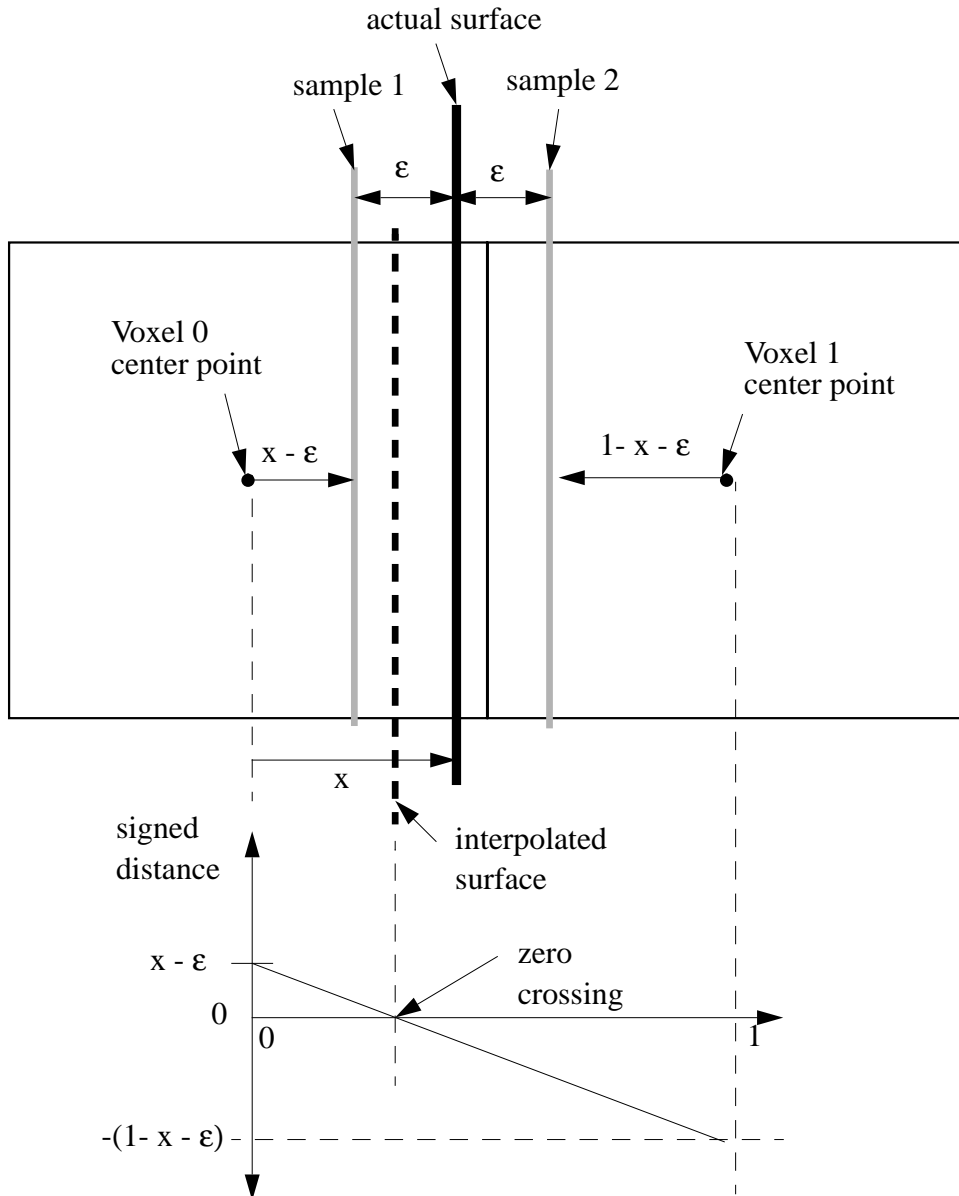


Figure 2.7: Graphical illustration of the error in zero-crossing interpolation using Algorithm *ClosestSignedDistance* with two noisy observations.

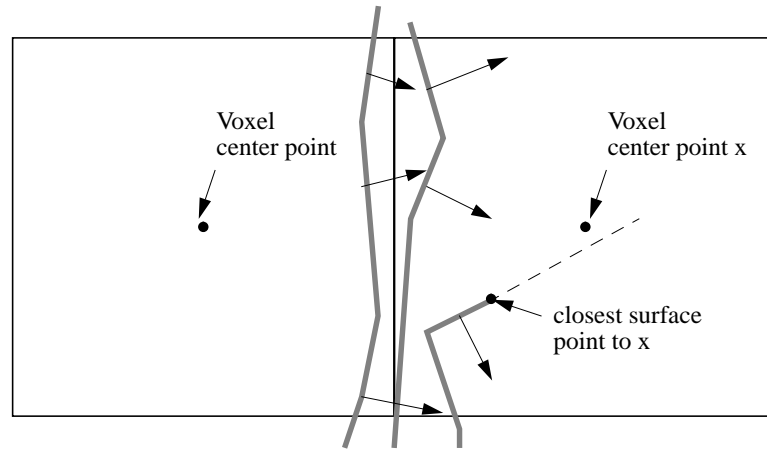


Figure 2.8: An example of inferring the incorrect sign of a voxel’s value,  $f(x)$ , due to a single noisy triangle. The algorithm incorrectly thinks point  $x$  is inside the surface based on the normal information from the closest point. The result will be a hole at that point in the surface since additional zero-crossings will result around the error at  $x$ .

other views which are potentially observations of the same surface. This search for nearby observations can be done efficiently using k-d trees [41] which is a structure for storing data of arbitrary dimensions for optimal nearest neighbors search. Here, a k-d tree is created for each view, and contains the 3D coordinates of all the triangle vertices in the view’s triangle surface set. Given a point in 3D space, we can quickly locate the nearest vertex in a given view by searching that view’s k-d tree much like a binary search [41].

If an insufficient number of observations are found, then the observation can be discarded as isolated/untrusted and the search can continue. Thus, we are requiring a quorum of observations before using them to build our model. The quorum of observations can then be averaged to produce a *consensus surface*. This process virtually eliminates the problems described previously (with respect to the naive algorithm).

As an improvement over using an equally weighted voting scheme, we can assign a confidence value  $\omega$  to each input surface triangle. A common technique is to weight the surface points/triangles from a range image by the cosine of the angle between the viewing direction and the surface normal [72]. This is simply computed by

$$\omega = \hat{v} \cdot \hat{n}$$

where  $\hat{v}$  and  $\hat{n}$  are the viewing direction and normal, respectively, of the given triangle. The consensus can now be measured as a sum of confident measures and the quorum is over this weighted sum. The rationale is that two low-confidence observations should not have the same impact on the result as two high-confidence observations. We can now



specify the consensus-surface algorithm. Instead of searching for the closest surface using Algorithm *ClosestSurface*, we can search for the closest *consensus surface*:

**Algorithm *ConsensusSignedDistance***

**Input:** point  $\mathbf{x}$

**Input:** triangle set  $T$

**Output:** the signed distance  $d$

(\* Compute the signed, implicit distance  $f(\mathbf{x})$  \*)

1.  $\langle \mathbf{p}, \hat{\mathbf{n}} \rangle \leftarrow \text{ClosestConsensusSurface}(\mathbf{x}, T)$
2.  $d \leftarrow \| \mathbf{x} - \mathbf{p} \|$
3. **if**  $(\hat{\mathbf{n}} \cdot (\mathbf{x} - \mathbf{p})) < 0$
4.     **then**  $d \leftarrow -d$
5. **return**  $d$

The only change from Algorithm *ClosestSignedDistance* is that Algorithm *ConsensusSignedDistance* computes the closest consensus-surface point and its normal in line 1. The algorithm for computing the closest consensus-surface point and its normal is as follows:

**Algorithm *ClosestConsensusSurface***

**Input:** point  $\mathbf{x}$

**Input:** triangle sets  $T_i, i = 1..N$

**Output:** the point and normal vector pair  $\langle \mathbf{p}, \hat{\mathbf{n}} \rangle$

1.  $O_{set} \leftarrow \emptyset$
- (\*  $O_{set}$  is the set of non-consensus neighbors \*)
2.  $C_{set} \leftarrow \emptyset$
- (\*  $C_{set}$  is the set of consensus neighbors \*)
3. **for** each triangulated set  $T_i$
4.     **do**  $\langle \mathbf{p}, \hat{\mathbf{n}} \rangle \leftarrow \text{ClosestSurface}(\mathbf{x}, T_i)$
5.      $\langle \mathbf{p}, \hat{\mathbf{n}}, \omega \rangle \leftarrow \text{ConsensusSurface}(\mathbf{p}, \hat{\mathbf{n}}, T)$
6.     **if**  $\omega \geq \theta_{quorum}$
7.         **then**  $C_{set} \leftarrow C_{set} \cup \langle \mathbf{p}, \hat{\mathbf{n}}, \omega \rangle$
8.         **else**  $O_{set} \leftarrow O_{set} \cup \langle \mathbf{p}, \hat{\mathbf{n}}, \omega \rangle$
9. **if**  $C_{set} \neq \emptyset$
10.     **then**  $\langle \mathbf{p}, \hat{\mathbf{n}}, \omega \rangle \leftarrow \arg \min_{\langle \mathbf{p}, \hat{\mathbf{n}}, \omega \rangle \in C_{set}} \| \mathbf{x} - \mathbf{p} \|$
11.     **else**  $\langle \mathbf{p}, \hat{\mathbf{n}}, \omega \rangle \leftarrow \arg \max_{\langle \mathbf{p}, \hat{\mathbf{n}}, \omega \rangle \in O_{set}} \omega$
12. **return**  $\langle \mathbf{p}, \hat{\mathbf{n}}, \omega \rangle$

Algorithm *ClosestConsensusSurface* examines the closest point in each view and searches for its consensus surface if one exists. After computing the closest consensus surfaces for each view, it chooses the closest of those from the consensus set  $C_{set}$ .  $C_{set}$  contains those locally averaged surfaces whose observations' confidence values sum to at least  $\theta_{quorum}$ . Note that two consensus surfaces are not differentiated based on their confidence sum  $\omega$  but rather on their proximity to  $\mathbf{x}$ . If none of the consensus surfaces exist, the algorithm selects the average surface which has the highest summed confidence out of set  $O_{set}$ .

For completeness, we outline Algorithm *ConsensusSurface* which is required by line 5 of Algorithm *ClosestConsensusSurface*. Algorithm *ConsensusSurface* basically finds all surface observations that are sufficiently similar to the given point and normal. These observations are then averaged to generate a consensus surface for the input surface. This algorithm relies on the predicate

$$\text{SameSurface}(\langle \mathbf{p}_0, \hat{\mathbf{n}}_0 \rangle, \langle \mathbf{p}_1, \hat{\mathbf{n}}_1 \rangle) = \begin{cases} \text{True} & (\| \mathbf{p}_0 - \mathbf{p}_1 \| \leq \delta_d) \wedge (\hat{\mathbf{n}}_0 \cdot \hat{\mathbf{n}}_1 \geq \cos \theta_n) \\ \text{False} & \text{otherwise} \end{cases} \quad (2.6)$$

which determines whether two surface observations are sufficiently close in terms of location and normal direction, where  $\delta_d$  is the maximum allowed distance and  $\theta_n$  is the maximum allowed difference in normal directions. Now we present the pseudo code for Algorithm *ConsensusSurface*:

**Algorithm *ConsensusSurface***

**Input:** point  $\mathbf{x}$

**Input:** normal  $\hat{\mathbf{v}}$

**Input:** triangle set  $T = \bigcup_i T_i$

**Output:** the point, normal vector, and the sum of the observations confidences  $\langle \mathbf{p}, \hat{\mathbf{n}}, \omega \rangle$

1.  $\mathbf{p} \leftarrow \mathbf{n} \leftarrow \omega \leftarrow 0$
2. **for**  $T_i \subset T$
3.     **do**  $\langle \mathbf{p}', \hat{\mathbf{n}}', \omega' \rangle \leftarrow \text{ClosestSurface}(\mathbf{x}, T_i)$
4.     **if**  $\text{SameSurface}(\langle \mathbf{x}, \hat{\mathbf{v}} \rangle, \langle \mathbf{p}', \hat{\mathbf{n}}' \rangle)$
5.         **then**  $\mathbf{p} \leftarrow \mathbf{p} + \omega' \mathbf{p}'$
6.          $\mathbf{n} \leftarrow \mathbf{n} + \omega' \hat{\mathbf{n}}'$
7.          $\omega \leftarrow \omega + \omega'$
8.  $\mathbf{p} \leftarrow \frac{1}{\omega} \mathbf{p}$
9.  $\hat{\mathbf{n}} \leftarrow \frac{\mathbf{n}}{\|\mathbf{n}\|}$
10. **return**  $\langle \mathbf{p}, \hat{\mathbf{n}}, \omega \rangle$

Note that in Algorithm *ConsensusSurface*, the definition of Algorithm *ClosestSurface* was slightly modified to also return the confidence  $\omega'$  of the closest surface triangle.

We refer to this algorithm as a whole as the consensus-surface algorithm. The following conditions are assumed:

1. Each part of the surface is covered by a number of observations whose confidences add up to more than  $\theta_{quorum}$ .
2. No set of false surfaces with a sufficient summed confidence will coincidentally be found to be similar (following the definition of Equation 2.6) or this occurrence is sufficiently unlikely.
3. Given  $N$  surface views, the real surface is closest to  $\mathbf{x}$  in at least one view.

If these assumptions are violated, mistakes in the surface mesh will result. From our experiments, a quorum requirement,  $\theta_{quorum}$ , of 1.5 to 3.0 is usually sufficient given a reasonable number of views.

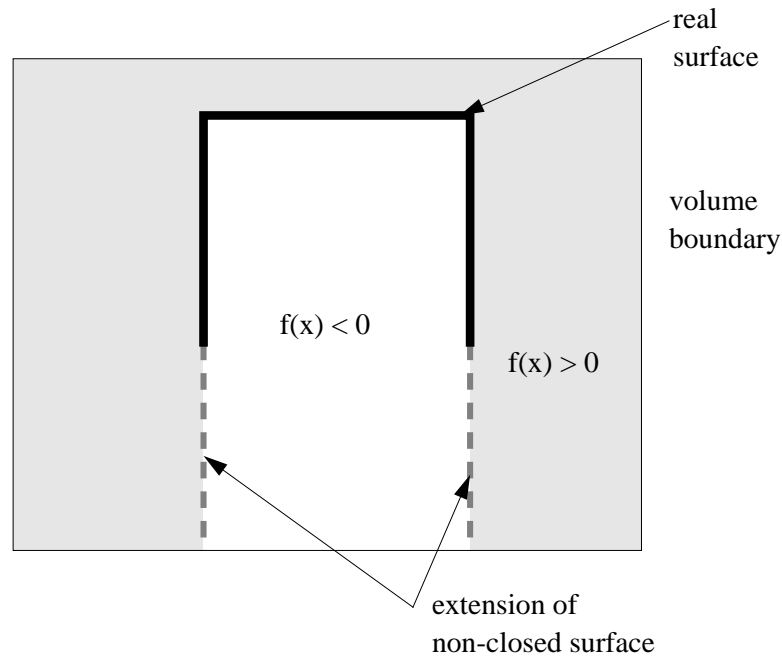


Figure 2.9: The extension of the implicit surface in unobserved regions of the grid.

### 2.4.3 Holes in the Marching-Cubes Algorithm

Using the standard marching-cubes algorithm [83] and consensus-surface algorithm as outlined above, there is a problem if there are holes or missing data. The marching-cubes algorithm works on the assumption that the surface is defined by zero crossings of the implicit surface function. It is almost always the case that parts of the object's surface are unobservable. For the regions of the volume where the surface is unobservable, the implicit surface we compute will be rather poorly justified and perhaps ill-defined.

For example, consider what happens when we sample the surfaces for all but the bottom of a cube. If we use the consensus-surface or naive algorithm described above, we will set the voxels directly underneath the cube to have a negative value and the rest positive. As shown in Figure 2.9, the effect is to literally extend the sides down to the bottom of the voxel grid. This is the best we can do using a local computation for  $f(x)$ . Essentially any non-closed boundaries of the observed surface will be extended until the side of the volume grid or another observed surface is reached.

Previous methods go to great lengths to prevent the difficulties presented by incomplete data. These workarounds involve creating special cases for dealing with regions near holes, greatly complicating implementation of the voxel filling as well as requiring a modification of the marching-cubes algorithm. For example, Curless and Levoy [28] try to detect this situation and label those voxels near holes as undefined or not on the surface. However, if

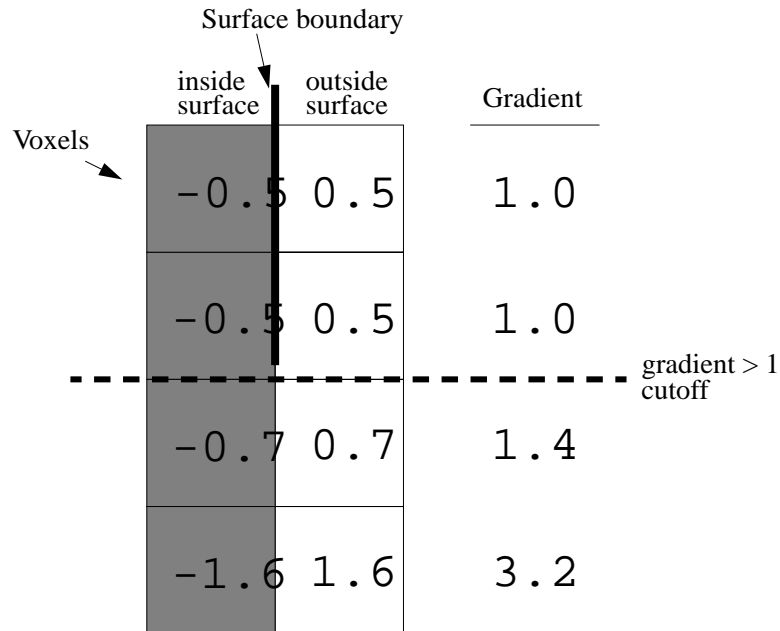


Figure 2.10: The use of gradient testing to eliminate false surfaces at zero crossings of the implicit surface, from the previous example. The gradient is labeled next to each zero crossing. Gradients greater than 1 can be ignored by the marching-cubes algorithm.

one examines the values of the distances around holes, there is one significant difference between the values around the real surface and those around holes. Around a surface, the gradient of the signed-distance function is at or near unity. However, when at a zero contour near a hole, the gradient at  $x$ ,  $\frac{\partial f}{\partial x}$ , is necessarily larger than one since the distances from points in the neighborhood of a hole must be greater than one voxel length. This fact provides a simple mechanism for eliminating holes without complicating the signed-distance function. Figure 2.10 shows our previous example of the bottomless cube with gradient values labeled over each zero-crossing—demonstrating how non-surfaces can be easily detected by the marching-cubes algorithm. We can simply test the gradient at each voxel before adding it to the surface. The gradient must be computed to find zeros already, so it is a simple matter to modify the marching-cubes algorithm to check the magnitude of the gradient.

#### 2.4.4 Accuracy and Efficiency

First, to achieve desired accuracy we must use a dense sampling of the volume. Since the memory requirements of a volume grid is cubic with respect to the density of the sampling for volumetric modeling, the first thing that gets sacrificed is accuracy. With our problem,

the standard volume-grid approach is also deficient in terms of computation time.

The straightforward use of volume grids presents several problems. One obvious problem with a voxel grid representation is that the number of voxels is  $n^3$  where each axis of the volume is discretized into  $n$  elements. This affects the achievable accuracy since we must choose the dimension to be small enough that the grid can fit in memory: we quickly reach the memory limits of our computers. In addition to storage cost, we must remember that for each voxel we must compute the signed distance; thus, the number of computations of our signed distance function  $f(\mathbf{x})$  will be cubic as well. Specifically, computation resources are wasted by computing signed distances in parts of the volume that are distant from the surface. For our purposes, the only voxels that we need to examine are those near the surface, a small fraction of the entire volume grid.

Curless and Levoy [28] alleviate this problem by run length encoding each 2D slice of the volume. This approach depends upon a complicated procedure which carves out voxels that are determined to be well outside of the surface. The procedure is tailored to their scheme for averaging the voxel values iteratively, view by view; it is not well suited to an algorithm such as ours which uses all the data simultaneously to compute each value. Their algorithm is discussed in more detail in Chapter 6.

Fortunately, there is a data structure, called an octree, that is perfect for our merging algorithm and requirements. Octrees [95] were developed as an efficient way for representing 3D occupancy grids for computer graphics and CAD modeling. An octree is a hierarchical representation of the volume in which we divide a given volume into eight octants and then we can subdivide each octant individually if necessary and so on to any level of subdivision desired.

For our purposes, we are only interested in the surface of our object, which octrees can efficiently represent. Octrees are designed just for this purpose: the sampling resolution can be adjusted to the level of detail necessary at each region of the volume. It is efficient in that respect—sampling finely near the surface and sampling coarsely away from the surface. Figure 2.11 shows a 2D slice of an octree representation of a simple surface.

The octree representation [95, 22] solves both the accuracy and the efficiency problems while keeping the algorithm implementation simple. Instead of iterating over all elements of the voxel grid, we can apply a recursive algorithm on an octree that samples more finely in octants only when necessary. To interpolate the zero crossings properly, we will need the implicit distance for the voxel containing the surface (the zero crossing) and all voxels neighboring this voxel: these voxels must all be represented at the finest level of precision. This constraint means that if we have a surface at one corner of an octant, the longest possible distance to the center of a neighboring octant is one and one-half diagonals of the voxel cube, which is a distance of  $\frac{3\sqrt{3}}{2}$  cube units.

Given the current octant, we can compute the signed distance. If the magnitude of the signed distance,  $|f(\mathbf{x})|$ , is larger than  $\frac{3\sqrt{3}}{2}$  of the octant width, then it is not possible for the surface to lie in the current or neighboring octant. If the surface is not in the current or neighboring octant, we do not care to further subdivide the current octant. The algorithm is as follows:

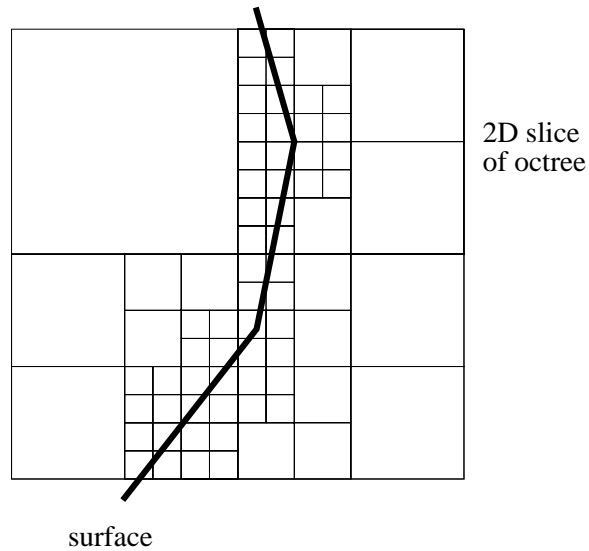


Figure 2.11: A 2D slice of an octree representation of a simple surface illustrates the adaptive resolution which is high around the surface and low elsewhere.

**Algorithm** *FillOctant*

**Input:** octant  $\langle v, \mathbf{x}, w, c \rangle$  with value  $v$ , center  $\mathbf{x}$ , width  $w$ , and children  $c$

(\* Fill the value and recursively subdivide the octant \*)

(\* to the finest required resolution. \*)

1.  $v \leftarrow \text{ConsensusSignedDistance}(\mathbf{x}, T)$
2. **if**  $(|v| < \frac{3\sqrt{3}}{2}w) \wedge (w > w_{\text{finest}})$
3.     **then** create sub-octants  $\langle v_i, \mathbf{x}_i, \frac{w}{2}, c_i \rangle$  for  $i = 0, \dots, 7$  by subdividing the current octant
4.     **for**  $i \leftarrow 0$  **to** 7
5.          $\text{FillOctant}(\langle v_i, \mathbf{x}_i, \frac{w}{2}, c_i \rangle)$
6.     **else**  $c \leftarrow \emptyset$

The octree in practice reduces the  $O(n^3)$  storage and computation requirement to  $O(n^2)$ . This is because the surfaces of 3D objects are, in general, 2D manifolds in a 3D space.<sup>6</sup>

Thus, the octree allows us to efficiently compute the implicit surface representation and uses memory efficiently—allowing us to achieve desirable levels of accuracy. The marching-cubes algorithm must be converted to manipulate octrees rather than voxels. This is handled by simply replacing the indexing of volume elements with macros that traverse the octree.

<sup>6</sup>Unless it behaves as a volume-filling surface or a porcupine—surfaces that seem to fill occupy 3D space—the surface of a 3D object will tend to sparsely occupy the 3D volume enclosing it.

### 2.4.5 Cost of the Consensus-Surface Algorithm

We can get rough estimate of the cost of our model-building algorithm by first considering the cost of the basic operation: computing a consensus surface. To simplify analysis, we assume that there are  $N$  views being merged and that for each view the triangle set  $T_i$  has  $n$  triangles.

Algorithm *ConsensusSurface* computes the closest surface for each view which on average will be an  $O(N \log n)$  operation assuming k-d trees [41] are used. Algorithm *ClosestConsensusSurface* computes the closest surface and then the respective consensus surface for each view, which adds up to a cost of  $O(N^2 \log n)$ . Since  $N$  will usually be much smaller than  $n$ , this operation is relatively cheap. Algorithm *ClosestConsensusSurface* is performed for each voxel or octree element.

Assuming that an  $M \times M \times M$  voxel grid is used, the modeling algorithm will cost  $O(M^3 N^2 \log n)$ . However, if octrees are used we may loosely assume that the number of voxels or octree elements which are evaluated will be proportional to the surface area of the object. For sake of approximation, we may assume that the area is  $O(M^2)$  where  $M$  is the number of elements in the equivalent voxel grid. This reduces the complexity of our modeling algorithm to  $O(M^2 N^2 \log n)$  which is a significant reduction since  $M$  will be relatively large in practice to enable accurate modeling.

## 2.5 3D Object Modeling Results

Here we present some experimental results of our implementation of the 3D object modeling algorithm described in this chapter.

The major limitation of our modeling system is the requirement for calibrated object positioning. Our calibrated image acquisition system—a Unimation Puma robot, and our range sensor, an Ogis light-stripe range finder—limits the objects which we are able to model. Due to this, the objects must be small enough to be imaged by the range finder and to be mountable on the Puma. As described in Chapter 1, we assume that the objects are rigid and opaque (lucent surfaces are not usually detectable by the range finder). However, despite these limitations there remains a large class of objects which we can use to test our model-building algorithm.

Since the object must be physically attached to the Puma, we are further limited by the surface area of the object which we can effectively observe. For this work, we do not attempt to model the undersides of the objects. A process of reattaching the object and aligning the new views would be required. Though it is feasible, reattaching the object to model its underside was not fundamental for testing our ideas.

For our experiments, we selected 5 objects to model using our system: a toy boxcar, a rubber duck, a piece of fruit, a ceramic mug, and a toy car. For each object, we manually determined the number of range images of the object to 1) maximally cover the viewable surface of the object, and 2) provide a sufficient amount of overlap between views for the

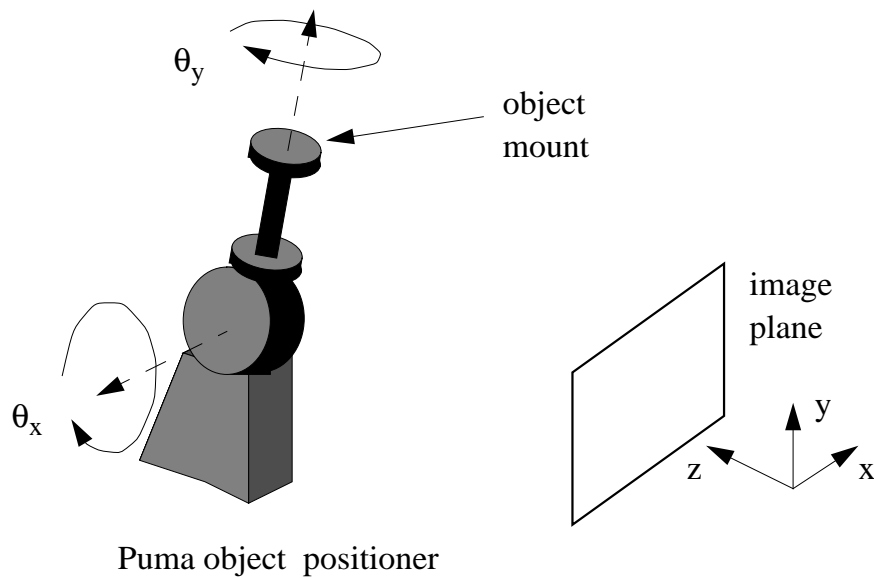


Figure 2.12: The two degrees of freedom,  $\theta_x$  and  $\theta_y$  of the Puma used to vary of object position with respect to the camera's coordinate system.

consensus-surface algorithm. The number of views required is related to the geometric complexity of the object: varying from 18 for the boxcar to 54 for the toy car.

The views were acquired by varying the angles of two rotation axes on the Puma's end effector: rotating around the camera's y-axis direction and rotating around the camera's x-axis direction. The Puma robot is capable of 6 degrees of freedom but for this work using only two was sufficient to detect the visible surfaces of our objects. We refer to these rotations as  $\theta_y$  and  $\theta_x$  respectively. Figure 2.12 shows a diagram of the rotational degrees of freedom used in our experiments. Generally, we would vary  $\theta_y$  from -180 degrees to 160 in increments of 20 degrees and would vary  $\theta_x$  from anywhere from -30 degrees to +30 degrees in 20 degree increments as well. For objects like the mug, we would have to add some views to observe difficult to view surfaces such as the bottom of the inside of the mug.

Each range image contained  $256 \times 240$  pixels with each pixel containing a 3D coordinate. The resolution of data is approximately 1 mm (i.e., the distance between two pixels on a flat surface at the nominal distance from the camera is roughly 1 mm). The accuracy of data is on the order of roughly 0.5 mm.

The results of our modeling algorithm for each object are shown in Figures 2.13- 2.22. Each of these figures show:

- an intensity image of the object



- a close-up of some of the triangulated range images used as input to the consensus-surface algorithm (shaded to better indicate the roughness of the original data)
- a slice of the volume grid where the grey-scale indicates the proximity to a surface point (black closest, white furthest)
- three views of the resulting triangulated model

The relevant statistics of the modeling experiments for each object are presented in Table 2.1. These statistics include the number of input images and triangles, the number of triangles in the resulting model, the resolution of the voxel/octree grid, the percentage of voxels in the volume grid which were actually represented in the octree structure, the execution time on an SGI Indy 5 (a 124 MIPS/49.0 MFLOPS machine), and the parameters for our consensus-surface algorithm (the quorum requirement  $\theta_{quorum}$ , the maximum distance,  $\delta_d$ , between similar points and the maximum angle,  $\theta_n$  between normal vectors of similar points). The volume grid was divided into at most 128 cubes along each dimension. Of the parameters used by the modeling algorithm, the quorum requirement parameter is the most difficult to determine. Proper choice of the quorum parameter depends on the number of views, the geometry of the object (i.e., how many views in which a given patch of the surface is visible), and the noise in the data. Setting this parameter automatically would be a difficult problem. It is very similar to the view-selection problem: how to choose an appropriate set of views to cover an object when the geometry is unknown.

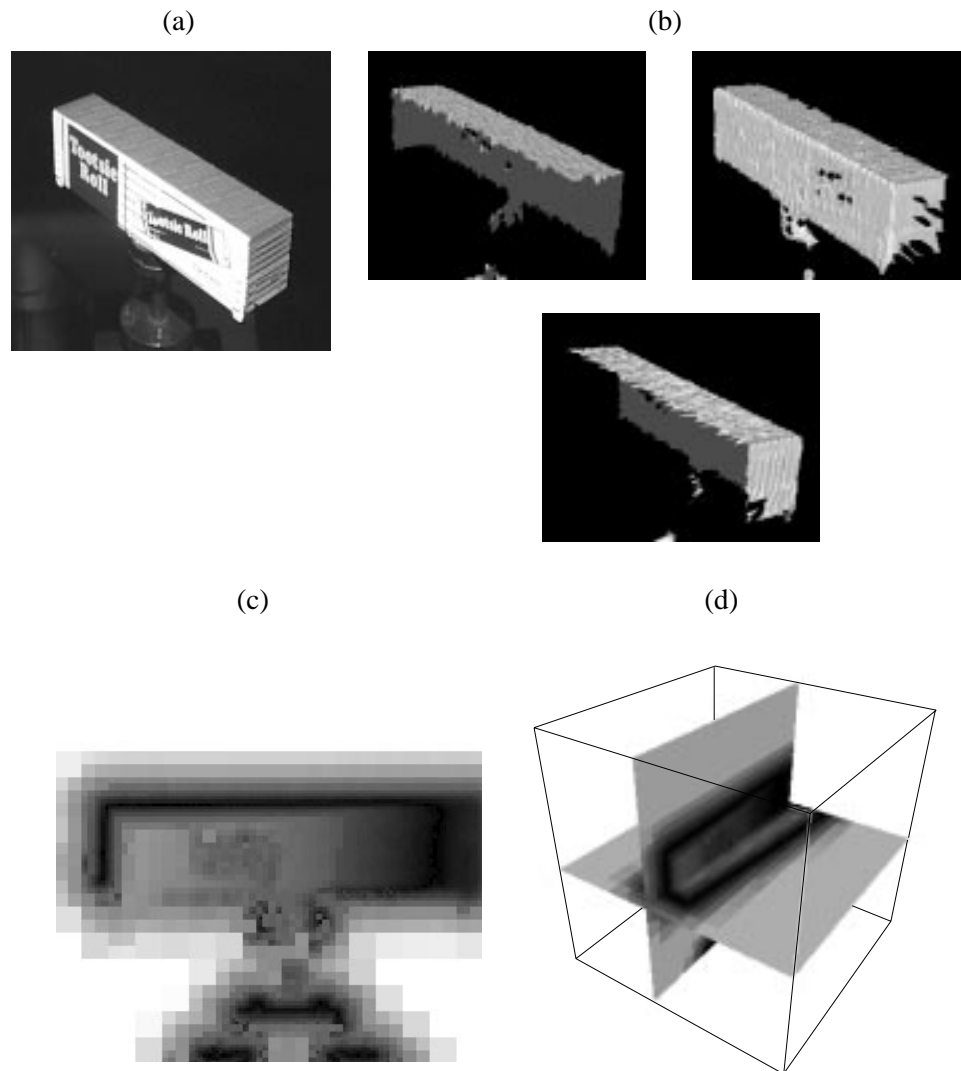


Figure 2.13: Results from modeling the boxcar. (a) An intensity image of the boxcar, (b) a close-up of some of the triangulated range images used as input to the consensus-surface algorithm, (c) a slice of the implicit-surface octree volume where the grey-scale indicates the proximity to a surface point (black closest, white furthest), and (d) a 3D view of two cross sections of the implicit-surface octree volume.

