User-Guided Inverse 3D Modeling

James Andrews

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-103
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-103.html

May 17, 2013

User-Guided Inverse 3D Modeling

by

James L. Andrews

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Carlo Séquin, Chair
Professor Jonathan Shewchuk
Professor Sara McMains

Spring 2013

User-Guided Inverse 3D Modeling

Abstract

User-Guided Inverse 3D Modeling

by

James L. Andrews

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Carlo Séquin, Chair

This thesis introduces and explores the idea of "user-guided inverse 3D modeling" – an interactive approach to shape construction and redesign that extracts well-structured, parameterized, procedural descriptions from unstructured, hierarchically flat input data, such as point clouds, boundary representation meshes, or even multiple pictorial views of a given inspirational prototype. This approach combines traditional "forward" 3D modeling tools with a system of user-guided extraction modules and optimization routines. With a few cursor strokes users can express their preferences regarding the type of modeling primitives to be used in a particular area of the given prototype to be approximated, and they can also select the degree of parameterization associated with each modeling routine. The results are then pliable, structured descriptions that are well suited to implement the particular design modifications intended by the user.

We present research on the components that make up an inverse 3D modeling system. Our research focuses have been (1) fitting of modeling primitives to data; (2) fitting higher-level structure (CSG expressions, symmetry, and hierarchy) to data; (3) minimal user interactions that guide these fitting processes; and finally (4) shape editing using the fitted structure. We also explore what is necessary to ultimately arrive at a clean, exportable result. A key research question guiding the design of our system has been how bringing the user into the loop affects the problem: i.e., where can the user's inputs simplify, accelerate, or otherwise improve the fitting process? What simple inputs can the user provide to *unambiguously* extract a desired primitive fit? And how can we let the user immediately begin using their fitted primitives to edit the shape? Answering these questions leads us to many new research findings, including: (1) We identify simple, lightweight inputs that allow the system to extract primitives with a great deal of user control, and we present detailed analysis of how these inputs can be ambiguous and what additional inputs may be needed to exactly capture the user's desired primitive fit. (2) We show multiple improvements to the state of the art in fast, effective fitting methods for sweeps and quadric surfaces. These improvements include identifying and fixing a major problem with previously-standard methods for kinematic surface fitting that led to grossly wrong results for data with small noise, and a new insight

into the nature of direct, algebraic fitting that leads us to more accurate type-specific quadric fitting. (3) We show how the user can guide a boundary-to-CSG reverse-engineering process: Specifically, we identify cases where the best result of such a process depends on the user's intent, and we show how simple user interactions can let users specify their intents. (4) In addition to the diverse problems faced by individual structure-fitting and shape editing modules, the combination of these modules also presents new research opportunities; we identify new interactions between these diverse modules, including transformations between primitive types and hierarchies that can be imposed across modules.

To time enough at last.

Contents

Contents **ii**

List of Figures **iv**

List of Tables **xii**

1 Introduction **1**
1.1 Related Work 4
1.2 System Overview 6

2 "Stationary" Swept Surface Primitives **8**
2.1 Background 9
2.2 Overview of Kinematic Field Fitting 10
2.3 Failure Cases of Previous Methods 13
2.4 Improved Fitting Methods 15
2.5 Robustness to Outliers 19
2.6 New Velocity Fields 20
2.7 User-Guided Segmentation 21

3 "Progressive" Swept Surface Primitives **29**
3.1 Background 29
3.2 User-Guided Segmentation and Fitting 31
3.3 Editing with Progressive Sweeps 35

4 Quadric surface primitives **40**
4.1 Background 41
4.2 Fitting method for hyperboloids, ellipsoids, and paraboloids 46
4.3 Fitting methods for lower-dimensional quadric types 52
4.4 User-Guided Segmentation and Fitting 56
4.5 Implementation Details 58

5 Smooth Surface Primitives **60**
5.1 Background 61

5.2 Implementation 64

6 General User-Guided Segmentation **67**
6.1 Overview of Methods 67
6.2 Easy Mesh Cutting 69

7 Fitting CSG Structure **72**
7.1 Automatic Boundary-to-CSG Conversion 74
7.2 Interactive Refinements of the CSG Structure 82
7.3 Handling Inexact Representations 85
7.4 Implementation 87
7.5 Limitations and Future Work 88

8 Higher Level Structure and Interaction Between Primitives **89**
8.1 Symmetries 89
8.2 Interactions Between Fitting Primitives 90

9 Additional Input Sources **94**
9.1 Related Work 95
9.2 Technical Approach 96
9.3 Results and Discussion 111
9.4 Limitations and Future Work 112

10 Cleaning Results for Export **115**
10.1 Cleaning General Polygon Mesh Surfaces 115
10.2 Cleaning the High Level Structure and Segmentation 118
10.3 Guaranteeing a Solid Model 119

11 Summary **125**

Bibliography **127**

List of Figures

1.1 Inspirational examples of redesigns using the manually-programmed Sculpture Generator I. . . . 2
1.2 Diverse redesigns of an input shape . . . 3
1.3 Diverse redesigns of another input shape . . . 4
1.4 Our pipeline for Inverse 3D Modeling. . . . 6

2.1 An example of a shape redesign enabled by our sweep-based modules. . . . 8
2.2 A simple example of kinematic surface fitting. . . . 10
2.3 Visualization of two kinematic motions of a cylinder. . . . 10
2.4 Example of a failure case for the rotation constraint. . . . 13
2.5 Example of a failure case for the unit constraint at small scales. . . . 14
2.6 Example of a failure case for the unit constraint at large scales. . . . 15
2.7 A spiral velocity field fit to a number of different selections using Taubin's constraint. 17
2.8 A spiral field (Eqn. 2.3) is fit to a cone using the HEIV method with varying values for $w_{\mathbf{p}}$. . . . 18
2.9 A spiral field (Eqn. 2.3) is fit to a cone with Gaussian noise applied to the base ($\sigma = 1\%$ bounding box size) using the HEIV method (converged in 3 iterations) and the Taubin method. . . . 19
2.10 Fitting an example with outliers using RANSAC. . . . 20
2.11 The general linear field (illustrated with red streamlines) is fit to a number of selections (in blue) on various objects. . . . 20
2.12 Optimization steps of the iterative segmentation and fitting algorithm for stationary sweeps. . . . 23
2.13 The iterations of our user-guided algorithm for selecting and fitting stationary sweeps. . . . 24
2.14 A cylinder fit with and without user constraints. . . . 26
2.15 An interesting example of shape editing using stationary sweeps. . . . 27

3.1 Fitting a progressive sweep with a knotted path. . . . 30
3.2 Overview of the progressive sweep-fitting initialization procedure. . . . 33
3.3 The user can quickly extract any number of diverse sweeps on an armadillo. . . . 34
3.4 Different initial strokes result in different sweep fits. . . . 35

3.5 Local and global edits to a sweep cross section. 36
3.6 Key elements of the user-interface. 37
3.7 A detail-preserving modification of the Armadillo surface. 38
3.8 From a stroke (in yellow) on the dragon mesh, we automatically extract the main hump of the dragon. 39

4.1 The difference between an ellipsoid and a hyperboloid can be very small in terms of local surface behavior, leading to ambiguity in the best-fitting quadric type. . 41
4.2 A chart of quadric types, not including rotationally symmetric subtypes. 44
4.3 Comparison of ellipsoid- and hyperboloid-specific fitting results for our method and the method of Allaire et al. 47
4.4 A sampling of the quadrics in a linear subspace formed by the best two eigenvectors of Taubin's fitting method. 48
4.5 Fitting errors at samples of a linear subspace of quadrics formed from the best two eigenvalues of Taubin's method. 49
4.6 Noisy data from a hyperbolic paraboloid fit by general quadric fitting and by paraboloid-specific fitting. 52
4.7 Red streamlines showing motion fields of (a) Eqn. 4.10, (b) Eqn. 4.11, and (c) Eqn. 4.12, fit to mesh data. 54
4.8 General and cone-specific fitting of a noisy cone. 55
4.9 Strokes (in yellow) are used to initialize various quadric fits on a variety of shapes. 56
4.10 Type-specific fitting can help disambiguate a user's selection. 57
4.11 Comparison of fits resulting from measuring error at mesh vertices vs. integrating error across the surfaces of a mesh 58

5.1 (a) The elephant's ear region is selected; (b) when optimized as a thin-plate spline, all surface details get smoothed away. (c) The ear is fixed in place as a constraint, and the foot marked by the blue patch is moved; continuity between the blue and yellow patches and the rest of the shape (gray) is preserved by Laplacian-preserving surface editing. 60
5.2 Non-linear formulations for smooth surfaces are very difficult to make fast and stable. We show simple cases where a state-of-the-art non-linear smooth surface method, FiberMesh [96], fails to converge in a number of simple configurations. In this figure, the thick blue or red curves are constraint curves: The FiberMesh surface interpolates these curves, and users edit the surface by adding or moving these curves. The top row shows an example where horizontally dragging a constraint curve drawn on an ellipsoid causes the surface to diverge to infinity. The bottom row shows configurations in which the FiberMesh system produces an oscillating surface that never converges. 62
5.3 Effect of the order of the Laplacian on the resulting smooth shape. 65

5.4 (a) A botijo model is approximated as a surface of revolution; (b) the corresponding profile; (c) edited profile, and (d) the resulting shape. The handles, modeled as detail-preserving smooth surfaces, move with the changes in the body. . . . 65

6.1 Segmentation of a shape where the desired primitive is not a close match to the data. . . . 70
6.2 Primitive-specific vs. "easy cut" segmentation of a botijo model. . . . 71

7.1 A selection of shapes easily defined by CSG operations. . . . 72
7.2 Simple examples of shape editing operations enabled by boundary-to-CSG conversion . . . 73
7.3 Example of the "cells" that can define a given shape (cyan). Here we have two circles and a box as our primitives, used to describe the cyan shape in the center of the diagram. Any region of the diagram with a different in/out classification with respect to any of the primitives, as well as the input shape, is numbered as a distinct "cell." Note that individual cells need not be contiguous, and the combination of cells is a disjunctive covering of the entire space (cell 0 is an unbounded cell covering everything not in any primitive). . . . 75
7.4 An example of boundary-to-CSG conversion where separators are needed. Several possible separators are shown. . . . 76
7.5 An example where separators can't easily help. . . . 76
7.6 A 3D example of partial CSG decomposition . . . 77
7.7 A shape composed of a circle, ellipse and some additional unlabeled geometry is shown (a). This decomposes into two representable parts (the circle and ellipse) and a non-representable part (red). The user edits the shape by moving the ellipse up and the circle down (b). We show two possible results of this edit (c,d). If we do not subtract non-editable copies of the circle and ellipse from the non-representable part, then corresponding areas of the non-representable part will remain as those primitives are moved (c). We find this growth non-intuitive, and prefer the result with non-editable primitive copies (d). . . . 77
7.8 The ray casting approach to finding all fundamental product cells for a given set of primitives: A ray (red) is cast through the primitives to identify cells. The start of the ray is identified as the first cell, and the yellow arrows show where the ray crosses a primitive boundary, potentially creating a new cell. . . . 78
7.9 (a) The cyan circle and box are reconstructed with a circle primitive and four linear halfspace primitives (arrows indicate the side of the linear halfspaces that is "inside"). (b) When the user moves the circle down, the initial CSG expression of Sec. 7.1.2 generates the solids shown on right. Note that the bottom of the circle is missing because no cells in the original model included an intersection of the bottom linear half space and the circle. . . . 79

7.10 A cyan shape with two boxes and a circular hole (a). If we reconstruct the shape and move the hole down in our system, the result without optimization extends the hole into the bottom box (b), while the result with optimization stops the hole at the second box, since the hole has been removed from the expression defining the second box (c). 82
7.11 (a) An arrow is represented as a CSG decomposition of an infinite cone, an infinite cylinder, and two planar half spaces. (b) The CSG decomposition arising from optimization leaves the cylinder at the base and the cone at the top as separate primitives. The user can click (click 1) the portion of the cylinder outside the cone, and then the cone (click 2), to add an intersection with the cone to the cylinder and arrive at shape (c). 83
7.12 (a) A boundary representation of a die, with a plane fit to one side (blue) and spheres fit to the corresponding dots (red, yellow, green, cyan). A "region of interest" separator bounding box, bounding all colored faces, is also added (shown in black wireframe). (b) A partial CSG representation, with the non-representable portion left in red, and the representable portion in green. Without the region of interest separator, the whole shape would be non-representable. (c) The user edits the green part by moving and scaling one of the spheres. 84
7.13 The target shape (cyan) is fit with an ellipse primitive, but the ellipse doesn't match the target shape exactly. This causes our analysis to find cells 1 and 3, which will make the entire shape non-representable (a). We propose two ways to eliminate these cells. The first option is to make the ellipse and the target shape match exactly (b). The second option is to compute the maximum deviation between the ellipse and the corresponding part of the target shape, and reject cells that are within that distance of both the matched primitive and the target shape. We show this in (c) with green and red curves following the ellipse and target shape respectively, with thickness equal to the maximum deviation: Cells 1 and 3 are discarded because they are completely under both the green and the red curves. 85
7.14 (a) A primitive (red dotted line) has been modified to match a target shape (cyan). (b) If the shape and primitive do not match exactly, or if the CSG operation is not performed exactly, then subtracting the primitive from the shape will leave some artifacts near the boundary of the primitive, whereever the target primitive is found to be slightly outside the subtracting primitive. (c) If we simply push the surface of the target shape to be slightly inside the subtracting primitive, we can eliminate these artifacts. 87

8.1 Given a symmetric object, our system can edit the symmetry, for example by changing the 3-fold rotational symmetry in (a) to a 4-fold rotational symmetry in (b). We do this by scaling a 120° slice of the original model into a 90° wedge and instancing it four times. 90

8.2 The sculpture model (a) is fit with multiple primitives (b): a progressive sweep on top, a cone in the lower half, and a plane at the bottom. The sweep path itself can be fit onto a sphere (c). This spherical structure can be retained along with any specified symmetry (here D_2) while editing the sweep path (d). 91
8.3 An example of progressive editing: The base (green) of a sculpture (reconstructed from a visual hull) is first approximated by a quadric surface shown in red (a). Projecting the base mesh to the quadric surface gives a cleaner result (b). The mesh is then converted to a surface of revolution and modified by editing the profile curve (c, d). Finally, the base is converted to a progressive sweep (e), and its sweep path (magenta) is edited to further deform the pedestal (f). 91