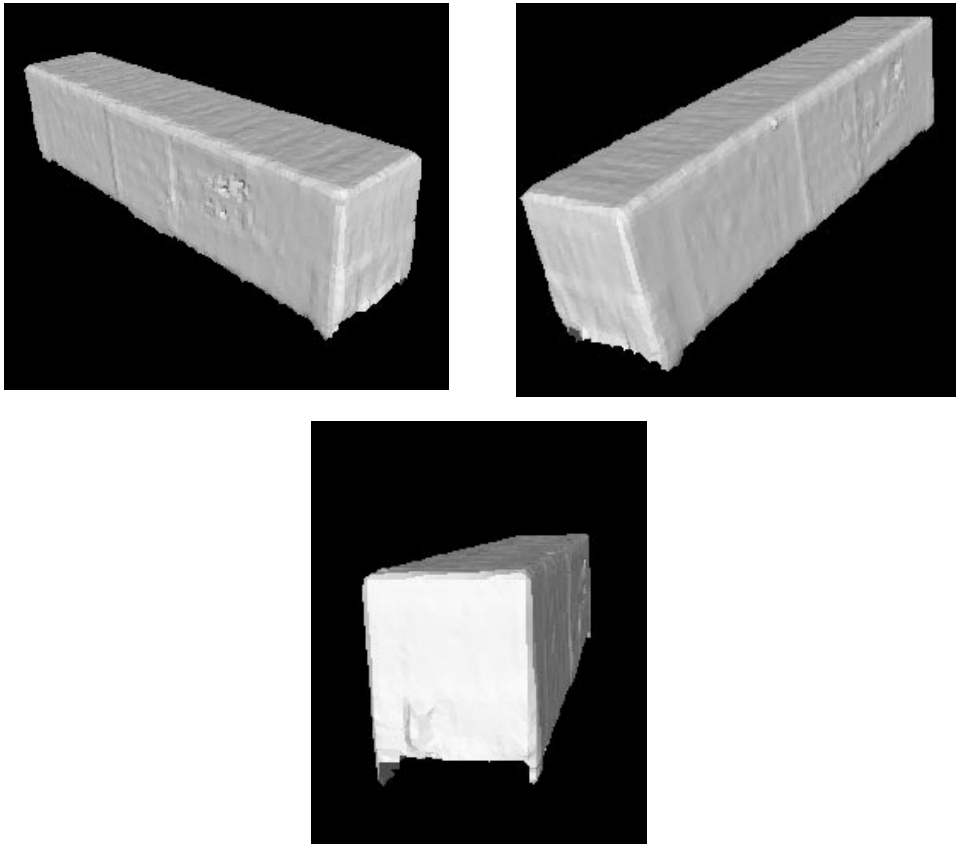


Figure 2.14: Three views of the resulting triangulated model of the boxcar.

The resulting triangulated models had to be cleaned to remove data corresponding to the Puma mounting device, which, despite the fact that its surface is black, was rather cleanly reproduced in the resulting models. If another object has the same rigid motion as the object being modeled, our algorithm considers this other object to be part of the object being modeled; it is unable to distinguish between objects with identical motion with respect to the camera.<sup>7</sup> Holes on the undersides of the objects were also filled during the cleaning process.

As an example of what the naive algorithm, Algorithm *ClosestSignedDistance* of Section 2.4.2, would produce we show the example of the the result of the naive algorithm on the duck data set in Figure 2.23. Notice how many extraneous surfaces exist near the duck

---

<sup>7</sup>Distinguishing two objects that have the same rigid motion is a fundamental limitation of algorithms that build models from sequences of data. This is the fundamental advantage of modeling by moving the object rather than modeling by moving the camera around the object. When the object is moved with respect to the camera, the rest of the background is unlikely to follow the same motion and thus will not be consistently detected by the data merging algorithm.

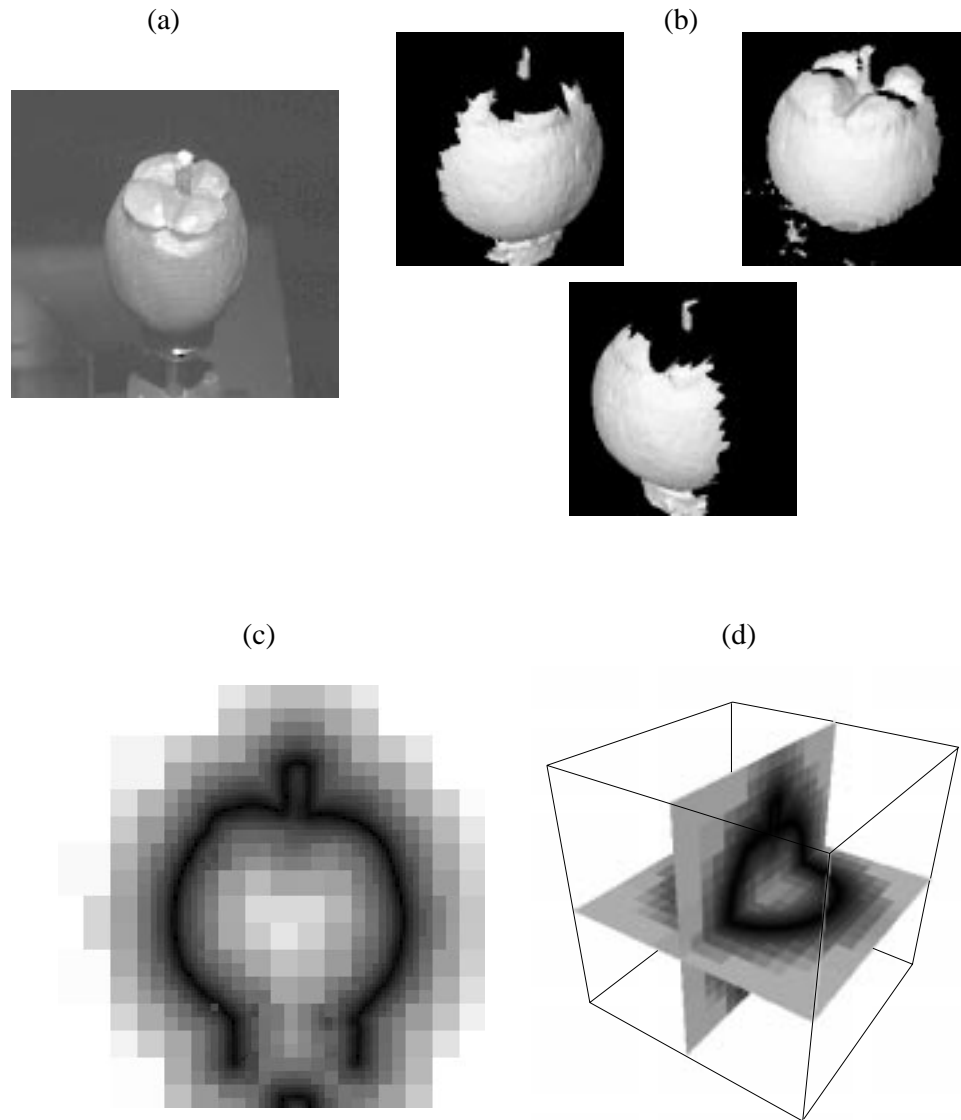


Figure 2.15: Results from modeling the fruit. (a) An intensity image of the fruit, (b) a close-up of some of the triangulated range images used as input to the consensus-surface algorithm, (c) a slice of the implicit-surface octree volume where the grey-scale indicates the proximity to a surface point (black closest, white furthest), and (d) a 3D view of two cross sections of the implicit-surface octree volume.

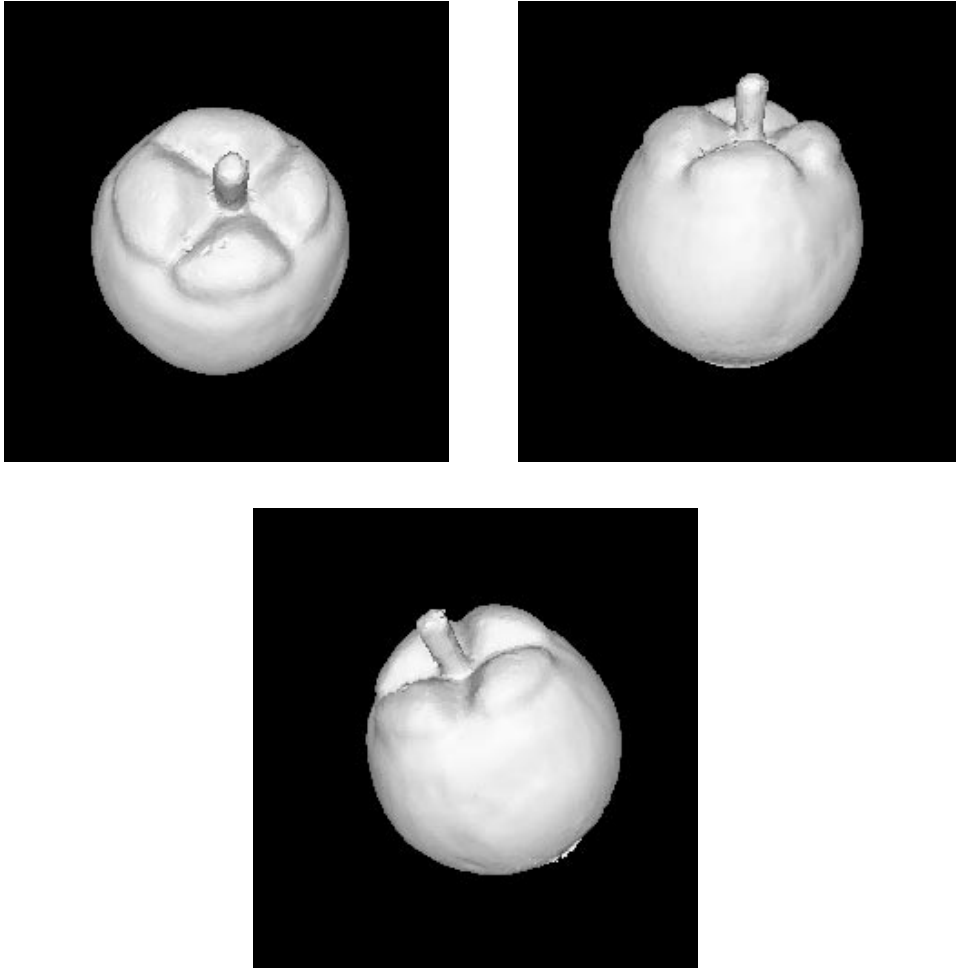


Figure 2.16: Three views of the resulting triangulated model of the fruit.

Object	Images	Tris In	Tris Out	Voxel Res.	Octree Usage	Time (minutes)	$\theta_{quorum}$	$\delta_d$ (mm)	$\theta_n$ (degrees)
boxcar	18	300k	23k	1.6mm	6%	17	1.5	2	45
fruit	36	370k	49k	1mm	6.8%	39	1.5	2	53
duck	48	555k	27k	1.8mm	4%	52	2.25	3	45
mug	50	680k	24k	2.5mm	23%	48	2.5	3	45
car	54	747k	26k	2mm	5.5%	86	1.5	2	53

Table 2.1: Statistics of the modeling experiments for each object.

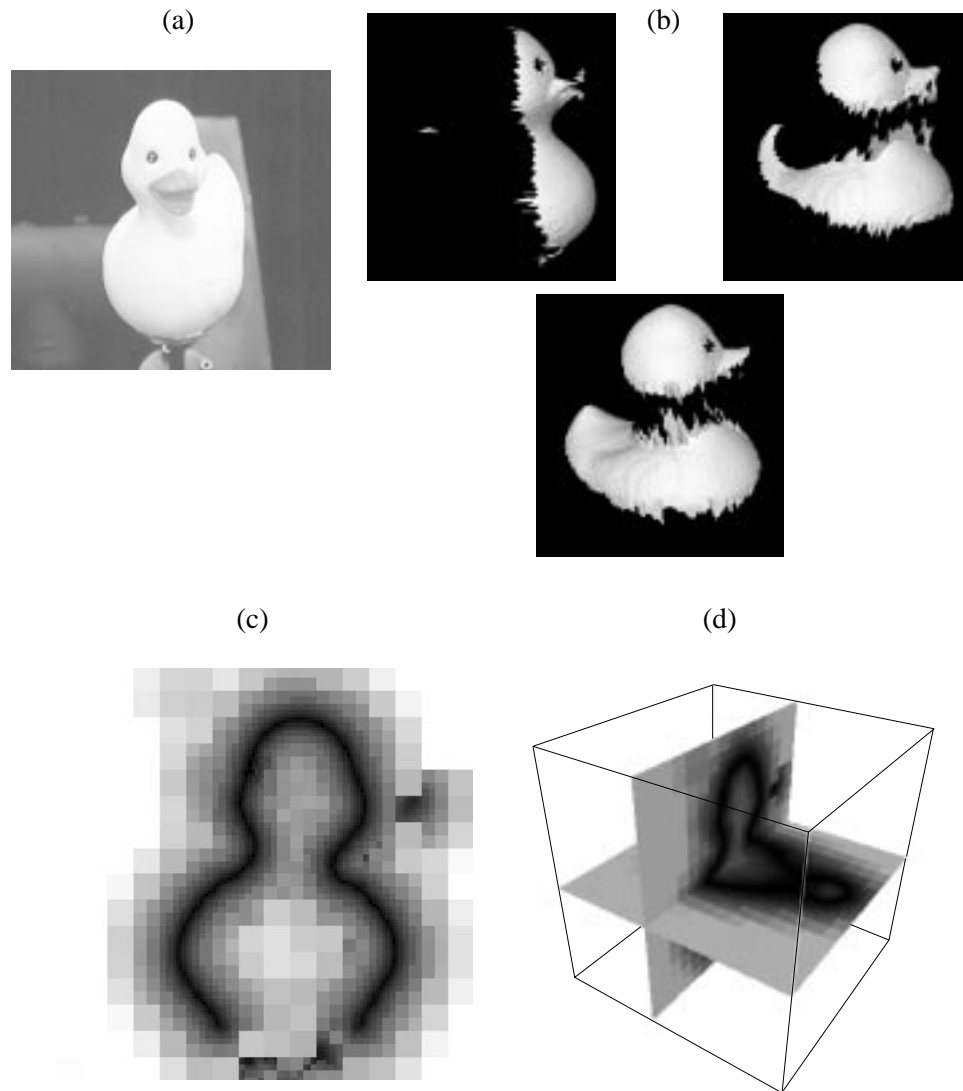


Figure 2.17: Results from modeling the rubber duck. (a) An intensity image of the duck, (b) a close-up of some of the triangulated range images used as input to the consensus-surface algorithm, (c) a slice of the implicit-surface octree volume where the grey-scale indicates the proximity to a surface point (black closest, white furthest), and (d) a 3D view of two cross sections of the implicit-surface octree volume.

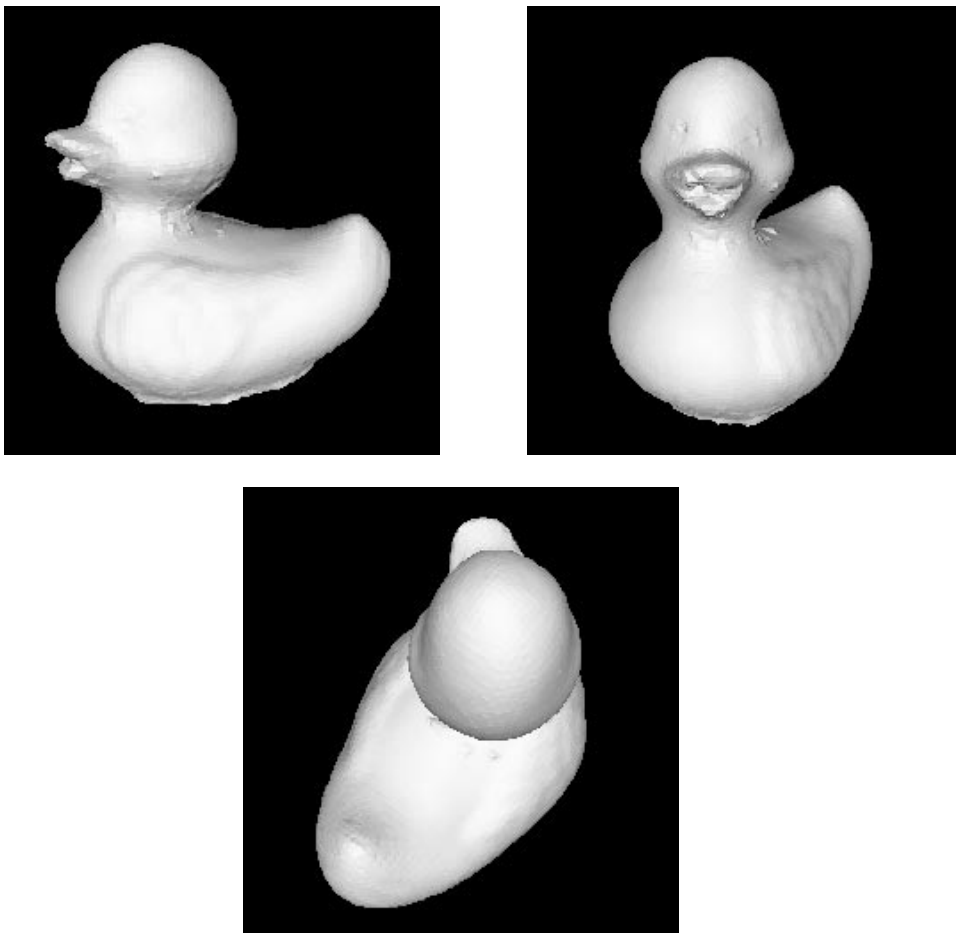


Figure 2.18: Three views of the resulting triangulated model of the duck.

from the input range-image data. Also notice the large number of holes and bumps over the resulting surface. The naive algorithm fails because it trusts that every surface observation is an accurate observation of the object surface. As can be seen from the sample range data of the duck in Figure 2.17, this is not the case.

To more clearly illustrate the accuracy of our modeling algorithm, Figures 2.24- 2.28 show cross sections of our final models and the original input range-image data. These examples demonstrate the ability of our consensus-surface algorithm to accurately locate the surface in very noisy data.

The range-image data was most noisy in dark regions of an object and regions of specular reflection and interreflection. For the most part, the consensus-surface algorithm was able to make sense of the data in spite of these significant errors. Small bumps sometimes resulted in those regions (e.g., on the boxcar and the duck) but that is to be expected when

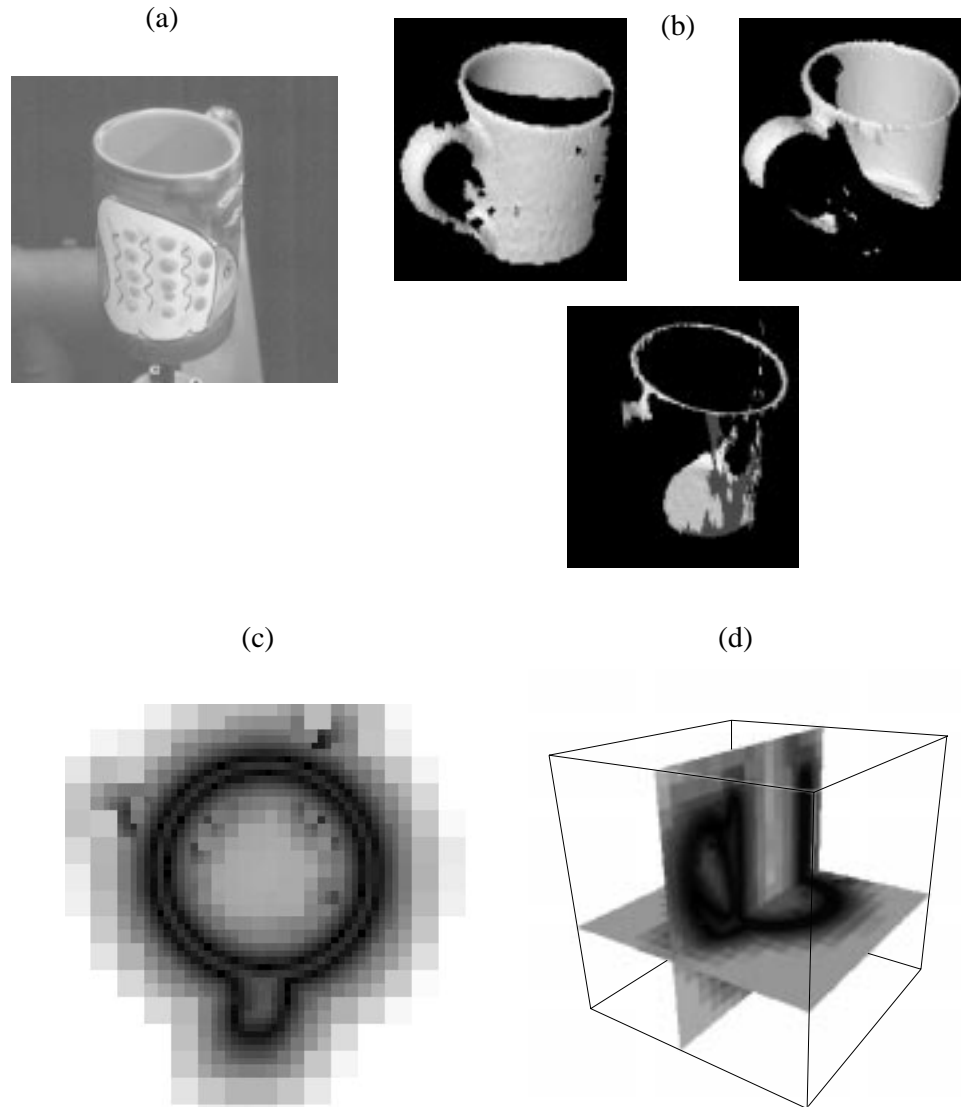


Figure 2.19: Results from modeling the ceramic mug. (a) An intensity image of the mug, (b) a close-up of some of the triangulated range images used as input to the consensus-surface algorithm, (c) a slice of the implicit-surface octree volume where the grey-scale indicates the proximity to a surface point (black closest, white furthest), and (d) a 3D view of two cross sections of the implicit-surface octree volume



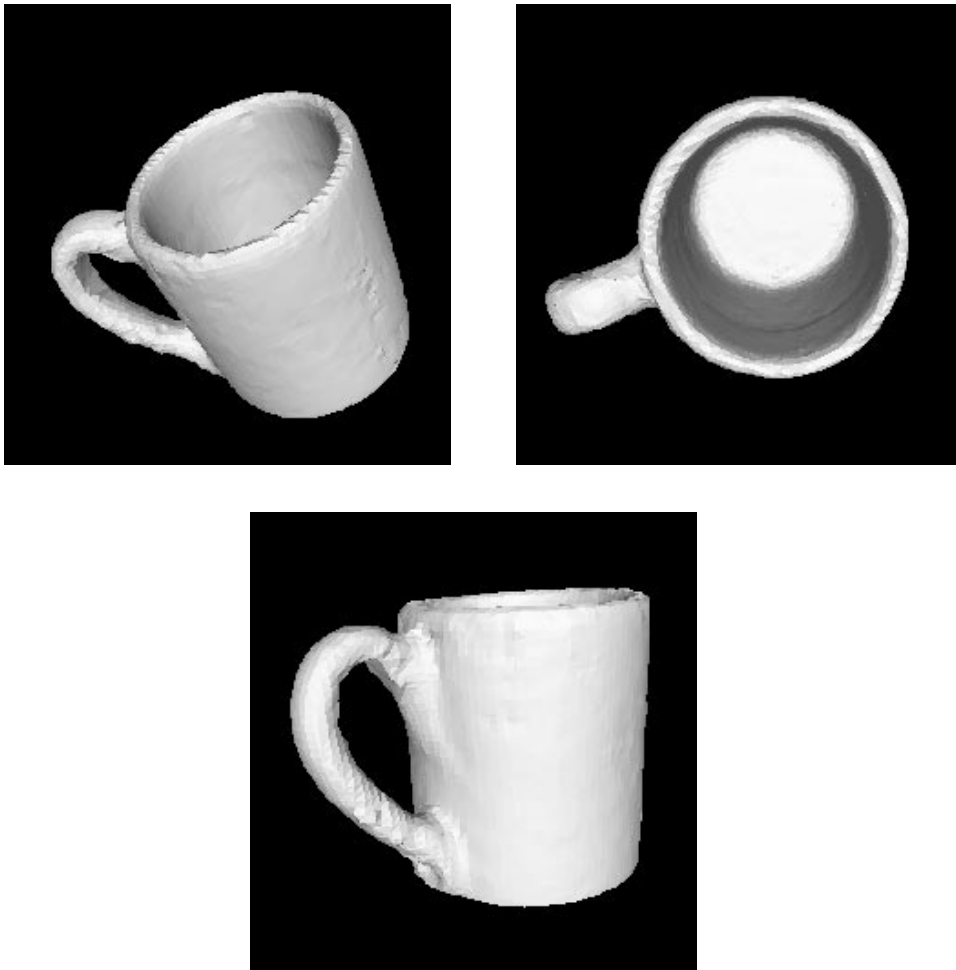


Figure 2.20: Three views of the resulting triangulated model of the mug.

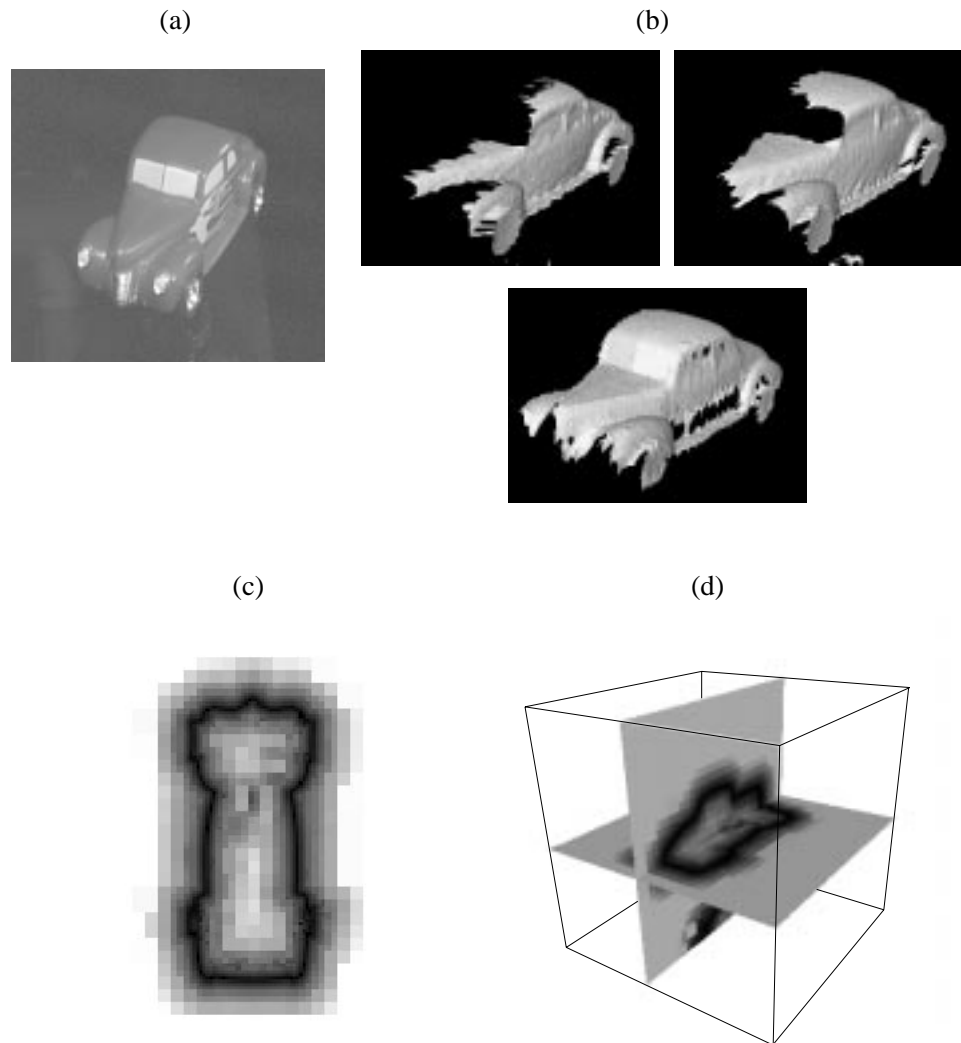


Figure 2.21: Results from modeling the toy car. (a) An intensity image of the toy car, (b) a close-up of some of the triangulated range images used as input to the consensus-surface algorithm, (c) a slice of the implicit-surface octree volume where the grey-scale indicates the proximity to a surface point (black closest, white furthest), and (d) a 3D view of two cross sections of the implicit-surface octree volume.

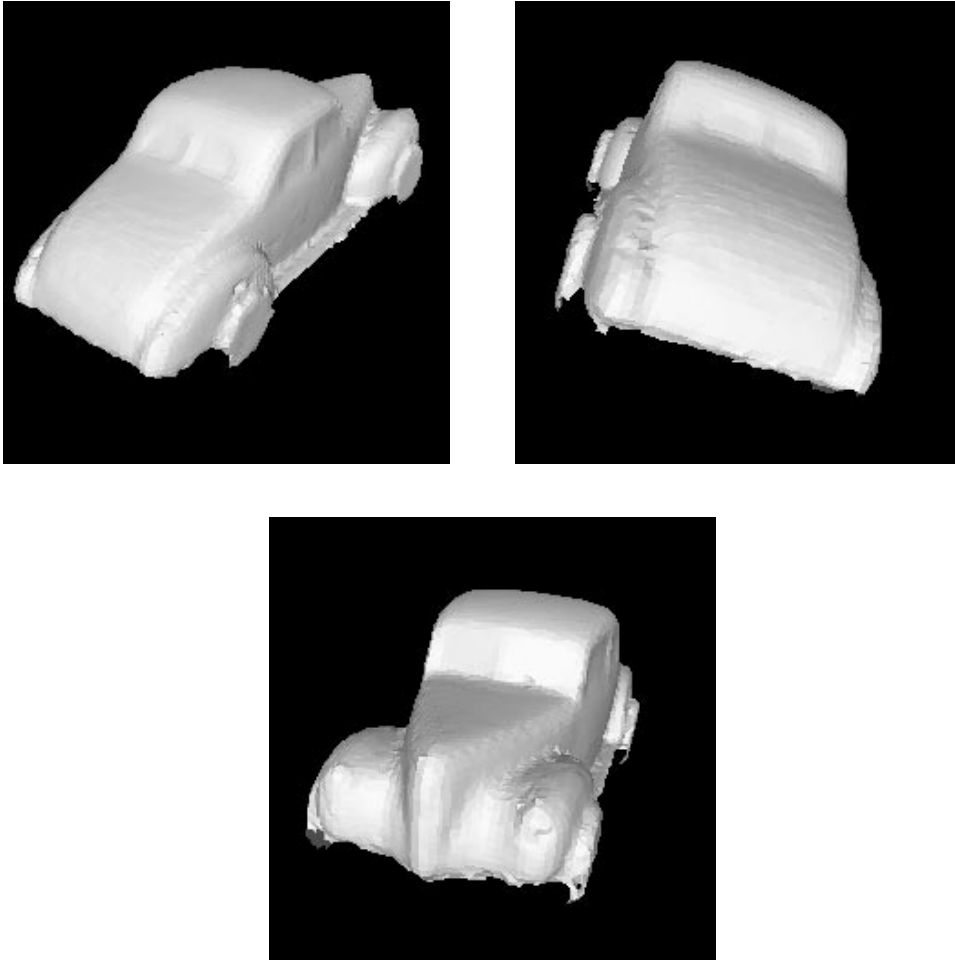


Figure 2.22: Three views of the resulting triangulated model of the car.

the data is consistently bad in a region. However, for the quality of the data, the model surfaces in those regions are still very good. One problem we were unable to solve was modeling the wheels on the toy car. The chrome wheel hubs refused to be imaged by our light-stripe range finder. This is due to the highly specular nature and interreflections on the wheel hub which gave no image data resembling a wheel surface from any views. To work around this, we placed white tape over the wheels of the car.

## 2.6 3D Modeling: Summary

We have described a method to create a triangulated surface mesh from  $N$  range images. Robotic calibration is used to acquire images of the object under known transformations,

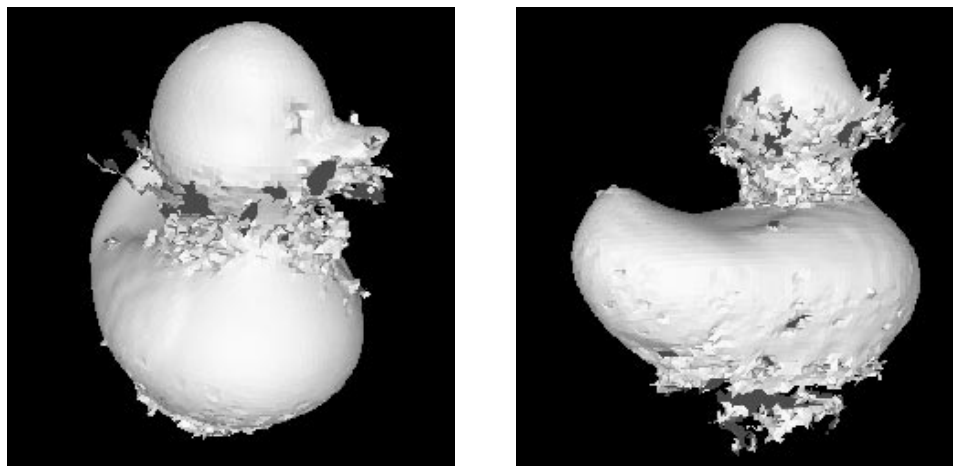


Figure 2.23: The result of the naive algorithm, Algorithm *ClosestSignedDistance*, on the duck image.

allowing us to align the images into one coordinate frame reliably. Our method for data merging takes advantage of the volumetric implicit-surface representation and the marching-cubes algorithm to eliminate topological problems.

The main contribution of this chapter is our algorithm for merging data from multiple views: the consensus-surface algorithm which attempts to answer the question

*What is the closest surface to a given point?*

With the answer to this question, we can easily compute the signed distance  $f(\mathbf{x})$  correctly. While other known methods (described in detail in Chapter 6) also implicitly address this question, their algorithms do not capture the essence of the problem and produce answers by taking averages of possibly unrelated observations. In contrast, our algorithm attempts to justify the selection of observations used to produce the average by finding a quorum or consensus of locally coherent observations. This process eliminates many of the troublesome effects of noise and extraneous surface observations in our data.

Consensus surfaces can be computed independently for any point in the volume. This feature makes it very easy to parallelize and allows us to straightforwardly use the octree representation. The octree representation enables us to model objects with high accuracy with greatly reduced computation and memory requirements. By modifying the marching-cubes algorithm to do a simple gradient test at zero crossings, we also are able to avoid special cases in our algorithm.

We have presented the results of our modeling algorithm on a number of example problems. These results demonstrate that our consensus-surface algorithm can construct

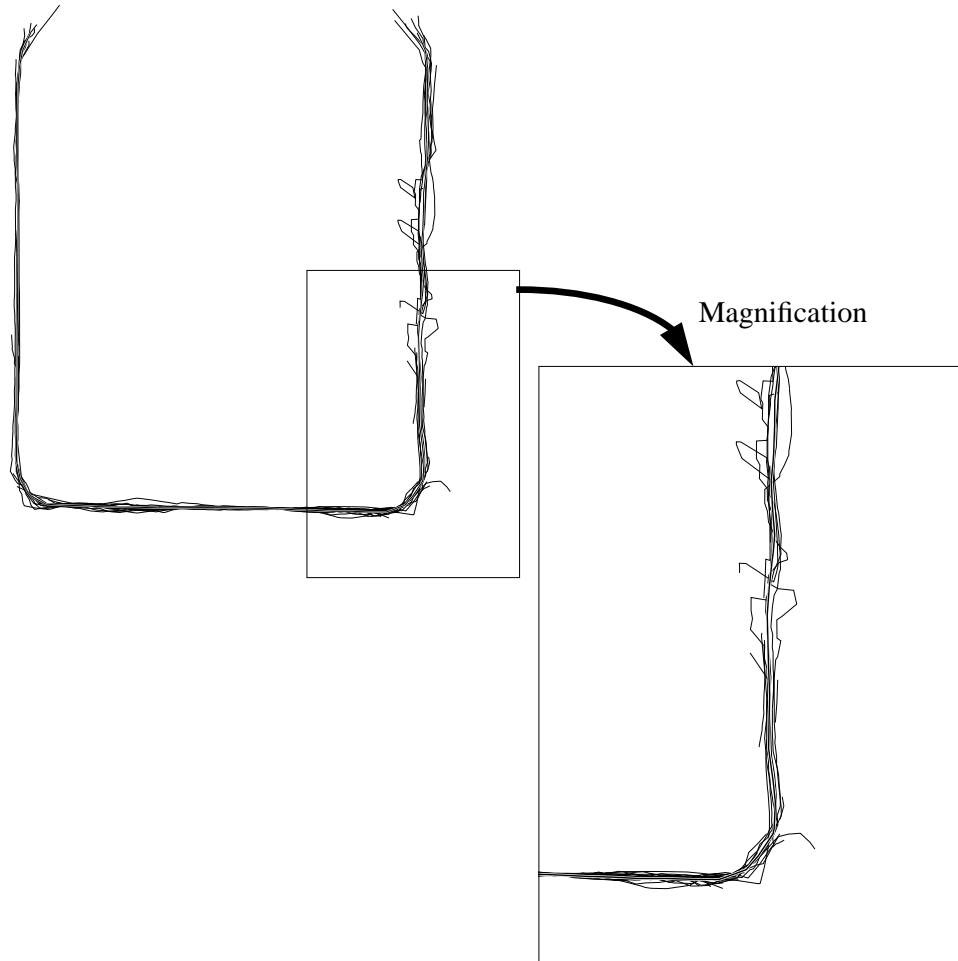


Figure 2.24: A cross section of the final model of the boxcar (thick black line) and the original range-image data (thin black lines) used to construct it.

accurate geometric models from rather noisy input range data and somewhat imperfect alignment.

We now discuss our algorithm for 3D-3D object localization.

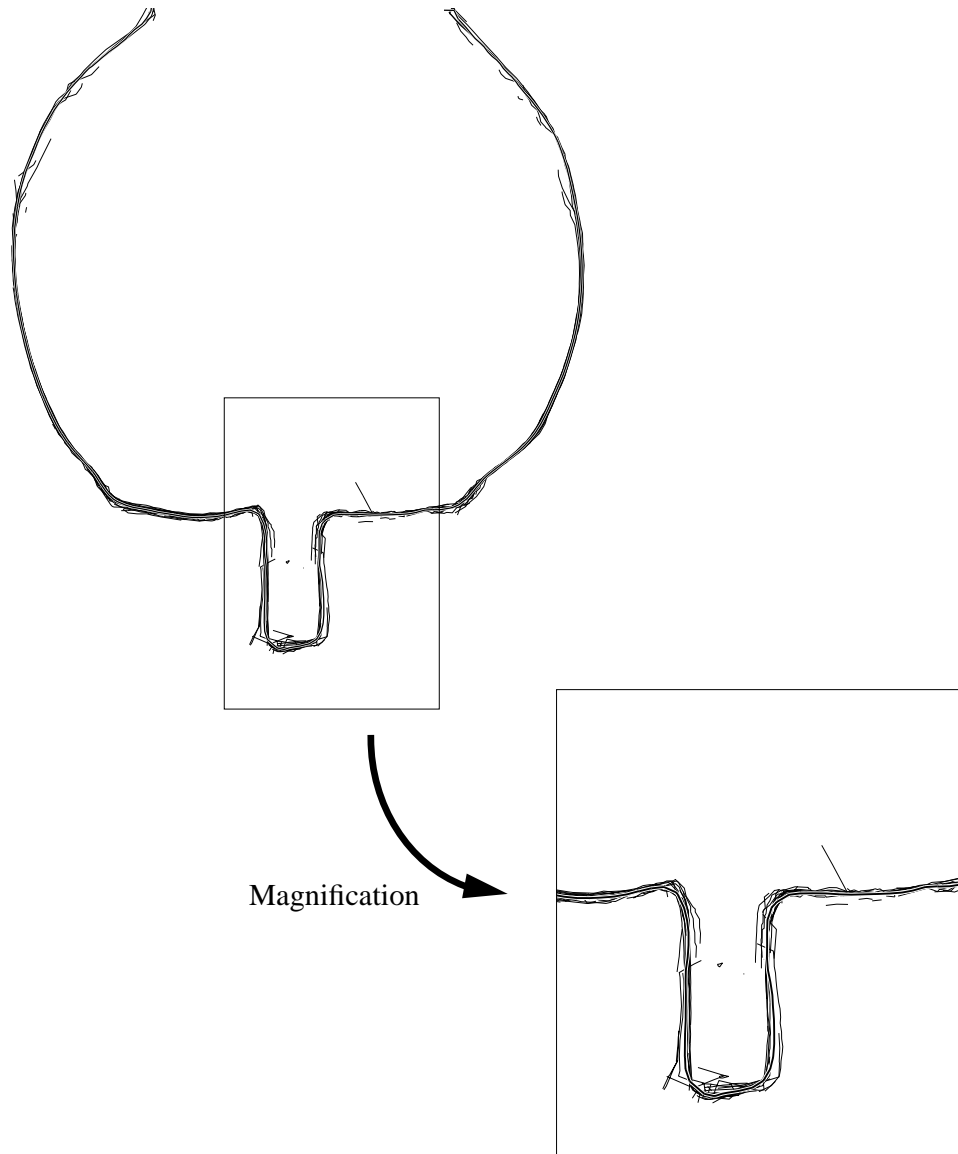


Figure 2.25: A cross section of the final model of the fruit (thick black line) and the original range-image data (thin black lines) used to construct it.

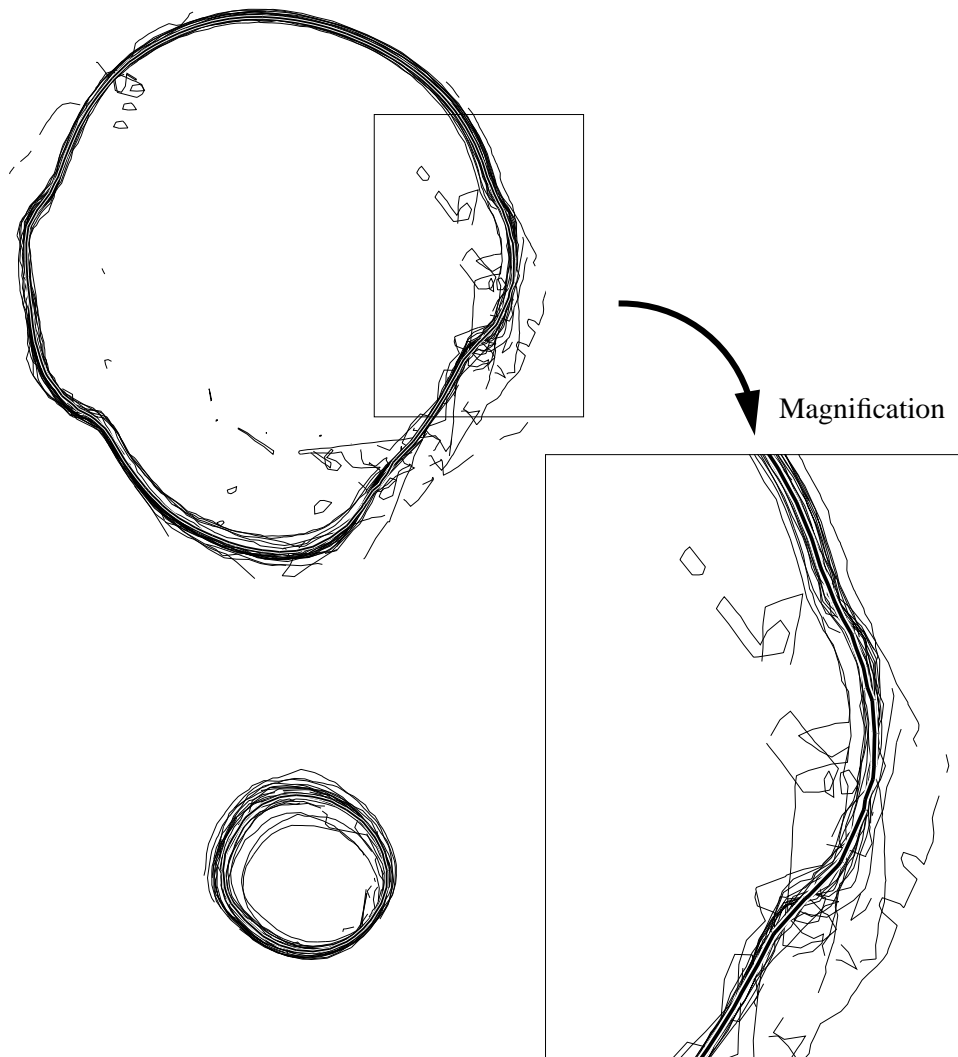


Figure 2.26: A cross section of the final model of the rubber duck (thick black line) and the original range-image data (thin black lines) used to construct it.

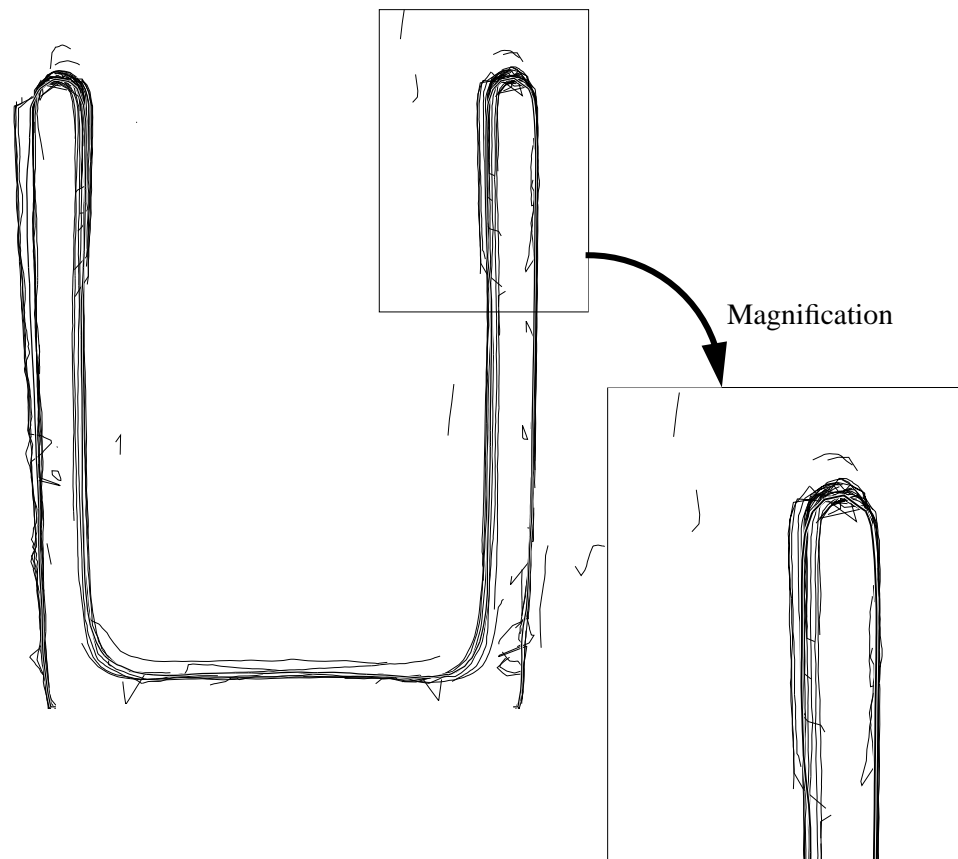


Figure 2.27: A cross section of the final model of the ceramic mug (thick black line) and the original range-image data (thin black lines) used to construct it.



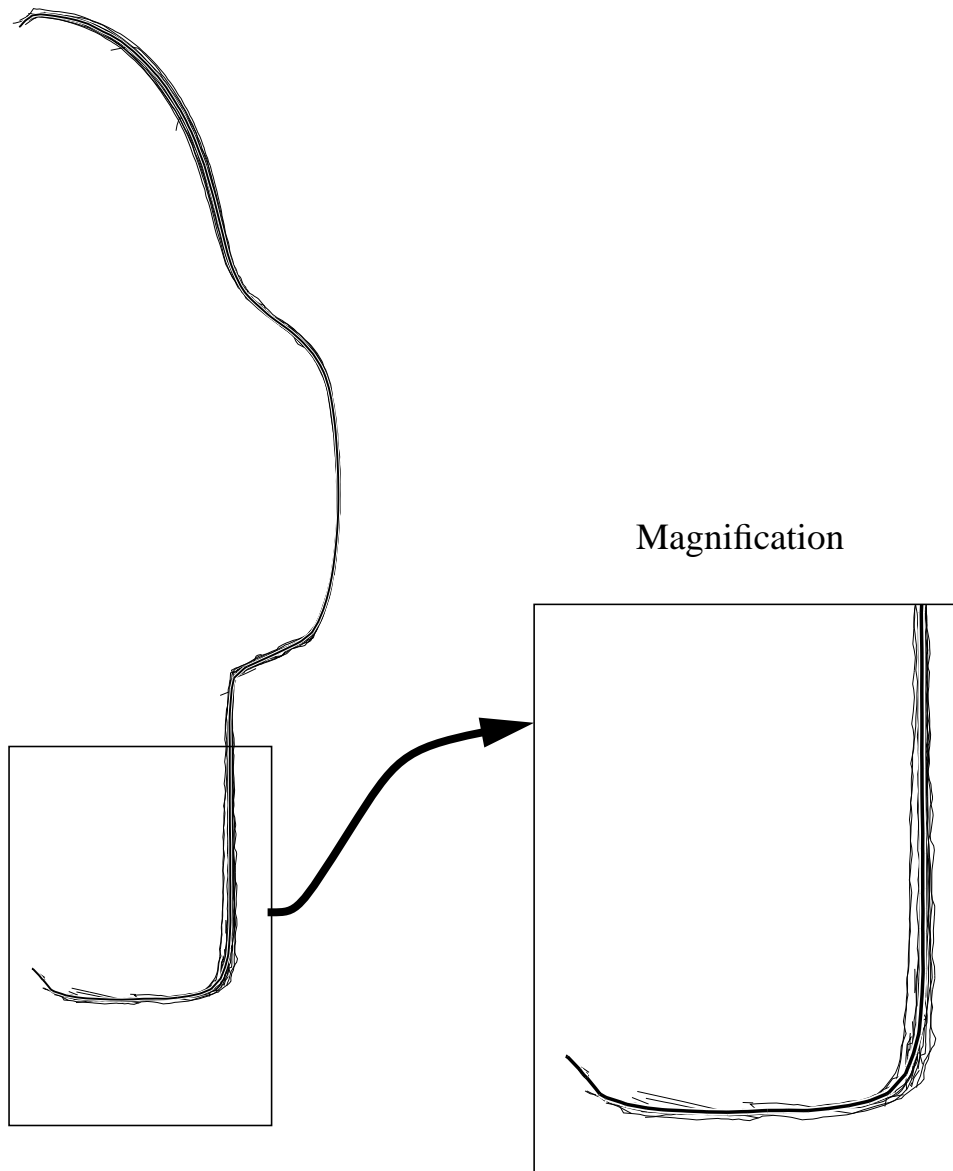


Figure 2.28: A cross section of the final model of the toy car (thick black line) and the original range-image data (thin black lines) used to construct it.

# Chapter 3

## 3D-3D Object Localization

The main goal of this thesis is to localize a known object in an image given a rough estimate of the object’s pose. In the previous chapter, we described a method for automatically building a 3D triangulated surface model from multiple range-image views. In this chapter, we will detail our approach to using such models for the task of localizing 3D objects in 3D range-image data—3D-3D object localization<sup>1</sup>.

For localization to be useful for recognition and tracking applications, it must be an efficient, robust operation and should be applicable to a wide variety of object shapes. There are many subtle problems to be solved to achieve robust and efficient localization in practice. We begin this chapter by briefly overviewing the problems involved and our approach to solving them.

### 3.1 Approach

Here, we assume that we are given an accurate, triangulated model of our 3D object’s surface (using the techniques presented in Chapter 2 or an appropriately triangulated CAD model), a range image, and a rough estimate of the object’s pose in the image. The localization task is to estimate the precise pose of the object in the range image. In approaching this task, we regard as axiomatic that localization is an optimization problem: we can evaluate any pose estimate, and the true pose has the optimal value. The primary problem is how to evaluate a pose candidate. Once pose candidates can be evaluated the next problem is how to efficiently and effectively search for the best pose candidate.

Our first decision is to evaluate the pose by measuring the distance between points on the model and points in the image. The rationale is that the range image provides us with samples of visible surface points in three dimensions. Thus, it makes sense to match points on the model surface with their samples in the image and measure the distance between them. In general, we use one point per triangle in the model though we could easily sample

---

<sup>1</sup>Earlier versions of this worked appeared in [144, 145, 146, 148].

more or fewer points.<sup>2</sup>

We prefer to rely on low-level data available from the sensor rather than higher-level features inferred from the data. 3D points (available directly from range images) are the simplest possible feature and points suit our purposes nicely: they are efficient to process and manipulate and relatively easy to match with one another. Points are also a very general representation of shape. If higher-level features are used, the shapes that could be modeled would most certainly be restricted (e.g., using algebraic surfaces).

At the highest level, our approach to localization comprises the following steps:

- Predict the appearance of model points in the image
- Match the model points to image points
- Refine the pose estimate using the point matches

The first problem we face is how to efficiently compute the visibility of the points of the model with respect to the range-image view. We present a local approximation method for efficiently predicting the visibility of points given the pose of the object and camera parameters. This method is general for all standard camera projection models and obviates the need for expensive ray-casting or z-buffering.

The second issue is how to compute correspondences between model and image points. We describe the use of k-d trees [41] to perform nearest-neighbor searches for efficiently computing these correspondences. We also describe a method for extending the nearest-neighbor search to consider attributes other than 3D location to improve the accuracy of correspondences when the error in the initial pose estimate is high.

The third problem is dealing with incorrect correspondences and noise. Our reason for computing the correspondences is to use them to refine our estimate of the pose. This closely resembles the classic pose estimation problem—computing the optimal pose from a set of correspondences. In pose estimation, the correspondences are usually assumed to be correct but that the data is possibly corrupted by noise (e.g., Gaussian noise). Here we are faced with a more difficult task—dealing with incorrect correspondences as well as noise, and no fixed/precomputed correspondences. Knowing that many of our correspondences will be incorrect, we draw upon the field of robust statistics [67, 96] to create a solution that is relatively insensitive to noise and outliers. The solution is more complicated than least-squares estimation—the standard solution for 3D-3D pose-estimation problems. The solution to localization requires non-linear optimization. In general, closed-form solutions do not exist for non-linear optimization problems—implying that an iterative solution scheme is necessary, as is the case here.

Our approach to optimization borrows much philosophically from Kass, Witkin and Terzopoulos's [75, 134] work on active contour models and energy minimizing snakes and

---

<sup>2</sup>The models built using the methods of Chapter 2 can be composed of triangles of arbitrary size. If the tessellation is too coarse, we can straightforwardly increase the density of triangles by interpolation using many schemes. If the tessellation is too fine, we can use decimation techniques (e.g., Johnson [73]) to reduce the triangle density.

from Besl and McKay's [6] iterative closest point (ICP) algorithm for 3D registration. Instead of assigning correspondences and statically solving for the pose, our method achieves robustness by allowing the model to dynamically settle on the optimal pose with respect to the constraints of the image. We accomplish this by making the correspondences a dynamic function of pose during optimization. The objective function that we minimize is specifically chosen to make the estimation robust to outliers and is based on solid statistical principles. As will be described later, this approach is intuitively and mathematically well justified.

The rest of this chapter will provide the details of our solutions to the problems described above:

- Point visibility
- Point-to-point correspondence
- Pose optimization

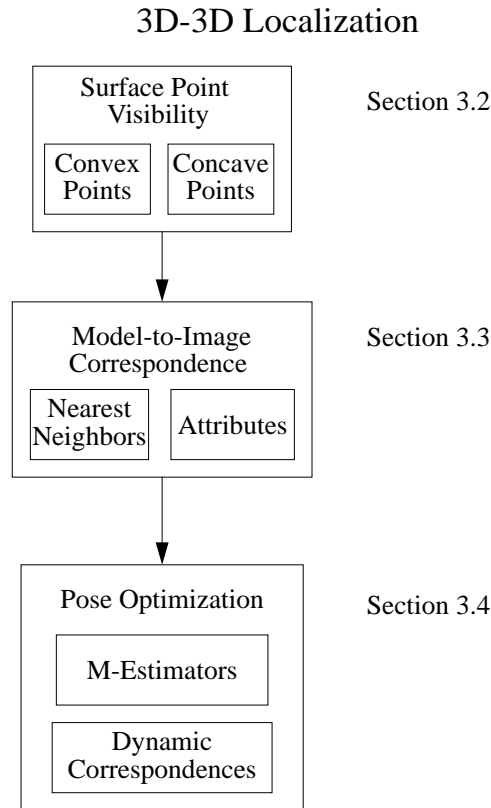
Figure 3.1 shows a diagram of the technical sections of the thesis. We begin by discussing an efficient approximation for surface point visibility computation. This is followed by a discussion of an efficient search technique for nearest neighbor correspondences and an extension to include attributes other than spatial coordinates. We then discuss the pose optimization problem and the main contribution of this chapter, our method for minimizing a robust M-estimator via dynamic correspondences with standard non-linear optimization techniques.

The chapter will conclude by summarizing the localization algorithm. The methods and algorithms described here are of a very practical nature. We will attempt to describe the steps with great attention to detail as there are many traps to catch the unsuspecting practitioner.

## 3.2 Point Visibility

Before matching a surface point of a model with a point in a range image, it is prudent to first determine if the model point is geometrically visible from the given pose. We need to answer the question of visibility for every point of the model. Since the visibility computation will be performed many times, the computation must be as efficient as possible. For an exact computation of the visible portions of an object model, there are two standard algorithms from the field of computer graphics: ray-casting and z-buffering.

Ray-casting [40, 141] works by casting a ray from the camera's center of projection through a given point on the model. The model point is visible if the ray does not first pass through any other point on the model surface. A ray is thus traced for each point of the model. Ray-casting is a rather complicated operation. For every triangle of the object, it is possible that we must test every other triangle on the object surface to determine if the triangle is the first surface that intersects the ray—resulting in  $O(n^2)$  ray-triangle



intersection tests for a surface model composed of  $n$  triangles. This is much too expensive to consider for localization in practice.

Z-buffering [40, 141] works by creating a depth image of the object. The object surface is projected into the image, triangle by triangle. If a surface point projects to the same image coordinate as a previous point, the point that is closest to the camera is placed in the depth image. Finally, only visible surface points are present in the depth image. Z-buffering is less expensive than ray-casting,  $O(n)$  operations<sup>3</sup> for a surface model composed of  $n$  triangles. Despite the existence of fast hardware implementations, z-buffering is still too time-consuming for our purposes. Z-buffering only works if the entire surface is projected triangle by triangle onto the depth image. Thus, there is no speedup benefit to be gained by using sparse collections of points. As a practical matter for efficiency, we must limit our localization search to use only a sparse set of points on surface of our object model.<sup>4</sup>

<sup>3</sup>Assuming a triangle fill is a constant time operation.

<sup>4</sup>For example, a reasonable coverage of a small object could have 40,000 triangles. Using that many triangles is inefficient and unnecessary to solve the task. Our experience is that using several hundred to a few thousand triangles is sufficient for localization of most objects.

Z-buffering necessarily solves for the visibility of *all* points on the surface (at the specified depth-image resolution); thus, z-buffering is not a desirable solution.

Since we are using a large set of points for localization, having a perfect visibility computation is not critical. Our localization algorithm should be resilient to a few mistakes out of a hundred. Thus, exact solutions such as z-buffering or ray tracing are not necessary for our purposes.

For 3D surfaces, we can reduce the problem to two cases: convex surface visibility and concave surface visibility. The convex case covers all points which lie on the convex hull of the surface: wherever the surface normal points towards the camera center, the point is visible (assuming outward pointing normals on every surface). The concave case covers all surface points which are not on the convex hull of the object. The solution for the concave case subsumes the convex case but has an added complication that the concave point will be occluded by other parts of the object surface from some viewpoints. In the following, we present an efficient approximation to solve for convex and concave surface visibility.

### 3.2.1 Convex Surface Visibility

We discuss the simplest case, the convex case, first. For this discussion, let us assume that the object is completely convex (e.g., a sphere or ellipsoid). As in Chapter 2, the surface is represented as a set of triangles. For any point  $x$  on a triangle, we can straightforwardly compute its visibility given the current viewing direction,  $\hat{v}$ —the vector from the camera’s center of projection to  $x$ .<sup>5</sup> Wherever the surface normal points towards the camera center, the point is visible. Without loss of generality, we focus our interest on the center points of each triangle. In the following,  $\tau_i$  denotes the  $i$ th triangle of the model,  $\hat{n}_i$  denotes  $\tau_i$ ’s outward pointing normal and  $c_i$  denotes  $\tau_i$ ’s center point. The visibility of point  $c_i$  is computed by the test

$$\text{visible}_{\text{convex}}(c_i) = \begin{cases} \text{true} & \hat{n}_i \cdot \hat{v} > 0 \\ \text{false} & \text{otherwise} \end{cases}$$

where  $\hat{v}$  is the viewing direction vector from the camera center of projection to  $c_i$ . This visibility test only requires a dot product and a comparison and, since it is local, can be computed independently for each triangle. Note that the definition of  $\hat{v}$  in this equation means that this test will work correctly regardless of camera parameterization.  $\hat{v}$  is the only information that we need to know about the camera projection to make the visibility determination.

In practice, a point on the surface will only be visible when the surface orientation is just less than 90 degrees from the viewing direction. Thus, we generally use a small threshold instead of 0 for the comparison in the above test. The slightly modified test is:

$$\text{visible}_{\text{convex}}(\tau_i) = \begin{cases} \text{true} & \hat{n}_i \cdot \hat{v} > \cos \theta \\ \text{false} & \text{otherwise} \end{cases} \quad (3.1)$$

---

<sup>5</sup>The camera’s center of projection is readily available from the camera calibration parameters as described in Chapter 1.

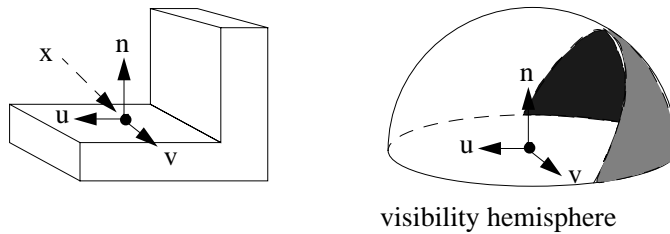


Figure 3.2: An example of the set of visible viewing directions (the unshaded portion of the hemisphere) for the point  $x$  on the L-shape.

where  $\theta$  is an angle close to 90 degrees which is the maximum orientation angle of the surface that will be detected by the range sensor.  $\theta$  is chosen empirically. This visibility test for the convex case nicely satisfies our requirements of efficiency and locality; however, we are not simply interested in localizing ovoids. The concave surface case is discussed next.

### 3.2.2 Concave Surface Visibility

In general, we must be prepared to compute point visibility for arbitrary shapes which will include concavities and self-occlusions where our simple convex visibility test (Equation 3.1) would be grossly insufficient.

As in the convex case, we prefer a local visibility test for the concave points. We can still make good use of our previous test for convex points. Since it is inexpensive, it makes sense to first check to see if the surface point in question is even oriented toward the camera. Once that is determined, we can then perform more expensive tests to determine whether the point is occluded by another part of the object or not. As stated previously, we cannot afford to perform ray-casting or z-buffering each time that we need to compute point visibility.

For each point, the visibility function is a binary function over the set of viewing directions. In practice, by assuming that the point lies on the planar center of the triangle, only the viewing directions lying in the unit hemisphere above the surface point (triangle) need to be tested for visibility. Figure 3.2 illustrates this for a simple L-shaped object. Thus, we can reduce the visibility problem to representing the set of viewing directions from which the point is visible. For the convex case, the visibility set is trivial—the hemisphere above the point's surface. Our main problem is how to represent the visibility set such that:

- the membership test is efficient, and
- the memory requirements are tolerable.

If it were necessary to represent the visibility set exactly, we would be in a lot of trouble. First, as one can imagine, the visibility set could take on arbitrarily complicated subsets of the viewing hemisphere—making the representation arbitrarily large. Secondly, it could become very expensive to evaluate membership in such a set. Fortunately, we can tolerate small errors: as described in Section 3.1 our localization algorithm must be robust to small numbers of errors. Thus, an approximation of the visibility set is sufficient.

We will use a discrete representation of the viewing hemisphere: a *lookup table* (LUT). We can tessellate the viewing hemisphere into discrete bins representing sets of similar viewing directions—all viewing directions in a particular bin are considered equivalent for the visibility computation. Such a scheme involves a time versus space trade-off. In general, we are always looking to reduce the time requirements of localization algorithms, thus, we will usually choose time savings over space savings. In practice, the more important trade-off is space versus accuracy. Do we prefer an LUT approximation which is accurate to 0.1 degrees or an approximation which is accurate to 3 degrees but can be efficiently stored and loaded.

One common LUT scheme [47, 81, 148] is to store in each bin a list of all object features (e.g., points, surfaces, lines) which are visible from the viewing directions corresponding to the bin. The problem with this approach is that it assumes that all points on the object are viewed from the same direction. This is adequate for an orthographic-projection camera model; however, under perspective projection the viewing direction depends on the projected image coordinates of the scene point. For example, the LUT scheme described above does not account for changes in the visibility set as an object translates along the central viewing direction.

Since we do not wish to restrict ourselves to a specific camera model, we must devise another approach. Instead of listing all visible entities in each bin, we allocate a separate LUT for each surface point used in the model (for example, the center points of each triangle). We refer to these LUTs as *visibility LUTs*. Each bin in a visibility LUT contains a binary value indicating whether the point is visible or not from viewing directions which map to the given bin. Since we have a visibility LUT for each point, each point's visibility can be independently computed using the viewing direction to *that* particular point, thus, allowing for arbitrary camera models.

The next issue is how to tessellate up the viewing hemisphere into discrete bins. We have two criteria for carving up the hemisphere:

1. The mapping from viewing directions to bins must be efficient.
2. The bins should cover approximately uniform areas of the hemisphere.

The first criteria is obvious: it is very important that the membership test is efficient as it will be evaluated frequently (once for every point of the model). The second criteria is a practical detail: if some bins are much larger than others, the error of the visibility approximation will vary with respect to viewing direction.

The simplest and most efficient mapping between viewing directions of a hemisphere and the bins of a 2D LUT is orthographic projection. Consider the 2D LUT as a grid lying



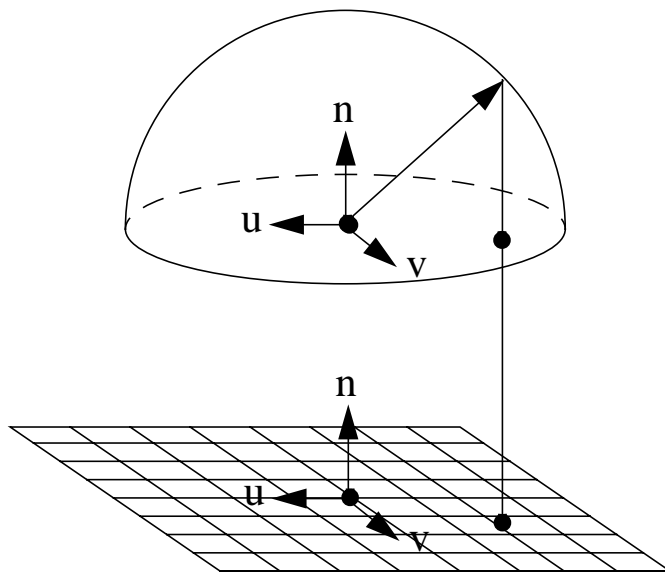


Figure 3.3: Orthographic projection of viewing directions onto a simple LUT array.

on the equatorial plane of the hemisphere. The 2D orthographic coordinates,  $[f \ g]^T$ , of a viewing direction projected onto the grids are simply the  $x$  and  $y$  components of the viewing direction  $\hat{v}$  (i.e.,  $f = x$  and  $g = y$ ). This is a bit unsatisfactory, however, because the result is that two different grid elements may cover vastly different areas on the viewing sphere (see Figure 3.3).

We would prefer a more uniform tessellation of the hemisphere. No perfectly uniform tessellation exists; however, there are several other options. One such improvement over orthographic projection is stereographic projection: each point on the sphere projects onto a grid lying on the equatorial plane by intersecting this plane with a ray passing through the point and the south pole (i.e., lowest point) of the sphere. Figure 3.5 shows the stereographic projection of a 2D LUT array. Stereographic projection is slightly more complicated than orthographic, but not beyond reason. The stereographic projection of  $\hat{v} = [x \ y \ z]^T$  to 2D stereographic coordinates,  $[f \ g]^T$ , is accomplished by

$$f = \frac{x}{z + 1} \quad (3.2)$$

$$g = \frac{y}{z + 1} \quad (3.3)$$

assuming that the radius of the sphere is 1, the sphere is centered at  $[0 \ 0 \ 0]^T$ , and the 2D grid  $[f \ g]^T$  lies on the  $z = 0$  plane. Using more advanced tessellations than orthographic and stereographic projection would require much more elaborate (and computationally expensive) mappings between viewing directions and LUT bins. One such tessellation is

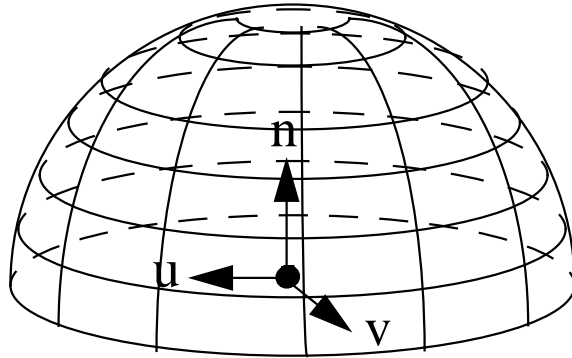


Figure 3.4: Latitudinal/Longitudinal discretization of a hemisphere.

based on latitude and longitude (see Figure 3.4). The problem with a latitudinal-longitudinal tessellation is the cost of indexing the LUT by viewing direction. Computing the latitude and longitude of the viewing direction requires the (relatively) expensive evaluation of trigonometric functions which we would like to avoid. What we can do, however, is essentially to use LUT methods again to compute the latitudinal-longitudinal coordinates. Since we are only really interested in mapping the viewing direction to a bin in the visibility LUT, the latitudinal-longitudinal coordinates can be ignored altogether. LUT methods can be used to index the indices of our visibility LUT. Thus, we have two levels of lookup tables:

- Visibility LUT: unique for each point, each bin contains the binary value indicating the visibility of the point from the set of viewing directions that map to the bin.
- Index LUT: identical structure for all points and models, each bin contains the indices of the Visibility LUT which correspond to the set of viewing directions that map to the bin.

It appears that we have the same problem as before: how do we tessellate the index LUT? The problem is a bit different for the index LUT than for the visibility LUTs.

Since one visibility LUT is required for each point of the localization model, the number of bins must necessarily be small; however, only one index LUT is necessary since the mapping from viewing directions to bin indices is the same for all points. Thus, it is feasible to use a much higher resolution table. The previous problem of uniform tessellation can be ignored as long as the resolution of the index LUT is substantially higher than resolution of the visibility LUTs (i.e., many index LUT bins correspond to each visibility LUT bin). The index LUT can be indexed by the  $f$  and  $g$  components of stereographic projection to provide the corresponding indices of the visibility LUT (implicitly providing the latitude and longitude, or any other mapping for an arbitrary tessellation of the hemisphere). Figure 3.6 demonstrates how the two-level indexing of viewing directions to visibility works.