9.1 (a) An example of an organic shape without an obvious closed network of feature curves. (b, c) Examples of more sculptural shapes that also lack obvious closed networks of feature curves, (B) having internal structure and material properties that could stymie automatic methods relying on photo consistency. 95
9.2 Overview of how our modeling primitives are combined. Starting from the silhouette curves defining a visual hull, we add in parametric primitives and feature curves, and finally combine these primitives into a unified model, using a series of optimization steps. 97
9.3 The standard visual hull of the Klein bottle (a) does not capture internal structure and has jagged edges. Allowing the user to draw silhouettes of negative space lets us capture internal structure (b), and a mesh smoothing optimization that takes silhouettes into account can smooth out unwanted sharp features from the hull (c). 98
9.4 A 2D example of a visual hull (in orange) obtained by intersecting the (pink) object silhouette cones from three cameras. We approximate the distance to the hull surface as a maximum of signed distances to the cone edges (with negative distances inside each cone). For point p_1 (inside hull) this gives the true distance to the hull, in solid green. For point p_2 (outside hull) the true distance is marked with a blue arrow, and the approximate distance is shown as a solid red line. Camera c_3 is shown with an additional cone (with dotted black outline) indicating the region that is visible in the image taken from that camera. We only precompute distances in the region of the input image, so since p_2 is outside this cone, we approximate the distance from p_2 to the c_3 cone by first projecting to the dotted cone (blue dotted line) and then projecting from there to the silhoutte cone (red dotted line). 100

9.5 An illustration of our simple primitive placement system. (a) The user clicks a point (red) where they would like to place a simple primitive. (b) The user chooses a cone primitive, and it is placed such that its centroid is directly under the user's point, in the part of the visual hull furthest from the surface. Its scale is proportional to the distance from that centroid point to the hull surface. (c) The user adjusts the cone orientation manually to roughly align with the desired cone. (d) An optimization deforms and translates the cone to conform to the visual hull. 102
9.6 Visualization of the curve fitting algorithm: Given a curve on an image (yellow), the curve is sampled and rays are shot through the image samples from the camera (green). These rays are intersected with the visual hull (blue), and candidate points inside the hull are allocated for each ray (spaced evenly – at a distance of a hundredth of the length of the diagonal bounding box of the visual hull, for scale). We then use dynamic programming to find a shortest path through these candidate points in the order of the rays, with some bias towards the center of the hull. 103
9.7 From a user-drawn compound Bézier curve in one view (a), a curve optimization taking into account 8 views and their silhouettes produces an initial 3D sweep curve. After the user has indicated a cross section scale, a tubular sweep is produced (b). Finally an optimization of the sweep spine control points and of the sweep cross section yields a final fitted sweep shape (c). 105
9.8 The single-view spine curve fitting process can also be used to fit tube sweeps to the elephant (a,b), quickly defining a smooth approximate shape with the correct topology, which can then be optimized to fit the visual hull (d,e) using the smooth surface optimization of Sec. 9.2.4. In contrast, starting from the noisy geometry of the visual hull itself (c) will be much more difficult for the smooth surface optimization (f) – the mesh surface optimization cannot discard some large ghost geometry, and could not ever correct a topological error. 106
9.9 Visualization of the hull-primitive combination algorithm, from left to right: (a) We start with an overlapping hull (red) and parametric primitive (green), and then (b) subtract from the original hull the "hull" of the primitive, grown by some fixed radius (pink). We then grow any remaining visual hull geometry (c), and add back in the intersection of the grown hull and the original (d). 107
9.10 A small deviation from a hull view (a) reveals large "ghost" geometry in the sweep half of the model, but the cone at the bottom looks reasonable (b). Here we combine a fitted sweep primitive for the top with the hull on the bottom, while still keeping the two parts connected at the join point (c). 108

9.11 Different methods for finding the curve-surface correspondence, illustrated on a simple 2D example: The curve is drawn in red, the surface in green, and the correspondences in black and green arrows. The surface after optimization is shown in dotted blue. (a) The naive approach of simply using the closest point. (b) The result of biasing the distance measure in a suitable direction. (c) The result of adding intermediate curve points, thus avoiding the generation of folding artifacts. 110
9.12 A set of 3D curves (in green) capture details of the elephant ear (a) constructed from lines drawn by the user in 3 of the 5 initial images. When these 3D curves are used in conjunction with the thin-plate spline reconstruction, the ear details appear on the elephant mesh (c). Without using these curves we would get the smooth, earless shape in (b). 110
9.13 Objects that we may attempt to handle with our modeling system. 112

10.1 Examples of surfaces that may be defined in a CAD file with open boundaries, self-intersections, and/or non-orientable components: (a) An open, symmetrical form of Girl's Cap [53]. (b) A Klein bottle. Although these surfaces can also be defined as solids, it is often easier to represent them as thin sheets, especially if the user intends to continue editing the shape. 116
10.2 Examples of different artifacts in the high-level structure that a user may want to clean before export: (a) Segments like the one shown in blue may have small gaps where the surface deviated from the fitted primitive. (b) The cylinder's segment (blue) includes a small region that would ideally only belong to the sphere's segment (green), demonstrating how neighboring segments may not initially have a clean boundary at the ideal transition point between the two shapes. 119
10.3 The Utah teapot (a) is a standard example of a surface with self-intersections and holes, as can be seen in a cross-section view (b). In cleaning up shapes like this, we aim to remove intersections and close gaps (c). 120
10.4 We attempt to resolve intersections in a complicated self-intersecting triangle mesh (16k triangles) (a). Robust BSP-based code [18] resolves most major self-intersections, but produces a result with an order of magnitude more triangles (142k triangles), many near-degenerate sliver triangles and still has 2k pairs of self-intersecting triangles due to floating point error in the output vertex positions (b). The excessive additional triangles are co-planar, so could largely be removed by mesh simplification, but the self-intersections may be more difficult to correct. 121

10.5 There is a gap between the top and body of the Utah teapot (a) that should be easily filled by connecting the boundary edges of the teapot top and body. However, a per-element inside/outside segmentation of some tetrahedralizations may be *incapable* of connecting these boundary edges, because the necessary edges or facets simply aren't present in that tetrahedralization. We illustrate this problem in 2D with a triangulation (b): In this example, we can at best either cover some of the teapot's top, by marking the green and blue triangles as "inside" (c) or connect the teapot top to the inside of the teapot body, by marking the green and blue triangles as "outside" (d). The equivalent problem occurs in some 3D constrained Delaunay tetrahedralizations of the Utah teapot: We show the result of covering some of the teapot top (e) or connecting the teapot top to the inside of the body (f) for one such example. 122
10.6 When two or more intersecting shapes (a) are repaired by the mesh cleanup algorithm, we typically want the region of the intersection to be solid (b), and not hollow (c). Parity-based mesh repair strategies that try to ensure that the shape changes from solid to hollow across facets of the input mesh tend to create hollow intersections. 123
10.7 (a) A 2D slice of a good inside/outside segmentation of the Utah teapot, with inside colored blue. (b) A 2D slice of Polymender's segmentation using a parity-based criteria for inside/outside segmentation [66], with inside colored blue. A 3D view of the region in the red box is shown in (c) – note the small tunnels connecting the spout to the body. 123

List of Tables

2.1 Timings for stationary sweep module 28

3.1 Timings for progressive sweep module 38

9.1 Timings for each optimization 112

Acknowledgments

Many thanks to my advisor, Carlo Séquin, for his patient help in developing these ideas with me, and for his boundless enthusiasm and support. The fascinating sculptures that fill Carlo's office, designed by Carlo and his colleagues in the art-math world, served as a constant source of both challenge and inspiration for much of this work.

Thanks to Sara McMains and Jonathan Shewchuk for their careful reading of earlier drafts of this thesis, and their many helpful questions and corrections.

Thanks to my collaborators during my internships at Adobe. Pushkar Joshi and Nathan Carr helped me think intuitively about smooth surfaces, and supported my ideas even when reviewers gave me a tough time. Hailin Jin gave me the opportunity to learn most everything I know about photo-based modeling, and his direction and advice was a driving force behind Chapter 10 of this thesis.

Thanks to the friends who I bothered with weird questions about my research, and who gave me hugely helpful advice in return – especially Sridhar Ramesh, Rahul Narain, Florian Hecht, and James Cook.

Thanks to the National Science Foundation (NSF award #CMMI-1029662 (EDI)) and Adobe Systems for funding much of this work.

Thanks to the sources that provided the example shapes I used to test my ideas – particularly the Image-based 3D Models Archive, Tlcom Paris, and the Stanford Computer Graphics Laboratory.

Thanks to my friends in the VCL and beyond, who make Soda Hall a great place to work and who have filled so many of my days with good stories and conversations.

Thanks to everyone who gave me great suggestions while I wrote my thesis. Charles Boebinger suggested the dedication. Rahul suggested my conclusions chapter should be a 100-page flipbook-style animation of a mic drop.

Thanks to my parents, for their endless love and support.

Thanks to Stephanie. ;)

And finally, thanks to all the helpful people who I appear to have forgotten here. Each error of grammar, spelling or fact in this thesis is a carefully placed clue to a hidden puzzle that, when solved, will reveal your names.

Chapter 1

Introduction

Many engineering tasks, redesign efforts, or artistic creations start from an existing prototype shape, which may exist only as a physical artifact, as a virtual shape in some computer file, or as a set of images or sketches. Unfortunately, because of insufficient record keeping or incompatibilities between different design systems, high-level design information that could readily be adapted to new requirements is often not available when a redesign or refinement becomes necessary. In other cases, previous high-level design information may not provide the best structure for making a desired change: for example, a vase modeled with a subdivision surface might be edited more easily by re-interpreting it as a surface of revolution with an editable profile curve. Likewise, when a model is reconstructed from point cloud data [81], photographs [46], or sketches [50], the representation used for the reconstruction may not be the most convenient for the desired re-design. For effective design optimization, it is desirable to capture the prototype shape as a well-structured, parameterized, procedural geometry description, which can then be fine-tuned to meet the current specifications and design requirements.

In this thesis, we present "user-guided, inverse 3D modeling," an interactive approach to extracting such a pliable geometry description from unstructured, hierarchically flat input data. Our approach consists of an integrated system of geometrical extraction and fitting routines, which are applied to user-designated portions of the given input shape, combined with a set of traditional "forward" 3D modeling tools that allow the extracted geometry to be edited and fine-tuned. We primarily work with boundary mesh representations, but discuss extensions to point cloud data and multiple pictorial views in Chapter 9.

While our inverse 3D modeling approach makes it possible to re-construct a proper geometrical model from some unstructured, incomplete, or ambiguous input data, its primary purpose is to create a well-structured, parameterized model that can easily be modified in some specific way. For instance, we may wish to make the handles of a vase (Fig. 2.13) loopier, or change the number of handles, or change the vase profile. We may want to adjust the pose of an organic creature, treating the limbs as editable sweeps (Fig. 3.7). We may want to arch the wings of a bird (Fig. 9.13b) or selectively thin a swept sculpture in order to recreate an originally plastic or wooden sculpture (Fig. 3.5) in bronze, at a large scale. While

the system could readily be used for a wide range of shapes from creatures to crankshafts, in this thesis we are especially interested in handling challenging, free-form shapes, which have been less extensively covered in the literature. We assume that the designer trying to extract a good geometrical description from a prototype shape already knows what the intended design modifications are, and thus can make sure that the appropriate degrees of freedom and edit-handles are incorporated in the extracted model description. We involve the user (redesigner) right from the beginning of the extraction process, and let them begin editing each primitive interactively as soon as it is extracted.

User-guided inverse 3D modeling can be seen as a generalization of an intuitive approach that has been used by Carlo Séquin since the mid-1990s in the development of several special-purpose computer programs to create models of abstract geometrical sculptures. One of the first examples started from a manually-generated wood sculpture by artist Brent Collins (Fig. 1.1a). The key geometric element, a sequence of biped saddles and holes, was captured in a parameterized program, called Sculpture Generator I [123]. This program allows the user to change the number of saddles and holes in the chain, the degree of the saddles, as well as details of their geometry such as the thickness and extension of the flanges. For a particular setting of the various parameters, the program can closely reproduce the original inspirational shape shown in Figure 1.1a; but it can also be used to generate a multitude of new sculptures that all seem to belong to the same "family" (Fig. 1.1b, 1.1c). Note that these redesigned shapes are topologically different and could not be created using shape warping alone.

The goal of our current effort is to generalize this approach to the reverse engineering of a much wider domain of shapes, ranging from engineering parts to consumer products and

a b c

Figure 1.1: Inspirational examples of redesigns using the manually-programmed Sculpture Generator I. (a) Collins' original Hyperbolic Hexagon; (b) the generated Hyperbolic Hexagon II, an enhanced version with 3rd-order saddle surfaces; (c) Heptoroid, with an added 7th "story" and some twist, resulting in a single-sided Moebius geometry. Shapes (b) and (c) were designed on Séquin's Sculpture Generator I and realized in wood by Brent Collins (b) from blueprints produced by Sculpture Generator I [123].

to the organic shapes of plants and creatures. We have constructed prototypes of a suite of fitting methods, specifically designed to work together for the application of user-guided inverse 3D modeling. To make these tools truly useful for the intended purpose, we believe they should be:

- Easy to direct by the user with one or two menu clicks or cursor strokes.
- Fast enough to permit re-fitting with only seconds of interaction and computation time.
- Flexible enough to fit primitives with substantial non-matching surface details.
- Controllable with respect to the extent to which they cover a part of the input shape.
- Integrated with instant editing facilities that permit making high-level shape changes.

In addition to presenting the overall vision behind our integrated framework, this thesis presents results (and some lessons learned) obtained with extraction modules realized during the last few years, in particular: two sweep modules that can produce generalized and highly parameterized rotational or translatory sweep geometries; modules that extract quadric primitives of user-specifed type and smooth surface patches; and modules that extract higher-level symmetry structure and Boolean CSG expressions of previously extracted primitives. Although most of our modules have focused on mesh-based data, we have also explored an image-based approach that starts with the visual hull constructed from a few 2D depictions of the given prototype. By supporting a diverse array of modules, and interactions between these modules, we enable great creative flexibility that would be difficult to achieve with any single tool – a single shape can be changed in diverse, surprising ways as we illustrate in Figs. 1.2 and 1.3.

The primary focus of our research is on identifying how to best bring the user into the loop of the primitive- and structure-fitting process, and how doing so can improve fitting and

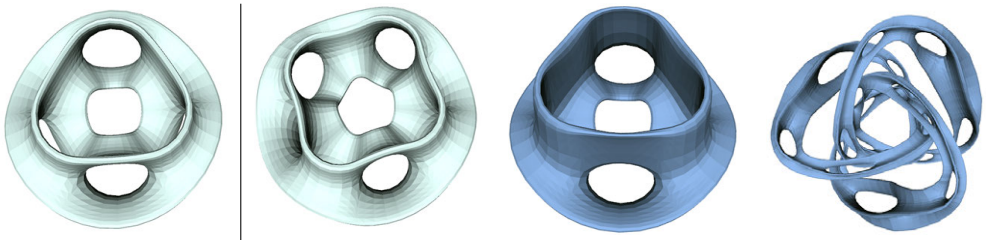


Figure 1.2: Starting from the input shape, shown at left, we use our inverse 3D modeling system to create diverse output shapes. (a) Changing 3-fold to 4-fold symmetry. (b) Using the stationary sweep module to change a flange. (c) Using the progressive sweep module (as well as symmetry, and conversion between primitives types) to create an interesting shape twisted along a complex path.