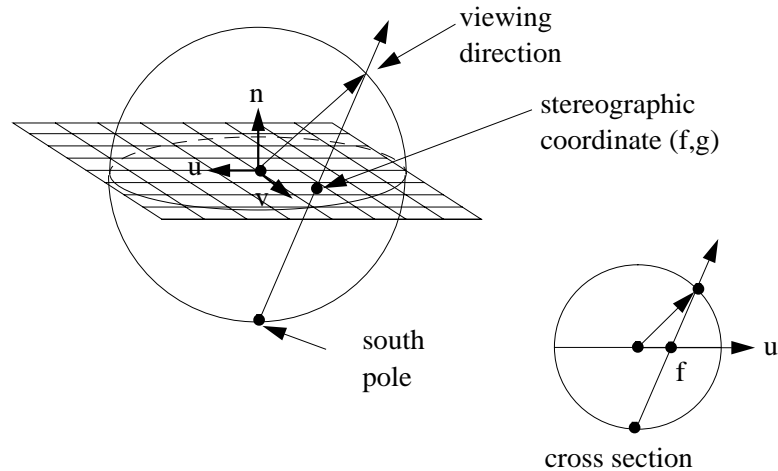


Figure 3.5: Stereographic projection of viewing directions onto a simple LUT array.

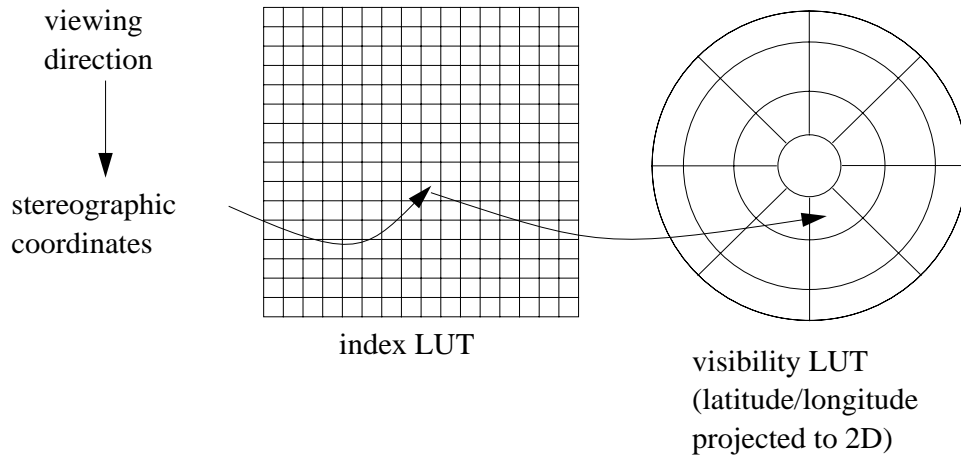


Figure 3.6: The viewing direction is mapped to the index LUT using stereographic projection (note, in practice the resolution of the index LUT would be extremely fine as opposed to the coarse LUT pictured here). The bin of the index LUT provides the index into the visibility LUT which may have an arbitrarily complex tessellation (latitude and longitude in this case).

This two-level indexing technique will work for any discretization of the viewing sphere. We can enumerate an endless list of possibilities, such as tessellations of pentagons or hexagons (as in a soccer ball), an icosahedron, or semi-regular triangulation of the sphere. The latitudinal-longitudinal tessellation allows us more control over the size of the bins than some of these other practical tessellations. Any tessellation can be used; the only requirement is that there is some (arbitrarily complicated/expensive) function that maps viewing directions to the index of the corresponding discrete chunk. This function can be used to generate the index LUT off-line, and then we have a simple and efficient way to map viewing directions to the bins of our viewing hemisphere tessellation.

One subtle point that was omitted from the preceding discussion was the coordinate system of the viewing direction vector  $\hat{\mathbf{v}}$ . Each point must have its own coordinate system since the viewing hemisphere is oriented in the direction of the point's normal; there is no single coordinate system which will work for all points on the model. The viewing direction must be in the point's local coordinate frame in order to index the correct element of the point's visibility LUT. We can define local coordinate system of triangle  $\tau_i$ 's center,  $\mathbf{c}_i$ , as follows. We require the  $z$  direction,  $\hat{\mathbf{z}}_i$ , to correspond with the  $\tau_i$ 's normal:

$$\hat{\mathbf{z}}_i = \hat{\mathbf{n}}_i.$$

The  $x$  and  $y$  directions can be arbitrarily chosen to be orthogonal to  $\hat{\mathbf{z}}_i$ . We choose the  $x$  direction,  $\hat{\mathbf{x}}_i$ , to be the direction from the center point,  $\mathbf{c}_i$ , to  $\tau_i$ 's first vertex  $\mathbf{p}_0$ :

$$\hat{\mathbf{x}}_i = \frac{\mathbf{p}_0 - \hat{\mathbf{c}}_i}{\|\mathbf{p}_0 - \hat{\mathbf{c}}_i\|}.$$

The  $y$  direction,  $\hat{\mathbf{y}}_i$ , follows directly from  $\hat{\mathbf{z}}_i$  and  $\hat{\mathbf{x}}_i$ :

$$\hat{\mathbf{y}}_i = \hat{\mathbf{z}}_i \times \hat{\mathbf{x}}_i.$$

To index our LUT all we need to do is convert  $\hat{\mathbf{v}}$  to local coordinates, which can be accomplished by the following matrix multiplication:

$$\hat{\mathbf{v}}_i = [\hat{\mathbf{x}}_i \ \hat{\mathbf{y}}_i \ \hat{\mathbf{z}}_i]^T \hat{\mathbf{v}} \quad (3.4)$$

$$\hat{\mathbf{v}}_i = \mathbf{R}_i \hat{\mathbf{v}} \quad (3.5)$$

The stereographic coordinates of  $\hat{\mathbf{v}}_i$  index a bin in the index LUT which gives us the index into the visibility LUT. The computation cost of this visibility test is also reasonable: one matrix-vector multiplication and two table lookups. The index LUT can be computed off-line and loaded at run-time, and for each concave point we must precompute and store its visibility LUT. The visibility LUT is most efficiently implemented as a bit vector. The resolution of the visibility LUT can be chosen based on memory limitations, required accuracy or efficiency requirements. For example, we can carve up the viewing hemisphere into 256 chunks and store this in eight, 32-bit words.

### 3.2.3 Off-line Lookup Table Creation

Now that we have chosen a representation for the visibility set, we must compute the LUT for each point of the model. Though this computation is relatively expensive, it can be computed in reasonable time (several minutes) off-line and stored with the model.

For each triangle  $\tau_i$ , the algorithm for filling the visibility LUT is as follows

**Algorithm** *CreateVisibilityLUT*

**Input:** point  $c_i$  of triangle  $\tau_i$

**Input:** triangle set  $T$

**Output:** the visibility LUT for point  $c_i$  of triangle  $\tau_i$

1. create the index LUT of dimension  $N \times N$
2. create bitmap LUT of dimension  $N \times N$ , initialize all bins to true
3. **for**  $\tau_j \in T$  s.t.  $\tau_j \neq \tau_i$
4.     **if**  $\tau_j$  occludes  $\tau_i$
5.         **then** project  $\tau_j$  into  $\tau_i$ 's local stereographic coordinates
6.             fill in the bitmap LUT bins corresponding to the projected triangle with false to denote occlusion
7. **if** all bitmap LUT bins are true
8.     **then** declare  $\tau_i$  convex
9.     **return**
10. **else** create and clear the visibility LUT
11.     **for** all bitmap LUT bins, initialize all bins to true
12.         **if** bin is false
13.             **then** set the corresponding visibility LUT bin to false
14. **return** visibility LUT

The index LUT is created and a bitmap of the same dimensions is created to record the visibility at each of these bins. The visibility of the bitmap's bins is computed by projecting all occluding triangles onto the bitmap and marking those bits which overlap the triangle as occluded (i.e., painting the triangle onto the bitmap). After doing this for all triangles of  $T$ , we can convert the bitmap to the (coarser) visibility LUT bitmap. If the bitmap indicates that all bins are visible, the point  $c_i$  is convex and no visibility LUT is necessary.

### 3.2.4 On-line Visibility Testing

When performing localization, we need to test the visibility of a point before using it for refining the pose. If we're dealing with orthographic projection, then we can use the same viewing direction for all points (and previous aspect LUT methods become an option). For perspective projection, the viewing direction depends on the point's location in the image. Thus, for perspective, we must first compute the image projection of the point and compute  $\hat{v}$  using the image coordinates and the known camera parameters.  $\hat{v}$  must then be rotated into the model's coordinate system and then the point's coordinate system. These transformations can be composed for efficiency. Once  $\hat{v}$  is transformed to the point's local

coordinate system, we first perform the convex visibility test from Equation 3.1. If the test returns true and the point is concave, then the visibility LUT is checked as described in Section 3.2.2.

The visibility tests described in this section are very efficient. Practical requirements force us to make approximations; however, the approximations as discussed here typically err on the safe side: points may be declared invisible when in fact they are visible, however, it is not possible that a point is declared visible when it is geometrically invisible. Since our localization algorithm will have to cope with a significant number of errors, declaring a few visible points to be invisible will not have a big effect on the result. On the other hand, declaring a few invisible points to be visible is pointless as they will never be detected.

### 3.3 3D-3D Correspondence

Once the set of visible model points has been computed, we need to compute the correspondences between these model points and points in the range image. The correspondences will be used to evaluate the current estimate of the pose.

The correspondence problem is circular. One cannot easily find the correct correspondences without first knowing the pose of the object (i.e., knowing where to look), but our interest in finding correspondences is to aid our search for the correct pose of the object.

One of the points of this thesis is that the exact correspondences are not necessary to refine the pose. What is necessary are correspondences which lead to improvements in the pose estimate. When starting with a small pose error, local search can often provide a set of correspondences that are close enough to the correct correspondences to guide the search to the correct pose. What we desire is a quick local search which will usually produce matches to points which are nearby the correct match. As the pose estimation is improved, these matches will approach the correct matches.

Given a point on the model and a current pose estimate, we must select a corresponding point from our range image. The obvious approach is to find the nearest point in 3D (Cartesian) space (nearest neighbor or closest point). Mathematically, the closest image point  $y$  to a given point  $x$  can be defined as

$$y = \arg \min_{y \in D} \|x - y\|$$

where  $D$  is the set of three-dimensional data points in the image. The theoretical complexity of the nearest-neighbor search is  $O(|D|)$ . However, geometry is on our side. 3D objects occupy a volume in 3D space and their surfaces occupy 2D manifolds in 3D space. The surface tends to sparsely occupy the 3D volume. Thus, it is possible to partition the surface points in the 3D space to more efficiently search for the nearest points in practice. We now describe a technique which utilizes this characteristic distribution of surface points in 3D space to make the search efficient.

### 3.3.1 K-D Trees for Efficient Correspondence Search

The key to efficient search in two or more dimensions is a generalization of binary-search trees called *k-d trees* [42] (“kd” is an abbreviation for *k-dimensional* where  $k$  is an arbitrary integer greater than zero).<sup>6</sup> The k-d tree is created by recursively splitting a data set down the middle of its dimension of greatest variance. The splitting continues until the leaf nodes contain a small enough number of data points. The result is a tree of depth  $O(\log n)$  where  $n$  is the number of points stored in the k-d tree. Figure 3.7 shows an example of how 2D points would be separated into leaf nodes using this technique.

The k-d tree can be searched efficiently by following the appropriate branches of the tree until a leaf node is reached (as in binary-tree search). The distance to all points in the leaf node is computed. A hyper-sphere centered at the key point (with radius of the distance to the current closest point) can be used to determine which, if any, neighboring leaf nodes in the k-d tree must be checked for closer points. This test can be performed very efficiently. Once we tested all the data in leaf nodes which could possibly be closer, we are guaranteed to have found the closest point in the tree.

To use k-d trees and nearest-neighbor search for point sets in a particular coordinate system, we need a measure of *dissimilarity* between a pair of points. The dissimilarity,  $\Delta$ , between k-d points  $\mathbf{x}$  and  $\mathbf{y}$  must have the form

$$\Delta(\mathbf{x}, \mathbf{y}) = F \left( \sum_{i=1}^k f_i(\mathbf{x}_i, \mathbf{y}_i) \right) \quad (3.6)$$

where the functions  $f_i$  are symmetric functions over a single dimension and functions  $f_i$  and  $F$  are monotonic. Most notable of these restrictions is that  $\Delta$  must be composed by a sum of dissimilarities along each individual coordinate. All metric distances satisfy these conditions—most importantly, the Euclidean distance

$$\Delta(\mathbf{x}, \mathbf{y}) = \|\mathbf{x} - \mathbf{y}\|. \quad (3.7)$$

Such a dissimilarity allows us to partition the k-d tree for optimal expected time of the nearest-neighbor search.

Though its worst case complexity is still  $O(n)$ , the expected number of operations is  $O(\log n)$ , which will be the case if the data is evenly distributed—as is the case for surfaces in 3D space. The overhead involved is that the k-d tree of range-image points must be built prior to the search. This is a one-time cost per image of  $O(n \log n)$  time since the points in the range image are static. For most cases of 3D localization, using closest points is sufficient. This is because surfaces sparsely occupy 3D space and points are not often found immediately above the surface of a visible object. As long as the majority of correspondences are to the correct surface in the image, localization will often succeed. The next section discusses an extension of this nearest-neighbor search to make the correspondence search a little more robust to initial position errors.

---

<sup>6</sup>The k-d tree [42] representation and nearest-neighbors search are extremely useful tools for many problems in computer vision, computer graphics, as well as computer science and artificial intelligence. K-d trees are used for many applications throughout this thesis, notably in Sections: 2.4, 4.3, 4.4, and 5.3.

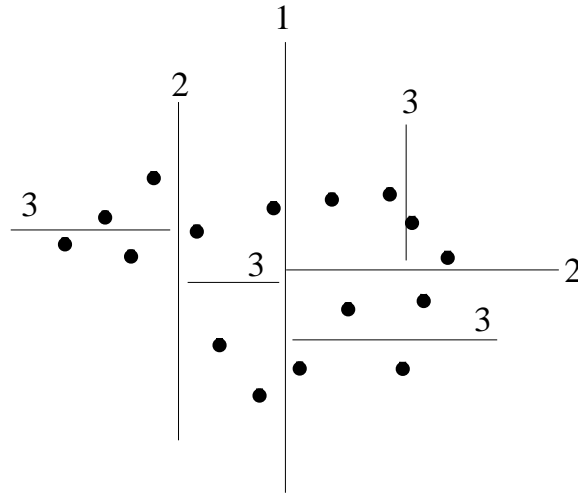


Figure 3.7: K-d tree subdivision of 2D points. Each line splits the data in half across the dimension of greatest spread. The number indicates which level of the tree at which the split occurs.

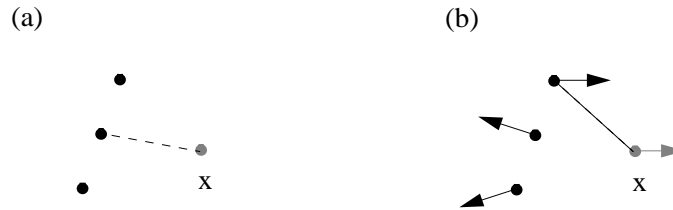


Figure 3.8: An example of nearest-neighbor correspondence (left) versus nearest neighbor+normal correspondence (right).

### 3.3.2 Attributes for Improving 3D-3D Correspondences

Besides proximity there are other possible constraints which we may want to consider when searching for correspondences. One such example is surface normal similarity—the surface normal of the model point should be similar to the normal of its matching image point. Figure 3.8 shows an example where, because of the error in the pose estimate, the simple nearest-neighbor search results in non-useful correspondences, while the addition of normal constraints on the local search improves the utility of the matching.

The ideal dissimilarity measure for comparing two unit vectors (normals) is the angle



between the two vectors

$$\Delta_\theta(\hat{\mathbf{n}}_1, \hat{\mathbf{n}}_2) = \cos^{-1}(\hat{\mathbf{n}}_1 \cdot \hat{\mathbf{n}}_2).$$

This metric is non-linear in terms of the two normals,  $n_1$  and  $n_2$ , that we are comparing and can not be decomposed into a dissimilarity function as specified in Equation 3.6.

As a matter of efficiency, we would prefer a linear metric that is of the form of Equation 3.7, such as the metric

$$\Delta_n(\hat{\mathbf{n}}_1, \hat{\mathbf{n}}_2) = \|\hat{\mathbf{n}}_1 - \hat{\mathbf{n}}_2\|. \quad (3.8)$$

We need to characterize  $\Delta_n$  to determine if it is suitable for our purposes: whether  $\Delta_n$  bears any relationship to our desired dissimilarity measure  $\Delta_\theta$ . Without loss of generality, we can rotate  $\hat{\mathbf{n}}_1$  and  $\hat{\mathbf{n}}_2$  such that

$$\hat{\mathbf{n}}'_1 = \mathbf{R} \hat{\mathbf{n}}_1 = [1 \ 0 \ 0]^T \quad (3.9)$$

$$\hat{\mathbf{n}}'_2 = \mathbf{R} \hat{\mathbf{n}}_2 = [\cos \theta \ \sin \theta \ 0]^T \quad (3.10)$$

where  $\theta$  is the angle between  $\hat{\mathbf{n}}_1$  and  $\hat{\mathbf{n}}_2$ . This step uses the facts that the normals are unit vectors and that 3D and angular distances are invariant to rigid rotation. Now we can simplify  $\Delta_n(\hat{\mathbf{n}}_1, \hat{\mathbf{n}}_2)$  to

$$\Delta_n(\hat{\mathbf{n}}'_1, \hat{\mathbf{n}}'_2) = \|\hat{\mathbf{n}}'_1 - \hat{\mathbf{n}}'_2\| \quad (3.11)$$

$$= \sqrt{(\cos \theta - 1)^2 + \sin^2 \theta} \quad (3.12)$$

$$= \sqrt{\cos^2 \theta - 2 \cos \theta + 1 + \sin^2 \theta} \quad (3.13)$$

$$= \sqrt{2(1 - \cos \theta)} \quad (3.14)$$

$$= \sqrt{4 \sin^2 \frac{\theta}{2}} \quad (3.15)$$

$$= 2 \sin \frac{\theta}{2}. \quad (3.16)$$

Using the small-angle approximation for  $\theta$ ,  $2 \sin \frac{\theta}{2} \approx \theta$  for small values of  $\theta$ . If we plot this function (see Figure 3.9) over the valid range of  $\theta$ , which is between 0 and 180 degrees, then we see that though this metric is not linear, it is close to linear and monotonic in  $\theta$ —only diverging near 100 degrees. This is qualitatively sufficient for our purposes.

Our immediate goal is to efficiently compute correspondences. Proximity is our first criteria, normal similarity is another. Comparing both of these quantities at the same time is a difficult proposition with no absolute solution. For example, assume we have three points  $\mathbf{x} = [0 \ 0 \ 0]^T$  with normal  $\hat{\mathbf{n}} = [1 \ 0 \ 0]^T$ ,  $\mathbf{x}_1 = [0 \ 0 \ 1]^T$  with normal  $\hat{\mathbf{n}}_1 = [1 \ 0 \ 0]^T$ , and  $\mathbf{x}_2 = [0 \ 0 \ 0]^T$  with normal  $\hat{\mathbf{n}}_1 = [0 \ 0 \ 1]^T$ . How do we decide which of  $\mathbf{x}_1$  and  $\mathbf{x}_2$  is closer to  $\mathbf{x}$ ?  $\mathbf{x}_1$  has a closer normal to point  $\mathbf{x}$ , while  $\mathbf{x}_2$  is closer to  $\mathbf{x}$  in position; the choice is not clear.

The answer depends on what the application's requirements are. Returning to the previous example, is it more important to have a point which is closer or a point which has a closer normal vector? We have to decide which constraint is more important and how

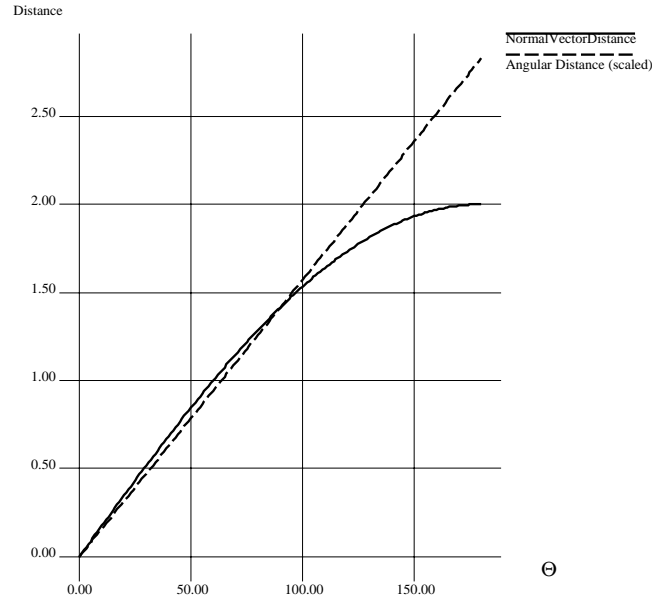


Figure 3.9: Plot of  $\Delta(\hat{\mathbf{n}}_1, \hat{\mathbf{n}}_2)$  (the 3D distance between two unit vectors) with respect to  $\theta$  (the angle between the two vectors). The (desired) angular distance is plotted for comparison.

much more important it is. We can do this easily by weighting either the point or normal and combining the vectors to form a higher-dimensional vector (e.g., six dimensions in the previous example).

Each data point is then stored in the k-d tree as the 6D vector

$$\mathbf{p} = [\mathbf{x}^T \ w \mathbf{n}^T]^T \quad (3.17)$$

$$= [x \ y \ z \ w n_x \ w n_y \ w n_z]^T \quad (3.18)$$

where  $w$  is the scaling factor for the normals. Applying the Euclidean distance metric to these points gives

$$\Delta(\mathbf{p}_1, \mathbf{p}_2) = \|\mathbf{p}_1 - \mathbf{p}_2\| \quad (3.19)$$

$$= \sqrt{\|\mathbf{x}_1 - \mathbf{x}_2\|^2 + \|w\hat{\mathbf{n}}_1 - w\hat{\mathbf{n}}_2\|^2} \quad (3.20)$$

$$= \sqrt{\|\mathbf{x}_1 - \mathbf{x}_2\|^2 + w^2 \|\hat{\mathbf{n}}_1 - \hat{\mathbf{n}}_2\|^2} \quad (3.21)$$

which is the desired effect. For example, if we decide that an error of one unit in distance is just as undesirable as an error of .25 units between normals, then we can scale the normals by setting  $w = 4$  in the previous equation.

Thus, no modification of the k-d tree and nearest-neighbor search technique is required to add the additional orientation attribute for correspondence search. The weighting factor  $w$  is used to effect the desired constraint on the correspondences.

Depending upon other constraints of the system, other attributes can be added in a similar manner. Curvature is another possible attribute for improving the correspondence search in 3D-3D localization. The main drawback of curvature is that computing curvature from range images is a particularly noise-sensitive operation. Another potential source for attributes is color information. The key is to select attributes that can be reliably measured and are invariant to object pose and other possible scene variations such as lighting.

### 3.3.3 3D-3D Correspondence Summary

Given an estimate of the pose, we have shown how to efficiently compute nearest neighbor correspondences between visible model points and points in the range image. We have also shown how the nearest-neighbor search can be extended to consider attributes other than 3D positional information.

One must be reminded that for 3D-3D localization, the use of additional attributes for correspondence search will only be necessary in extreme situations. In general, if the pose is reasonably accurate, correspondences based on proximity will yield useful results (i.e., model point is matched to a point near the correct point of the object surface). This is due to the nature of range-image data. The points in the image sparsely occupy 3D space. Thus, as a point moves away from a surface, the space between the point and the surface is usually empty. If the initial position estimate is too far or, perhaps, the current pose estimate is closer to an object other than the desired object, then proximity may not be sufficient for effective localization. Using additional attributes for the correspondence search may alleviate the problem, but there are limits. If the pose estimate is nearly correct, proximity will almost always be sufficient for 3D-3D localization to converge to the true pose.

Now that we have a method for efficiently generating local correspondences, our goal is to use these correspondences to improve our estimate of the pose; this is the topic of the next section.

## 3.4 3D-3D Pose Optimization

This section discusses a variety of techniques dealing with the computation of pose of a 3D object (point set) with respect to observed 3D points. We refer to this general area as *3D-3D pose optimization* which includes the problems of 3D-3D pose estimation and 3D-3D pose refinement.

The desired results of pose estimation and pose refinement are much the same: find the pose of the object in the image. The assumptions are quite different; pose estimation is *given a set of correspondences* while pose refinement is *given an image and a rough initial pose estimate*. For pose estimation, the given correspondences are assumed to be correct and are fixed. Pose estimation is very much a static problem. Because of this, the pose-estimation

problem is well suited to abstract formulations and theoretical analysis. In contrast, the problem of primary interest here, localization, is a dynamic operation involving possibly all information available from the image.

It must be noted that there is a medium ground between pose estimation and pose refinement—hybrids of pose estimation and pose refinement. Some examples include the work of Grimson [52, 49] and Lowe [85, 84, 89, 90] in which the pose is iteratively computed by a sequence consisting of pose estimation and correspondence search. During each iteration a new correspondence (found using local search) is added to the previously accumulated set of correspondences and pose estimation is performed on the whole set. The goal is to gradually increase the accuracy of the pose estimate by adding constraints (correspondences).

Instead of jumping straight to our solution for 3D-3D localization, we will build up to it by first discussing a simpler version of the problem—pose estimation. The form of the pose-estimation problem of interest here is 3D-3D pose estimation—to compute the pose of an object given a number of correspondences between a set of measured 3D (image) points and a set of 3D points of our prior model. The 3D-3D localization techniques presented in this thesis borrow much from the theory and practice of 3D-3D pose estimation.

We begin by considering the problem of 3D-3D pose estimation—a problem which has a long history [36, 1, 66, 58] and is generally considered to be solved.

### 3.4.1 3D-3D Pose Estimation

The 3D-3D pose-estimation problem is to compute the pose (a rigid transformation) which aligns the 3D model points  $\mathbf{x}_i$  with their corresponding image points  $\mathbf{y}_i$  where  $i = 1, \dots, n$ . The rigid transformation is specified by the matrix-vector pair  $\langle \mathbf{R}, \mathbf{t} \rangle$  where  $\mathbf{R}$  is a  $3 \times 3$  rotation matrix and  $\mathbf{t}$  is a 3D translation vector.

Each correspondence provides three linear constraints on our unknown pose variables via the rigid transformation equation

$$\mathbf{y}_i = \mathbf{R}\mathbf{x}_i + \mathbf{t}.$$

In general, the sensed points  $\mathbf{y}_i$  will be contaminated by noise:

$$\mathbf{y}_i = \mathbf{y}_i^{actual} + \beta$$

where  $\beta$  is a random 3D variable. Assuming that  $\beta$  is unbiased (i.e.,  $\beta$ 's mean is  $[0 \ 0 \ 0]^T$ ) and follows a normal distribution (i.e.,  $P(\beta) \propto e^{-\frac{\beta^T \beta}{2\sigma^2}}$ ), then the optimal transformation is the least-squared error solution—the values  $\langle \mathbf{R}, \mathbf{t} \rangle$  that minimize

$$f(\mathbf{R}, \mathbf{t}) = \sum_i \|\mathbf{R}\mathbf{x}_i + \mathbf{t} - \mathbf{y}_i\|^2. \quad (3.22)$$

This seems like an easy enough problem to solve— $f$  is quadratic in its input parameters—until we consider rotation. A rotation in 3D space has only three degrees of freedom, yet

is represented in our linear formula by a  $3 \times 3$  matrix  $\mathbf{R}$ . The three degrees of freedom restrict the values of  $\mathbf{R}$  in a non-linear way.  $\mathbf{R}$  must satisfy the constraints

$$\mathbf{R}\mathbf{R}^T = \mathbf{I} \quad (3.23)$$

$$|\mathbf{R}| = 1 \quad (3.24)$$

where  $\mathbf{I}$  is the  $3 \times 3$  identity matrix. The first constraint requires the rows of  $\mathbf{R}$  to be orthonormal. The second constraint ensures that the rotation is rigid and not a reflection. Capturing all these constraints while taking advantage of the linear matrix form of rotation is difficult.

The generally accepted solution to the rotation problem is to represent the rotation using a unit 4-vector called a quaternion. A quaternion is composed of a 3-vector  $[u \ v \ w]^T$  and a scalar  $s$ . We will use the 4-vector

$$\mathbf{q} = [u \ v \ w \ s]^T$$

to denote the rotation. The rotation matrix,  $\mathbf{R}$ , corresponding to  $\mathbf{q}$  is denoted by  $\mathbf{R}(\mathbf{q})$  (refer to Appendix B for the derivation).

The 7-vector

$$\mathbf{p} = [\mathbf{q}^T \ \mathbf{t}^T]^T$$

denotes the complete set of pose parameters/rigid transformation. Now we have three equivalent notations for the pose which will be used interchangeably:  $\langle \mathbf{R}, \mathbf{t} \rangle$ ,  $\mathbf{p}$ , and  $\langle \mathbf{R}(\mathbf{q}), \mathbf{t} \rangle$ .

The best intuitive description of a quaternion is that the 3-vector of the quaternion represents the axis of rotation and the scalar represents the angle of rotation. This is not entirely accurate, as the quaternion is not simply an axis and angle representation. Quaternions have many nice mathematical properties which have proven useful for deriving several solutions as will be described below. For the reader's convenience, an overview of quaternions is provided in Appendix B.

The importance of the quaternion representation for the 3D-3D pose-estimation problem was first demonstrated by Sanso [116] in the field of photogrammetry and then later introduced to the field of computer vision by the work of Hebert and Faugeras [36] and Horn [66]. They independently showed that a closed form solution for  $\langle \mathbf{R}, \mathbf{t} \rangle$  existed by formulating the minimization of Equation 3.22 using quaternions to represent rotation. Their methods are slightly different, but we will describe the general idea central to both. First, both sets of points  $\mathbf{x}_i$  and  $\mathbf{y}_i$  are translated so that the centers of gravity of both translated sets are located at the origin. The centered points are denoted by  $\mathbf{x}_i^c$  and  $\mathbf{y}_i^c$  respectively. Using the centered points we can solve for the rotation quaternion  $\mathbf{q}$  independent of the translation  $\mathbf{t}$ . The optimal quaternion can be computed as the eigenvector corresponding to the maximum eigenvalue of a matrix composed from the points  $\mathbf{x}_i^c$  and  $\mathbf{y}_i^c$ . The reader should consult [116, 36, 66] for the derivation of this matrix. With  $\mathbf{q}$  and, hence,  $\mathbf{R}$  determined,  $\mathbf{t}$  is easily determined to be the difference between the centroid of the point set  $\mathbf{y}_i$  and the centroid of the rotated point set  $\mathbf{R}\mathbf{x}_i$ .

Horn also showed that the closed-form solution generalizes to weighted least squares as well. Haralick [58] and Arun [1] presented a closed-form solution for 3D-3D pose estimation which compute  $\mathbf{R}$  directly using singular value decomposition of a matrix composed from the points  $\mathbf{x}_i^c$  and  $\mathbf{y}_i^c$ . The quaternion formulation is favored over the SVD formulation which is not as robust, numerically speaking.

If the errors in the observed data are not normally distributed, least-squares estimation may be inappropriate as the resulting estimate is optimal only for normally distributed errors. It may be necessary to consider a different objective function which is the optimal estimator with respect to the error distribution of the data. Also, if we have statistically significant errors or outliers, then the closed-form solutions for  $\langle \mathbf{R}, \mathbf{t} \rangle$  are no longer useful. In fact, least-squares estimation will usually fail when outliers are present as the estimation is very sensitive to large errors.

Unfortunately, the existence of outliers is the rule for computer vision applications rather than the exception. When viewing objects, it is often true that parts of the object surfaces are occluded by other objects in the scene or shadows. When points on an object are occluded or go undetected, the errors for the corresponding point of the model will be much larger than the errors for visible points of the object. This is because these unobserved points violate the assumption that the observation actually corresponds to a point in our model.

Since the closed form solution is no longer valid, an iterative approach is probably necessary to solve this problem. The best known method for iterative search is gradient-descent search. The basic algorithm is to follow the path of best improvement (the gradient direction) until no local move improves the estimate (i.e., until the minimum of our objective function is reached).

There are a couple of well known problems with gradient-descent techniques. First, they assume a starting position is available. This is not a problem for pose-refinement since a starting point is already assumed. Second, gradient descent is a slave to local minima. Depending on the starting point, the desired minima may or may not be reachable. For example, Figure 3.10 shows an example search of a one dimensional function with two starting positions,  $s_1$  and  $s_2$  that lead to different local minima  $r_1$  and  $r_2$ . In this example,  $r_2$  is the desired result. The range  $[x_{min}, x_{max}]$  denotes the set of starting points that will lead to the solution  $r_2$ ; this range is referred to as  $r_2$ 's basin of attraction. To ensure a high probability of success, we would like a good initial starting guess (i.e.,  $|s - r|$  is small) and a wide basin of attraction (i.e.,  $|x_{min} - x_{max}|$  is large). In general, the size of the basin of attraction and the initial error will depend on the complex interaction of the model and data points that are matched as well as the number and size of any outlier correspondences.

We can now step back and examine how our localization problem relates to the pose estimation problem. The main similarity is that local minima will be a problem, since we will in general have some number of incorrect correspondences to deal with. However, the localization problem is still more difficult. First, we are not given absolute correspondences, only a rough pose as a starting point of the search. Correspondences are acquired using the efficient local search described in Section 3.3; however, it is unlikely that any significant portion of these correspondences will be correct unless the pose estimate is also correct.

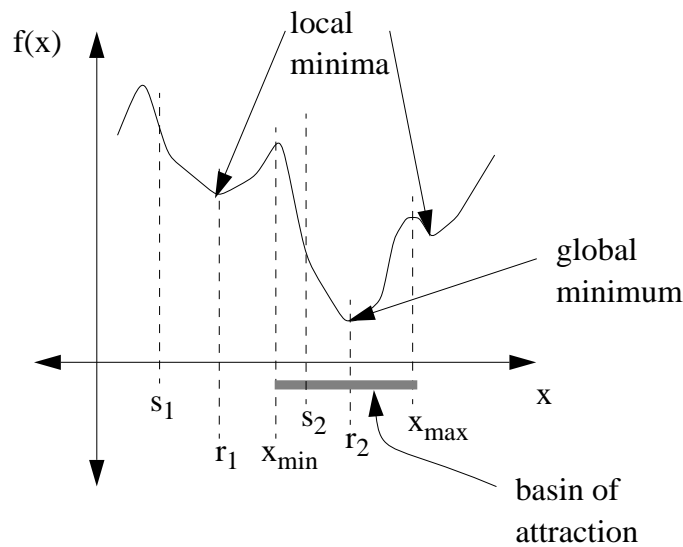


Figure 3.10: Examples of local versus global minima and basins of attraction.

We must deal with two problems:

- Poor initial correspondences: we must assume that at the start of the search most of the correspondences will not be correct.
- Outliers: even when most of the correspondences are correct, we must be able to handle outliers gracefully.

The next section will describe methods from the field of robust statistics to handle outliers. The methods described there will lead to an approach that also overcomes the problem of poor initial correspondences.

### 3.4.2 Robust Estimation

We return to the pose estimation problem described in the previous section. We are given a set of  $n$  observed points  $y_i$  and corresponding model points  $x_i$ , and we want to compute the pose  $\langle \mathbf{R}, \mathbf{t} \rangle$  which will align the two sets. The additional complication is that some percentage of the  $n$  correspondences will be incorrect, and we do not know which ones are incorrect *a priori*. The errors for these incorrect correspondences will not fit a normal distribution that describes the expected errors when comparing a point with its observation.