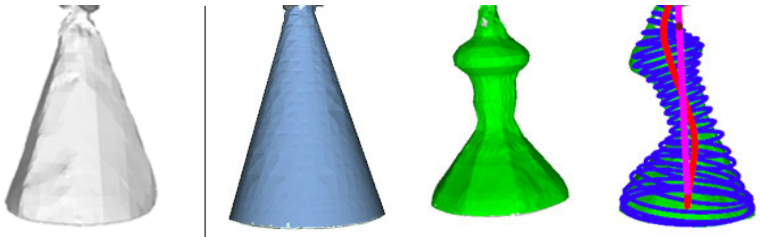


Figure 1.3: Starting from the input shape, shown at left, we use our inverse 3D modeling system to create diverse output shapes. (a) Using quadric fitting to cleanup and simplify a noisy shape; (b) Using a rotational sweep to extract and edit the profile; (c) Using the progressive sweep module to bend the shape.

editing results. We examine how the user can disambiguate primitive- and structure-fitting processes, improving the speed at which results are obtained as well as the quality of the results. For our primitive fitting modules, we identify minimal user inputs that can initialize a fit, letting us fit a primitive in seconds. We then examine ambiguities that can arise in the fitting and segmentation process – where the user may reasonably prefer a result that is not the numerical "best" by any metric, or where there are many results that have similar quality – and we introduce new, simple inputs to allow the user to ensure the system arrives at the desired parameterized structure. Similar ambiguities arise in many other modules, such as the Boundary-to-CSG module (Chapter 7) and the mesh-cleanup module (Chapter 10), and in each case we research new ways to let the user quickly resolve these ambiguities. In addition to these findings, we also present more fundamental improvements to the state of the art in primitive fitting – for example, in the case of simple sweep fitting, we identify significant problems with previous common fitting methods, which led to very poor results on data with small noise, and show how to fix these problems.

1.1 Related Work

The idea of fitting modeling primitives and other high-level structures to a given shape in order to facilitate editing and redesign has been pursued by many other researchers. Reverse engineering for CAD applications [7, 81, 143, 142] has traditionally focused on taking potentially noisy, unstructured data (for example from a 3D scan), and automatically segmenting the shape into a set of clean, non-overlapping primitive shapes that fully capture the desired structure and which can be imported into a CAD program. In these systems, user interaction is minimized (ideally, eliminated), and the goal is an error-minimizing fit. Work on primitive-based shape segmentation has covered similar conceptual ground [48, 149]. Some work has focused on reverse-engineering to a more easily edited primitive, for example a kind of sweep-morph primitive [112], but still with the general goal of finding a single best fit. We use many of the techniques developed for these systems, but instead of focusing on finding

a single, ideally-fitting set of primitives, we focus on building a flexible system that may abstract some detail in favor of a more structured, high-level representation that can easily re-interpret a shape on the fly as the user's ideas and redesign goals may change. Where a traditional reverse-engineering goal is often to minimize user-interaction, ideally resulting in a fully automatic system, our goal is to have a fully-controllable, interactive system.

Some ground-breaking previous systems have introduced user-guided abstraction of a shape for redesign purposes. One such system is the method of Krishnamurthy and Levoy [75] in which a B-spline patch network is fit to an input shape in the form of a dense triangle mesh, using optimization routines to perfect the fit; the B-spline control points then allow artists to edit the reconstructed mesh more easily. Another stepping stone for our development is a sweep-specific shape editing system that lets a user manually fit a set of sweeps to a model and then use the parameters of the sweep representation to edit the underlying shape [151]. Several sketch-based modeling methods have also allowed the user to interactively specify primitives fitting a target shape, for modeling purposes: for example, by placing generalized cylinders, ellipsoids, and symmetry constraints over a sketched object [50]. Our focus is on integrating many such modules in a unified framework, where each module provides the few control parameters that are most useful for the redesign intensions of the user.

In addition to fitting modeling primitives, recent work has also demonstrated progress in automatically finding higher-level structure in a given input shape, for instance, finding the symmetries and alignments between parts [79, 100]. This permits inverse procedural modeling, in which a generative model describing an existing shape is discovered [11]. Interactively modifying the underlying symmetry of a given shape also played a key part in Sculpture Generator I [123], and we therefore include some symmetry-based editing tools in our framework.

Much recent work on variational surface editing has focused on factoring a model into a smooth surface with low-frequency shape elements and a fine, detailed surface, which can then be moved, transferred or edited separately [22, 132, 135]. This concept is broadly useful, and all our modules are designed to permit some fine details to be handled separately from the fitted high-level primitive structure.

Sketch-based methods for shape editing have often focused on direct editing of feature curves such as silhouette or contour lines [156], or more general feature lines [69]. Another sketch-based editing system focuses on editing meshes by editing the shading [51]. Focusing on such direct features, rather than primitives, is an alternative approach that can work in tandem with our system. In most cases, a general, mesh-based, smooth surface representation is used as the underlying primitive representation to support such edits – this is a powerful, general primitive that we use as well.

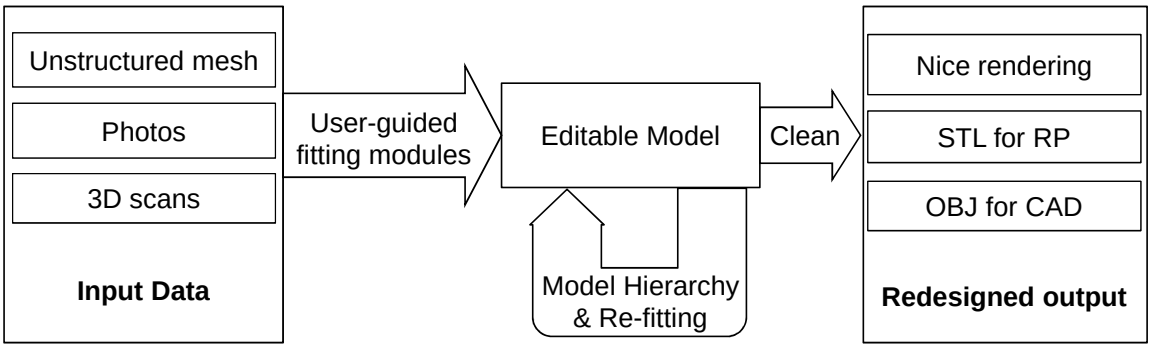


Figure 1.4: Our pipeline for Inverse 3D Modeling.

1.2 System Overview

Our approach to building a useful Inverse 3D Modeling system is to start by building a robust library of "user-guided fitting modules" for diverse 3D modeling primitives. These primitives will serve as a basis for a useful high-level reconstruction of a given artifact. We focus on the parametric primitives most often found in today's CAD tools, because these familiar, practical primitives for shape design form a baseline of what can be expected from a 3D modeling tool. In many cases, suitable fitting methods for these primitives already exist, but they may need to be adapted to suit our interactive context. We require our fitting methods to be fast and user-controllable. We prefer that the user spend a few seconds indicating the primitive they would like to extract and the degrees of parameterization they desire, rather than relying on a fully automated process "to do the right thing." In addition, we would like our extraction methods to be able to either overlook or preserve, if desired, intricate surface details that cannot be described conveniently with the higher-level primitive itself. The difference between the original data and the simplified extracted primitive is remembered as a decoration, which, if so desired, can be re-applied to a modified version of the extracted primitive. The primitives we aim to fit can describe a shape directly (such as quadric surfaces or swept surfaces), or can describe higher-level features like symmetries, or a Boolean CSG expression of other, previously-fitted shapes.

Once we have fit various structures to a surface, the user can manipulate these structures to redesign the shape as they desire. If the user's goals or editing needs change, they can re-fit new structures or transform existing structures into more general primitives – for example, a quadric primitive may be transformed into a sweep primitive. The user may also recursively fit structure to the model. For example, a quadric surface (e.g., an ellipsoid) may be fit to the points of a sweep path; subsequently, the system may impose this structure onto any modifications of the sweep path by reprojecting the sweep path onto the quadric surface when either the sweep path control points or the quadric parameters are edited.

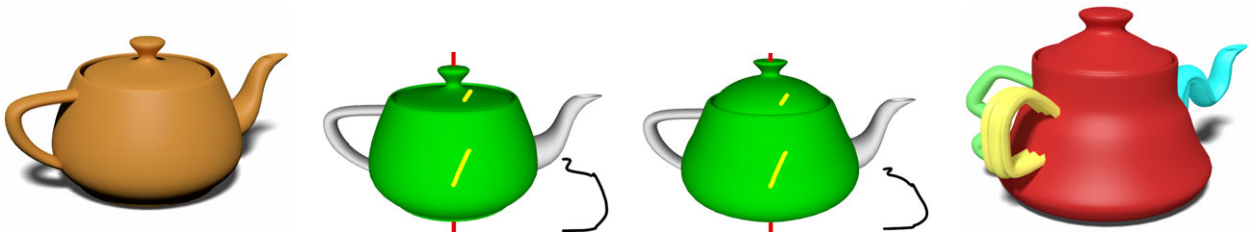
Finally, once a user is satisfied with the redesigned shape, the system helps the user to clean up the resulting model, ideally providing a nice, watertight shape ready to be used in various applications – rendering, rapid prototyping, simulation, or further editing in other

CAD tools. While this process can be partially automated, we again explore how lightweight user input can clear up ambiguities and improve results.

Chapter 2

"Stationary" Swept Surface Primitives

The first extraction modules that we focused on were based on generalized sweeps. Sweeps are very powerful procedural geometry generators; fully general sweeps can have so much parametric flexibility that sweep fitting can be made trivial: if the cross section is allowed to morph arbitrarily, then a given shape can be fit by any arbitrary sweep path. To ensure more meaningful fitting results, we define two useful, more-constrained sub-classes of general sweep: First, we define "stationary sweeps" as sweeps that are describable by a single profile curve swept along a sweep path that follows a constant rotation, translation or scaling motion – such as simple rotational or helical sweeps based on a fixed rotational axis in space. Second, we define "progressive sweeps" to handle generalized translatory sweeps, which we discuss in the next chapter. We show an example of the kind of shape edits these two modules will allow us in Fig. 2.1.



a b c d

Figure 2.1: An example of a shape redesign enabled by our sweep-based modules. (a) An input surface. (b) User draws two strokes (in yellow) and clicks the "fit stationary sweep" button to fit a stationary (rotational) sweep to the teapot body. (c) User edits the cross section to change the shape of the teapot body. (d) Final re-designed shape, including further edits to the body, as well as edits to the spout and handle using the progressive sweep module (Chapter 3). Two different modified versions of the handle were added for variety.

Stationary sweeps are a relatively restricted class of sweep – following sweep paths that can generally be characterized by only 6 or 7 global parameters – and because of that they can be extracted in an extremely efficient and unambiguous manner. We can find a stationary sweep fit for a mesh with a million triangles in about a quarter of a second on a typical laptop computer. Our extraction method is based on the "kinematic surface fitting" concepts pioneered by Pottmann and Randrup [108].

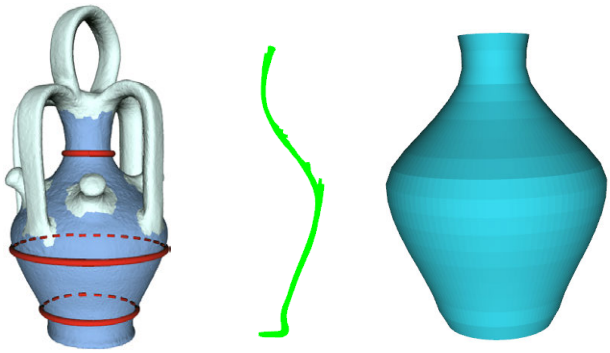
The kinematic surface fitting problem consists of several sub-problems: segmentation of a surface into subsets that can be fit by separate kinematic surfaces [48, 59]; fitting a kinematic motion to a given set of data points; and finally, finding a generator curve [115]. In this chapter, we show that previous methods for fitting general kinematic motion can fail on some simple, common cases such as a box or a cone with small noise, and then show how to fix these problems by applying more robust fitting methods that have previously not been applied to kinematic surface fitting. We then show how to apply this kinematic motion fitting algorithm in a simple, user-guided segmentation algorithm. Finally, we show how to use this fitted motion to edit a stationary sweep.

2.1 Background

Sweeps are often described by a cross-section (or profile) curve, and a sweep path: By moving the cross-section curve along the sweep path, one can generate the swept geometry. However, for stationary sweep surfaces, we replace the standard sweep path with a "velocity field" defined over all space, and we then *advect* the cross-section curve through the velocity field to generate the swept geometry. This allows us to consider our stationary sweeps as *kinematic surfaces*: Surfaces that are tangent everywhere to some easily parameterizeable, linear velocity field over space (such as Eqns. 2.1, 2.2, 2.3). Given such a field and a curve in space (which we refer to as the "generator curve"), advecting the generator curve by the velocity field will generate a kinematic surface (Fig. 2.2), which is a specialized class of swept surface; we refer to this class as "stationary sweeps."

Such tangent-everywhere velocity fields are called the "slippable motions" of a surface [48]. A point $\mathbf{p}$ with normal $\mathbf{n}$, is considered slippable with respect to a field $\mathbf{v}()$ if $\mathbf{v}(\mathbf{p})$ is orthogonal to $\mathbf{n}$, which can be tested by checking that $\mathbf{v}(\mathbf{p}) \cdot \mathbf{n} = 0$. The full basis of slippable motions of a surface can be used to classify the surface type: for example, if a surface is slippable by both a pure rotation and a pure translation in the direction of the rotation axis (as in Fig. 2.3), then the surface is a cylinder.

Previous methods for fitting kinematic motions have been either basis-dependent (leading to results that change significantly depending on the scale of the data, under small noise; see Figs. 2.5 and 2.6), or they are specialized to pure-translation or rotation-dominant motion (e.g., cylinders or helices) and do not work well for the full range of surface types expressable by standard kinematic surface equations. We give a detailed analysis of the problems caused by basis dependence, and then show how a method from a related problem provides a new, basis-independent solution that is generally applicable to any linear velocity field. We also



a b c

Figure 2.2: A simple example of kinematic surface fitting: (a) A slippable motion (illustrated by red streamlines) is found for some selection of the data (in blue). (b) The data (in green) is projected to some shared sweep plane, where it can be fit by a generator curve. (c) Advecting the generator curve along the slippable motion field generates the kinematic surface.