We look to the field of robust statistics [67, 96] for a solution to this problem. Robust statistics provides solutions to the problem of estimating statistics reliably despite data contaminated by outliers—data which do not belong to the desired sample population.

There are three classes of robust-estimation techniques that we will consider here: outlier thresholding, median/rank estimation, and M-estimation.

### Outlier Thresholding

The first class of solutions, outlier thresholding, is the simplest and, hence, the most prominent robust-estimation technique used in computer-vision applications such as localization. Outlier thresholding is also the most efficient of the three methods, but, unfortunately, the most unreliable. The basic idea is to estimate the standard deviation,  $\sigma$ , of the errors in the data and eliminate data points which have errors outside the range  $[-k\sigma, k\sigma]$  where  $k$  is typically greater than or equal to 3. One problem with this is that an estimate of  $\sigma$  may be grossly incorrect if there are many outliers. Another problem is that outliers with errors less than  $k\sigma$  are possible. If a large number of these outliers exist, then least-squares estimation will still be inaccurate. Another popular (and somewhat similar technique) is to find the first mode of the distribution of data and throw out the data past that point [151]. This assumes that the number of outliers is much smaller than the correct data and that the outliers do not harmfully alter the shape of the mode.

The essential problem with these methods is that a hard threshold must be used to eliminate the outliers. This is an unfortunate situation since regardless of where the threshold is chosen, some number of valid data points will be classified as outliers and some number of outliers will be classified as valid. In this sense, it is unlikely that a perfect method for selecting the threshold exists unless the outliers (or perhaps their distribution) are known *a priori*. As will be discussed in Sections 3.4.2 and 3.6, a hard threshold also creates a highly non-linear (non-smooth) objective function which causes difficulties for numerical optimization techniques.

The above methods may not always be useful for the localization problem since the initial correspondences and pose are likely to be incorrect anyway. Initially, these errors may not have any unimodal distribution and the valid correspondences and outliers may have indistinguishable error values.

### Median/Rank Estimation

The second class of robust estimators that we will discuss is the median/rank estimation methods. The basic idea is to select the median or  $k$ th value (for some percentile  $k$ ) with respect to the errors for each observation and use that value as our error estimate. The logic behind this is that the median is almost guaranteed not to be an outlier as long as half of the data is valid. In fact, median estimation has the optimal breakdown point of any estimator (smallest percentage of outliers that are capable of forcing the estimate outside some finite range).

An example of median estimators is the least-median-of-squares method (LMedS) [96, 79]. LMedS computes the parameters  $\mathbf{p}$  which minimizes the median of the squared error:

$$\mathbf{p} = \arg \min_{\mathbf{p}} \left( \text{med}_i z_i(\mathbf{p})^2 \right)$$



where  $z_i(\mathbf{p})$  is the error in the  $i$ th observation and  $\text{med}_i$  returns the median value of its argument over all values of  $i$ . Generally, to solve this problem, we must perform exhaustive search of possible values  $\mathbf{p}$  by testing least-squares estimates of  $\mathbf{p}$  for all possible combinations of matches from model to observed data. Techniques like randomization and Monte Carlo methods are necessary to make searches such as LMedS feasible. The need to exhaustively try all combinations of estimates  $\mathbf{p}$  underscores a major limitation of median methods for localization. Localization is based on the idea that local search can solve the task efficiently. As will become clear later in this section, the efficiency is gained via local optimization of a smooth objective function. LMedS and other median methods are not well suited to this type of search, since the objective is the error from a single data point.

LMedS bears a striking resemblance to a couple of classic methods in pose optimization and object recognition: random sample consensus (better known as RANSAC) by Fischler and Bolles [38], the alignment method of Huttenlocher and Ullman [70], and the interpretation-tree (IT) search of Grimson and Lozano-Perez [52]. The common idea is that combinations of subsets of the data are used to estimate the (pose) parameters, after which the evaluation of the parameters can be performed. In the case of RANSAC, alignment, and IT search, the evaluation involves checking the correspondence of other model points with the observed data set, while in LMedS it involves measuring the median of the errors for all the model points. These methods solve a slightly more difficult problem than localization, however, since no set of correspondences or initial pose estimate is assumed.

While median/rank techniques can be very robust, they are also extremely computationally expensive.

## M-estimation

The third and final class of robust-estimation techniques which we will discuss is M-estimation. M-estimation is a generalization of least squares. The “M” refers to maximum-likelihood estimation.

The general form of M-estimators allows us to define a probability distribution which can be maximized by minimizing a function of the form

$$E(z) = \sum_i \rho(z_i) \quad (3.25)$$

where  $\rho(z)$  is an arbitrary function of the errors,  $z_i$ , in the observations. The equivalent probability distribution to  $E(z)$  is

$$P(z) = e^{-E(z)}$$

which, not coincidentally, is maximized by minimizing  $E(z)$ . Thus, the M-estimate is the maximum-likelihood estimate of  $P(z)$ . Our choice of  $\rho(z)$  determines  $P(z)$  which is our prior model of the distribution of errors in our observations. It also determines whether our estimate will be sensitive to unusually large numbers of outliers.

As described previously, least-squares estimation is very sensitive to outliers. Least-squares estimation corresponds to M-estimation with  $\rho(z) = z^2$ —equivalent to performing maximum likelihood estimation of

$$P(z) = e^{-\sum_i z_i^2}$$

which is readily identifiable as the probability of  $n$  independent observations of a normally distributed variable.

We can find the parameters  $\mathbf{p}$  that minimize  $E$  by taking the derivative of  $E$  with respect to  $\mathbf{p}$  and setting it to 0:

$$\frac{\partial E}{\partial \mathbf{p}} = \sum_i \frac{\partial \rho}{\partial z_i} \cdot \frac{\partial z_i}{\partial \mathbf{p}} = 0$$

By substituting

$$w(z) = \frac{1}{z} \frac{\partial \rho}{\partial z}$$

we get

$$\frac{\partial E}{\partial \mathbf{p}} = \sum_i w(z_i) z_i \frac{\partial z_i}{\partial \mathbf{p}}. \quad (3.26)$$

which, if we momentarily forget that  $w$  is a function of  $z$ , has the same form as if  $\rho(z) = wz^2$ —readily recognized as weighted-least squares. In this interpretation, the term  $w(z)$  measures the weight of the contribution of errors of magnitude  $z$  towards a WLS estimate.

The weight term can also be interpreted as confidence in a given observation. For pure least squares, we have  $w(z) = 1$ —indicating that each error has equal confidence, regardless of how large the error. A weight function,  $w(z)$ , can be defined that has the effect as outlier thresholding:

$$w(z) = \begin{cases} 1 & |z| \leq \theta \\ 0 & |z| > \theta \end{cases}. \quad (3.27)$$

In words, observations which have errors above the threshold  $\theta$  are ignored. Thus, we have another way of specifying an M-estimator, as a weight function.

There are many other possible choices of  $\rho(z)$  to reduce the sensitivity to outliers on the estimation. Table 3.1 lists several possible functions that can be used for M-estimation. Plots of the respective weight functions are shown in Figure 3.11 for comparison. All except the Gaussian function (least squares) can be considered for robust estimation.

The five robust M-estimators—the threshold function, Sigmoid function and functions by Tukey, Huber, and Lorentz—described in Table 3.1 strongly weight observations with small errors while discounting observations with large errors. Looking at the plots of the equivalent probability distributions in Figure 3.12, we see that the distributions are all similar to a normal distribution near the center of the distribution, but have noticeably larger tails (i.e., a higher prior likelihood for large errors).

In what follows, when we refer to robust M-estimation, we will be referring to the minimization of a function of the form of Equation 3.25 where  $\rho(z)$  is derived from one

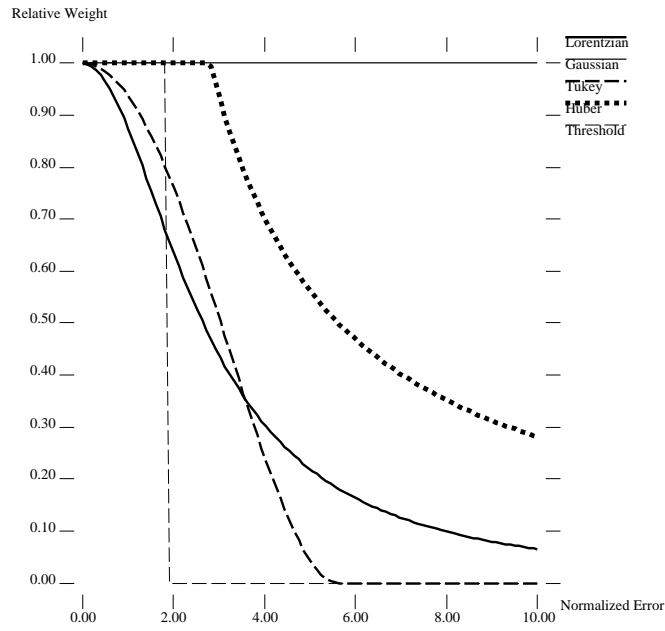


Figure 3.11: Plots of  $w(z)$  for each of the M-estimators listed in Table 3.1 .

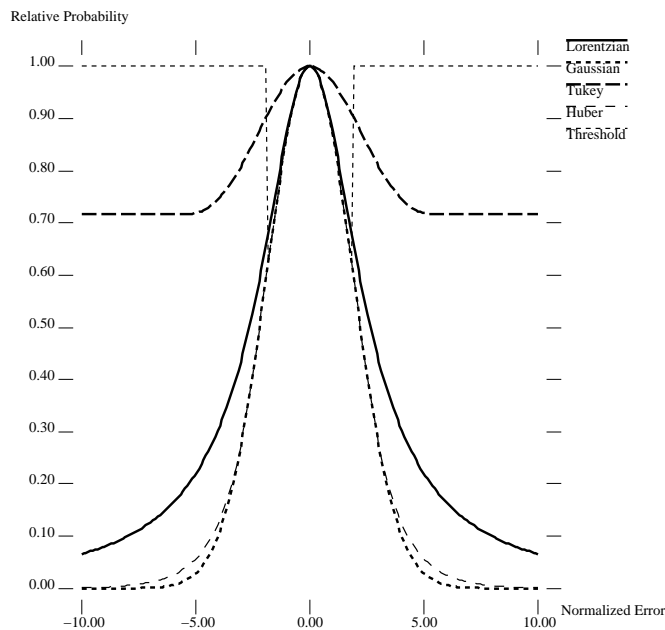


Figure 3.12: Plots of  $P(z)$  for each of the M-estimators listed in Table 3.1 .

Function Name	$w(z)$	Comments
Gaussian	1	least-squares estimation
Threshold	$w(z) = \begin{cases} 1 &  z  \leq \sigma \\ 0 &  z  > \sigma \end{cases}$	outlier thresholding, $\sigma$ is the threshold
Lorentz's function	$w(z) = \frac{1}{(1+\frac{1}{2}(\frac{z}{\sigma})^2)}$	$\sigma$ controls width of distribution [111]
Tukey's function	$w(z) = \begin{cases} (1 - (\frac{z}{\sigma})^2)^2 &  z  \leq \sigma \\ 0 &  z  > \sigma \end{cases}$	$\sigma$ controls width of distribution [137, 58]
Huber's function	$w(z) = \begin{cases} 1 &  z  \leq \sigma \\ \frac{\sigma}{ z } &  z  > \sigma \end{cases}$	$\sigma$ controls width of distribution [67]
Sigmoid function	$w(z) = \frac{1}{1+e^{\beta(z-c)}}$	$\beta$ controls falloff and $c$ is the center of the falloff, related to mixture of Gaussian and uniform background [62, 143]

Table 3.1: Table of weight functions for M-estimation.

of the five robust weight functions described above or a function with similar weighting characteristics to one of the above five functions in Table 3.1.

M-estimators are often loosely mixed with weighted least-squares (WLS) estimation. Haralick et al. [58] use an iterative reweighting technique based on WLS for the optimal pose with respect to a robust M-estimator. In our experience, weighted least squares is only appropriate when the correspondences are fixed as is standard for pose-estimation problems.

As one can see, the weight term in M-estimation is a function of the pose  $p$ . If we attempt to solve the M-estimation as a WLS problem, the function being minimized is different from the objective function, and the computed step may take the pose into a different minima. Thus the closed form (WLS) solution is no longer valid. The problem is that if the magnitude of the derivative of the weight function is great, small changes in the pose will have large changes on the function being minimized. A proper implementation of M-estimation is likely more computationally intensive than outlier thresholding, but it is likely to be better behaved.

### 3.4.3 Robust Pose Refinement

Our original problem is to solve for the pose,  $\mathbf{p}$ , of an object given a set of correspondences between visible object points  $\mathbf{x}_i$  and image points  $\mathbf{y}_i$ . Since we know that we will have many outliers in our set of correspondences, we will use a robust M-estimator to solve for  $\mathbf{p}$ . We will minimize

$$E(\mathbf{p}) = \frac{1}{|V(\mathbf{p})|} \sum_{i \in V(\mathbf{p})} \rho(z_i(\mathbf{p})) \quad (3.28)$$

where  $V(\mathbf{p})$  is the set of visible model points for the model pose parameters  $\mathbf{p}$  (computed using the methods of Section 3.2),  $z_i(\mathbf{p})$  is the 3D distance between the  $i$ th pair of correspondences. We define

$$z_i(\mathbf{p}) = \|\mathbf{R}(\mathbf{q})\mathbf{x}_i + \mathbf{t} - \mathbf{y}_i\|$$

where  $\mathbf{R}(\mathbf{q})$  is the 3x3 rotation matrix for the rotation component of  $\mathbf{p}$  and  $\mathbf{t}$  is the translation component. Since there is no closed form solution for this problem, an iterative optimization technique such as gradient descent is necessary.

Unfortunately, minimizing  $E$  is not so straightforward. The corresponding image points,  $\mathbf{y}_i$ , are a function of  $\mathbf{p}$  as well<sup>7</sup>:

$$\mathbf{y}_i = \mathbf{y}(\mathbf{x}_i, \mathbf{p}) = \arg \min_{\mathbf{y} \in D} \|\mathbf{R}(\mathbf{q})\mathbf{x}_i + \mathbf{t} - \mathbf{y}\|$$

Ignoring this fact can result in inefficiency and possibly incorrect results. This complicates the minimization algorithm in practice.

The whole idea of localization is that we can start with a crude set of correspondences and gradually converge to the correct correspondences and at the same time find the true pose of the object in the image—much like active contour approach of Witkin, Kass and Terzopoulos [75, 134] and the ICP algorithm of Besl and McKay [6]. Thus, as we search the pose space, an improvement in  $E$  should correspond to an improvement in  $\mathbf{p}$  as well as an improvement in the correspondences. If we hold the correspondences fixed for any search, we are minimizing a different function.

Fortunately, with a little work, we can perform the gradient-descent search to correctly minimize the desired function  $E$ . The basic algorithm for gradient descent search of a function  $f(x)$  with initial guess  $x_0$  is

**Algorithm** *GradientDescent*

**Input:** starting point  $\mathbf{x}$

**Input:** function  $f(\mathbf{x})$

**Input:** gradient of  $f$ ,  $\nabla f(\mathbf{x})$

**Output:** the location,  $\mathbf{x}_f$  of the local minimum of  $f(\mathbf{x})$

1. **repeat**

---

<sup>7</sup>If attributes are used to find correspondences as described in Section 3.3.2, then this function would have to include the additional error term in this equation.

2.  $f \leftarrow f(\mathbf{x})$
3.  $d\mathbf{x} \leftarrow -\nabla f(\mathbf{x})$
4.  $\lambda \leftarrow \arg \min_{\lambda} f(\mathbf{x} + \lambda d\mathbf{x})$
5.  $\mathbf{x} \leftarrow \mathbf{x} + \lambda d\mathbf{x}$
6. **until**  $f - f(\mathbf{x}) < \epsilon$
7. **return**  $\mathbf{x}$

The crucial step of Algorithm *GradientDescent* is the computation of  $\lambda$  in Line 4—generally referred to as a line minimization [111]. If we take a step  $\lambda$  in the gradient direction, a proper evaluation of  $E$  requires that we reevaluate the correspondences at that step to determine if  $E$  decreases with respect to the correspondences at that point. By doing so, the correspondences become *dynamic*, and the search begins to simulate the effect of model points floating in a potential field [75, 134, 29, 104, 133] in which the only force felt by each point is from the attraction of the nearest point—its correspondence.

Another complication is that the set of points that are visible in the model is also a function of  $\mathbf{p}$ . With a slight change of  $\mathbf{p}$ , we could see a large change in the set of visible points. This can cause great problems for a line-minimization routine since changes in visibility will cause discontinuous jumps in  $E$  as  $\lambda$  varies. Even though  $E$  (Equation 3.28) is normalized, adding and removing many observations will often have large changes on the value of  $E$ . For example, consider a cube rotating in front of a camera. As the cube rotates, different sides become visible. The equivalent to this in localization is when the localization search rotates the model to improve the pose estimate. As the estimate of the model's orientation changes, surfaces of the model may make the transition from occluded to visible and vice versa. This will cause the value of  $E$  to abruptly change. Figure 3.13 shows an example of this effect with a cross section of a cube and the value of  $E$  as the orientation changes.

To implement line minimization efficiently and reliably,  $E(\mathbf{p})$  must be smooth along the line (at least within the current basin of attraction). If we allow  $V(\mathbf{p})$  to vary, then we cannot achieve smoothness during line-minimization: this is an unfortunate fact which we must accept. Our solution is to compute the visibility set,  $V(\mathbf{p})$ , before the line minimization and keep this set fixed during the line minimization.

One may wonder what the effect of changing correspondences will have on the smoothness of  $E(\mathbf{p})$ . The answer is that  $E(\mathbf{p})$  is *usually* smooth when a change of correspondence occurs. This is because the changes of correspondences usually occur continuously with respect to  $z_i$ . For example, consider the minimum distance  $z$  between point  $\mathbf{x}$  and two other points  $\mathbf{x}_0$  and  $\mathbf{x}_1$ . If point  $\mathbf{x}$  is in between  $\mathbf{x}_0$  and  $\mathbf{x}_1$ , the distance function,  $z(\mathbf{x})$ , is maximum when the point is halfway between the two points and linearly decreases as it approaches either of the points (see Figure 3.14). The important fact is that the function is continuous as the point moves. It is not  $C^1$  as there are first derivative discontinuities at each point and the midpoint between them as can be seen in the graph of  $z$  in Figure 3.14.

When using correspondences with attributes (e.g., points and normals as described in Section 3.3.2), it is indeed possible that the minimum distance function will have discontinuities. For example, we can modify the previous example to give each point a normal as

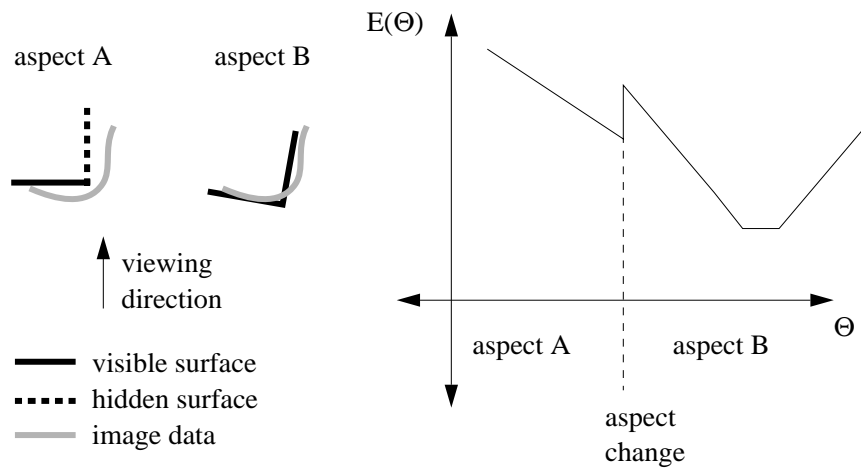


Figure 3.13: Example of the fluctuation of  $E$  due to changes in the pose and the set of visible points. The graph shows the value of  $E$  as the object rotates from one aspect to another with respect to the camera.

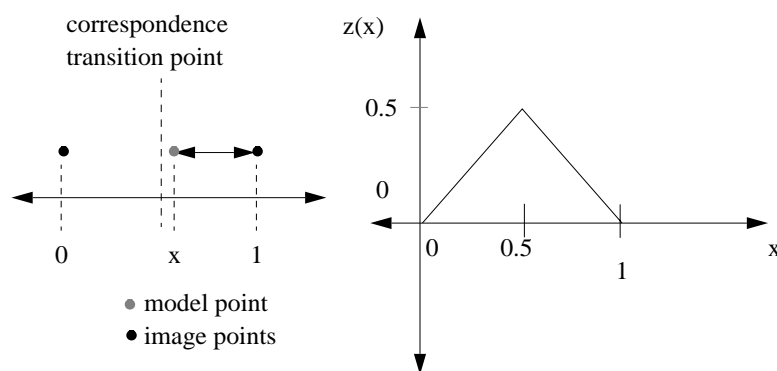


Figure 3.14: The minimum distance function with respect to the dynamic correspondences.

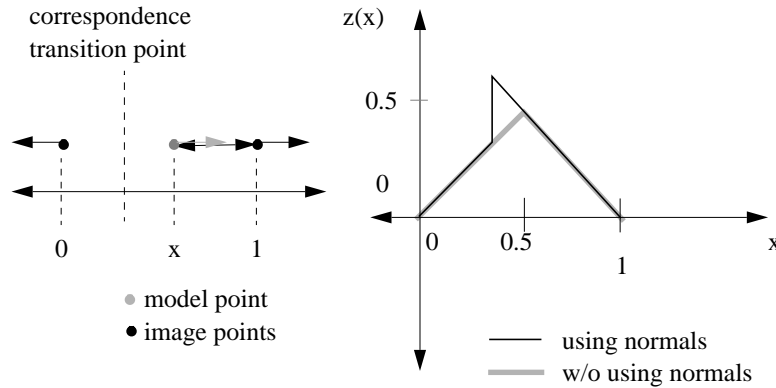


Figure 3.15: Correspondences which compare normals can create discontinuities in the distance function  $z$ . Note how the transition point (where the correspondences switch) has moved due to the influence of the normal constraint.

an attribute. If the two points have different normals, then the point at which the correspondence shifts is no longer the midpoint (depending on the weight of the normal constraint). Figure 3.15 shows this effect. Point  $x$  has a left facing normal and is, thus, more strongly attracted to  $x_0$ , whose normal is also facing left. Thus, as  $x$  approaches  $x_1$ , the transition point—the point where  $x$  becomes closer to  $x_1$  than  $x_0$  in the 6D point and normal space—is closer to  $x_1$  than  $x_0$  since the correspondence prefers  $x_0$  because of its similar normal. At some point, the proximity to  $x_1$  will overwhelm the difference in surface normal directions. As can be seen in Figure 3.15, the minimum distance function has a noticeable discontinuity at the transition point. However, such discontinuities are relatively small compared to the discontinuities caused by changes in the set of visible points.

To complete our minimization algorithm, the only thing remaining to discuss is the computation of the gradient of  $E(\mathbf{p})$ . The presence of dynamic correspondences does not cause any problems since the gradient is an instantaneous measure and the likelihood of a model point lying exactly on a discontinuity is small enough to ignore.

When computing the gradient of  $E(\mathbf{p})$  (Equation 3.28), we have from Equation 3.26

$$\frac{\partial E}{\partial \mathbf{p}} = \sum_i w(z_i) z_i \frac{\partial z_i}{\partial \mathbf{p}}.$$

It turns out that we can greatly simplify the following derivations with a few algebraic manipulations. First, instead of using

$$z_i(\mathbf{p}) = \|\mathbf{R}(\mathbf{q})\mathbf{x}_i + \mathbf{t} - \mathbf{y}_i\|$$

we redefine  $z_i$  to be

$$z_i(\mathbf{p}) = \|\mathbf{R}(\mathbf{q})\mathbf{x}_i + \mathbf{t} - \mathbf{y}_i\|^2.$$



This only requires a slight change in  $w(z)$  to compensate for having arguments that are the square of the distance rather than the distance itself. We will also assume that the model points have been pre-rotated so that the current quaternion is  $\mathbf{q}_I = [0 \ 0 \ 0 \ 1]^T$  which has the property that  $\mathbf{R}(\mathbf{q}_I) = \mathbf{I}$ . This allows us to take advantage of the fact that the gradient of  $\mathbf{R}(\mathbf{q})\mathbf{x}$  has a very simple form when evaluated at  $\mathbf{q} = \mathbf{q}_I$ :

$$\left. \frac{\partial(\mathbf{R}\mathbf{x})}{\partial\mathbf{q}} \right|_{\mathbf{q}_I} \mathbf{x} = 2\mathbf{C}(\mathbf{x})^T. \quad (3.29)$$

where  $\mathbf{C}(\mathbf{x})$  is the  $3 \times 3$ , skew-symmetric matrix (i.e.,  $\mathbf{C}(\mathbf{x}) = -\mathbf{C}(\mathbf{x})^T$ ) of the vector  $\mathbf{x}$  (the derivation of Equation 3.29 is presented in Appendix B). Multiplying by this matrix and a vector is equivalent to taking the cross product of  $\mathbf{x}$  and that vector, i.e.,

$$\mathbf{C}(\mathbf{x})\mathbf{v} = \mathbf{x} \times \mathbf{v}.$$

Using these facts,  $\frac{\partial z}{\partial \mathbf{p}}$  can easily be derived:

$$\frac{\partial z}{\partial \mathbf{p}} = 2(\mathbf{R}(\mathbf{q})\mathbf{x}_i + \mathbf{t} - \mathbf{y}_i) \frac{\partial(\mathbf{R}(\mathbf{q})\mathbf{x}_i + \mathbf{t} - \mathbf{y}_i)}{\partial \mathbf{p}} \quad (3.30)$$

$$= \begin{bmatrix} 2(\mathbf{x}_i + \mathbf{t} - \mathbf{y}_i) \\ 4\mathbf{C}(\mathbf{x})^T(\mathbf{x}_i + \mathbf{t} - \mathbf{y}_i) \end{bmatrix} \quad (3.31)$$

$$= \begin{bmatrix} 2(\mathbf{x}_i + \mathbf{t} - \mathbf{y}_i) \\ -4\mathbf{x}_i \times (\mathbf{t} - \mathbf{y}_i) \end{bmatrix}. \quad (3.32)$$

Thus, very simple formulas define the gradient with respect to the rotation quaternion and translation vector.

We now have covered the point visibility problem, the correspondence problem and pose optimization as they relate to the 3D-3D localization problem. In this section, we described an optimization method, gradient descent with dynamic correspondences, that solves the localization problem using a robust M-estimator. We can now put together the methods discussed so far into a complete algorithm for performing robust 3D-3D localization.

### 3.5 Putting It Together: 3D-3D Localization

We have described the principal components of our 3D-3D localization algorithm: point visibility, point-to-point correspondence, and robust pose optimization.

Here, we put everything together to present a pseudocode description of our complete 3D-3D localization algorithm:

**Algorithm** *3D-3D Localization*

**Input:** initial pose  $\mathbf{p}$

**Input:** range image point set  $D$

**Input:** object localization model

**Output:** final pose  $\mathbf{p}$

1. create k-d tree for the points  $D$
2. **repeat**
3.     compute the set of visible model points:  $V(\mathbf{p})$
4.      $E_0 \leftarrow E(\mathbf{p})$
5.      $\mathbf{dp} \leftarrow -\nabla E(\mathbf{p})$
6.      $\lambda \leftarrow \arg \min_{\lambda} E(\mathbf{p} + \lambda \mathbf{dp})$
7.      $\mathbf{p} \leftarrow \mathbf{p} + \lambda \mathbf{dp}$
8. **until**  $E_0 - E(\mathbf{p}) < \epsilon$
9. **return**  $\mathbf{p}$

The above algorithm is more or less a complete description of the localization as we have implemented it. There are however a few practical details which were omitted for clarity and are discussed below.

### 3.5.1 Scale's Effect on Gradient Directions

A problem affecting gradient-based search methods is that the amount of change due to rotation and translation is related to the scale of the data. The problem is that a unit change of rotation results in a change in the error that is dependent on the scale of the vector being rotated. At the same time, a unit change in translation is independent of the scale of the data. Roughly, as the scale increases, the sensitivity of the rotation parameters increases at a rate that is much faster than that of the translation parameters. This is not a good situation for a gradient-based search. The implication is that if we have a given problem and simply scale the data, the algorithm which uses gradients will follow two different solution paths when rescaled.

From the derivation of the rotational Jacobian in Equation 3.29, one can quickly derive the gradient of the following example, a typical squared error between a point  $\mathbf{x}$  at pose  $\langle \mathbf{q}, \mathbf{t} \rangle$  and an observation  $\mathbf{y}$

$$f = (\mathbf{R}(\mathbf{q}_c) \mathbf{x} + \mathbf{t} - \mathbf{y})^2.$$

Assuming that  $\mathbf{q} = \mathbf{q}_I$  (i.e., the point  $\mathbf{x}$  has already absorbed the current rotation), the rotational gradient is of the form

$$\frac{\partial f}{\partial \mathbf{p}} = 4\mathbf{C}(\mathbf{x})^T(\mathbf{x} + \mathbf{t} - \mathbf{y}) \quad (3.33)$$

$$= -4\mathbf{x} \times (\mathbf{t} - \mathbf{y}) \quad (3.34)$$

and the translational gradient is of the form

$$\frac{\partial f}{\partial \mathbf{t}} = 2(\mathbf{x} + \mathbf{t} - \mathbf{y}). \quad (3.35)$$

If the scale of our data,  $\mathbf{x}$  and  $\mathbf{y}$ , (assume  $\mathbf{t}$  is zero, or equivalently that it is already subtracted from  $\mathbf{y}$ ), is roughly  $\Sigma$ , then

$$\left| \frac{\partial f}{\partial \mathbf{p}} \right| \approx \sin \theta |\mathbf{x}| |\mathbf{y}| \propto \Sigma^2$$

using the relation between the angle,  $\theta$ , between two vectors and the magnitude of their cross product, while

$$\left| \frac{\partial f}{\partial t} \right| \approx |\mathbf{x} - \mathbf{y}| \approx \frac{\sin \theta}{\sin(90 - \frac{\theta}{2})} |\mathbf{x}| \propto \Sigma$$

using the law of sines and assuming that  $\mathbf{x}$  and  $\mathbf{y}$  approximately form an isosceles triangle. Thus, as the scale of our data increases, the gradient direction shifts towards a pure rotation, and vice versa as the scale decreases.

When the gradient is a pure translation or pure rotation, gradient-based searches will have poor convergence characteristics. Imagine that the desired pose is a pure translation. The gradient with respect to rotation will dominate, and infinitesimal steps will have to be taken during each line minimization to ensure that progress is made<sup>8</sup>. What would be desirable for most applications is that the solution to a problem at a scale where rotation and translation have roughly the same influence would be the same (modulo scaling of  $t$ ) regardless if the data were rescaled or not. To affect this is not difficult, it simply involves normalizing the data to some canonical frame (say a unit cube). It is a good idea to include the normalization in any algorithm that will deal with objects of different sizes, otherwise algorithm performance may unexplainedly get worse when applying it to a new domain.

This is related to numerical conditioning of matrices for solving linear systems [61]. For purely quadratic functions, a standard technique for rescaling the variables to form spherical objective functions is called preconditioning. Preconditioning involves computing an approximation of the inverse of the quadratic coefficient matrix of the objective function. The system is premultiplied with this approximation and if the inverse approximation is accurate the condition number of the linear system can be improved (the objective is transformed to be more spherical than elongated). Preconditioning and scaling the data will generally improve the performance of gradient-based searches.

Hartley [61] recently showed that the scale of the data can cause numerical problems for Longuet-Higgins' 8-point algorithm for computing the fundamental matrix of two uncalibrated cameras. He showed that by simply normalizing the data before applying the algorithm, the condition number of the solution matrix is greatly improved resulting in reduced error. The same logic applies to evaluation and use of the gradient in our case.

### 3.5.2 Efficient Gradient Search

Gradient-descent search (as described in Section 3.4.3) is notoriously inefficient in practice [7, 111]. While gradient descent solves the minimization problem, it does so inefficiently since each gradient step requires the evaluation of the objective function's gradient—Equation 3.26, which is relatively expensive to compute—and a line minimization. Ideally, the gradient would point directly to the local minimum of the objective function, and the

---

<sup>8</sup>A seemingly straightforward solution is to iteratively move in pure translations or pure rotations. This may be effective in some cases, but it suffers from the same problems of pure gradient-descent search. For example, if the error is pure translation, a step in rotation that reduces the error will later have to be undone to reach the desired result.

solution would be found in one line-minimization step. In practice, this is usually not the case.

By definition, each successive gradient step direction is perpendicular to the previous step direction (otherwise the directional derivative in the current search direction would still be non-zero). This means the search must travel between two points by making a sequence of 90 degree turns. Unless pointed directly toward the minimum, the search will be accomplished by a potentially large number of 90 degree steps which zig-zag back and forth towards the local minimum.

The problem is that each gradient step undoes some of the work of the previous step unless the *change* in the gradient direction along the new step direction is perpendicular to the previous step (i.e., the change in gradient is in the same direction as the current gradient). Such a direction is referred to as a conjugate direction. As the search moves away from the current point along the conjugate direction, the gradient direction of the point on the line of search continues to be perpendicular to the previous step.

For efficiency, a variation of gradient descent that follows conjugate directions, the conjugate-gradient algorithm [7, 111], is used. Conjugate gradient search avoids much of the zig-zagging that pure gradient descent will often suffer from. For purely quadratic functions, conjugate-gradient search can be shown to converge to the local minima in  $n$  line minimization steps, where  $n$  is the number of free parameters of the objective function. The modification of Algorithms *GradientDescent* and *3D-3D Localization* to utilize conjugate-gradient search is trivial [7, 111].

### Line Minimization

Line minimization (Line 4 of Algorithm *GradientDescent*) is a critical step for any localization method using gradient-based optimization. The critical issue is that the choice of line-minimization method will determine the number of evaluations of the objective function being minimized. In our case, the objective function,  $E(\mathbf{p})$  (Equation 3.28), is very expensive to evaluate—computing correspondences and summing up the weighted errors between model and image points. It is, thus, important to minimize the number of evaluations of  $E(\mathbf{p})$  to compute the minimum point,  $\lambda$ , along the search direction. For the line minimization, there are many techniques to choose from [7, 111]. We use a combination of golden-ratio bracketing and parabolic fits [111] to quickly isolate the minima in the gradient direction with as few evaluations of  $E(\mathbf{p})$  and  $\frac{\delta E}{\delta \mathbf{p}}$  as possible. If the bracket cannot be found quickly, we stop the search and return the best value of  $\lambda$  so far.

## 3.6 3D-3D Localization: Experimental Results

In this section, we present the results of our experimental evaluation of our 3D-3D localization algorithm.

We first present examples of the performance of the surface-point visibility approximation (described in Section 3.2) used by our algorithm. This gives us a qualitative feel for the accuracy and errors in the approximation.

Next, we present a qualitative evaluation of the smoothness of our objective function  $E$  (Equation 3.28) with respect to the various M-estimator functions  $\rho(z)$  described in Section 3.4.2.

We then present quantitative results demonstrating the convergence of our algorithm with respect to the various M-estimators.

To provide a baseline for comparing our algorithm with the state of the art, the convergence experiments were also performed using Besl and McKay's iterative closest point algorithm [6]—the most widely used algorithm for 3D-3D alignment and localization.

Finally, we present convergence results for a randomized version of our 3D-3D localization algorithm which shows that there is a potential for improving both accuracy *and* efficiency of our algorithm.

Note, the models used in this section (with one exception<sup>9</sup>) were created using the method of Chapter 2. The models created using this method generally contain on the order of 30,000 triangles in the mesh. Currently it is not practical to perform localization on surface meshes this dense. For the purposes of localization, we must decimate (reduce the density of triangles) the triangle mesh.

There are several algorithms developed for triangle-mesh decimation including [119]. However, most of these algorithms are designed with graphics applications in mind—all that is desired is an accurate geometric approximation. For localization it is important that the triangulation is relatively uniform across the object surface. Otherwise, the pose estimate will be biased by the dense regions of the surface—the objective function will weight dense regions more than sparse regions and the dense regions will thus be closer aligned to the image data than the sparse regions. Fortunately, there is a decimation algorithm designed with object recognition and localization tasks in mind. Johnson's decimation algorithm [73] optimizes the decimated mesh to keep the lengths of triangles nearly uniform, the result is that the surface is triangulated by nearly uniform area triangles—no long, thin or extremely large triangles are generated as is the case with other algorithms. Johnson's algorithm was used to produce lower resolution surface meshes for each object using approximately two to three-thousand triangles per object. This number of triangles is reasonable for testing our 3D-3D algorithm on the SPARC 20 workstations (resulting localization time approximately 20 to 30 seconds for large initial errors). The geometric accuracy of the decimated model is preserved in all but the high curvature regions of the surface. The decimated surface meshes are sufficiently accurate for 3D-3D localization in range images with relatively coarse resolution as used in this work— $256 \times 240$  pixels with a resolution of approximately 1 mm.

---

<sup>9</sup>The model of the Sharpei dog is courtesy of Heung-Yeung Shum [127]

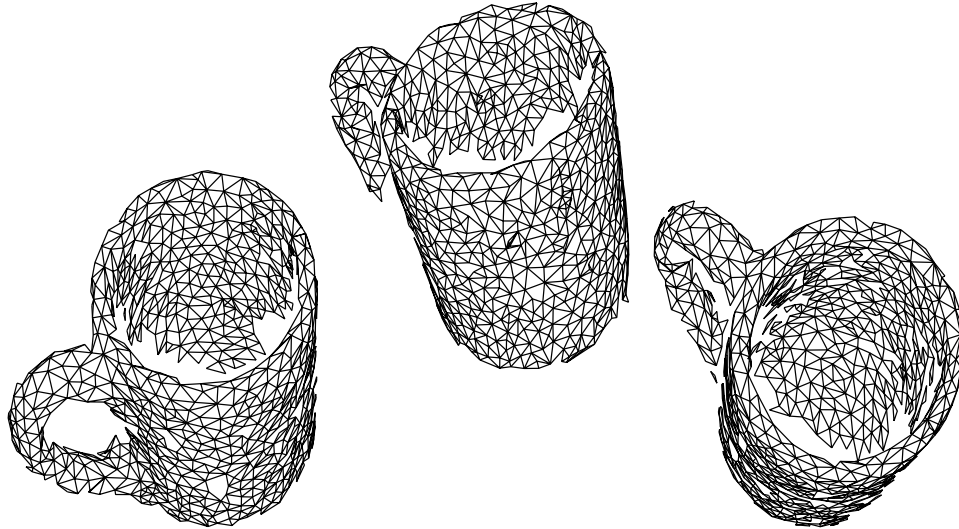


Figure 3.16: Three views of the mug object generated using our surface point visibility approximation.

### 3.6.1 Qualitative Analysis of the Surface-Point Visibility Approximation

We first briefly demonstrate the accuracy (or rather inaccuracy) of our surface-point visibility approximation as described in Section 3.2. Presenting a few simple examples will give us a qualitative feel for the accuracy and errors in our visibility approximation.

The case of interest is the approximation of hidden surface point removal. As described in Section 3.2, the visibility function for a given surface point is efficiently approximated by a lookup-table (LUT) approach. The discrete sampling of the visibility LUT results in errors since there will invariably be a bin in the point's visibility LUT which contains both occluded and unoccluded viewing directions.

The best example of these approximation errors is the ceramic mug model from Chapter 2. Figure 3.16 shows several views of the mug in which the visible surface triangles are drawn using our approximation for visibility. For our purposes of localization, we only use the center point of each triangle. A visibility LUT is defined for each center point and the triangle is drawn if the center point is determined to be visible. Note that, in all views, several triangles are missing along the border between visible and occluded surface points on the mug. Also note that no triangles are drawn if their center point is occluded by other surface triangles. This is true because our algorithm for LUT generation errs on the safe side with respect to our localization task. Generally, the number of such incorrectly classified points will be very small in relation to the total number of visible surface points.

### 3.6.2 Qualitative Analysis of the Objective Function

We now consider the smoothness<sup>10</sup> and shape of our objective function  $E$  (see Equation 3.28) with respect to the various M-estimators,  $\rho(z)$ , that we may use for robust localization. For 3D-3D localization to be successful, it is necessary that  $E$  is smooth and has a wide basin of attraction. That is, we would prefer  $E$  to be easily and reliably minimized using standard non-linear optimization techniques<sup>11</sup> over wide ranges of initial pose estimates.

Smoothness is not an obvious property of  $E(\mathbf{p})$ . We have changes in the set of visible points ( $V(\mathbf{p})$ ) as  $\mathbf{p}$  changes. The correspondences vary with respect to  $\mathbf{p}$  as well. Add to that the non-linear effect of the robust M-estimators (described in Section 3.4.2) and it is not obvious how smooth  $E(\mathbf{p})$  might be.

To inspect the shape of  $E(\mathbf{p})$ , we computed the values of  $E(\mathbf{q})$  over several range images containing known objects in known positions. Here, we show some plots of  $E(\mathbf{q})$  for a range image of a ceramic dog<sup>12</sup>. The intensity image corresponding to the range image used for this analysis is shown in Figure 3.21 (a), an off-center view of the range data is shown in Figure 3.21 (b), and the alignment of the dog model in the image is shown in Figure 3.21 (c). The correct pose of the dog was manually estimated.

Barring large amounts of noise, partial occlusion, or shadows, we would expect the correct pose estimate to be a local minima of any of the robust estimator functions described previously and this is indeed the case.

For each of the M-estimator functions  $\rho(z)$  from Section 3.4.2, we plotted the value of  $E$  along various lines in pose space which cross the desired pose. The plot in Figure 3.17 shows the values of  $E$  for each function  $\rho(z)$  along a pure translation through the correct pose (at point  $x = 0$ ). The axis dimensions of the plot is in millimeters and the range of the plot is 120 mm. The dog is approximately 120 mm high. Figure 3.18 shows the value of  $E$  for each function  $\rho(z)$  along a pure rotation through the correct pose (at point 0). The axis dimensions of the plot is in degrees and the range of the plot is 80 degrees. Finally, Figure 3.19 shows the value of  $E$  for each function  $\rho(z)$  along a rotation and translation through the correct pose (at point 0). The axis dimensions of the plot is in combined degrees and millimeters and the range of the plot is 80 degrees and 120 mm.

One can see that the  $E$  is relatively smooth for all M-estimator functions except the threshold function. Once the pose is close enough to the desired pose, the threshold version of  $E$  begins to become smoother but still has many bumps and local minima created by the differential nature of the threshold. On a closer look (see Figure 3.20), we see that  $E$  is smoother for the smooth down-weighting M-estimators such as the Lorentzian than it is for the Gaussian. In practice, this difference in relative smoothness results in poorer performance by the Gaussian.

---

<sup>10</sup>By smoothness of a function, we refer to lack of bumps and local minima along the function's landscape. Optimally, we prefer a purely quadratic function which has a single global minimum and no other local minima.

<sup>11</sup>Randomized optimization techniques such as simulated annealing [43, 62, 111, 140] are not considered here but are discussed in the future work in Chapter 7.

<sup>12</sup>The model of the Sharpei dog is courtesy of Heung-Yeung Shum [127].

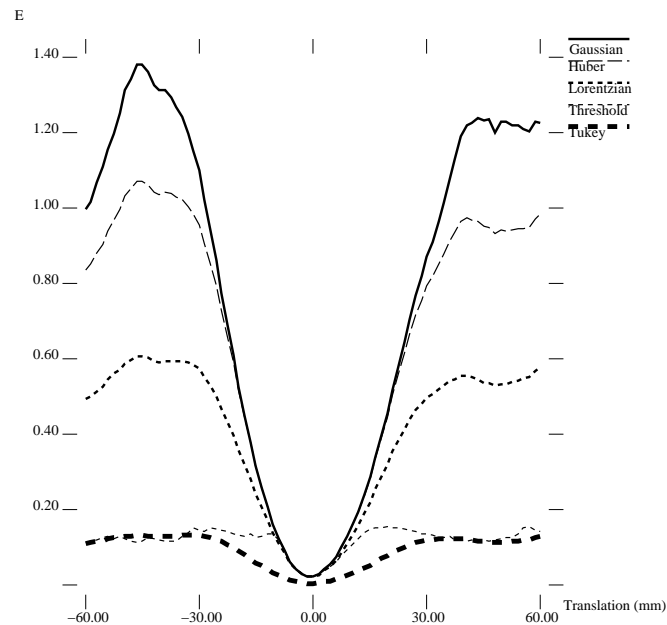


Figure 3.17: The variation of  $E$  along a line of translation in pose space for five M-estimators. The true pose is at  $x = 0$ . The  $x$ -axis represents translation in millimeters. The rough range of the graph is  $[-60, 60]$  millimeters (about half the length of the dog).

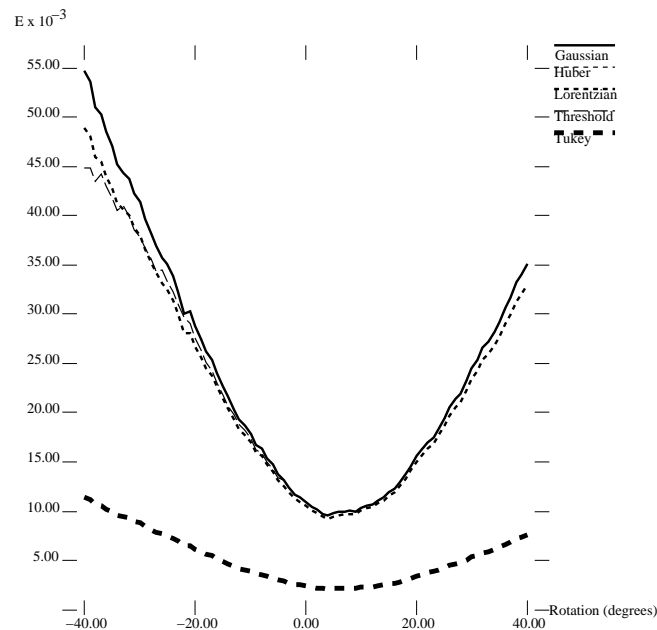


Figure 3.18: The variation of  $E$  along a line of pure rotation in pose space for five M-estimators. The true pose is at  $x = 0$ . The  $x$ -axis represents rotation in degrees. The rough range of the graph is  $[-40, 40]$  degrees.



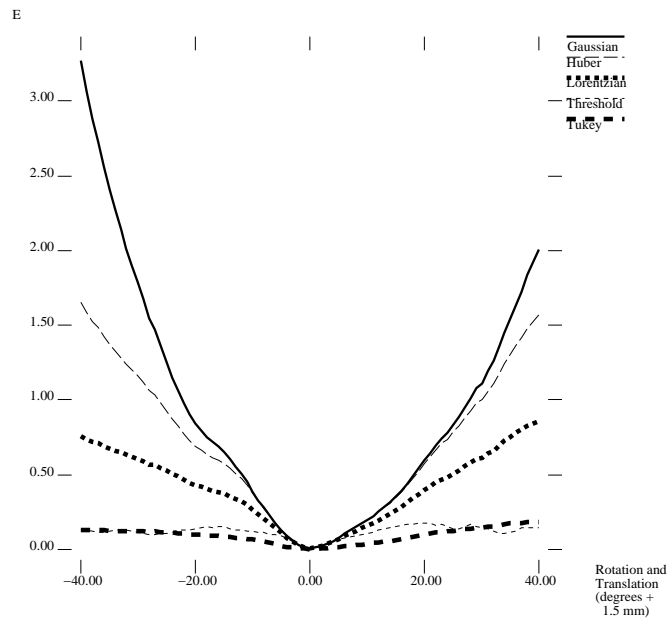


Figure 3.19: The variation of  $E$  along a line of translation and rotation in pose space for five M-estimators. The true pose is at  $x = 0$ . The  $x$ -axis represents simultaneous rotation and translation. The rough range of the graph is  $[-60, 60]$  millimeters and  $[-40, 40]$  degrees.

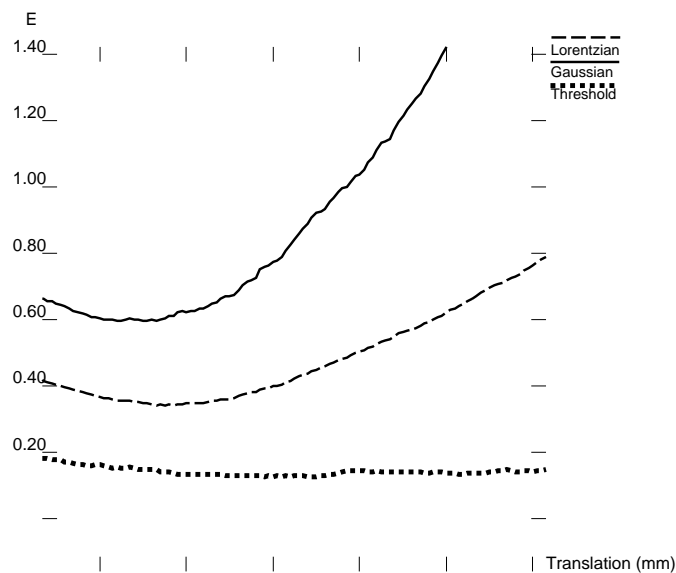


Figure 3.20: A close-up of a piece of the graphs of the function  $E$  for the Lorentzian, Gaussian and Threshold along a line of translation in pose space. The Threshold and Gaussian versions of  $E$  have many small local minima which causes problems for optimization algorithms in practice.

### 3.6.3 3D-3D Localization Convergence Results

We now present quantitative results demonstrating the convergence of our algorithm using the various M-estimators described in Section 3.4.2. To provide some basis for comparison with the state of the art, we also repeated each experiment using Besl and McKay’s iterative closest point algorithm [6]—which is currently the most widely used algorithm for 3D-3D alignment and localization.

We performed experiments on images of 4 objects: the dog, the mug, the car, and the duck. For each experiment, we took a range image of our object in a known position—using the robotic positioner or by manually estimating the object’s precise position. Each image contains  $256 \times 240$  pixels, each pixel containing a 3D point. The test images are shown in Figures 3.21, 3.22, 3.23, and 3.24. These figures show the intensity image view corresponding to the range finder’s view, an off-center view of the range-image points, an overlay example of an initial (incorrect) pose estimate as used in the experiments and an overlay of the object model on the intensity image at the estimated position.

Each experiment consists of 100 trials of the 3D-3D localization algorithm from a randomly generated initial pose estimate. The initial pose estimates were generated by perturbing each pose by a random translation vector and a random rotation. Each trial was performed with the same magnitude of initial pose error: 20 millimeters in translation and 30 degrees of rotation. We wish to emphasize that the initial errors are not uniformly distributed between 0 and 20 millimeters of translation, and 0 and 30 degrees of rotation, but are exactly 20 millimeters and 30 degrees, respectively. Thus, there are no easy (i.e., close to the actual pose) initial estimates in our set of trials. Our algorithm is capable of convergence from larger errors but characterizing the convergence range in general is difficult since convergence will vary greatly with respect to the input data. The objects presented here have dimensions of at most 200 millimeters, thus 20 mm translation errors are significant. Combining this with 30-degree rotation errors makes for a difficult localization task in general.

For each image, we performed the convergence experiment on four versions of our 3D-3D localization algorithm. The versions differ only by the M-estimator weight function which is used. The following weight functions were used: Gaussian (constant weight), Lorentzian, Tukey, and Huber. See Section 3.4.2 and Figure 3.11 for a review of these functions. To account for large initial pose errors it is important that the errors  $z$  are normalized accordingly so that the robust estimators do not immediately discount correct correspondences. The errors are normalized by a progressively decreasing normalization factor  $\sigma$  (i.e., use  $z' = \frac{z}{\sigma}$ ). This is effectively the standard deviation which controls the width of the M-estimator weight-functions (e.g., a  $3\sigma$  threshold). In these experiments, we begin with  $\sigma = 12$  mm and reduce it to 6 mm and then to 3 mm as the algorithm progresses.

In addition to the four versions of our algorithm, we also executed the same experiment using Besl and McKay’s iterative closest point algorithm [6]. ICP iteratively solves for the closed-form pose estimate (using techniques such as those by Sanso [116], Faugeras and Hebert [36], Horn [66], or Arun [1] as described in Section 3.4.1) given correspondences between model points and the closest points in the 3D image data. As described previously,

these closed-form techniques are very sensitive to incorrect correspondences and outliers. Our implementation of ICP follows the typical approach and uses a hard threshold for removing outliers. The threshold is progressively decreased from  $\sigma = 12$  mm to  $\sigma = 3$  mm as in the implementation of our algorithm as described above.

We manually verified the results of each trial and determined the number of correct trials—those that correctly align the model with the image—for each experiment. The results are listed in Table 3.2.

From these results we see that the robust weight functions have much better convergence properties than a pure least-squared error objective function (corresponding to the Gaussian). The Tukey and Lorentzian weight functions perform slightly better than Huber's weight function. We speculate that this may be due the first-order discontinuity in the Huber weight function which may create more local minima than the smooth Lorentzian and Tukey weight functions. ICP is not successful with such large initial pose errors. For each convergence trial, our localization algorithm typically performed 40 conjugate-gradient steps and 180 function evaluations for each trial (approximately 25 seconds on a SPARC 20 workstation). The ICP algorithm typically performed 30 iterations per trial.

There is nothing special about the images used for these tests (i.e., they weren't hand chosen, but rather randomly selected from a large set of sample images). They were taken with a plain black background which would suggest that the localization task would be trivial. This is a bit deceptive since they do contain a considerable amount of background noise (random points floating in space), much of which is near the object. The range data of the dog image (Figure 3.21) is the cleanest of the four which partly explains the better convergence results for that image. This noise is typical of the light-stripe range finder when imaging black points in the scene. The duck and car images are made difficult by the fact that they are nearly singular views. A slight twist of the duck will reveal many previously hidden surface points. The mug image is a tough problem for any localization algorithm because other than the handle the object is symmetric. Thus, a localization search could settle on any rotation that matches the body of the mug to the image while down-weighting or ignoring the large errors for the handle. Surprisingly the robust M-estimators were very adept at locating the precise pose of the mug, though the above problem was the main source for the trial failures. The dog object is also very tricky, the problem is that it has a lot of low curvature surface area which leaves two degrees of freedom. The localization task is also more difficult when one considers the large change in point visibility from the initial to final pose. So though these localization tasks may appear simple or trivial to most readers, there are many subtleties to beware of.

Seeing the poor convergence results for ICP in the experiments, we performed two more experiments with variations of ICP. We repeated the duck image experiments using a weighted-least squares version of ICP and using the Gaussian and Lorentzian weight functions. This was done to determine if the use of hard thresholds (and their known susceptibility to local minima as demonstrated in Section 3.6.2) were the reason for ICP's failure and whether smoothly weighting the errors (as done by the other robust M-estimators) would offer an improvement. The results of these experiments are shown in Table 3.3. The results showed that by changing to a smooth down-weighting function (the ICP w/

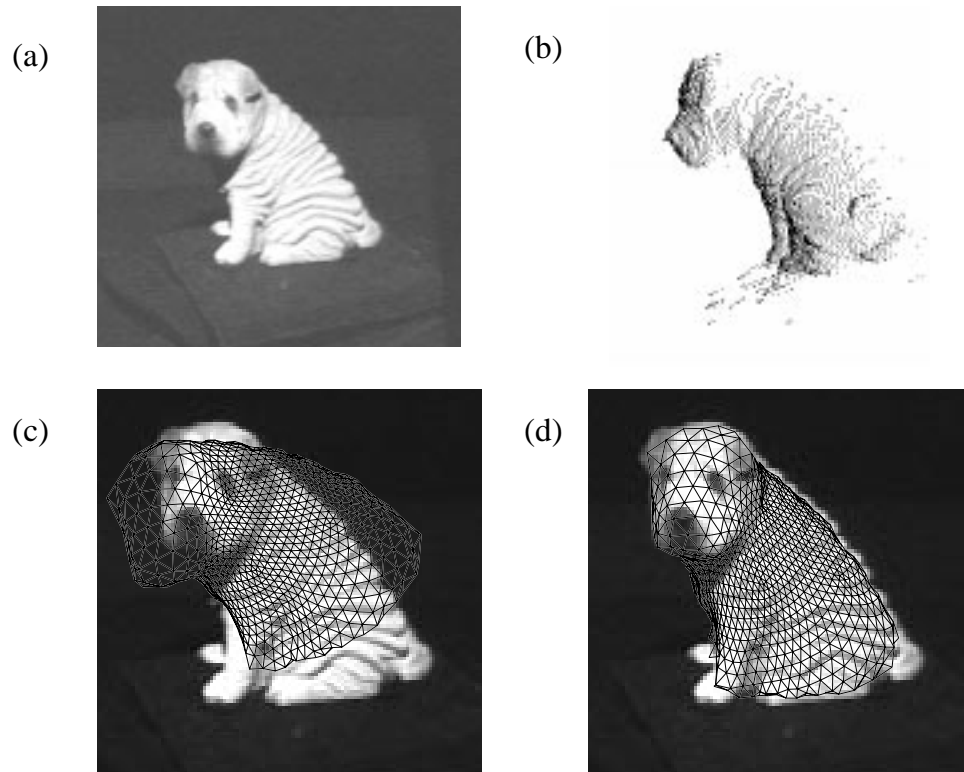


Figure 3.21: The Sharpei dog convergence test data: (a) the intensity image view, (b) an off-center view of the 3D range data points, (c) a typical initial starting point for the convergence tests (20 mm translation error and 30 degrees rotation error) (d) an overlay of the dog model at its estimated actual location.

Object	% Correct				
	Gaussian	Lorentzian	Tukey	Huber	ICP
dog	65	98	100	85	24
mug	52	70	67	67	2
car	37	54	61	57	4
duck	50	60	58	58	9

Table 3.2: Results of the convergence experiments with variations of our 3D-3D localization algorithm and ICP. The initial errors were of uniform magnitudes: 30 degrees of rotation and 20 mm of translation.

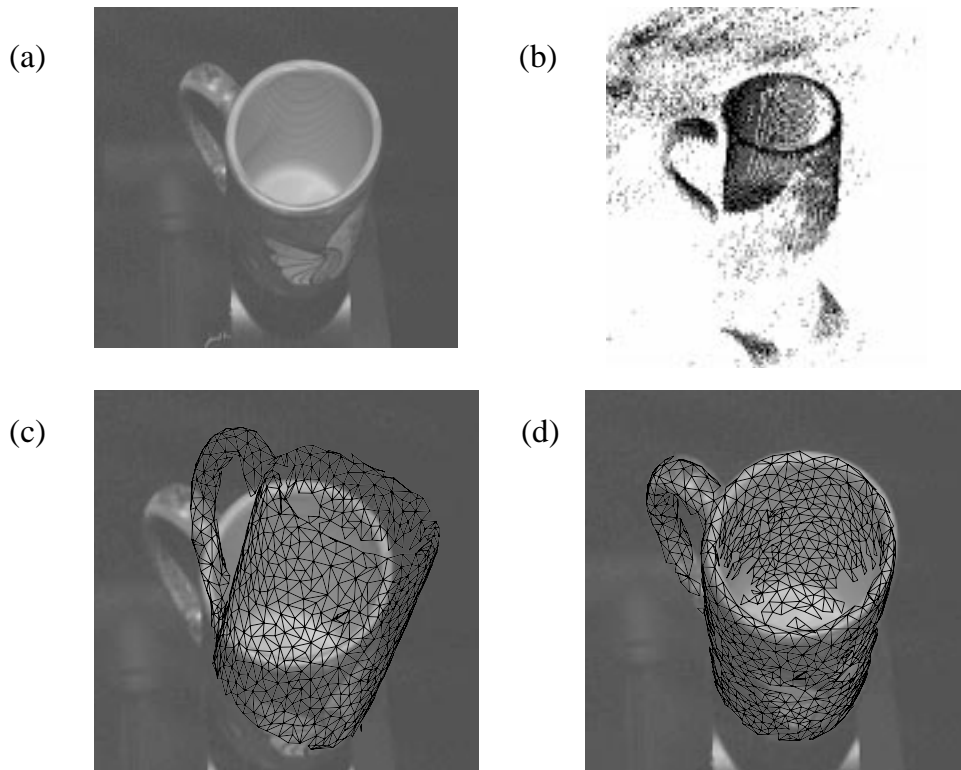


Figure 3.22: The ceramic mug convergence test data: (a) the intensity image view, (b) an off-center view of the 3D range data points, (c) a typical initial starting point for the convergence tests (20 mm translation error and 30 degrees rotation error), and (d) an overlay of the mug model at the estimated location.

Lorentzian case) the convergence is improved. It also showed that using a hard threshold is slightly better than none at all (the ICP w/ Gaussian case). However, neither attempts managed to improve the convergence to approach that of any of the versions of our 3D-3D localization algorithm.

We offer a few hypotheses about ICP's relative inability to converge to the desired location for the above images:

1. ICP does not anticipate changes in visibility sets when taking a step in pose space. It often inadvertently forces itself out of the local basin of attraction by stepping to a view with a significantly different set of visible points. For example, in the duck example (see Figure 3.24), the view is singular. A slight rotation introduces many newly visible surface points on the duck. Thus, creating a situation where ICP has jumped to a position of a function which it has little information about. ICP was originally designed to register two (true) 3D data sets. Range images are incomplete

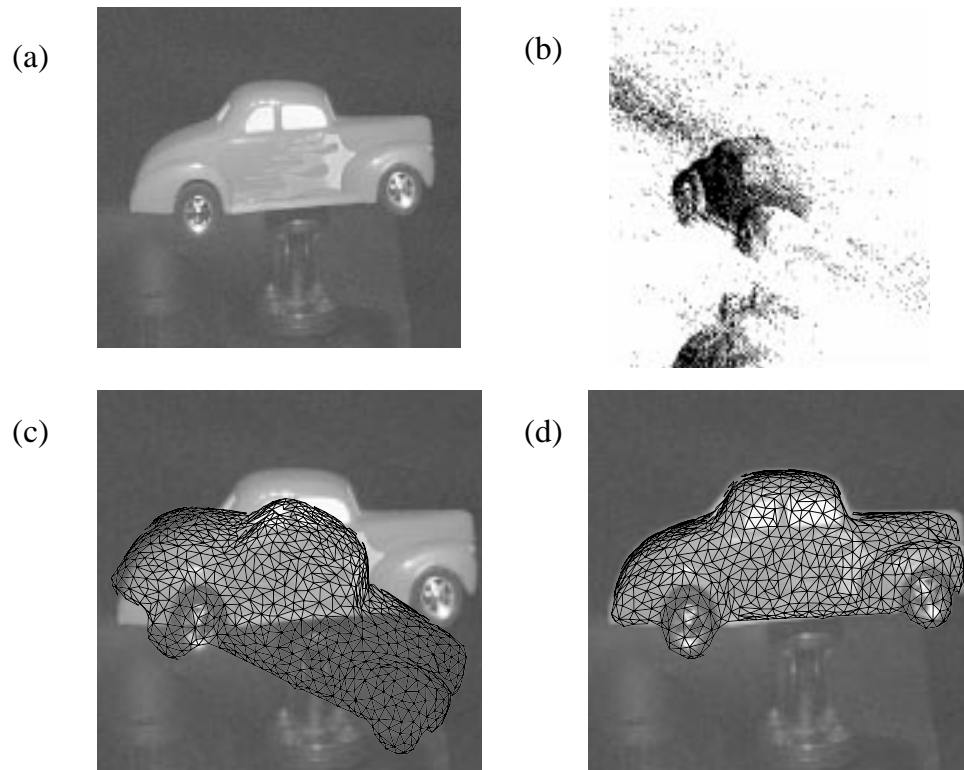


Figure 3.23: The toy car convergence test data: (a) the intensity image view, (b) an off-center view of the 3D range data points, (c) a typical initial starting point for the convergence tests (20 mm translation error and 30 degrees rotation error), and (d) an overlay of the car model at the estimated location.

3D data sets with respect to an object. The overlap between model and image data will necessarily be limited and ICP was not designed with this in mind.

2. ICP does not cope well with inaccurate correspondences. In our experience, it does well if all or most of the image data belongs to the object or if the initial pose error is small. The images (a) through (c) have a substantial amount of noise. Range data points are randomly scattered about in the background (see Figure 3.24) and are often found to be correspondences. The closed-form pose estimation methods do not deal gracefully with such inconsistent correspondences.
3. ICP attempts to make a global decision using information (correspondences) which is often only valid locally.

These results demonstrate that using dynamic correspondences within a gradient-descent search of a robust objective function is a key component to achieve a wide degree of

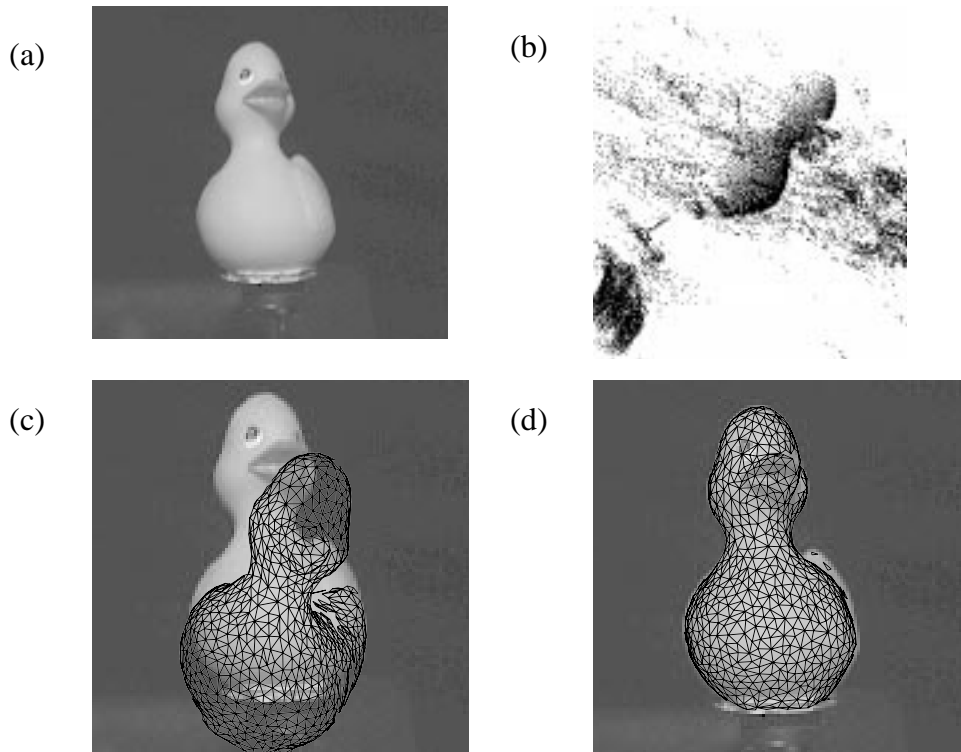


Figure 3.24: The rubber duck convergence test data: (a) the intensity image view, (b) an off-center view of the 3D range data points, (c) a typical initial starting point for the convergence tests (20 mm translation error and 30 degrees rotation error), and (d) an overlay of the duck model at the estimated location.

Object	% Correct			
	Lorentzian	ICP w/ Lorentzian	ICP w/ Threshold	ICP w/ Gaussian
duck	60	17	9	7

Table 3.3: Results of the convergence experiments with variations of ICP on the duck image. The leftmost entry is the result of our 3D-3D localization algorithm using the Lorentzian weight function and is intended as a point of reference.

convergence for object localization in noisy image data.

### 3.6.4 Randomization Experiments

Viola [140] demonstrated a method for aligning shapes by maximizing the entropy between two functions. A key result of his work was the demonstration of a randomized gradient-

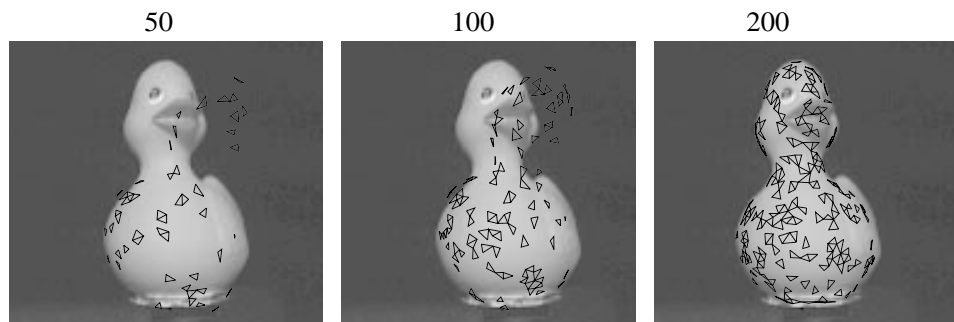


Figure 3.25: Depiction of three iterations of the randomized localization search. Each image shows an overlay of the randomly selected points from the model surface which were used for localization during the iteration.

descent algorithm which increased the speed of the algorithm and reduced the susceptibility of his algorithm to local minima (the randomness helps the algorithm to get “unstuck”). While local minima is also a problem for almost every localization algorithm, including ours, the opportunity for algorithm speedups via randomization may be critical for practical applications.

In this section, we present the result of an experiment to test the convergence of a modification of our 3D-3D localization algorithm to use a simple randomization strategy. The modification is to perform each gradient computation and line minimization (Lines 6 and 7 of Algorithm *3D-3D Localization*) of our search using only a small randomly sampled subset of our model data to measure the objective function. Since the accuracy of the localization result will increase with the number of samples of the error, we progressively increase the number of random samples just as we decrease the standard deviation  $\sigma$  in the above convergence trials. In these randomization experiments, we doubled the number of surface point samples as we halved the value of  $\sigma$ . The rationale is similar to the simulated-annealing search algorithm [111, 43, 62, 24] which starts out with a high degree of randomness in the search and gradually settles to non-random search. With a small number of point samples we may not be able to accurately estimate the pose, but we may be able to quickly find a better estimate. As we get closer to the true pose, the convergence rate improves thus we can afford to add more points as we get closer to get the desired accuracy of the pose estimate. Figure 3.25 shows a snapshot of three iterations of the randomized localization algorithm. Each overlay shows the randomly selected points from the model surface which were used for localization during the iteration. The number of sampled points is gradually increased as the pose estimate converges to the true pose.

We performed three experiments with the following randomization schedules: 200 to 400 to 800, 100 to 200 to 400, and 50 to 100 to 200. The results of these experiments (again performed using the duck image of Figure 3.24) are presented in Table 3.4.



Object	% Correct			
	Non-Random (3000)	200,400,800	100,200,400	50,100,200
duck	60	57	52	34
speed (seconds)	27.5	8.5	6	4.5

Table 3.4: Results of the randomization convergence experiments. Each column lists the number of surface point samples used in the randomization schedule, the number of percentage of correct results and the execution time in seconds. All these experiments used the randomization modification of our 3D-3D localization algorithm with the Lorentzian weight function. The initial errors were of uniform magnitudes: 30 degrees of rotation and 20 mm of translation.