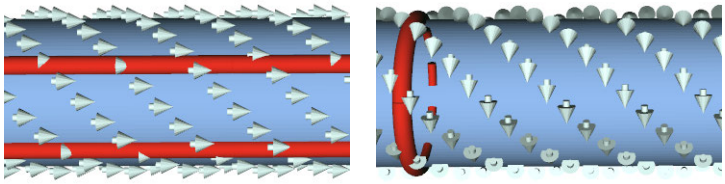


Figure 2.3: Visualization of two kinematic motions of a cylinder. The arrows show the direction of the fields at individual points, and the red lines are streamlines showing paths traced by following the velocity fields.

discuss iterative methods to improve these solutions. Our improvements to kinematic motion fitting can easily plug into any kinematic surface fitting system.

2.2 Overview of Kinematic Field Fitting

Previous work on fitting the velocity fields of kinematic surfaces has used a common fitting method (Sec. 2.2.2) and problem formulation: Given a set of n points with corresponding unit-length surface normals $\{\mathbf{p}_i, \mathbf{n}_i\}$, and a velocity field (Sec. 2.2.1) parameterized by some vector $\mathbf{m}$, find field parameters that are "most slippable" with respect to the data points.

2.2.1 Velocity Field Types

Three velocity fields have been proposed. In order of increasing generality, they are: First, a *constant field*, which accepts only translational motion [115]:

$$\mathbf{v}(\mathbf{p}) = \mathbf{c}. \tag{2.1}$$

Second, a *helical field*, which adds optional rotational motion for helices and surfaces of revolution [108]:

$$\mathbf{v}(\mathbf{p}) = \mathbf{r} \times \mathbf{p} + \mathbf{c}. \tag{2.2}$$

Finally, a *spiral field*, which adds optional scaling motion for cones and logarithmic spirals [107]:

$$\mathbf{v}(\mathbf{p}) = \mathbf{r} \times \mathbf{p} + \mathbf{c} + \gamma\mathbf{p}. \tag{2.3}$$

2.2.2 Common Fitting Method

Almost all kinematic surface fitting papers use a common direct fitting algorithm [108] to find the field parameters.

The input data for the common fitting algorithm is a set of points with unit surface normals, $\{\mathbf{p}_i, \mathbf{n}_i\}$, which we express as a concatenated vector, $\mathbf{x} := [p_x, p_y, p_z, n_x, n_y, n_z]^T$. We also set a weight w_i per data point, to control the contribution of that point to the algorithm's error metric. To apply the algorithm to a polygon mesh input, in this chapter we take the mesh vertices as the points $\mathbf{p}_i$, and the average vertex normals as the associated normals $\mathbf{n}_i$. Because vertices may not be uniformly distributed over the surface, we let weight w_i be the total area of the faces adjacent to the vertex. An alternative approach is to triangulate the mesh and use Dunavant's Gaussian quadrature rules [31] to integrate sums over the mesh surface. In our experience, these two approaches to sampling a mesh give nearly equivalent results for kinematic surface fitting.

The common fitting algorithm expresses the slippability of the field with respect to the data in terms of some symmetric covariance matrix $\mathbf{M}$, such that $(\mathbf{v}(\mathbf{p}) \cdot \mathbf{n})^\mathbf{2} = \mathbf{m}^T\mathbf{M}\mathbf{m}$, where $\mathbf{m}$ is the vector of velocity field parameters. For example, for the spiral field with parameters $\mathbf{m} = [\mathbf{r}_x, \mathbf{r}_y, \mathbf{r}_z, \mathbf{c}_x, \mathbf{c}_y, \mathbf{c}_z, \gamma]^T$, matrix $\mathbf{M}$ will be

$$\mathbf{M} := \sum_i w_i \mathbf{f}(\mathbf{x}_i)\mathbf{f}(\mathbf{x}_i)^T, \tag{2.4}$$

$$\text{where } \mathbf{f}(\mathbf{x}) := [(\mathbf{p} \times \mathbf{n})_x, (\mathbf{p} \times \mathbf{n})_y, (\mathbf{p} \times \mathbf{n})_z, n_x, n_y, n_z, p \cdot n]^T.$$

The common fitting algorithm introduces a normalization to avoid degenerate solutions such as the field with $\mathbf{v}(\mathbf{p}) = \mathbf{0}$ everywhere. The most straightforward solution would be to normalize $\mathbf{v}(\mathbf{p})$ by $||\mathbf{v}(\mathbf{p})||$:

$$\frac{\mathbf{v}(\mathbf{p}_i) \cdot \mathbf{n}_i}{||\mathbf{v}(\mathbf{p}_i)||}. \tag{2.5}$$

However, this normalization has been largely avoided because it has been considered too computationally expensive to work with [108]. Instead, previous work has normalized the cumulative squared error by some quadratic function q (Sec. 2.2.3) of the motion parameters $\mathbf{m}$:

$$\underset{\mathbf{m}}{\operatorname{argmin}} \frac{\sum_{i=1}^{n} w_i(\mathbf{v}(\mathbf{p}_i) \cdot \mathbf{n}_i)^2}{q(\mathbf{m})}. \tag{2.6}$$

Note that this normalization can alternatively be viewed as a constraint on the solution vector: minimizing Eqn. 2.6 is equivalent to minimizing the non-normalized metric under the constraint $q(\mathbf{m}) = 1$.

Finally, this quadratic function q is expressed in terms of some (often singular) symmetric matrix $\mathbf{N}$: $q(\mathbf{m}) = \mathbf{m}^T\mathbf{N}\mathbf{m}$. Using the method of Lagrange multipliers to solve the constrained minimization problem, $\mathbf{m}$ must be a solution to the generalized eigenvalue problem

$$(\mathbf{M} - \lambda\mathbf{N})\mathbf{m} = 0. \tag{2.7}$$

All eigenvectors corresponding to small eigenvalues of the matrix pencil $(\mathbf{M} - \lambda\mathbf{N})$ are then slippable motions of the data points.

Some systems augment this core fitting method by iterative re-weighting (for example to downweight outliers) [108, 59] or subsequent application of a general, non-linear fitting technique (which requires a reasonable "initial estimate" from the core fitting method to perform well) [85].

2.2.3 Quadratic Normalization Functions

A few different quadratic normalizations have been proposed. The two commonly used normalizations are: First, the *unit constraint*, in which the full parameter vector is constrained to have unit magnitude [48, 59]. For the spiral field, this becomes

$$(||\mathbf{r}||^2 + ||\mathbf{c}||^2 + \gamma^2) = 1. \tag{2.8}$$

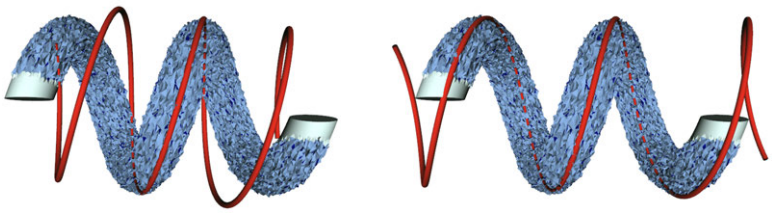
Second, the *rotation constraint*, which just constrains the rotation axis $\mathbf{r}$ [108]:

$$||\mathbf{r}||^2 = 1. \tag{2.9}$$

The constant field is simple enough so that the constraint $||\mathbf{c}||^2 = 1$ solves the problem perfectly; however this is only applicable to the constant field.

The choice of normalization fundamentally affects the resulting fit: The rotation constraint requires the solution to include a rotation component of constant magnitude, so it should only be used when it is known in advance that rotation is included in the desired solution [108].

The unit constraint is "basis-dependent": the best fit will change if the data is scaled or translated. Previous work using the unit constraint suggested re-scaling the data to a



Rotation constraint
a

Taubin constraint
b

Figure 2.4: A spiral field (Eqn 2.3) (red streamlines) is fit to a blue selection of a helix with randomly perturbed vertices (Gaussian noise with $\sigma = 0.4\%$ of bounding box size) using (a) the rotation constraint and (b) Taubin's constraint.

fixed size to address this issue [48, 59]. This mitigates but does not eliminate the problem: We show in Sec. 2.3 that there is no fixed scale that can eliminate the artifacts of basis dependence.

2.3 Failure Cases of Previous Methods

The kinematic surface fitting methods described in Sec. 2.1 work on many examples, as shown in previous work [108, 48, 59]. However, we found that they also fail on some common, simple cases. In this section, we demonstrate and explain the cases that cause these methods to fail.

In each failure case, the key problem is that the slippability measured for each data point is scaled by $||\mathbf{v}(\mathbf{p})||$, a quantity that is unrelated to the actual tangency of the field to the surface at that point. While the quadratic normalizations of Sec. 2.2.3 avoid the degenerate case of $\mathbf{v}(\mathbf{p}) = 0$ everywhere, they still permit "low velocity" fields for which $||\mathbf{v}(\mathbf{p})||$ is reduced at every data point. The "most slippable" solutions under these constraints are therefore biased towards such "low velocity" fields. For noisy data, where the expected slippable motions have some error, these bias-favored solutions can be erroneously chosen as the most slippable fields.

We demonstrate the failure cases in practice on simple synthetic example meshes, for which the ideal solutions are readily apparent. Each example mesh is generated with approximately uniform vertex sampling. We introduce a small amount of Gaussian noise (with σ less than 0.5% of the bounding box size and smaller than half the average edge length), and we recompute the normal for each sample point by averaging face normals. These small-noise examples should not be challenging, but they cause the previous methods to perform poorly due to their systematic biases.

2.3.1 Rotation Constraint Failure Cases

The rotation constraint, $||\mathbf{r}|| = 1$, requires a fixed magnitude rotation to be part of the solution field, but does not specify the constraints on the translational or scaling motion of the field. Any translational or scaling motion in the field will therefore increase the magnitude of $||\mathbf{v}(\mathbf{p})||$ everywhere, beyond what is mandated by the rotation constraint. This additional velocity scales the error at each data point, causing the rotation constraint to be systematically biased against translational or scaling motion in the presence of noise.

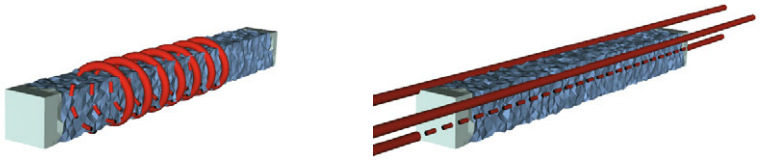
To demonstrate this, we fit a helix with small noise using the rotation constraint in Fig. 2.4. The rotation constraint underestimates the pitch (translational motion) of the helix. In contrast, a fitting method without this bias [137] (described in Sec. 2.4.1) recovers the expected pitch.

2.3.2 Unit Constraint Failure Cases

The unit constraint (Eqn 2.8) is basis-dependent: Fitting results depend on the scale and translation of the data points. Therefore, previous authors who use this constraint [48, 59] first center the data points around the origin, and scale the bounding box to a fixed size (e.g., so that the longest edge of the box has unit length). The bias of the unit constraint depends on the chosen size.

One source of bias in the unit constraint favors scaling and rotation at small scales: Velocities of linear scaling and rotational motions are proportional to the distance from the center or axis of the motion; so as data points come closer together, the velocities (and thus errors) from scaling and rotation become smaller. To demonstrate this bias, we fit four sides of a box with small noise using the unit constraint in Fig. 2.5. At scales with bounding box size 4 or smaller, the resulting fit has a significant, erroneous rotational component (Fig. 2.5a).

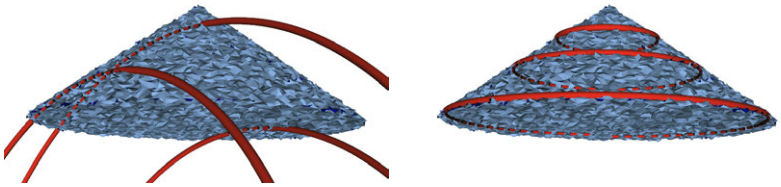
Another source of bias favors offsetting the rotation axis from the origin. As the constant parameter $\mathbf{c}$ increases to achieve the offset, the rotation axis $||\mathbf{r}||$ must scale down proportionally (to satisfy the unit constraint). Scaling down the rotation axis scales down



Unit constraint, size=4
a

Taubin constraint
b

Figure 2.5: A spiral field (Eqn 2.3) (red streamlines) is fit to a blue selection of a box with randomly perturbed vertices (Gaussian noise with $\sigma = 0.2\%$ of bounding box size) using (a) the unit constraint with bounding box size 4 and (b) Taubin's constraint.



Unit constraint, size=4 a

Taubin constraint b

Figure 2.6: A spiral field (Eqn 2.3) (red streamlines) is fit to a cone with randomly perturbed vertices (Gaussian noise with $\sigma = 0.4\%$ of bounding box size) using (a) the unit constraint with bounding box size 4 and (b) the Taubin constraint.

the velocity (and thus error) of rotation. To demonstrate this bias, we fit a cone with small noise using the unit constraint in Fig. 2.6. At scales with bounding box size 4 or greater, the resulting fit erroneously offsets the rotation axis (Fig. 2.6a).

From these two examples, we see that the bias of the unit constraint can cause problems at small scales (sizes ≤ 4) and large scales (sizes ≥ 4) alike: no single scale works well for all cases.

2.4 Improved Fitting Methods

The problems we have identified in kinematic surface fitting methods are similar to those faced by early methods for algebraic surface fitting [111]. "Approximate maximum likelihood" (AML) methods are a general class of methods for fitting algebraic surfaces and general parametric models, which have been applied to many other problems [137, 68, 23], although they have not been applied to kinematic surface fitting before. In this section, we show how to apply AML methods to the kinematic surface fitting problem to create an improved kinematic surface fitting method.

AML methods can apply to any parametric model that takes the form $\mathbf{m} \cdot \mathbf{f}(\mathbf{x}) = 0$. For our kinematic equations (Sec. 2.2.1) this holds – in the case of a spiral field, for example, $\mathbf{m}$ would be the parameter vector $[r_x, r_y, r_z, c_x, c_y, c_z, \gamma]^T$, and $\mathbf{f}(\mathbf{x})$ would be the transformation defined in Eqn. 2.4.