As these results show, the convergence accuracy gradually decreases as the random set size decreases. The speedup is significant, however, thus the potential exists for applying randomization techniques to capitalize on the relationship between speedup and convergence and produce an algorithm that is faster and has wider convergence on average. For example, running the randomized algorithm multiple times from slightly different starting points may be more efficient *and* more likely to converge to the desired global minima. This work is beyond the scope of the current thesis.

### 3.7 3D-3D Object Localization: Summary

We have described the principal components of our 3D-3D localization algorithm: point visibility, point-to-point correspondence, and robust pose optimization. Our algorithm iteratively optimizes an objective function which is specified to reduce the effect of noise and outliers which are prevalent in real image data.

Since an iterative algorithm is required, the computation of the objective function must be as efficient as possible. Our solution utilizes an efficient and accurate approximation to compute point visibility with respect to the pose and camera parameters. Point correspondences are efficiently computed using k-d trees and nearest-neighbor search. We showed how to extend the point correspondence search to include attributes, such as surface normals, in the nearest-neighbor search.

The most significant contribution of this chapter is our method for minimizing a robust M-estimator via dynamic correspondences with standard non-linear optimization techniques. We described the use of robust M-estimators to define an objective function over which to optimize the pose such that the effect of noise and outliers is greatly reduced. We showed that optimizing such objective functions requires the use of dynamic correspondences to properly evaluate the objective function. The optimization was implemented using standard techniques such as conjugate-gradient minimization.

Our results demonstrate that our 3D-3D localization algorithm using a smooth down-weighting M-estimator, such as the Lorentzian, can consistently and efficiently converge from rather large initial pose errors despite the presence of large number of extraneous range-image points around the object. The convergence results showed that our algorithm has much wider convergence ranges than Besl and McKay's ICP algorithm [6]. We also showed that through randomization, the potential exists for a substantial speedup and convergence improvement.

This ends our description of our 3D-3D localization algorithm. Experiments presented in Section 3.6 show that this algorithm has a acceptable convergence time and produces accurate results. We will now describe our extensions of this work to 3D-2D localization. Our discussion begins with building object models for the 3D-2D localization task and is followed by a description of our algorithm for 3D-2D localization.

# Chapter 4

## Object Modeling for 3D-2D Localization

The previous two chapters have described our approach for 3D object modeling and 3D-3D localization. This chapter focuses on the problem of building models for the purpose of localizing 3D objects in 2D intensity images—from here on referred to as 3D-2D localization.

In many ways, model building for 3D-2D localization is similar to the problem of building 3D models, which was addressed in Chapter 2. As in that work, here we seek to build a model—for the task of localization—from several real images of the object while automating the process as much as possible. 3D-2D localization is an inherently more difficult problem than 3D-3D localization because the appearance of an object in an intensity image is dependent on a combination of geometric and photometric properties of the object rather than geometry alone as in the 3D-3D case.<sup>1</sup> The choice of model representation is also not immediately obvious as the intensity images are 2D projections of 3D phenomena.

### 4.1 Problems and Approach

In this section, we overview the issues, problems, and our approach to building the models for 3D-2D localization.

Before we can begin to build our object models for 3D-2D localization, we need to know what information about the object will be required to solve this task. In order to localize an object in an intensity image, we must establish some correspondence between a localizable feature of the object and its 2D projection in the image. What features should be used? Once we have decided on a certain set of features, we need to decide upon a model representation that allows us to predict the appearance of those features in various images of the object. After the features and model representation are decided upon, we

---

<sup>1</sup>Photometric properties and effects do play a role in range-image acquisition (e.g., effects such as interreflectance and specular reflection can cause problems for light-stripe range finders and stereo vision). In general, these effects are ignored except during image processing. The range sensor effectively abstracts away these effects.

then face the problem of building such models from real images of the object. We break the following discussion into three sections:

1. What features should we use for 3D-2D localization?
2. How should we model the features?
3. How do we build such a model from a set of images of the object?

The focus of this chapter is on Question 3, but before jumping in we will answer the first two questions—thus motivating our approach to 3D-2D localization (Chapter 5).

### 4.1.1 Features for 3D-2D Localization

If we are to align a 3D object in a 2D intensity image, we need to correlate the model to its the image. There are several types of features which have been used to correlate/matching an object with its intensity image:

- Pixels/Regions: intensity/color/texture
- Interest points: corners, junctions, inflection points, distinguished points
- Edge curves: lines/arcs/ellipses fit to edgel chains
- Edgels: edge points extracted by the edge operator.

We will discuss the use of each of these features for model-to-image correlation.

One possibility is to correlate regions of constant surface properties on the model with intensity, color or texture of pixels/regions in the image. This is difficult because the image intensity of an object region may vary greatly because of changes in surface orientation, lighting, shadows and reflections. Predicting pixel intensity or color requires detailed knowledge of scene geometry, surface reflectance properties, light source properties and positions, and the camera's geometric and photometric parameters. Without being able to predict sensed intensities or colors, matching prediction with observation for localization will be very error prone.

There have been several recent findings and results which may make pixel/region correlation feasible in the near future. Nayar and Bolle [100] discovered a photometric invariant involving the ratios of intensities between two neighboring points on a smooth surface. Maxwell [93] has developed techniques to analyze and segment color images into connected regions of similar material properties. Texture information may vary because of the factors which influence intensities. However, texture the spatial properties of certain textures may be reliably detectable—using methods such as Krumm's [78] local spatial frequency—over a wide range of lighting and scene variations. Changes in texture appearance caused by changes in orientation and scale can be predicted since the rough object location is known during the localization process. A drawback is that while many objects have distinctive texture, most objects do not. Thus, texture alone will not be sufficient to

localize most objects. Viola [140] presented an approach that minimizes the entropy of the correlation of the model shape with the intensity image rather than simple pixel intensity correlation. This approach avoids some of the above problems for some simple object shapes and reflectance properties. His approach is discussed further in Chapter 6.

A popular class of features used for structure from motion [135, 130, 124, 106] and robot navigation [98, 91, 92, 135] are interest points and corners. Definitions of these interest points may differ but most follow the same principal: small image windows with distinct intensity profiles. Some definitions include image windows with high standard deviations in intensity, zero crossings of the Laplacian of the intensity, or corners. Tomasi [135] defines a criterion for interest points based on the likelihood that the point can be tracked from image to image. A similar feature is found by analyzing edge curves extracted from the image and selecting junctions of multiple curves and high curvature (corner) points. These kind of features are useful for motion estimation because they are distinct and can reliably tracked from image to image. Such interest points are usually too sparse for localization and verification applications; however, they may be good candidates for recognition indexing schemes where sparseness is more of an asset.

Another possibility is intensity edges—discontinuities in the image’s intensity profile. Using present day techniques, intensity edges are the most reliable, well-understood, practical, and, not coincidentally, the most popular feature for object recognition and localization in intensity images. Intensity edges occur at points in the scene where there are geometric discontinuities (e.g., occluding contours of objects), surface orientation discontinuities (e.g., peak edges or corners) or surface reflectance discontinuities (i.e., transitions of surface color or reflectance). These intensity edges have a direct correlation with points on the object surface; however, there are other sources of intensity edges which may appear on an object but are not intrinsic to the object—examples include cast shadows on the object, specular reflections, or digitization noise. The intensity edges that are intrinsic to the object are themselves subject to external sources of variation. For example, occluding contours will only be apparent in the image if the background reflects a sufficiently different intensity than the object, or surface markings may only be visible when viewed from certain directions or under certain light source conditions. However, in general, the intrinsic discontinuity sources of an object are detectable over wide changes in illumination and can be accurately predicted without precise knowledge of the light source.

There are several types of intensity edge features. Curves (e.g., line segments, arcs, ellipses, etc.) fit to a collection of edgel data is the most common [47, 85, 110, 48, 49, 122, 54, 3, 15, 113]. Junctions and corners of the fitted curves are also a popular choice of feature [81, 15, 110, 54]. Junctions and corners of edge curves are most similar to the interest points described previously. Models consisting of a collection of curves and junctions can be localized by matching them to the image curves and junctions. These features are simple to specify and the process of curve fitting greatly reduces the combinatorics of matching since the number of curves and junctions will invariably be much smaller than the number of edgels in the image. However, using edge curves immediately constrains the type of objects for which localization/recognition systems will be applicable.

Another option is to dispense with relying on a high-level feature extractor and use the raw edgels themselves. This is the approach taken by [14, 143, 124]. Other than the pixel intensities, edgels are likely the lowest-level information available from the image that we can use for localization. This approach has been used recently by quite a few researchers [143, 17, 14, 124] developing methods following the alignment paradigm of Basri and Ullman [139]. Using the lowest-level edge data to perform localization makes some sense since localization is not necessarily concerned with the same combinatoric matching problem as in the general object-recognition problem. Thus, more data is not necessarily a bad thing. This reduces the number of assumptions made by the algorithm and restrictions placed on our system. The algorithm will also be more robust to the small variations in edgel data which have significant effects on curve fitting or higher level features extracted from the edgel data. Also, collections of edgels can represent a wide variety of shapes.

For the reasons discussed above, we will use intensity edgels as our basis for localizing 3D objects in intensity images. Of course, using combinations of color, intensity, texture, and intensity-edge information would be preferable if practical, but for this thesis, we focus on using intensity edgels for 3D-2D localization.

### 4.1.2 Edgel Models for 3D-2D Localization

Now that we have decided on the type of feature to use for localization, we must choose an appropriate representation. The key requirement for our representation is the ability to efficiently predict the appearance of the object edgels from any given viewing direction or arbitrary camera parameters. We must also keep in mind the difficulty of automatically building the model from real images. In the following we consider two general approaches: 2D representations and 3D representations.

#### 2D Edgel Models

By 2D representation, we refer to representing an object by a set of representative 2D images (views). In our case, each view would be an edge image.

Each view is a discrete sample from the space of object locations and orientations, as well as lighting conditions and camera parameters. To model an object using this approach, we must sample a sufficient number of views to cover the expected variations for the given task or application. The number of required views can be quite large for even a few degrees of freedom since the number of required views is roughly exponential with respect to the number of degrees of freedom which will be modeled. For example, for each degree of freedom, if we take  $n$  samples along each degree of freedom, we will have  $n^d$  samples in total where  $d$  is the degrees of freedom. In addition to sampling the view space of the object, for the task of localization, this space must be made continuous, this is done by connecting edges in neighboring views so that intermediate views can be interpolated. Computing these correspondences between two views is nearly an impossible task in general, since given any two neighboring views, the set of visible points is almost never the same.

Shashua [124] proposed an approach for modeling reflectance and geometry by computing correspondences using optical flow. Gros [54] described a framework for matching line segments and points by approximating the motion between views as an affine transformation and searching for an affine transformation using a clustering technique similar to the Hough transform [2].

Much work has been done using 2D representations dating back to Basri and Ullman [139]; this and other work using 2D representations is discussed in Chapter 6. This representation is most popular for ease of building models, however, the price that is paid is accuracy and applicability. The motivation for most of the 2D representations is for the problem of object identification or indexing. Only the alignment method of Basri and Ullman [139] is suited and intended to localize and verify an hypothesized pose; however, their method is only applicable to orthography or scaled orthography.

Other basic types of 2D representation include efficient methods for image correlation and pattern recognition. Murase and Nayar [99] introduced the eigenspace representation for image sets. Using their method, the object is represented by a large number of intensity images. Eigenspace analysis is used to reduce the images to points in a low-dimensional subspace (the principal components of the eigenspace). Image matching can then be efficiently performed in the subspace. Huttenlocher, Lilien and Olson [69] used this method to represent binary edgel images of the object, which is more robust to changes in lighting. They cleverly showed that the correlation between vectors in the subspace is approximately the same as a Hausdorff distance between the binary edgel images. The Hausdorff relationship makes their formulation somewhat insensitive to partial occlusion as well. The eigenspace techniques are essentially an efficient form of image correlation and suffer from all the same problems as correlation. Another related technique is the use of artificial neural networks to solve pattern matching problems [107, 33, 114, 97]. Implicitly, the networks learn a functional mapping between images and object/view identification—equivalent to image correlation and pattern recognition. Localization and verification is not possible in this framework, though these methods show promise for solving the indexing problem.

### 3D Edgel Models

The other choice for representation is a 3D representation. Here each edgel is represented as an edgel in three dimensions in, preferably, the coordinate system of the object. The advantage of this representation is that we only need one model for all possible object poses and camera parameters. The main drawbacks of such a representation are how to acquire such a model from 2D data and how to efficiently predict the appearance from a given view.

Much of the previous work [47, 85, 122, 3, 15] on 3D representations of edges for 2D recognition and localization tasks have been based on CAD modeling and acquiring the 3D edge model from CAD geometry; however, most of this work has been limited to simple features such as straight edge segments, arcs, and junctions [81, 15, 110, 54]. In general, CAD modeling is labor intensive and often results in crude models.

Our approach is to treat edgels just like surface points in the 3D-3D case. That is, the model is a collection of edgels (oriented points) on the object where each edgel has a computationally efficient, local visibility criteria associated with it. Given such a model, we can easily and accurately predict appearances of the object's edges continuously with respect to pose for any given camera model. The rest of this chapter describes our approach to build an object centered, 3D edge model.

### 4.1.3 Building 3D Edgel Models from Real Images

We have decided to build an object-centered, 3D edgel model from a set of intensity images of the object. Many of the problems involved with this task are very similar to the problems of Chapter 2. The steps of view acquisition, view alignment, and data merging seem relevant for the 3D edgel modeling problem as well. The basic problems involved in building 3D edgel models from real images are:

1. Distinguishing background from foreground edgels
2. Converting 2D data to 3D data
3. Aligning all of the data into the same coordinate system
4. Merging all the edgel data to form a single 3D edgel model
5. Dealing with occluding contours

We now will briefly overview our method for building 3D edgel models for 2D localization, while addressing these problems along the way.

We begin by acquiring a set of intensity images of the object that adequately samples the range of viewing directions and, if desired, lighting conditions that are expected when applying 3D-2D localization. The intensity images are then processed to provide sets of 2D edgels. We must then deal with a non-trivial complication—creating a 3D model from a set of 2D projections of the 3D object. There are possible ways to get 3D information from sets of 2D information such as structure from motion [82, 135, 130, 124, 106] or epipolar/stereo analysis [45, 131]. None of these are currently practical, however.

Fortunately, we have a trick up our sleeve, namely, the 3D modeling techniques described in Chapter 2. That is, we are able to build a 3D model of an object's surface geometry. Using the 3D surface model solves the first three problems: the foreground/background problem, 2D-to-3D conversion, and alignment of the 3D data. In fact, a common theme of our work on 3D-2D localization is to use 3D information, whenever possible, to solve the 2D problems.

Once all the 2D edgel data is mapped and aligned in 3D model coordinates, the 3D edgel data must be merged into a single, unified edgel model. Our algorithm for merging edgels is closely related to the consensus-surface algorithm of Chapter 2 and is appropriately called the *consensus-edgel* algorithm. The idea is to find clusters of similar edgels from multiple views. Much like the surfaces from range images, the edgels that are extracted



from intensity images are often noisy, and spurious edgels are frequently detected due to reflections, specularities and image noise. Thus, we do not want every observed edgel as part of the final model. Only stable edges—edges that are consistently detected over many views—are desired. By searching for a consensus of edges from the observed edgel data, we can guarantee that only stable edgel generators are part of the edgel model.

The consensus edgel algorithm is not sufficient for our purposes however; it is only applicable for building models of rigid edgel generators. That is, edgels whose position with respect to the 3D model is fixed (e.g., surface markings or corners). The other class of edgels, non-attached edgels, are those resulting from occluding contours of an object such as the side of a cylinder. Occluding contours appear to be detectable over many views. For example, as a cylinder rotates about its axis, the projection of the side contours remains stationary in the 2D image. However, the object coordinates of the apparent contour generator is constantly changing as the cylinder rotates. This is because the contour generator is changing between views (i.e., a different point is generating the perceived contour). To model the edgels generated by occluding contours, we can do no better than modeling the geometry of the object surface. Since we already have an accurate surface model, we utilize the surface geometry to predict the appearance of occluding-contour edgels for localization.

Figure 4.1 shows a diagram of the technical sections of the thesis. This chapter has two principal components dealing with the modeling of rigid edgels and occluding-contour edgels. Rigid edgels (including surface markings and corner edgels) are modeled from a large set of sample images of the object. We first describing in detail each step of our rigid-edgel-modeling algorithm: view acquisition, view alignment via 2D to 3D mapping, and edgel merging using the consensus-edgel algorithm. We then discuss the geometric curvature analysis required to model occluding contour edgels of an object.

## 4.2 View Acquisition

The first step of the edgel-modeling process is view acquisition—sampling intensity images of the object from various views and processing them to produce 2D edgel sets representing each view. As in Chapter 2, we do not explicitly deal with the problem of view selection. We simply assume that we are given a set of views that provides adequate coverage for our purposes.

To compute edgels from an intensity image, the usual process involves convolving the image with an edge operator and linking the maxima of this operator into edgel chains. The choice of edge operator is rather arbitrary, and there are a number of them to choose from. The Canny operator [16] and Deriche operator [30] are the two most popular edge operators for computer-vision applications.

We use the Canny operator, although other operators would give similar results. The Canny operator is the most common edge operator and implementations are found in nearly every computer-vision software package. It is implemented as a first derivative of a Gaussian convolution operator, which Canny [16] showed to be a very close approximation of the optimal convolution operator for detecting step edges. The Canny operator measures

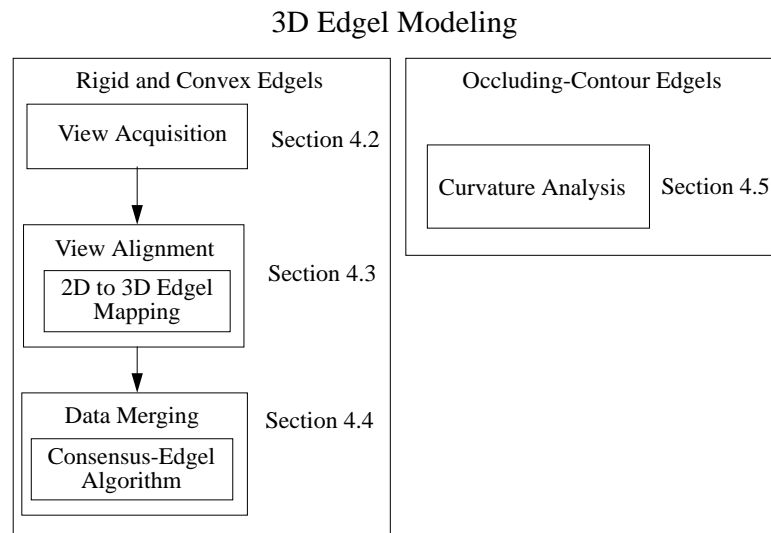


Figure 4.1: Organization of this chapter.

the edge strength at each point in the image. All pixels that are not local maxima of the edge measure are suppressed. The remaining pixels are local maxima of the Canny operator and are linked to form edgel chains. The linker uses hysteresis thresholding [16] to connect adjacent edgels that may not be as strong as other edgels on the chain. This prevents edges from being broken into smaller pieces randomly due to fluctuations in the measured edge strength along the chain. Linking also eliminates many spurious points from the immediate neighborhoods of strong intensity edges.

The result of the Canny detector and edgel linking is a set of raw edgel chains. Figure 4.2 (a) shows an example of typical raw edgel chains from a Canny operator. As can be seen in that figure, these chains are jagged—much like a staircase—due to the discrete nature of the image pixels. For the purposes of modeling and localization, smooth edge data is very desirable (this will be made clear later in this and the next chapter).

Edge smoothness can be achieved by applying a smoothing operator over each edge chain [88, 147]. Figure 4.2 (b) shows an overlay of the edgel chains resulting from such a smoothing operation. As can be seen, the smoothing removes much of the noise and aliasing effects (kinks) from the edgel chains while maintaining accuracy of the edgel positions (an important aspect for localization). In fact the edgel positions for most edgels are improved to sub-pixel accuracy after the smoothing removes the discrete kinks in the chains. For details of the smoothing operation, the reader is referred to the paper by Wheeler and Ikeuchi [147]. Smoothing also makes the computation of edgel normals much more reliable. After smoothing the edgel chains, it is possible to subsample the edgel chains if desired to get a denser set of edgels than pixel resolution.

The result after image acquisition, edge detection, and smoothing is a set of edgel chains

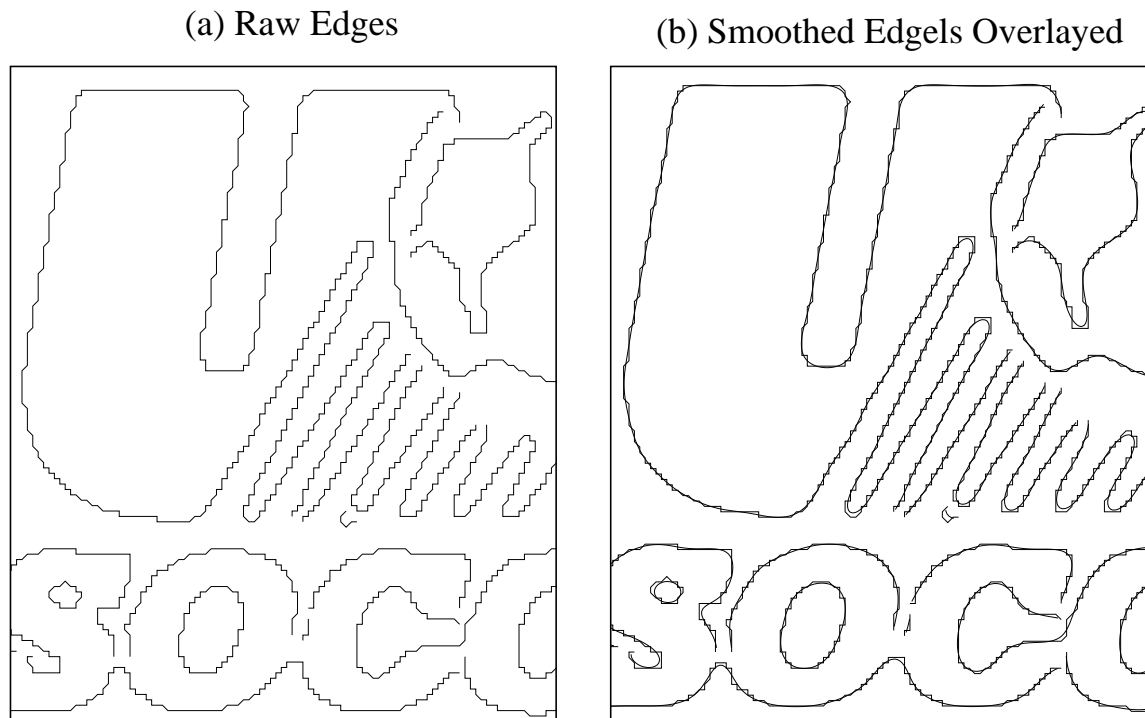


Figure 4.2: An example of (a) raw and (b) smoothed edgel chains (overlaid over the raw chains).

for each image.  $E_i = \cup_j e_{i,j}$  denotes the set of edgel chains for view  $i$ .  $e_{i,j} = \{\mathbf{U}_0, \dots, \mathbf{U}_n\}$  denotes the  $j$ th edgel chain of view  $i$ . Each edgel chain is an ordered list of 2D edgel points where  $\mathbf{U}_k$  denotes the (continuous) 2D coordinate of the  $k$ th edgel point in the chain.

After acquiring 2D edgel sets from various views of the object, these edgels must somehow be mapped into a common 3D coordinate system. Next, we describe our method for mapping and aligning the acquired 2D edgels in 3D model coordinates.

### 4.3 2D to 3D: View Alignment

The goal of this section is to map all the 2D edgels into a 3D coordinate system so that the data from all views can be merged to form the localization model. The problems (repeated from Section 4.1) that we face include:

- Distinguishing background from foreground edgels
- Converting 2D data to 3D data

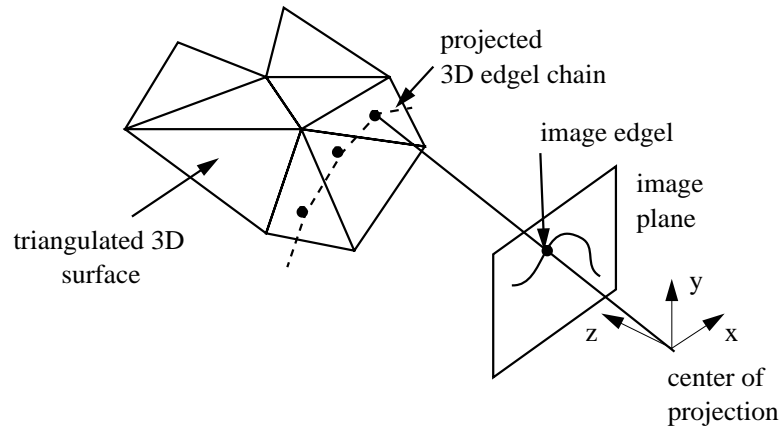


Figure 4.3: An example of mapping a 2D edgel onto a 3D triangulated surface.

- Aligning all the data into the same coordinate system

We can use the 3D modeling technique and calibrated-positioning system of Chapter 2 to virtually eliminate the above problems while enabling us to align all of the edgel data in the object's 3D coordinate system.

To begin, a triangulated surface model of the object is built using the techniques of Chapter 2. We assume that the pose of the object in each intensity image is known—for example, using the calibrated object positioner as described in Section 2.3. The position is denoted by the rigid transform  $\mathbf{R}_{0 \leftarrow i}$ , which represents the motion between view 0 and the view  $i$ .<sup>2</sup> We can use view 0 as the object coordinate system. All the 2D edgel data will be transformed to this central view.

Given the 3D surface model and its pose  $\mathbf{R}_{0 \leftarrow i}$  in the image with respect to the object's coordinate system, we can project the 2D edgels onto the 3D model—in a sense, texture mapping the edges onto the model's surface. This is accomplished much like ray tracing; for each edge point in the image, we follow the ray from the camera's center of projection, through the edgel in the image plane, and into the scene. We then determine which surface triangles intersect with the ray and select the closest of these triangles. Figure 4.3 shows an example of the edgel mapping process.

Though this operation is intuitively simple, in practice there are several changes of coordinate system involved in mapping the 2D edgel to the model's coordinate system. In particular, we have 3 coordinate systems to deal with: image, camera, and object. The image coordinate system is in terms of pixels while the others are defined as 3D Cartesian frames in metric units. The transformations between coordinate systems are:

$$\mathbf{x}_c = \mathbf{R}_{c \leftarrow m} \mathbf{x}_m \quad (4.1)$$

<sup>2</sup>As in Section 2.3,  $\mathbf{R}_{0 \leftarrow i}$  is a  $4 \times 4$  homogeneous transformation matrix.

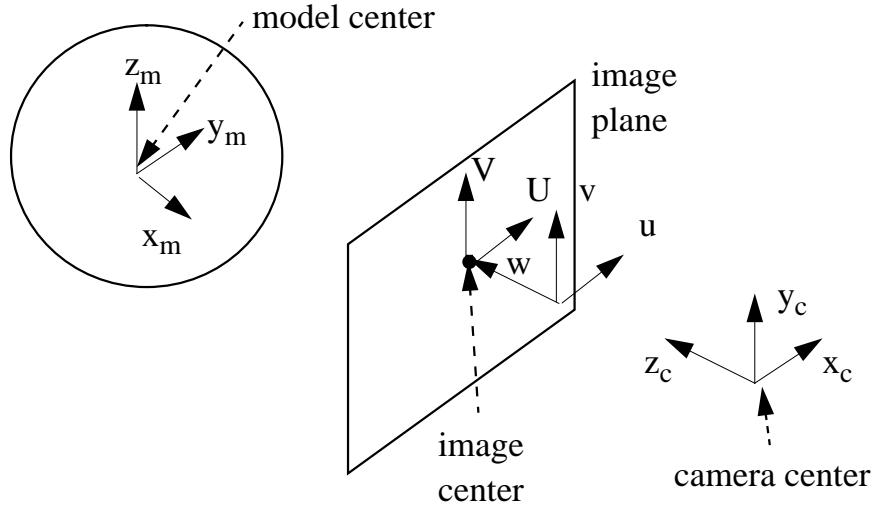


Figure 4.4: A diagram representing the relationships between the various coordinate frames used in the analysis in this chapter.

$$\mathbf{u} = \mathbf{R}_{u \leftarrow c} \mathbf{x}_c \quad (4.2)$$

$$\mathbf{U} = \begin{bmatrix} \frac{u}{w} \\ \frac{v}{w} \end{bmatrix} \quad (4.3)$$

where  $\mathbf{x}_m$  and  $\mathbf{x}_c$  are the positions in model and camera coordinates respectively,  $\mathbf{u} = [u \ v \ w]^T$  are the image projection coordinates, and  $\mathbf{U} = [U \ V]^T$  are the 2D image pixel coordinates.  $\mathbf{R}_{c \leftarrow m}$  is a rigid transformation that transforms model coordinates to an orthographic camera-centered frame. The camera-centered frame is defined such that  $[0 \ 0 \ 0]^T$  is the center of projection and  $[0 \ 0 \ 1]^T$  is the camera's focal axis.  $\mathbf{R}_{u \leftarrow c}$  transforms the Euclidean camera coordinates to (possibly non-Euclidean) 3D projection coordinates. This transform accounts for scaling factors (such as aspect ratio) and translation of the image center to non-zero image coordinates. The image projection coordinates are specified so that the final transformation is purely a perspective projection. Figure 4.4 shows the relationship between these coordinate frames.

The 2D edgel observations are defined in the image coordinate system, but to build 3D edgel models of an object, it is necessary to have these measurements in a 3D coordinate system, preferably the model coordinate system. There are several ways that one could go about mapping these edges. We begin by projecting the triangles of the 3D surface model into the image (much like z-buffering) using the equations presented above. Then, given the 2D edgel coordinates,  $\mathbf{U}$ , we can quickly determine which triangle intersects the ray corresponding to the edgel. First, we convert the edgel  $\mathbf{U}$  to a viewing direction  $\mathbf{v}$  in camera

coordinates by inverting the projection transform

$$\mathbf{v} = \mathbf{R}_{u \leftarrow c}^{-1} \begin{bmatrix} U \\ V \\ 1 \end{bmatrix}. \quad (4.4)$$

We then compute the precise intersection of ray  $\hat{\mathbf{v}}$  with triangle  $\tau_i$

$$\mathbf{x}_c = \frac{\mathbf{n}_i^c \cdot \mathbf{c}_i^c}{\mathbf{n}_i^c \cdot \hat{\mathbf{v}}} \hat{\mathbf{v}} \quad (4.5)$$

where  $\mathbf{n}_i^c$  and  $\mathbf{c}_i^c$  are the normal and center point, respectively, of triangle  $\tau_i$  in camera-centered coordinates. Once the intersection point is computed, we can then transform the point to the model coordinates by inverting the model-to-camera transform:

$$\mathbf{x}_m = \mathbf{R}_{c \leftarrow m}^{-1} \mathbf{x}_c.$$

We have now shown one way to map 2D edgels to 3D model coordinates. There are many other possible strategies to perform this mapping; however, none will be significantly more efficient than the one above. In any case, since model building will be done off-line, efficiency is not a real concern, thus, there is no need to discuss other possible methods.

As each 2D edgel in a chain is mapped to 3D model coordinates, the ordering of the edgels in the chain is maintained so that the 3D tangent direction for each edgel may be estimated. For example, the tangent vector,  $\hat{\mathbf{t}}_i$ , of the  $i$ th edgel in a chain is

$$\hat{\mathbf{t}}_i = \frac{\mathbf{x}_{i+1} - \mathbf{x}_i}{\|\mathbf{x}_{i+1} - \mathbf{x}_i\|}$$

where each  $\mathbf{x}_i$  is the mapped point corresponding to 2D edgel  $\mathbf{U}_i$ . The  $i$ th 3D edgel in a chain is then represented as the point and tangent vector pair  $\langle \mathbf{x}_i, \hat{\mathbf{t}}_i \rangle$ . This mapping solves the foreground/background problem. Only the edgels originating from the surface of the object are mapped to three dimensions; the other edgels may be ignored for purposes of modeling the object.

The basic approach that we have presented is not fool-proof however. It will often fail for edgels on or near occluding boundaries when the edgel rays do not intersect with our surface model. Figure 4.5 shows an example of this problem. Several factors contribute to the misalignment of occluding contour edgels with the surface model. Some degree of variability in 2D edgel location is due to edge detection, image discretization, and image noise. These errors will be propagated to the 3D mapping. Also, we can expect some errors to be introduced by the 3D mapping because of alignment/calibration error (in our calibrated image acquisition system) and 3D surface model error. Due to all of these factors an edgel corresponding to an occluding contour will randomly fall on either the object side or non-object side of the actual edge position. Thus, there will always be many instances of edgel rays that are generated by occluding contours of our object but do not intersect the object. We can rectify this problem by testing not only for intersection but also for near misses—edgel rays passing near triangles on the occluding boundary of the surface model.

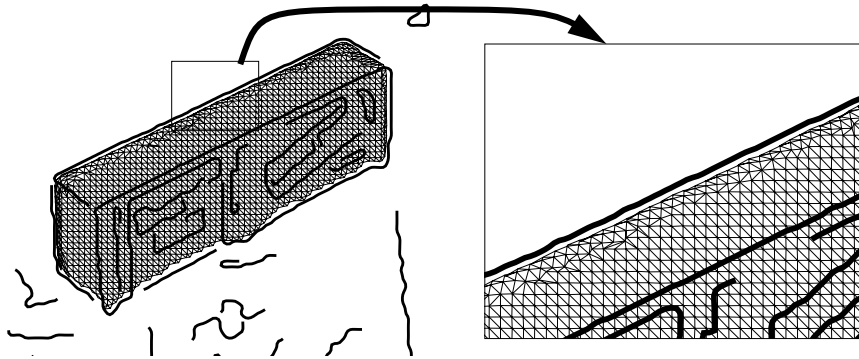


Figure 4.5: An occluding boundary edge which does not project onto the model surface.

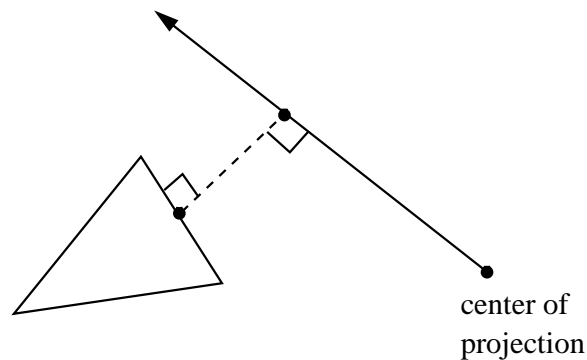


Figure 4.6: The shortest distance between a ray and the surface triangles when they do not intersect.

Occluding boundary triangles are easily detected when the model is projected into the image. These triangles will be adjacent to a triangle which is not visible (using the visibility tests of Section 3.2) from the given pose—thus creating an occluding boundary. The occluding boundary triangles can be checked separately for the near miss condition. We calculate the shortest distance between the edgel ray and any triangle on the model surface (see Figure 4.6). If this distance is within some threshold (for example, say 2 mm), then we can accept it as a potential 3D edgel in the model. The 3D coordinate of this edgel is the point on the edgel ray that is closest to the model (again see Figure 4.6).