The "maximum likelihood" (ML) method seeks to minimize the squared distance from each data point $\mathbf{x_i}$ to the nearest corresponding point on the model surface $\bar{\mathbf{x}}_\mathbf{i}$; in other words, to minimize

$$\sum_i w_i ||\mathbf{x_i} - \bar{\mathbf{x}}_\mathbf{i}||^2, \text{ subject to } \mathbf{m} \cdot \mathbf{f}(\bar{\mathbf{x}}_\mathbf{i}) = 0.$$

Because $\bar{\mathbf{x}}_\mathbf{i}$ is the zero of $\mathbf{m} \cdot \mathbf{f}(\mathbf{x})$ that is closest to $\mathbf{x_i}$, the distance to this root can be approximated to first order by the magnitude of one step of Newton's method. This gives

the AML distance

$$\sum_i w_i \frac{(\mathbf{m} \cdot \mathbf{f}(\mathbf{x_i}))^\mathbf{2}}{||\nabla_\mathbf{x}(\mathbf{m} \cdot \mathbf{f}(\mathbf{x_i}))||^\mathbf{2}}. \tag{2.10}$$

Rewritten in terms of kinematic surface fitting, the AML method becomes

$$\sum_i w_i \frac{(\mathbf{v}(\mathbf{p_i}) \cdot \mathbf{n_i})^\mathbf{2}}{||\nabla_\mathbf{p}(\mathbf{v}(\mathbf{p_i}) \cdot \mathbf{n_i})||^\mathbf{2} + ||\mathbf{v}(\mathbf{p_i})||^\mathbf{2}}. \tag{2.11}$$

Note that the AML and ML methods are both scale dependent, because the closest element $\bar{\mathbf{x}}_i$ can be different from $\mathbf{x}_i$ in both position and normal. If the points are scaled up, differences in the normal are unchanged, but differences in the position increase. We can make this trade-off explicit: scale the data points to a fixed size bounding box (we scale it so the longest axis has length 1), then introduce a weight parameter $w_\mathbf{p}$ that scales the contribution of the position-based term:

$$\sum_i w_i \frac{(\mathbf{v}(\mathbf{p}_i) \cdot \mathbf{n}_i)^2}{w_\mathbf{p}||\nabla_\mathbf{p}(\mathbf{v}(\mathbf{p}_i) \cdot \mathbf{n}_i)||^2 + ||\mathbf{v}(\mathbf{p}_i)||^2}. \tag{2.12}$$

Note that as $w_\mathbf{p}$ goes to zero, the AML fitting equation becomes the non-linear minimization (Eqn 2.5) previously proposed for kinematic surface fitting [108]. Therefore, in theory, all results on AML fitting apply directly to this fitting problem. However, this original non-linear minimization has numerical issues around singularities where $||\mathbf{v}(\mathbf{p})||$ goes to zero: at these points, that slippability metric is undefined. A small, non-zero value for $w_\mathbf{p}$ gives a more stable metric with no undefined points.

2.4.1 Direct AML Method: Taubin's Constraint

The AML distance metric is non-linear, requiring iterative methods or approximation to find a solution. One popular approximation is Taubin's method [137], which approximates the non-linear AML metric (Eqn. 2.10) by summing all numerator and denominator elements separately:

$$\frac{\sum_i w_i(\mathbf{m} \cdot \mathbf{f}(\mathbf{x}_i))^2}{\sum_i w_i||\nabla_\mathbf{x}(\mathbf{m} \cdot \mathbf{f}(\mathbf{x}_i))||^2}. \tag{2.13}$$

Like the previous kinematic surface fitting methods (Sec. 2.1), this is a direct method solveable by a small generalized eigenvalue problem. Like those methods, it rescales the cumulative squared error (as in Eqn. 2.6), so individual points are still scaled by local velocity. However, when $w_\mathbf{p} = 0$, this constraint ensures that the overall average squared velocities have a fixed magnitude. Taubin's constraint then becomes

$$\sum_{i=1}^{n} w_i||\mathbf{v}(\mathbf{p}_i)||^2 = 1. \tag{2.14}$$

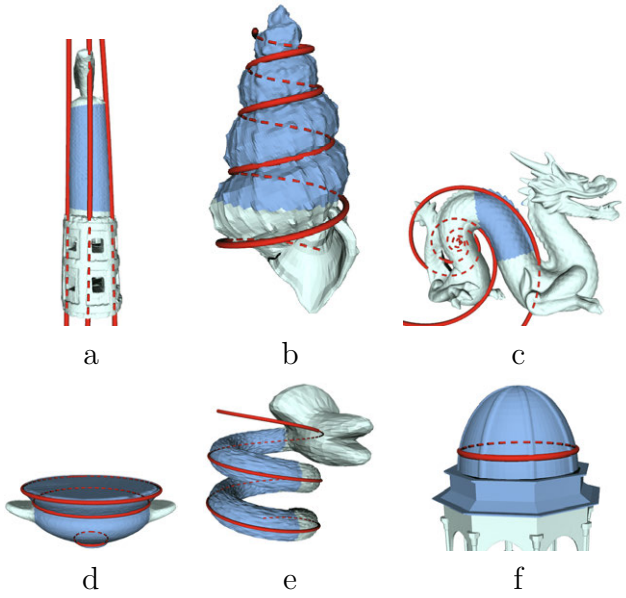


Figure 2.7: A spiral velocity field fit to a number of different selections (in blue) using Taubin's constraint. Streamlines tracing the best fitting field are shown for each image.

Because the average squared magnitude velocity is directly constrained, we can't "cheat" the Taubin-constrained error metric by choosing a field that globally reduces the velocity at all data points. This prevents failure cases of the variety described in Sec. 2.3.

Note that letting $w_{\mathbf{p}} = 0$ ensures that Taubin's method is basis independent, and does not cause stability issues: degenerate points where $\mathbf{v}(\mathbf{p}_i)$ goes to zero simply do not contribute to either the numerator nor denominator sums.

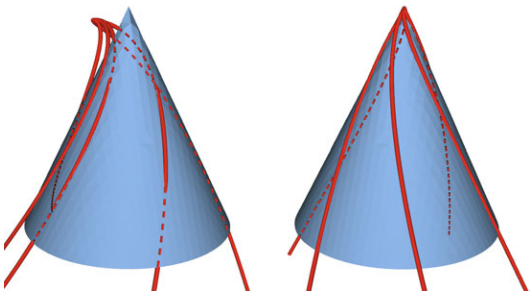
To implement Taubin's method, we express the normalization in the form $\mathbf{m}^T\mathbf{N}\mathbf{m}$ required by the standard fitting algorithm (Sec. 2.2.2). The matrix $\mathbf{N} = \sum_i w_i(\nabla_{\mathbf{x}}\mathbf{f}(\mathbf{x}_i))(\nabla_{\mathbf{x}}\mathbf{f}(\mathbf{x}_i))^T$; for the spiral field (Eqn 2.3) this is

$$\mathbf{N} = \sum_{i=1}^{n} w_i \begin{bmatrix} [\mathbf{p_i}]_\times^T[\mathbf{p_i}]_\times & -[\mathbf{p_i}]_\times & 0 \\ -[\mathbf{p_i}]_\times^T & \mathbf{I} & \mathbf{p_i} \\ 0 & \mathbf{p_i}^T & \mathbf{p_i}\cdot\mathbf{p_i} \end{bmatrix} \tag{2.15}$$

$$+w_{\mathbf{p}} \sum_{i=1}^{n} w_i \begin{bmatrix} [\mathbf{n_i}]_\times^T[\mathbf{n_i}]_\times & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \mathbf{n_i}\cdot\mathbf{n_i} \end{bmatrix}$$

$$\text{with } \mathbf{m} = [r_x, r_y, r_z, c_x, c_y, c_z, \gamma]^T.$$

For generality, we have included the $w_{\mathbf{p}}$ terms here; when applying the Taubin constraint this weight should be zero, but in a non-linear, iterative method (Sec. 2.4.2) it can be non-zero.



$w_{\mathbf{p}} := 10^{-5}$ a $w_{\mathbf{p}} := 10^{-3}$ b

Figure 2.8: A spiral field (Eqn 2.3) is fit to a cone using the HEIV method with varying values for $w_{\mathbf{p}}$. The HEIV method never converges for $w_{\mathbf{p}} := 10^{-5}$; we show the state after 101 iterations. The method converges in 2 iterations for $w_{\mathbf{p}} := 10^{-3}$.

To demonstrate the Taubin constraint in practice, we show a number of practical test cases of the Taubin constraint in Figs. 2.4–2.7 and 2.9–2.11.

While the Taubin constraint works well in practice, it remains a biased approximation — it systematically places less weight than ideal on points where the velocity field is small, and more where the velocity field is large. To address this, we turn to iterative, non-linear AML methods.

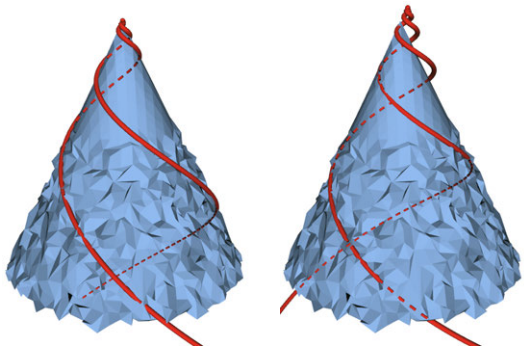
2.4.2 Iterative AML Methods: HEIV and Reduced

To minimize the true non-linear error term (either Eqn. 2.5 or Eqn. 2.12), we must use an iterative method. Fortunately, a number of iterative AML methods have been developed [23, 67, 78] – all of which have been shown to converge very quickly in theory for data with small noise [23]. All of these methods are based on solving an eigenvalue problem similar to the direct methods (Sec. 2.2.2) but iteratively adjusted to correct the weights of the data points. Because the weights can only be corrected with respect to one field, these iterative methods focus on finding a single best solution rather than the full basis of solutions provided by direct methods.

Previously the "reduced method" [67] has been suggested for use on the kinematic surface fitting problem [109], although without evaluation. This method simply iteratively re-weights the unit-constraint method. Specifically, it repeatedly solves the eigenvalue problem described in Sec. 2.2.2, with normalization matrix $\mathbf{N} = \mathbf{I}$ and error matrix $\mathbf{M}$ recomputed at the $(j+1)^{\text{th}}$ iteration as

$$\mathbf{M}_{j+1} := \sum_i w_i \frac{\mathbf{f}(\mathbf{x}_i)\mathbf{f}(\mathbf{x}_i)^T}{||\nabla_{\mathbf{x}}(\mathbf{m}_j \cdot \mathbf{f}(\mathbf{x}_i))||^2} \ , \tag{2.16}$$

where $\mathbf{m}_j$ is the parameter vector at iteration j; $\mathbf{m}_0$ can be initialized by solving with any direct method. Unfortunately, this fails in the same way as the non-iterative unit



$HEIV, w_{\mathbf{p}} := 10^{-3}$ $Taubin$

a b

Figure 2.9: A spiral field (Eqn. 2.3) is fit to a cone with Gaussian noise applied to the base ($\sigma = 1\%$ bounding box size) using the HEIV method (converged in 3 iterations) and the Taubin method.

constraint method: the inherent biases of that method are not addressed by re-weighting, and similar results to those of Figs. 2.5a and 2.6a occur. This result is consistent with the poor performance observed for the reduced method on algebraic curve fitting problems under mild noise [23].

In contrast, the "heteroscedastic errors-in-variables" (HEIV) method [78] performed much more successfully when evaluated in the context of algebraic curve fitting [23] – replicating the robustness under noise of the Taubin method, but with lower error. Intuitively this may be expected because this method can be seen as an iterative re-weighting of Taubin's method. At each iteration, it solves a generalized eigenvalue problem with $\mathbf{M}$ reweighted as in Eqn. 2.16 above, and $\mathbf{N}$ reweighted as

$$\mathbf{N}_{j+1} := \sum_i w_i \left(\frac{(\mathbf{m_j} \cdot \mathbf{f}(\mathbf{x_i}))^2}{||\nabla_{\mathbf{x}}(\mathbf{m_j} \cdot \mathbf{f}(\mathbf{x_i}))||^4} \right) (\nabla_{\mathbf{x}}\mathbf{f}(\mathbf{x}_i))(\nabla_{\mathbf{x}}\mathbf{f}(\mathbf{x}_i))^T.$$

This method performs similarly to the Taubin method on most examples we tested. Some care must be taken to choose the $w_{\mathbf{p}}$ large enough for stability; we found $w_{\mathbf{p}} \geq .001$ worked consistently, while smaller values could be unstable and thus fail to converge as shown in Fig. 2.8. We therefore set $w_{\mathbf{p}} := .001$ for our tests. Advantages of HEIV over Taubin become evident when noise is distributed unevenly over the surface, in areas which Taubin will systematically over-weight, as shown in Fig. 2.9.

2.5 Robustness to Outliers

Like any least-squares fitting method, these methods are all sensitive to outliers. Many methods can be, and have been, used to reduce the impact of such outlier points; in particular, M-estimators [61] and RANSAC [38] have both been suggested [108, 59]. The RANSAC

approach is explained in detail, in the context of kinematic surface fitting, by [59]. We recommend this procedure, albeit using the new fitting techniques and normalized distances presented here instead of Eqn. 2.8. An example demonstrating the effectiveness of this approach is shown in Fig. 2.10. For this example we assumed that outliers have an error (computed by Eqn. 2.5) greater than .1, and that 90% of points are not outliers.

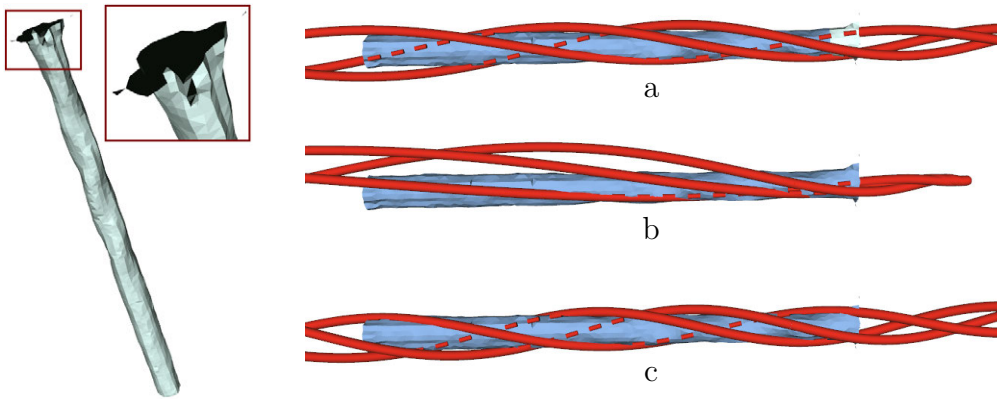


Figure 2.10: A spiral field (Eqn 2.3) is fit to a scanned model of a drill bit, on which outliers are concentrated at one end. The best-fitting vector field is visualized by red stream lines. (a) and (b) are fit without using RANSAC; in (a) the outliers at the end have been omitted manually by not selecting that portion of the mesh, while in (b) the outliers are included, significantly affecting the result. (c) is fit using RANSAC, and it gives a result similar to manually avoiding the outliers. The Taubin method is used for each fit.

2.6 New Velocity Fields

Previous methods for kinematic surface fitting did not generalize well beyond the few field types listed in Sec. 2.2.1: the unit constraint's problems only become worse with more com-

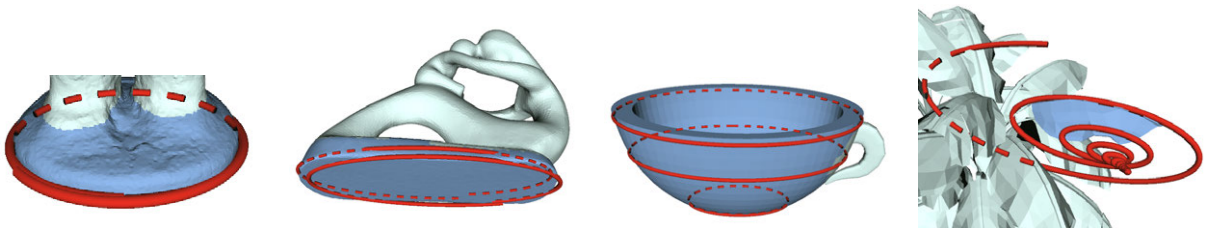


Figure 2.11: The general linear field (illustrated with red streamlines) is fit to a number of selections (in blue) on various objects.

plex fields, and it is unclear how to apply the rotation constraint unless the field prominently features a rotation axis parameter. The Taubin and HEIV methods, in contrast, apply to any velocity field linear in its parameters $\mathbf{m}$ – that is, any velocity field that can be expressed in the form $\mathbf{v}(\mathbf{p}) := \sum_i m_i \mathbf{f}_i(\mathbf{p})$ where m_i are the elements of the parameter vector $\mathbf{m}$, and the functions $\mathbf{f}_i(\mathbf{p})$ can be any functions from positions to vectors. Therefore these methods can be used to fit new, more general velocity fields. For sensible results, the class of fields chosen should also be closed under the Lie bracket operator – in other words, composing motions of multiple velocity fields in a class should result in motions that are also in that class.

The space of possible classes of velocity fields is enormous, and not easy to understand intuitively. But it opens doors to fitting some new, interesting primitives – and some simple primitives that were notably missing from the past repertoire of kinematic surfaces. For example: although spheres are handled by the more traditional kinematic surfaces, ellipsoids and general quadrics are not, because there is no support for rotation combined with some scaling. This would correspond to a non-linear field with some scaling matrix $\mathbf{S}$ as a new parameter:

$$\mathbf{v}(\mathbf{p}) := \mathbf{S^{-1}}(\mathbf{r} \times (\mathbf{Sp})) + \mathbf{c}. \tag{2.17}$$

Just by adding this scale factor, kinematic surfaces would now include all quadric surfaces as a subtype they could handle. This scaled equation is no longer linear in the parameters, but if we multiply out (letting $\mathbf{A} = \mathbf{S^{-1}}[\mathbf{r}]_\times \mathbf{S}$, where $[\mathbf{r}]_\times$ is the matrix form of a cross product by $\mathbf{r}$) we see that its fields are a subset of a class of general linear fields:

$$\mathbf{v}(\mathbf{p}) := \mathbf{Ap} + \mathbf{c}, \tag{2.18}$$

where $\mathbf{A}$ is an arbitrary 3×3 matrix. This general linear field can be used to fit fields with rotation combined with some scaling, as we demonstrate in Fig. 2.11. In Fig. 2.11a-c, the fields follow elliptical paths for shapes that are (approximately) stretched surfaces of revolution. In Fig. 2.11d, the field still fits the data well, but follows a path with exponential scaling – essentially a stretched logarithmic spiral.

From this example it is clear that the more general fields do include at least one additional, useful shape primitive, and thus seem worthy of further investigation. These more general fields could find even broader applications, but also present new challenges: with more complex velocity fields, a useful interpretation of the resulting parameters becomes more difficult. Developing a complete surface reconstruction pipeline that exploits the full range of possible kinematic surfaces will likely require exploration of additional new field types and new algorithms to fit and interpret those field types. We consider this to be outside of the scope of this thesis – an enticing direction for future work.

2.7 User-Guided Segmentation

Up to this point, we have focused on fitting a field *given* a selection of data — showing how previous methods for kinematic surface fitting fail, and how methods from other fitting

domains can be applied to fix those problems. In this section, we tackle the problem of segmentation: selecting the data to be fit. Our goal is to provide the users with great control over what they choose to interpret as a stationary sweep, while keeping the user input small and simple. Note that it may be useful to interpet a surface as a stationary sweep even if that results in high error of fit – for example, the fit in Fig. 2.15 has high error, but the fit is still meaningful and enables interesting shape editing possibilities – so an automatic segmentation which focuses on error minimization [48, 59] is not suitable for our purposes.

To explore user control of the stationary sweep fitting process, we present an interactive system to allow the user to quickly select and edit stationary sweeps. We start from a minimal user input: a single small stroke on part of the surface that the user would like to extract as a stationary sweep. We present a simple region-growing approach that can often extract a reasonable result from this input alone. We then explore ambiguous cases where the system may not initially give the desired result, and we present lightweight additional inputs that we show can clarify these ambiguous cases.

2.7.1 Basic Segmentation Algorithm

To make an initial selection in our system, the users stroke the surface that they would like to extract as a stationary sweep, and then press the "extract stationary sweep" button. The system uses a straightforward, region-growing approach to find a stationary sweep under the user's stroke: First, the system selects the data under the user stroke, then it alternatingly fits a kinematic field to the selected data and expands the selection to neighboring surfaces that also match the fitted field. An overview of this process is given as Algorithm 1.

In detail, the algorithm proceeds as follows: Starting from the selection of surface elements (in our case, mesh facets) under the user's stroke, our algorithm first fits an initial kinematic field to that selection. For the fitting step (line 3 in Algorithm 1), we use the direct Taubin method of Sec. 2.4.1 – this is the fastest method, and gives good results for reasonably clean data. Once a tentative motion field has been estimated, the algorithm establishes an error bound as the maximum deviation between the motion field and the user-selected points. Error is computed using Eqn. 2.5. Surface elements that can be reached with a flood-fill from the area seeded by the user, and which have a deviation within the current error bounds, are added to our selection. This larger selection is used to obtain a better estimate of the motion field. The error bound is recomputed as the deviation between the new field and the surface elements originally marked by the user. Then the process repeats with another flood-fill from the user-selected surface elements; it stops after a round in which no new elements have been accepted. This process tends to converge after just 2-3 iterations (see Table 2.1). The growth process is illustrated in Fig. 2.12.

Algorithm 1 Fit stationary sweep

1: Initialize sweep s to set of surface elements m marked by user
2: **while** s continues to change **do**
3: Estimate parameters $\mathbf{p}$ from s (Section 2.4.1)
4: Find max distance t from m to $\mathbf{p}$
5: s = floodfill from m to neighboring surfaces closer to $\mathbf{p}$ than t
6: **end while**

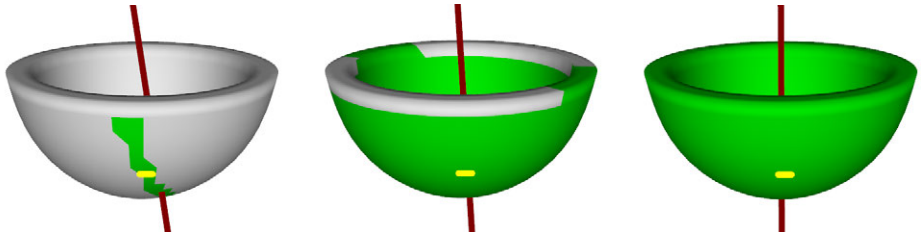


Figure 2.12: Stationary Sweep Fitting: A stroke (yellow) by the user defines an initial sweep region that immediately provides a (potentially incorrect) axis of rotation (red). The system automatically grows the sweep region (green) and extracts improved sweep parameters iteratively until the region can no longer be extended.

2.7.2 User Guidance and Refinement of Fit

The problem of fitting and segmenting stationary sweeps is inherently ambiguous: There are often multiple reasonable selections and sweep fits that could be chosen for a given surface. To address cases of ambiguity, we support additional user input that lets the user disambiguate the problem, ensuring that the selection and fit are exactly what the user intends. We identify and address two main types of ambiguity: First, the size of the desired selection depends on the user's intent, and the default selection may be too small or too large. To address this ambiguity, we give the users a simple way to control the error threshold that limits the size of the initial selection, and we also give the users tools to exclude undesired parts of a selection even if their error of fit is as low as that of the desired selection. The second type of ambiguity is that the fitted field may not have the desired motion: in ambiguous cases such as a portion of a sphere or a cylinder, there are mutliple fields that fit a given surface equally well. To address this ambiguity, we give users the option to force the fitted field to follow the direction of their input stroke, allowing for fine-grained control of the fitting result.

2.7.2.1 User controls to correct a too-small selection

The algorithm initially selects surface elements based on error-of-fit: It selects elements that fit at-least-as-well-as the user-marked elements. This approach makes it very natural to extend the selection further: If the surface area covered by the extracted sweep does not extend as far as the user would like, she can simply mark a few additional surface

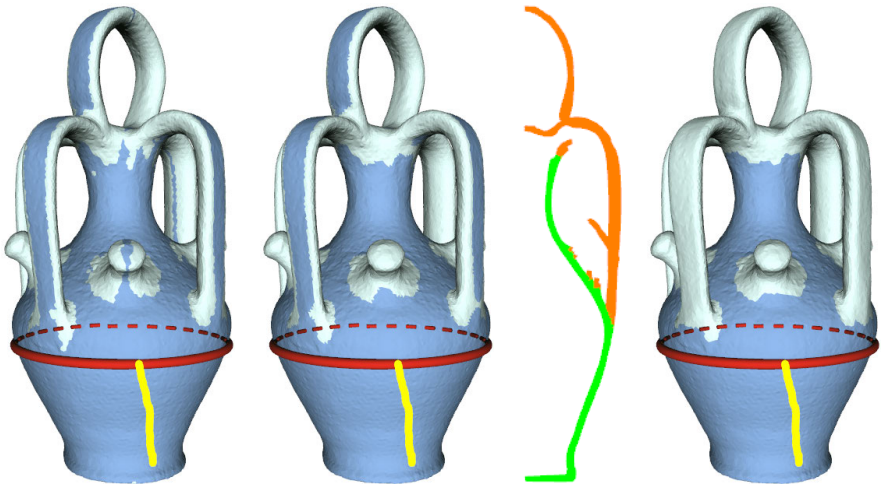


Figure 2.13: (a) From an initial stroke (yellow) drawn by the user, the motion field (red streamlines) is extracted, and matching surface elements are marked in blue. (b) The result is filtered by morphological opening and closing [59], removing some spurious selections. (c) Undoing the rotational sweep component generates a 2D "profile" view. By de-selecting portions of this profile (orange) the user can manually cull undesired areas from (d) the body sweep.

elements that should be included. The algorithm will re-fit the kinematic surface, using the newly marked elements as well as the original elements when computing error bounds. This implicitly increases the error bounds to accommodate the newly marked elements, ensuring that the fit will include more of the surface. The fitting algorithm region-grows from all marked parts simultaneously, so this also allows the user to fit disconnected parts with the same sweep motion – for example, the top and body of a teapot (Fig. 2.1b).

2.7.2.2 User controls to correct a too-large selection

There are two common reasons that too much of the surface may be included in the selection. First, the error threshold may have been raised so high that it isn't useful for segmentation. This can occur when the user wants to interpret a shape as a sweep even though it deviates significantly from an ideal sweep surface. In this case, the segmentation problem is no longer closely tied to stationary sweep fitting, so we turn to more general user-guided segmentation techniques that we discuss in detail in Chapter 6. An example of one such general approach is to allow the users to mark surfaces they would like to exclude from the selection, in addition to surfaces that they want to include [63].

In the second case, the selection can be too large even when the error threshold is reasonably small, because some surface elements can have low error-of-fit despite not being part of the intended selection. Hofer et al. [59] note that it is common for curve-like pieces of adja-

cent, conceptually-unrelated surfaces to be included in a kinematic field-based segmentation, because they are also approximately tangent to that globally-defined motion field. We see this occur for example for the Botijo model in Fig. 2.13a: The user has stroked to select the body of the model, but parts of the handles are also tangent to that same velocity field and so are also included in the initial selection. Hofer et al. suggest using morphological operators to remove these features [59]: specifically, the "closing" operator, which deselects selected elements touching the boundary of the selection, followed by the "opening" operator, which selects unselected elements touching the boundary of the selection. We show the result of applying these morphological operators in Fig. 2.13b: The operators successfully remove some narrow strips along the handles (e.g., the front-most nub, and the back-right handle), but fail to remove the larger selections along the other handles. Because there is no well-defined metric to remove these larger selections, additional user interaction is needed. While generic selection-refinement approaches (Chapter 6) could be used, there is also an elegant stationary-sweep specific approach to disambiguating these cases: We let the user refine the selection in a 2D "profile" view. To generate the profile view, our system collapses all the extracted surface parts into a common 2D plane, by moving the parts along the fitted motion field. In this profile view (Fig. 2.13c) the undesirable portions (shown in orange) can readily be selected and eliminated from the extracted sweep surface. The same 2D profile view can also be used to generate an idealized sweep profile curve by applying a skeletonization to the green areas of the profile view, or for shape editing as we discuss below.

2.7.2.3 User controls to correct the sweep motion

Many simple shapes can be fit well by multiple fields: for example, a spherical shape can be fit by any rotational field with the axis passing through its center. A cylinder (Fig. 2.14) can be fit by a purely translational field, a purely rotational field, or any helical field in the subspace of linear combinations of these two fields. In such cases, we provide ways for the user to express and enforce a preference over which field is extracted. One common case is that the user wants a pure rotation: In this case, we simply provide an option to enforce a pure-rotation fit. We enforce pure rotation simply by fitting a helical field and removing any pitch from the field after each fit. (An iterative method to enforce a pure rotation fit also exists [106] but we did not explore this because this simple, direct method seemed to work fine already.) In the more general case, we let users constrain the fit to follow the *direction* of their strokes, which gives users fine-grained control of the motion field they fit. We demonstrate this control in Fig. 2.14.