This strategy seems like a reasonable solution. The result, however, is still unsatisfactory. The problem is that while the 2D projection of the surface model's occluding contour may appear smooth, the contour is not smooth in three dimensions. Figure 4.7 (b)

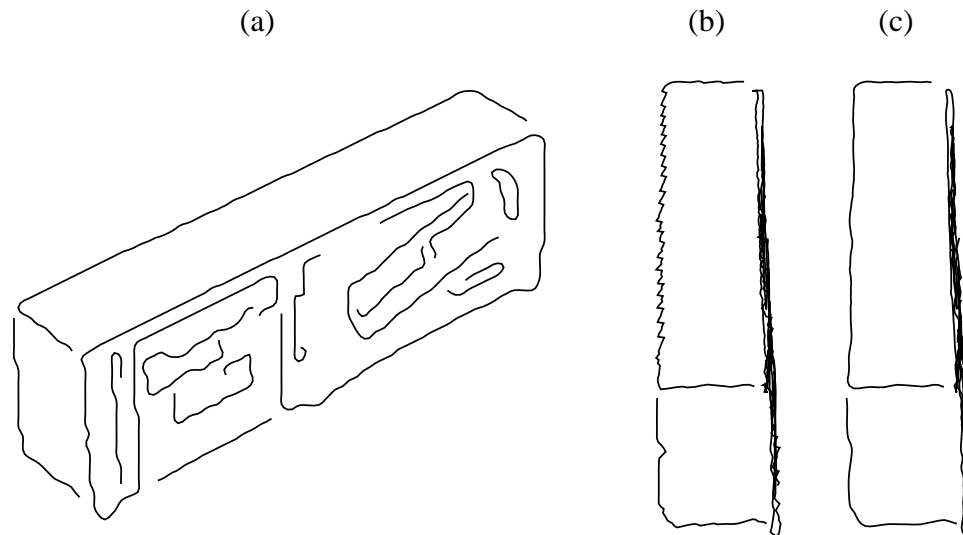


Figure 4.7: An occluding contour mapped to the object. (a) original 2D view, (b) another view of the mapped contour in three dimensions shows that it is not smooth in three dimensions, (c) the same contour after smoothing.

shows this phenomenon. The cause of the jagged 3D edgel is the triangulation along the occluding contour of the model surface. Much like the problem with jagged edges after edge detection in Section 4.2, we can use a smoothing operator to remove the kinks in the edge while retaining most of the original edge shape. In this case, we are smoothing a 3D edge chain. The smoothing operator of Wheeler and Ikeuchi [147] is as easily applied to 3D edgel chains as it is to 2D edgel chains. The smoothed result is suitable for our purposes. Figure 4.7 shows an example of such a 3D edgel before and after smoothing.

As one may imagine, the use of a threshold to attach edgels to occluding boundaries may be prone to errors. Often there will be edgels from the background that are near occluding boundaries of our object. Such mistakes are not critical as they are indistinguishable from spurious edgels that will invariably appear on the object. A successful algorithm for merging the 3D edgel data must account for the presence of extraneous and spurious edgels in the set of observations. The algorithm must minimize the possibility of including such edgels in the final model.

We have described how 2D edgels are acquired from a view of our object and how these edgels are mapped onto the surface of our object. In this section, we have shown how to align all of our edgel data into an object-centered coordinate system via an edgel-mapping process using a surface model. This process nicely eliminates two difficult problems: first, it directly solves the foreground/background problem—a problem that is likely unsolvable automatically using any other method; second, the correspondence problem is now tractable as all of the edgels are transformed to the same coordinate system which allows a natural



and reliable way to match and compare edgels.

The next problem is how to convert a large set of edgel observations to a single model representation of these observations. Our solution, the consensus-edgel algorithm, is the subject of the next section.

## 4.4 View Merging: the Consensus-Edgel Algorithm

In this section, we will describe an algorithm for merging the 3D edgel sets acquired by using the methods of the previous two sections. Given multiple aligned edgel sets, the question is how to combine the edgels from all the views to produce a unified representation of the edgel generators of the object. The main problem here is distinguishing random and extraneous edgels from those which will be visible over a wide variety of scenes. We must also be prepared for a significant amount of uncertainty in the edgel positions due to:

- Edge detection and discretization
- Image noise
- View alignment/calibration error
- 3D surface model error

This problem bears much similarity to the merging problem for 3D surface model building described in Section 2.4.

One difference between the surface and edgel case is that the topological problem is much simpler with edgels. In fact, topology is not necessary at all as we only need to build a model of edgels (points and tangent directions). Modeling the connectivity of the edgels may be useful for other tasks but is not necessary for the current localization task. We do not have the equivalent of the implicit surface representation to simplify the data merging process, nor do we need such a device. The main reason for using the implicit surface representation in Chapter 2 was to obviate a difficult problem of topology. If edgel connectivity is desired, this information would be easily obtainable as the topology of edgel chains is quite simple.

As in the 3D surface-merging problem, our edgel-merging algorithm must compensate for noisy/spurious edgel observations. Again, in the presence of such observations, there is little or no basis to trust that a single observation should be part of the object model. Our goal is to build models of edgel generators which are stable over many views of the object. We wish to take advantage of multiple observations to get a better estimate of the true source of the edgels with respect to the object's coordinate system. Simultaneously, we wish to exclude spurious and random edgels from the final model. Again, we are drawn to the concept of *consensus* as the basis for merging the observed data.

The basis of the merging algorithm is to find consensus sets of similar edgels from various views and compute an average of these edgels to add to the model. As in the surface-merging problem of Chapter 2, the concepts of consensus and similarity are relevant for

edgel merging. For the edgel-merging case, similar edgels are defined to be observations of the same edgel generator on the object. In the surface-merging case, similarity was defined in terms of surface location and normal direction. In the edgel-merging case, it is much the same; we define edgel similarity in terms of edgel location and tangent direction. Two edgels are grouped as similar if the distance between the two is below a threshold and the angle between their tangent directions is smaller than a threshold. We define the predicate

$$\text{SameEdgel}(\langle \mathbf{x}_0, \hat{\mathbf{t}}_0 \rangle, \langle \mathbf{x}_1, \hat{\mathbf{t}}_1 \rangle) = \begin{cases} \text{true} & (\| \mathbf{x}_0 - \mathbf{x}_1 \| \leq \delta_d) \wedge (|\hat{\mathbf{t}}_0 \cdot \hat{\mathbf{t}}_1| \geq \cos \theta_n) \\ \text{false} & \text{otherwise} \end{cases} \quad (4.6)$$

which determines whether two surface observations are sufficiently close in terms of location and tangent direction, where  $\delta_d$  is the maximum allowed distance and  $\theta_n$  is the maximum allowed difference in tangent directions. Much like the consensus-surface algorithm, consensus is defined as the minimum number of similar observed edgels required to instantiate an edgel generator in the model.

Our problem now is where to begin the search for consensus edgels. In the surface-merging case, we only needed to fill all the voxels in the volume grid; the order in which we proceed is irrelevant as each voxel is independent of the other voxels. In the current problem, all we have is an unorganized set of edgels with no obvious way to proceed.

Having no other information than the edge sets themselves and without the benefit of a representational aid like the volumetric representation for surfaces, it makes sense to begin the search for consensus edgels at one of the observed edgels. A good edgel to start with is one which is likely to be a consensus edgel (i.e., an observation of a stable edgel generator). One possibility is to begin with large edgel chains first. It is unlikely that noise or random external events will result in the detection of very long edgel chains. Also, long edgel chains are often indicative of significant geometric or photometric features of the object which are often detectable over many views.

Our strategy will be to take each 3D edgel chain—longest first—and check each edgel of the chain to see if it belongs to a consensus edgel. We perform a local search of the edgels from all other views to find the nearest edgel in each view. These edgels are tested for similarity to the current edgel and if a consensus is found, we add the averaged, consensus edgel to our final model.

Figure 4.8 shows an example of the local search for a case where we have four views. The edgels found to be in the consensus can be eliminated from further consideration. Thus, we are able to identify all consensus edgels from our set of 3D edgel observations.

As in the consensus-surface search, the local search for nearest edgels can be implemented efficiently using k-d trees [42]. A k-d tree of 3D edgels is built for each different view of the object. A nearest-neighbor search of the respective k-d trees can be computed in  $O(\log n)$  expected time where  $n$  is the number of edgels in the given tree.

The consensus-edgel algorithm can now be specified more precisely. In the algorithm below  $e_{i,j}$  denotes the  $j$ th edgel chain of view  $i$ . Again, an edgel chain comprises a list of 3D edgels—3D points and tangent direction pairs  $\langle \mathbf{x}_i, \mathbf{t}_i \rangle$ .

**Algorithm** *BuildConsensusEdgelModel*

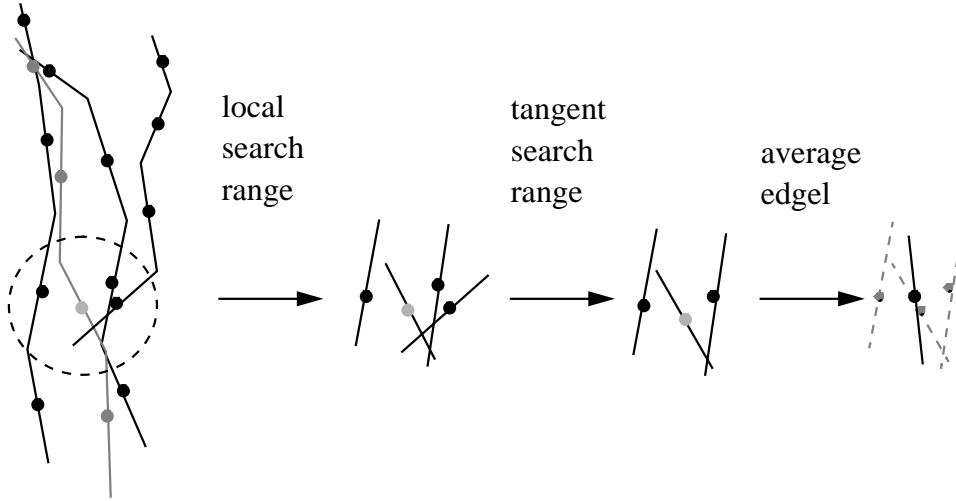


Figure 4.8: The stages of a consensus-edgel search for the grey edgel (center left). Eliminate candidates first using proximity, then the tangent direction constraint, then average the remaining edgels to form a consensus edgel.

**Input:** 3D edgel sets  $E_i, i = 1, \dots, N$

**Output:** consensus edgel model  $E_f$

1.  $E_f \leftarrow \emptyset$
2. **for each**  $E_i$
3.     **do** *Build K-D Tree*( $E_i$ )
4.     **for each** edgel chain  $e_{i,j} \in \cup_i E_i$ , largest  $|e_{i,j}|$  first
5.         **do for all**  $\langle \mathbf{x}, \hat{\mathbf{t}} \rangle \in e_{i,j}$
6.             **do**  $\langle \mathbf{x}_c, \hat{\mathbf{t}}_c, count \rangle \leftarrow \text{ConsensusEdgel}(\langle \mathbf{x}, \hat{\mathbf{t}} \rangle, E_k)$
7.             **if**  $count \geq \theta_{quorum}$
8.                 **then**  $E_f \leftarrow E_f \cup \langle \mathbf{x}_c, \hat{\mathbf{t}}_c \rangle$
9. **return**  $E_f$

Algorithm *BuildConsensusEdgelModel* makes use of an important subroutine: *ConsensusEdgel*. Algorithm *ConsensusEdgel* performs the local search around the neighborhood of the given edgel to estimate the average of the edgels that are determined to correspond to the same edgel generator on the object. Algorithm *BuildConsensusEdgelModel* uses a threshold  $\theta_{quorum}$  to determine if the support for a given edgel is sufficient to add it to the final edgel model.

**Algorithm** *ConsensusEdgel*

**Input:** edgel  $\langle \mathbf{x}, \hat{\mathbf{t}} \rangle$  of chain  $e_j$  in edgel set  $E_i$

**Input:** 3D edgel sets  $E_k, k = 1, \dots, N$

**Output:** average consensus edgel and count:  $\langle \mathbf{x}_{avg}, \hat{\mathbf{t}}_{avg}, count \rangle$

1.  $\mathbf{x}_{avg} \leftarrow \mathbf{x}$
2.  $\hat{\mathbf{t}}_{avg} \leftarrow \hat{\mathbf{t}}$
3.  $count \leftarrow 1$
4. **for** each edgel set  $E_j \neq E_i$
5.     **do**  $\langle \mathbf{x}_c, \hat{\mathbf{t}}_c \rangle \leftarrow \text{NearestEdgel}(E_j, \mathbf{x})$
6.     **if**  $\text{SameEdgel}(\langle \mathbf{x}, \hat{\mathbf{t}} \rangle, \langle \mathbf{x}_c, \hat{\mathbf{t}}_c \rangle)$
7.         **then**  $\mathbf{x}_{avg} \leftarrow \mathbf{x}_{avg} + \mathbf{x}_c$
8.          $\hat{\mathbf{t}}_{avg} \leftarrow \hat{\mathbf{t}}_{avg} + \hat{\mathbf{t}}_c$
9.          $count \leftarrow count + 1$
10.  $\mathbf{x}_{avg} \leftarrow \frac{1}{n} \mathbf{x}_{avg}$
11.  $\hat{\mathbf{t}}_{avg} \leftarrow \frac{\hat{\mathbf{t}}_{avg}}{\|\hat{\mathbf{t}}_{avg}\|}$
12. **return**  $\langle \mathbf{x}_{avg}, \hat{\mathbf{t}}_{avg}, count \rangle$

Algorithm *NearestEdgel* returns the closest edgel (in terms of position) from a set of edgels to the given edgel. Three thresholds are required and are almost identical to those required for the consensus surface algorithm (see Section 2.4).  $\delta_d$  and  $\theta_n$  are required for the predicate *SameEdgel*, which is defined in Equation 4.6, to determine how close the edgel points and tangent directions must be to consider them as similar. These can be estimated based on the variations expected from the observed data.  $\theta_{quorum}$  is the consensus threshold which determines the required number of similar edgels for consensus. This threshold must be chosen while considering the number of total views and the number of views from which the edgel sources may be visible.  $\theta_{quorum}$  will determine how stable the edgels must be for inclusion in the model.

The result of Algorithm *BuildConsensusEdgelModel* will be the set of consensus edgels over the edgel sets from all views. The above description omits one bookkeeping detail for clarity. We must keep track of which edgels have been used to add a consensus edgel to the final model and eliminate these edgels from further consideration.

We now have a method to merge rigidly attached object edgels (i.e., edgel sources whose position is fixed on the object). This method overlooks a large source of edgels, namely occluding contours. Occluding contours are quite different from rigid edgels and thus require slightly different representation and treatment for localization. It is unlikely that the same point on the contour generator would be observable from a sufficient number of views for consensus to be applicable. Since the appearance of occluding contours is based purely on local geometry of the generator's surface, we can utilize our 3D surface model to predict the appearance of the surface's occluding contours. This is the topic of the next section.

## 4.5 Modeling Occluding Contours for 3D-2D Localization

Occluding contours are the boundaries observed at points on a smooth surface where there is a transition from visible surface points to occluded surface points from a particular

viewing direction. Occluding contours are often detected as edges in images since the object surface and background are often of distinctly different shades, colors or intensities. Thus, occluding contours are prominent features for object recognition and localization. For smooth convex surfaces, every point on the surface can generate an occluding contour from many viewing directions. For example, each point on a sphere generates an occluding contour in the image for all viewing directions  $\hat{v}$  such that

$$\hat{v} \cdot \hat{n} = 0$$

where  $\hat{n}$  is the surface normal at the given point. This set of  $\hat{v}$ 's spans the tangent plane of the surface at that point.

For localization, we need to be able to predict the appearance of occluding contours from a given viewing direction. Appearance prediction needs to be an efficient operation as well. Another consideration is that our representation must be derived from our 3D surface model.

One possible solution would be to represent the surface of the object using an algebraic polynomial, and then analytically solve for occluding contours. This is the approach taken by Ponce and Kriegman [77]. The problem is that these algebraic equations can become very large even for simple shapes. Another complication for algebraic representations of occluding contours is how to build models of algebraic surfaces. In practice, objects will have to be modelled using a collection of algebraic surfaces. Inferring the algebraic representation from a triangulated surface is an extremely difficult problem—fundamentally the same as range-image segmentation [5], for which no completely satisfactory solution exists. Even if segmentation were solved, the boundaries between surfaces greatly complicates the solution of the algebraic surfaces.

Another possibility, and the approach that we use in this thesis, is to use a piecewise planar approximation of the surface. It is a reasonable option for several reasons:

- Our surface is already approximated by a piecewise planar model (triangular patches).
- The approximation error of the contour projection can be made arbitrarily small (but is limited by the original surface-model resolution).
- We can use a point-based representation consistent with our rigid edgel representation and 3D surface representation.

The piecewise-planar approximation idea is made clear by considering the piecewise-linear approximation of a 2D circle (cross section of a sphere). Figure 4.9 (a) shows a circle and a set of viewing directions (equivalent to tangent directions) at the corresponding contour generators. As the viewing direction changes, the contour generator traverses the circle. If the circle is represented by a set of connected line segments (see Figure 4.9 (b)), the contour generator still changes as the viewing direction changes, but now it changes in discrete intervals corresponding to the endpoints of the line segments.

For the localization task, the most important aspect of the approximation is the accuracy of the projected contour. The worst-case error is the distance between the piecewise-linear

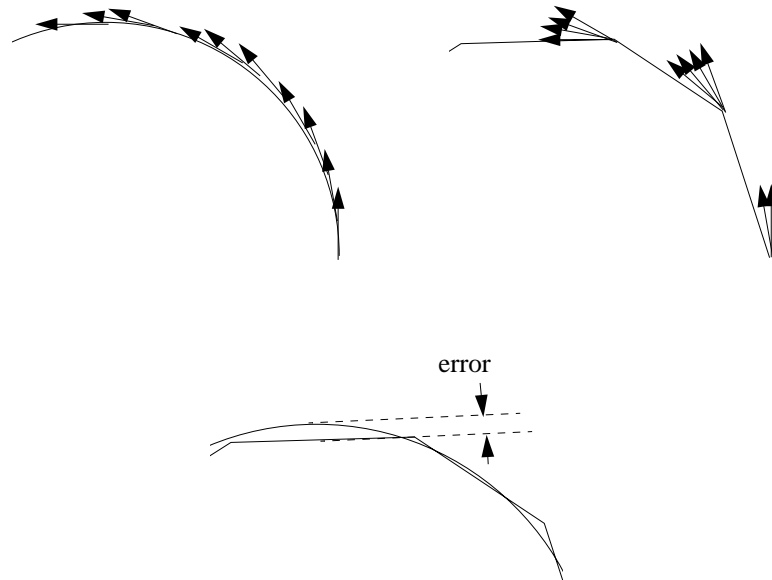


Figure 4.9: A circle and its occluding contour points (a) continuous circle, (b) piecewise-linear circle, (c) the error of approximation

circle and the true circle and is maximum at the midpoint of each approximating line segment (see Figure 4.9 (c)). We can represent a curve's occluding contours to a desired accuracy using piecewise-linear approximation. This naturally extends to 3D using piecewise-planar models (such as our triangulated surface models).

Applying z-buffering [40, 141] to our triangulated surface model is one possibility for computing the occluding-contour edges. Basically, the occluding-contour points can be identified by finding points which lie on edges which border a visible triangle on one side and an invisible triangle on the other side. However, as noted in Section 3.2, z-buffering is too expensive for our localization application. Using the same idea with our surface-point visibility approximation produces very poor results since the mistakes of the approximation will always imply occluding-contours where none should exist.

Instead we concentrate on defining a local computation over points on the object surface to determine whether they generate a contour or not from a given viewing direction. Our triangulated surface model consists of vertices which are connected to form triangles that cover the surface. Each vertex or surface point is a potential contour generator. We are interested in finding a quick local computation to determine if the vertex is on the occluding contour generated by the surface model from a certain viewing direction. The curvature at a point provides the information necessary to predict the appearance of contour points.

### 4.5.1 Curvature

Curvature is, technically, a property of a curve, not a surface [102]. The curvature of a surface usually refers to the curvature of a specific curve that lies on the surface. For a length-parameterized curve in three dimensions,  $\beta(t) = [x(t) \ y(t) \ z(t)]^T$ , the curvature is the magnitude of the change in the tangent vector,  $\beta'(t)$ . Curvature is also described as the magnitude of the bending of the curve where the direction of the bending is  $\beta''(t)$ . Thus, the curvature of  $\beta(t)$  is defined as

$$\kappa(t) = \|\beta''(t)\| = \sqrt{x''(t)^2 + y''(t)^2 + z''(t)^2}$$

For a circle of radius  $r$ ,  $\kappa = \frac{1}{r}$ . A rough characterization of curvature is the inverse of the local radius of the curve.

For curves, the Frenet frame [102] defines a coordinate system at each point. The coordinate system is defined such that the normal  $\hat{n} = \frac{\beta''}{\|\beta''\|}$ . Thus, with the normal direction already specified, the magnitude of bending is sufficient to describe the curvature at a point on a curve.

Getting back to smooth surfaces, we are interested in whether a surface point generates a contour from a given viewing direction. The first requirement is that the viewing direction  $\hat{v}$  is perpendicular to the surface normal  $\hat{n}$  at the point (i.e.,  $\hat{n} \cdot \hat{v} = 0$ ). Given  $\hat{v}$  and  $\hat{n}$ , we can extract a particular curve from the surface—the intersection of the surface with the plane spanned by  $\hat{v}$  and  $\hat{n}$ . This curve is called a *normal section* [102] since the surface normal lies in the plane of the curve. For example, the normal section of a point on a sphere of radius  $r$  is a circle of radius  $r$  (see Figure 4.10). The curvature of the normal section at the given point is appropriately called the *normal curvature*.

The curvature of a normal section is not simply a magnitude (as in the 2D case) but also indicates the direction of bending by its sign. Surfaces have a unique normal  $\hat{n}$  at each point while a normal section at a point may bend in either the  $\hat{n}$  or  $-\hat{n}$  directions (see Figure 4.10). Thus, the sign of curvature of a 3D point differentiates between normal-section curves that bend away from the surface normal and those that bend in the direction of the surface normal.

To generate an occluding contour for a surface with an outward pointing normal, we require that the normal section has negative normal curvature (see Figure 4.10 (b)). Negative normal curvature implies that the point is a transition from visible to invisible points—thus generating the contour.

For a 3D surface point, the curvature depends on the specific normal section. Fortunately, Euler's formula gives us a simple characterization of the curvatures of all normal sections given an arbitrary viewing direction  $\hat{v}$  such that  $\hat{n} \cdot \hat{v} = 0$ . Euler's formula is

$$\kappa(\hat{v}) = \kappa_1(\hat{e}_1 \cdot \hat{v})^2 + \kappa_2(\hat{e}_2 \cdot \hat{v})^2 \quad (4.7)$$

where  $\kappa_1$  and  $\kappa_2$  are the maximum and minimum curvatures, respectively, and are referred to collectively as the principal curvatures.  $\hat{e}_1$  and  $\hat{e}_2$  are the tangent directions corresponding to the maximum and minimum curvatures, respectively, and are referred to as the principal

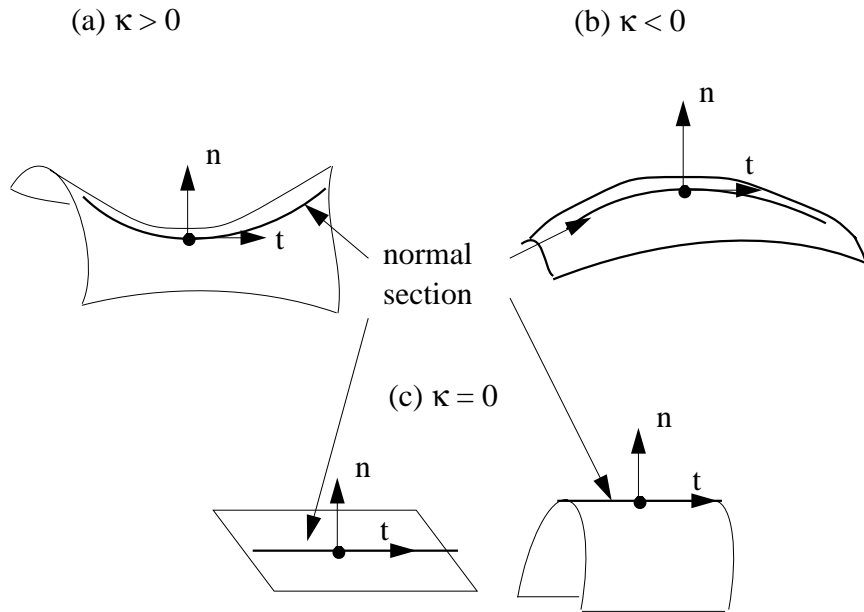


Figure 4.10: Example of normal sections with (a) positive normal curvature and (b) negative normal curvature.

directions. The principal directions are orthogonal to each other and form the basis of the tangent plane of the point.  $\kappa_1$ ,  $\kappa_2$ ,  $\hat{e}_1$ , and  $\hat{e}_2$  completely characterize the curvatures of normal sections at a point. Along any normal section, the curvature of a point can be classified into three categories: positive, negative and zero. Only surface points with negative curvature (convex normal sections) generate occluding contour edges in a 2D projection of the 3D object. However, the curvature at a surface point depends on which normal section is taken. Since it is possible for a point to have  $\kappa_1 > 0$  and  $\kappa_2 < 0$ , some viewing directions in the point's tangent plane would generate occluding contours ( $\kappa(\hat{v}) < 0$ ) while others would not ( $\kappa(\hat{v}) \geq 0$ ).

### Estimating Curvature from a Triangulated Surface Model

Using curvature information, we can determine which points on our 3D surface model are potential contour generators. Determining the curvature of the surface points of the model is tricky. The problem is that we have a triangulated (sampled) model of a surface: much shape information has been lost. The surface model is also not likely to be perfect—the surface will invariably be contaminated by noise. Since curvature is a second-order differential property, it is very computationally sensitive to noise and the effects of discrete sampling (e.g., aliasing).

Several solutions for computing the surface curvature from discrete data have been



proposed in the literature. Besl and Jain [5] presented a formulation for computing curvature from range images. Their formulation relies on the regular grid sampling of the depth (as is available directly from range images) so is not directly applicable to our problem. Chen and Schmitt [18] proposed a solution for computing curvature at points of a triangulated model. Their method involves fitting circles to triples of points to approximate normal sections and then use a least squares fit to Equation 4.7 to solve for the principal curvatures and directions. Unfortunately, the best fit circle for three haphazardly selected points on a surface can give wildly inaccurate estimates of normal curvature. In addition, there is a singularity for collinear triples which makes fitting circles to these points unreliable.

A more stable and natural method is the one proposed by Koenderink [76]. His idea is to fit a quadric surface in the neighborhood of each point and then use the algebraic representation of the surface to derive the principal curvatures and directions. This is a natural approach for estimating principal curvatures since, fundamentally, the principal curvatures and directions define a locally quadratic approximation of the surface shape. A variation of Koenderink's approach was used by Shi et al. [125] to measure surface curvature in tomographic medical data.

We also use a variation of Koenderink's approach to compute the principal curvatures and directions for each vertex on our triangulated surface model. While fitting a quadric surface seems simple enough, there are some subtleties that may interest the reader. The description of the quadric fitting and curvature estimation is presented in Appendix C. The main point is that we can easily compute the principal curvatures,  $\kappa_1$  and  $\kappa_2$ , and the principal directions,  $\hat{e}_1$  and  $\hat{e}_2$ , from the local quadric fit.

Given  $\kappa_1$  and  $\kappa_2$ , we can quickly classify each point into one of three categories:

1.  $(\kappa_1 \geq 0) \wedge (\kappa_2 \geq 0)$ : This point is not a contour generator, and we can eliminate it from consideration for our purposes.
2.  $(\kappa_1 \geq 0) \wedge (\kappa_2 < 0)$ : This point is either a cylindrical point (i.e.,  $\kappa_1 = 0$ ) or a saddle point and generates contours from some tangent viewing directions—those with negative normal curvature.
3.  $(\kappa_1 < 0) \wedge (\kappa_2 < 0)$ : This point is an elliptic point and generates contours for all tangent viewing directions.

Figure 4.11 shows examples of each of these three surface classes and the set of viewing directions from which the point generates an occluding contour (i.e.,  $\kappa(\hat{v}) < 0$ ).

Thus, we can classify all points on our model surface as belonging to one of the above three classes. We use this information to predict the visibility of contour edges in images of the object. The details of the visibility computation will be described in the next chapter on 3D-2D localization.

One point worth noting is the effect of corners and very high curvature points. These points are fundamentally different to occluding contours but also share some similarities. They are given a separate class: convex edgels.

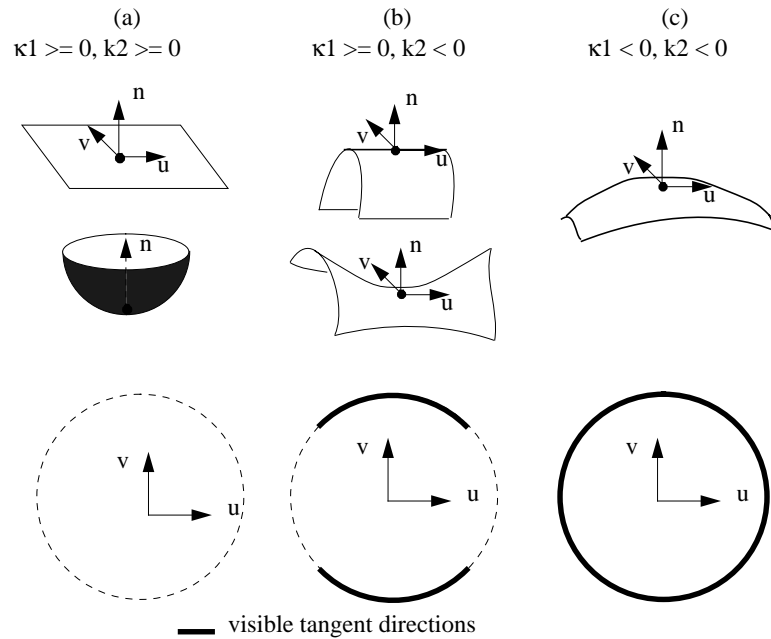


Figure 4.11: Classes of surfaces based on curvature. The third row shows a tangent plane for each class on which the dark shaded regions indicate viewing directions from which a contour is visible at the point in question.

### Convex Edgels

High-curvature points (e.g., corners) are a hybrid of the rigid surface and occluding-contour edgels. From some viewing directions, high curvature (corner) points generate occluding contours. In this sense, the convex edgel is like an occluding-contour edgel. From other directions, they may generate intensity discontinuities because of the surface normal discontinuity across the edgel. In this sense, the convex edgel is like a rigid surface edgel.

For the curvature analysis presented in this section, we need not distinguish high curvature points since our method for acquiring rigid edgel models (described in Section 2.4) will usually detect this type of edgel as rigid. However, as we will see in the next chapter, the visibility constraints for convex edgels demands treatment separate from rigid surface edgels and occluding-contour edgels.

## 4.6 Object Modeling for 3D-2D Localization: Results

Here we present some experimental results of our implementation of the 3D edgel modeling algorithm described in this chapter.

As discussed above, we make use of a 3D surface model of the object constructed using the method described in Chapter 2. Thus, we are bound by the same constraints of the modeling system described there: the objects must be small enough to be imaged by the range finder and to be mountable on the Puma. We assume, as before, that the objects are rigid and opaque.

For our 3D edgel modeling experiments, we selected 7 objects to model using our system. We use the toy boxcar, rubber duck, ceramic mug, and toy car which were used in the experiments of Chapter 2. We also model 3 other simpler (planar) objects: a stop sign, a sign with a T on it (T-sign), and a bulls-eye like target.

For each object, we first built the 3D surface model following the methods of Chapter 2. We then acquired intensity image views of the objects, again using our calibrated image acquisition system. We manually determined the number of intensity image views for each object to 1) maximally cover the viewable surface of the object, and 2) provide a sufficient amount of overlap between views for the consensus-edgel algorithm to extract rigid surface edgels. The number of views required is related to the geometric complexity of the object: varying from 21 for the three planar objects to 54 for the toy car.

The views were acquired by varying the joint angles  $\theta_y$  and  $\theta_x$  of the robot's end effector as in Section 2.5 (see Figure 2.12 and Section 2.5 for a review of these details). Generally, we would vary  $\theta_y$  from -180 degrees to 160 in increments of 20 degrees and would vary  $\theta_x$  from anywhere from -30 degrees to +30 degrees in 20 degree increments as well.

In addition to varying the pose of the object, we also varied the illumination in order to reduce the dependence of our models on any single light source condition. We used three different illumination configurations for this purpose. The illumination configuration was switched after every image in the sequence.

The acquired intensity images contained  $256 \times 240$  pixels. Each image was processed as described in Section 4.2 to produce a set of smoothed 2D edgel chains. The 2D edgel chains were then projected into 3D object coordinates using the calibrated position and 3D surface model of the object. The edgels were then resampled in 3D coordinates at a resolution of 1 mm. The resulting 3D edgel sets were then provided to the consensus-edgel algorithm to extract the significant 3D edgel generators from the observed data.

The results of the model acquisition for the test objects are shown in Figures 4.12- 4.18. Each of these figures show:

- an intensity image of the object
- an example from the set of observed intensity edge images
- the complete set of 3D edgels used as input to the consensus-surface algorithm
- a view of the 3D rigid edgels extracted from the data
- three views of the full edgel model, including 3D rigid edgels and occluding contour edgels extracted from the 3D surface model (the views are displayed using hidden edgel removal for clarity)

Object	Images	Edgels In	Edgels Out	Time (seconds)	$\theta_{quorum}$	$\delta_d$ (mm)	$\theta_n$ (degrees)
stop	21	22k	914	10	11	1.5	25
T-sign	21	26k	970	14	8	1.5	25
target	21	18k	602	9	11	2.5	25
boxcar	33	39k	1537	48	7	3	45
mug	36	39k	1510	240	8	2	25
car	54	40k	1387	134	6	1	36
duck	49	21k	185	46	12	2	45

Table 4.1: Statistics of the edgel modeling experiments for each object.

The relevant statistics of the modeling experiments for each object are presented in Table 4.1. These statistics include the number of input images, the number of edgels input to the consensus-edgel algorithm, the number of edgels in the resulting model, the execution time on an SGI Indy 5 (a 124 MIPS/49.0 MFLOPS machine), and the parameters for our consensus-edgel algorithm (the quorum limit  $\theta_{quorum}$ , the maximum distance,  $\delta_d$ , between similar points and the maximum angle,  $\theta_n$  between tangent vectors of similar edgels).

The results demonstrate that we can create fairly accurate models of the edgel generators of an object given many sample views of the object and a 3D surface model of the object. In particular, the consensus-edgel algorithm is able to extract the significant rigid edgels from a large set of rather noisy edgel observations.

## 4.7 Object Modeling for 3D-2D Localization: Summary

In this chapter, we have described our methods for automatically constructing a model for use in localizing a 3D object in 2D intensity images—3D-2D localization. For this localization task, we have chosen to match model edgels with image edgels using a 3D model representation rather than a view-based (2D) approach.

In order to build a 3D model of the edgel generators of the object, we make use of the 3D surface modeling and calibrated view alignment described in Chapter 2. Knowing the pose of the object and its shape allows us to convert 2D observations into the object’s 3D coordinate system, while eliminating background edgels from consideration. We presented the consensus edgel algorithm which extracts the stable rigid edgel generators from the sample views of the object. To account for occluding-contour edgels, we proposed a method for computing and analyzing the curvature at all points on our 3D surface model. Knowing the curvature of a point, we can classify it as a contour generator or not. The principal curvatures and directions of a point makes it possible to predict the viewing directions from which a contour generator will generate a contour.

We have postponed discussion of the visibility computation for our edgel model. At best, from our sampled data, we can collect a set of viewing directions from which each model edgel was detected. Visibility conditions of an edgel cannot be reliably or accurately

(a)



(b)