To constrain the motion field fit to follow the direction of the user stroke, we use a method based on [49]: we sample the user's stroke path and generate a set of stroke points projected to the mesh surface, $\mathbf{s}_i$, with corresponding stroke directions $\mathbf{t}_i$. We project the stroke directions to be orthogonal to the surface normals $\mathbf{n}_i$ of the mesh at points $\mathbf{s}_i$. We define the path normal as this surface normal, and we define corresponding path binormal as $\mathbf{b}_i := \mathbf{t}_i \times \mathbf{n}_i$. We can then measure whether a motion field is tangent to the path in the same

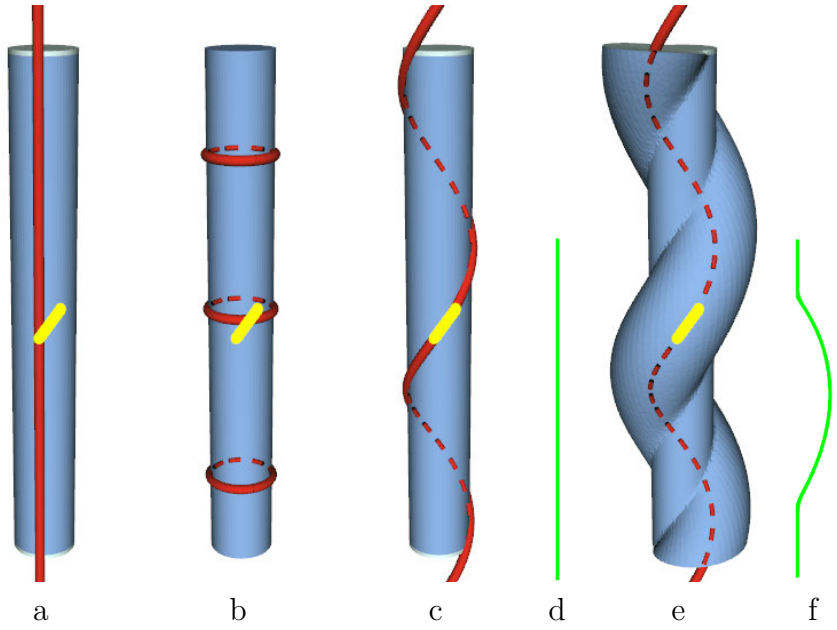


Figure 2.14: A cylinder fit by a stroke, (a) with an unconstrained helical field, (b) with a rotation-specific fit, and (c) with a helical field following the stroke direction. Selected surface is shown in blue, resulting fields are visualized by red streamlines. The helical field allows interesting new edits; for example we can transform the profile from (d) to (f), resulting in a spiral deformation (e).

way we measure whether a field is tangent to a surface: by measuring how orthogonal the field at the sampled points $\mathbf{v}(\mathbf{s}_i)$ is to the corresponding normal (and now binormal) vectors, $\mathbf{n}_i, \mathbf{b}_i$. These sampled points and normals, $\{\mathbf{s}_i, \mathbf{n}_i\}$ and $\{\mathbf{s}_i, \mathbf{b}_i\}$, can be added directly to the set of points and normals over which we already optimize. The new stroke points must have some weight relative to the original data points, which represents a trade-off: the field could follow the user's stroke more closely, at the expense of following the data less closely, or vice versa. This trade-off is exposed to the user as a simple slider, with the system following the stroke as closely as possible at one end of the slider, and following the data as closely as possible at the other end. As a default value for this slider, when stroke-following is enabled, we give the data fit and stroke fit equal weight overall.

2.7.3 Shape Editing

As soon as a stationary sweep is fit to the data, we allow a user to begin using that representation to perform shape edits. These edits are performed in the 2D profile view generated by advecting all the surfaces facets fit by a given sweep into a common 2D plane via the fitted motion field. Edits in this 2D view are immediately transferred back to 3D, so users can see the results of their edits interactively. We demonstrate some example edits for a ro-

tational sweep in Fig. 2.1c; note the black curve shows the 2D profile view where the user edits are performed.

This approach works even for shapes where the stationary sweep model is a very inexact fit. In Fig. 2.15 a not-very-circular statue has been fit with a rotational sweep (a). The corresponding 2D profile view (b) has substantial thickness due to the variation of the radial distance to the central rotation axis. A small portion of this profile has been highlighted and selected for editing (b, c). If the highlighted profile portion is moved to the right, the parts of the sculpture containing the four noses and eight cheeks are being bulged outwards (d). Alternatively, if just the portions of the profile corresponding to the nose-tips are selected (e), then only the four noses are elongated; the cheeks of the face do not bulge outwards (f).

Note that this approach does not extract an explicit representation of the sweep profile – it simply collapses the 3D mesh to a 2D "profile" space. If an explicit representation of the profile is desired, one can perform a skeletonization [115] of the projected profile elements. This can be desirable for some more advanced editing operations: for example, to beautify a sweep, one can project all points to the nearest point on an "ideal" profile curve.

2.7.4 Performance

Thanks to the use of a fast direct method for fitting the kinematic motion field, performance is not an issue: Even for relatively large input surfaces, the stationary sweep finder converges in less than a second (Table 2.1) on our test machine – a laptop with 2GB RAM and a 2.4 GHz Core Duo processor. The fitting time is proportional to the number of points processed. All examples converged in 2-3 iterations, where an iteration includes one step of region growing

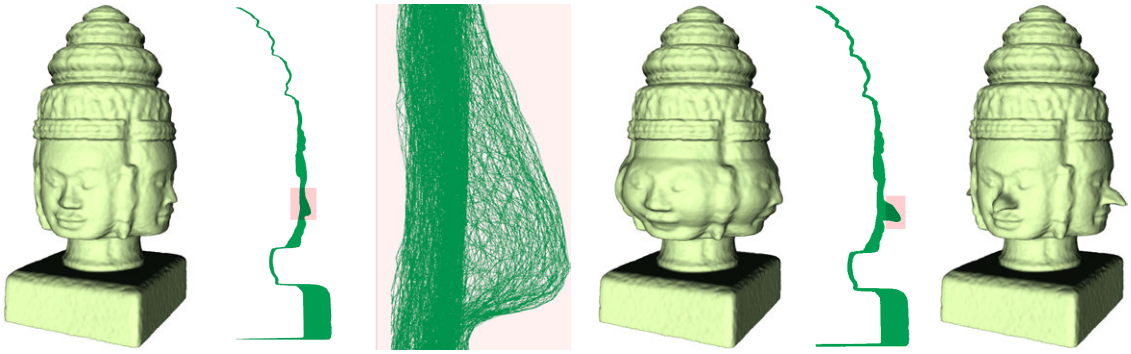


Figure 2.15: (a) A not-very-circular statue is fit with a rotational sweep. (b) The corresponding "thick 2D profile" view, generated by rotating all surface facets to a single 2D plane. (c) A small portion of the "profile" selected for editing. (d) Modified statue after the selected profile portion has been moved to the right. (e) Just the nose has been selected in cross section view. (f) Four pointy noses have been created by moving the selected portion to the right.

and re-adjusting the parameters.

Table 2.1: Timings for stationary sweep module

Model	# Tris	Section Fit	Interaction	Iterations	Time (ms)
Bowl (Fig. 2.12)	4k	Full shape	One stroke	3	16 ms
Teapot (Fig. 2.1)	25k	Body and top	Two strokes	2	94 ms
Botijo (Fig. 2.13)	82k	Body	One stroke	2	171 ms
Angkor Wat (Fig. 2.15)	163k	Full shape	One stroke	3	375 ms

Chapter 3

"Progressive" Swept Surface Primitives

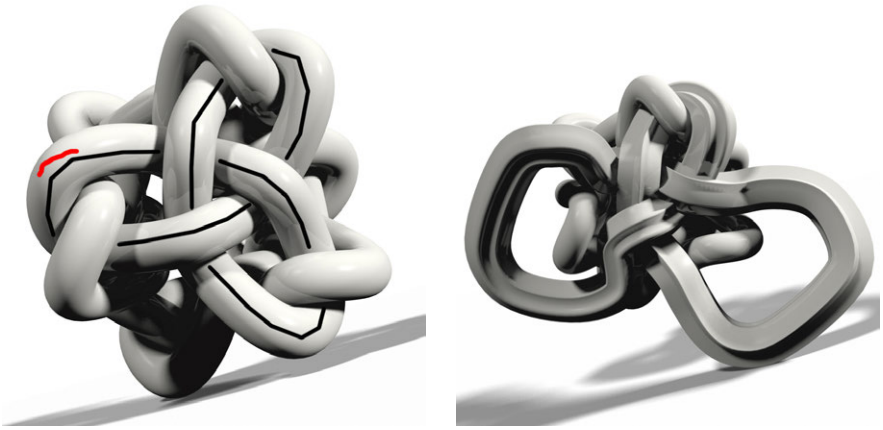
We now describe our "progressive" sweep fitting module – our second sweep fitting module. It is aimed at fitting more complex swept structures than the stationary sweep module, at the cost of slightly more user involvement and computational effort. We define "progressive" sweeps as a class of generalized translatory sweeps, characterized by an arbitrary smooth space curve that serves as a sweep path, along which we sweep a planar cross section that is allowed to be rotated or even affinely distorted as it moves along the path. The necessary information is captured by a relatively large number of local parameters.

In this chapter, we present a simple algorithm for fitting progressive sweeps from a single user stroke that indicates the desired location and direction of the sweep. The users can optionally make additional strokes to refine the sweep fit in ambiguous cases – for example, if they would like the sweep to progress past a region where the fitting error is high, or to take a different path in a branching structure. We show that our sweep fitting algorithm handles complex shapes in less than 10 seconds, making it a reasonable tool to use in the middle of an interactive redesign session. Finally, we discuss the shape redesigns our progressive fit enables, showing a number of examples.

3.1 Background

A number of previous systems have fit sweep-like structures to enable general shape edits. Dion et al. [29] and Ueng et al. [140] describe a system for extracting sweeps from surfaces that have a fixed orientation of the sweep cross section. Ramamoorthi and Arvo [114] describe a general sweep fitting algorithm that extracts some relatively simple sweeps by asking the user to place a cylinder as a first approximation, and then iteratively optimizing a subset of the cylinder parameters to deform it into the given model shape; for complex sweep surfaces (such as those in Fig. 3.1 and Fig. 3.5), it is often not clear where to place the cylinder to align it with a contorted sweep surface. Moreover, the general framework in [114]

simultaneously optimizes too many parameters to permit a global optimization of the whole sweep surface at a speed suitable for interactive use. Yoon and Kim [151] describe a system for sweep-based free-form deformation, in which the sweep fitting step is almost fully manual: The user must provide a tubular "cage" that encloses the entire sweep and follows the desired sweep path. Constructing such a cage for a complex sweep (Fig. 3.1) is essentially as hard as re-building the geometry from scratch. Our contribution is a straightforward algorithm that fits a fairly general class of sweeps, and handles complex examples with minimal, simple user input.



a b

Figure 3.1: (a) An input surface with several touching components, with one trefoil component shown with a black overlay. The system can easily fit a progressive sweep to that component after the user draws a short stroke (shown in red) on its surface. (b) After the closed trefoil sweep has been extracted, the user can then interactively edit its cross section and sweep path.

3.1.1 Progressive sweep representation

We define our progressive sweeps as general translatory sweeps that handle diverse tube-like structures with arbitrary 3D sweep paths and 2D cross sections. Choosing how to parameterize such a sweep structure is a matter of making trade-offs: Additional degrees of freedom in the representation can allow us to handle new, more complex examples, but also make the fitting problem harder and more ambiguous. With excessive degrees of freedom, the fitting process can take longer and end up delivering qualitatively worse results – since the added degrees of freedom can increase the possibility of undesirable but technically well-fitting results – which in turn will require more user interaction to refine and correct. In the trade-off between generality and efficiency, we found the following parameters seemed to encapsulate a reasonably flexible, but still easy-to-fit, domain of swept surfaces:

1. A polyline in 2D defining a planar *cross-section template*, which will be transformed to generate the local sweep cross sections along the sweep path.

2. A polyline in 3D defining the path that the sweep follows. Cross sections are placed perpendicular to this path.

3. A set of transformation parameters at each control point defining the sweep path, specifying a transformation applied to the cross-section template to generate the local cross section at that point.

Each set of transformation parameter values generates a 2D transformation matrix to be applied to the cross-section template: This could be a 2D rotation, a uniform scaling, or an arbitrary 2×2 matrix. The tangent of the sweep path at this point is taken as the average directions of the two segments adjacent to it, and the cross-section template is always kept perpendicular to that direction. When moving our cross section along the sweep path, we orient the cross section relative to the rotation minimizing frame [145] before the 2×2 matrix transformation is applied.

Note that our representation does not allow 3D rotations of the cross section with respect to the tangent vector, or the possibility to morph the cross section between arbitrary shapes (also called "lofting"). The additional flexibility gained did not seem worth the greatly increased ambiguity, as we found that simply allowing affine scaling of the cross section can already let the system handle a surprisingly diverse set of examples (e.g., Figs. 3.3 and 3.4b).

3.2 User-Guided Segmentation and Fitting

As with stationary sweeps, we provide a simple, fast fitting module for progressive sweeps. This module lets the users make a small stroke on a surface that they would like to fit with a progressive sweep, and then automatically fits a corresponding sweep surface. To help disambiguate the desired sweep, the direction of the user stroke should be roughly aligned with the desired initial direction for the sweep path to follow.

3.2.1 Fitting Algorithm

In contrast to stationary sweeps, there is no direct linear method for fitting progressive sweep surfaces: We must instead rely on a relatively expensive non-linear optimization. To achieve efficiency, we reduce the size of the optimization task by breaking the overall problem into a chain of smaller optimization problems, each of which has relatively few parameters. As an overview, our process is to generate an initial cross-sectional template from the user stroke, then alternate between adding a new segment to the sweep path and optimizing the parameters of this newest segment, while simultaneously also fine-tuning the parameters of the last few segments. This incremental, segment-by-segment expansion of the sweep continues until no further segment can be added without exceeding the current error

bounds. The individual error for each sweep segment is calculated as the maximum distance of any computed sweep surface point from the original input mesh. As a heuristic, our initial error bounds are set automatically at a small constant factor above the error of the initial sweep segment. We later discuss how the user may increase the error bounds interactively by drawing a stroke in a not-yet-covered region to designate additional surface elements to be incorporated into the sweep. Our fitting process is given as Algorithm 2.

Algorithm 2 Fit progressive sweep

```
1: Initialize template T based on stroke drawn by user
2: for d = −1 to 1 step 2 do {Go forwards and backwards}
3:   while error of fit below threshold do
4:     Add segment to sweep in direction d
5:     k := 2
6:     repeat
7:       Optimize newest k segments
8:       k := 2 + k
9:     until error of fit below threshold or k > 6
10:  end while
11:  Remove last segment (for which the fit failed)
12: end for
```

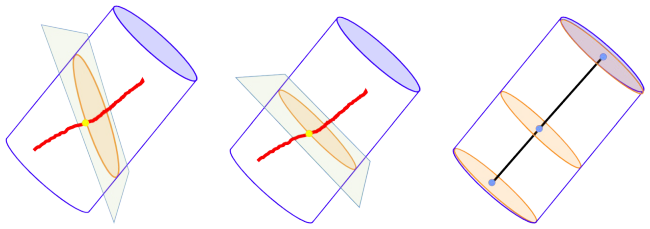
3.2.1.1 Initializing the Cross-Section Template

First, the user draws an initial stroke on the mesh surface to indicate a starting location and direction for the sweep (Fig. 3.2a). The system then extracts an estimate of a cross-section template near the middle of the user stroke. To do so, it traces a local "cross-section" curve on the mesh surface, starting from the user's stroke and tracing in the plane perpendicular to the user's stroke direction. This template will be updated during subsequent optimization steps: When the direction of the first segment of the sweep is optimized, the system re-generates the template with a cutting plane perpendicular to the new direction. This allows the system to adapt robustly to imprecisely drawn user strokes, so a casually-drawn user stroke is still sufficient to obtain a good sweep initialization.

3.2.1.2 Optimization Procedure

Our system iteratively generates new sweep segments along a piecewise linear polyline sweep path. For each new segment, it initializes a new control point by extrapolating from the last two control points. We start out with a very conservative, short extension of the sweep path, and then let the penalty function below (Eqn. 3.1) determine the most effective length for the next segment.

We start out by optimizing the new sweep segment in the context of the last segment extracted. To improve the speed of the optimization, we initially only optimize over these



a b c

Figure 3.2: Initializing a progressive sweep: A stroke (red) by the user defines the approximate direction for the selected sweep. An initial cross-section template is extracted in a plane near the middle of the stroke and perpendicular to it (a). Plane orientation and template are updated as the direction of the first sweep segment is optimized (b). The forwards and backwards sweep-extent of the first segment is pushed as far as possible within the set error bounds (c).

two segments: For much of the length of many sweeps, this will be sufficient to find a good fit. However, in ambiguous regions we may need to optimize more of the sweep to find a good fit. If the error of fit is above the error bounds, we increase the number of segments included in the optimization, and try again. We give up at some maximum number of segments k, and assume we have reached the end of the sweep. In all cases, we regularly sample points p_i on the last k segments of the sweep surface and use the standard Levenberg-Marquardt algorithm [87] to minimize the following energy:

$$\begin{aligned} \Sigma_i d(\mathbf{p}_i)^2 + w\Big(&\Sigma_{j=N-k}^{N}\big(\frac{1}{||\mathbf{s}_j - \mathbf{s}_{j-1}||}\big)^2 \\ &+\Sigma_{j=N-k}^{N}||\mathbf{x}_j - \mathbf{x}_{j-1}||^2 \\ &+\Sigma_{j=N-k}^{N-1}\kappa(\mathbf{s}_{j-1},\mathbf{s}_j,\mathbf{s}_{j+1})^2\Big). \end{aligned} \tag{3.1}$$

The first term is the squared distance of a sweep surface point from the original mesh. The remaining terms are regularization terms that penalize, respectively: short segments (avoid zero-length steps), changes in the transformation parameters x_j (avoid un-needed parameter wobbles), and curvature of the sweep path (prefer straight and smooth curves). A weight value w is chosen to scale all the regularization terms. This weight controls how much the sweep is willing to bend to fit the data: A larger weight w generates a simpler, smoother sweep path at the cost of potentially following the data less closely. The desired smoothness of the sweep path is typically scale-dependent, and tied to the size of the cross section: A sweep with a smaller cross section can bend more before it self intersects, and intuitively we expect thinner tubular shapes to have less resistance to bending. To give weight value w an intuitive meaning for sweeps of different scales, we re-scale the data before fitting so that the initial cross section fits in a unit-sized box.

To calculate the curvature-based penalty we use the discrete integrated curvature metric [8] based on the last 3 control points of the sweep path $\kappa(\mathbf{s}_{j-1},\mathbf{s}_j,\mathbf{s}_{j+1})$. Specifically, given

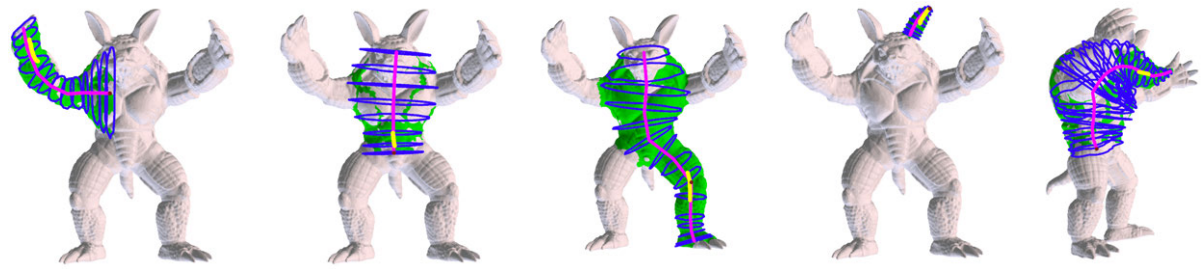


Figure 3.3: The user can quickly extract any number of diverse sweeps on an armadillo.

two contiguous sweep path vectors $\mathbf{e}_{j-1} = \mathbf{s}_j - \mathbf{s}_{j-1}$ and $\mathbf{e}_j = \mathbf{s}_{j+1} - \mathbf{s}_j$, the discrete curvature is calculated as

$$\frac{2\mathbf{e}_{j-1} \times \mathbf{e}_j}{|\mathbf{e}_{j-1}||\mathbf{e}_j| + \mathbf{e}_{j-1} \cdot \mathbf{e}_j} .$$

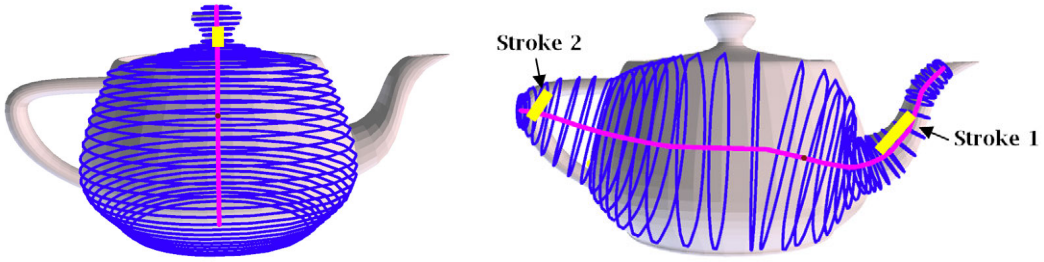
To compute distances $d(\mathbf{p}_i)$ efficiently, we use a precomputed adaptively-sampled distance field [44]. Gradients required by the Levenberg-Marquardt method are computed by finite difference approximation.

Note that this procedure, like most non-linear optimizations, relies on a number of arbitrary constants: the number of samples $\mathbf{p}_i$ of the sweep surface, the weight w of the regularization terms, and the maximum number of segments k over which we optimize. There is no fundamentally correct value for these constants: They entail performance and accuracy trade offs in this system. Based on our investigations so far, we use six cross sections per segment, with each cross section sampled at the vertices of the initially extracted 2D cross-section template. We use a default regularization weight w of .0001 to ensure that the distance term $d(\mathbf{p}_i)$ dominates; however we find that the trade-off between smoothness and willingness to exactly follow the data is something relatively intuitive, so we could also expose parameter w to the user as a smoothness slider.

3.2.2 User Control of Fitting Process

The initial location and direction of the user stroke yields significant control over how the shape will be interpreted, as illustrated in Figs. 3.3 and 3.4. In Fig. 3.4a, a short vertical stroke on the lid of the teapot produces a vertical sweep, but in Fig. 3.4b, a stroke on the spout produces a mostly horizontal sweep. In both cases, free-form scaling of the cross section by an arbitrary 2×2 matrix was permitted. Depending on how the user plans to use and/or modify the teapot model, one or the other representation may be preferable.

In some cases, the sweep fit may stop sooner (or later) than the user desires. In the situation shown in Fig. 3.4b, the sweep fit, which starts in the middle of the spout, would naturally stop at the tip of the spout and where the spout meets the teapot body. However, the user allows the sweep fit to continue throughout the body of the teapot by adding an additional sweep path stroke. Internally, the error margins that specify what surface



a b

Figure 3.4: Different initial strokes result in different sweep fits. (a) A vertical stroke on the lid produces a vertical sweep. (b) Stroke 1, on the spout, produces a mostly horizontal sweep. Stroke 2, near the handle, is used to extend the range of the sweep fit across the teapot body.

elements are acceptable to be included in the sweep fit are increased appropriately, so that the sweep can continue. If the sweep goes beyond what the user had in mind, the user also has a command to halt the sweep fitting process, and the ability to clip the ends of the fitted sweep at any point.

In other cases, thanks to the inherent ambiguity of the sweep-fitting problem, the progressive sweep module may find an error-minimizing fit that is not the sweep envisioned by the user. In these cases the user can draw a new target path, starting from the point where the previous fit went awry. We then re-run the fit from that point, with a new regularization term to ensure it follows the corrected path.

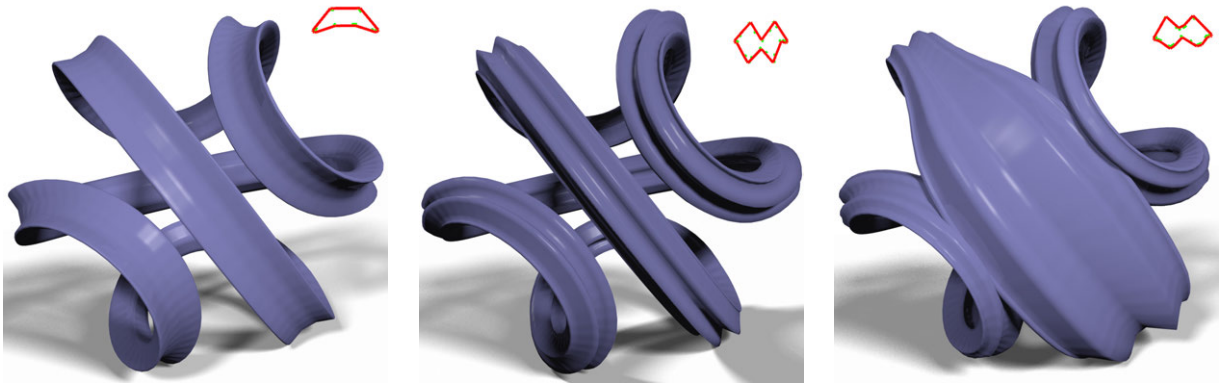
3.3 Editing with Progressive Sweeps

We allow users to edit a surface using progressive sweep structures as soon as the parameterized structures have been fitted to the input data. To enable sweep-based editing, we create a correspondence between the vertices of the original surface and their closest images on the extracted sweep model. Note that this is different from the distance used to extract the sweep: For sweep extraction, we efficiently compute the distance from sweep to surface via a pre-computed adaptively sampled distance field, but doing so does not give a correspondence. The correspondence could be ambiguous near sharp bends in the sweep path where multiple parts of the sweep may be almost equi-distant from a given point on the surface, but for our initial prototype we simply use the first closest point found, and this gives the results shown in this chapter.

The correspondence between the sweep and the original surface is represented as follows: Each vertex on the surface that corresponds to the sweep is given a parameter t, indicating where the vertex maps to along the sweep path, and a 2D position p_2, indicating the position of the vertex with respect to the cross section in that frame. When the user updates the pa-

rameters of the sweep, the system re-computes where these new sweep parameters map the corresponded vertices, and updates their 3D positions accordingly. Note that this representation is detail-preserving: The exact position of the vertex relative to the sweep is encoded in p_2, so the deviations between the sweep surface and the edited surface are maintained across transformations of the sweep parameters, as shown in Fig. 3.7.

When the user edits the 2D cross section template, in our initial prototype system we simply move each 2D position p_2 exactly as the closest point on the cross section template moves. This approach worked well enough in practice to generate the results shown in this chapter, but has the same potential ambiguities of correspondence as the correspondence from surface to sweep has. An alternative approach that avoids these amibiguities is to use mean value coordinates [42] to define a smooth deformation of the 2D cross section space.



a b c

Figure 3.5: (a) The input surface with a cross-section template (shown in red) that twists as it moves along the loop. (b) A new surface based on a modified cross section. (c) Another modified surface with yet a new cross section, which was also scaled up locally in one part of the sweep.

Results of the editing process are shown in Figs. 2.1, 3.1, 3.7, and 3.5. In Fig. 2.1 the Utah teapot has been decomposed into a stationary sweep for the body and two progressive sweeps for the spout and the handle. The geometry of these three parts has then been altered at a high level: The sweep paths of the spout and the handles, as well as the cross-sectional profiles of the body and of one handle-instance have been modified. The modified parts have been re-assembled in an imaginative new way. Fig. 3.6 shows the additional editing controls provided for a progressive sweep: the sweep path itself, the cross-section template, and the adjustable transformation parameters along the path (in this case, scale). Fig. 3.5 illustrates the effect of the latter two controls: The user can make a global change to the cross-section template (Fig. 3.5b) or introduce a dramatic localized change in the scale of the profile (Fig. 3.5c).

Figs. 3.6 and 3.7 demonstrate the detail-preserving capabilities of our system. The left side of Fig. 3.6 shows the leg of the Armadillo model being extracted as a progressive line-

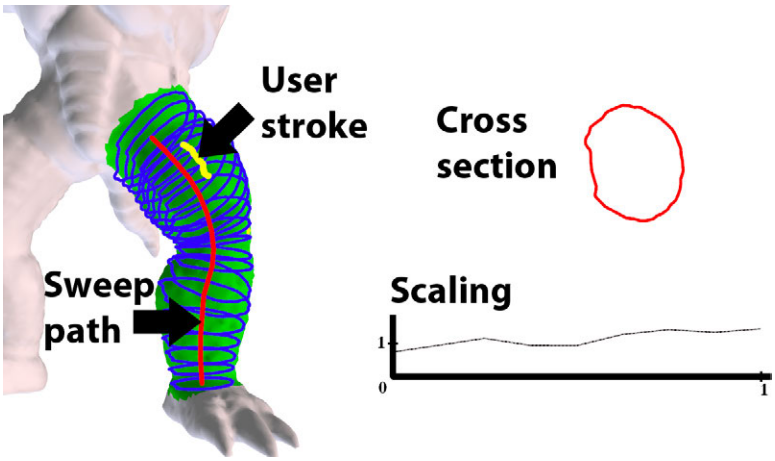
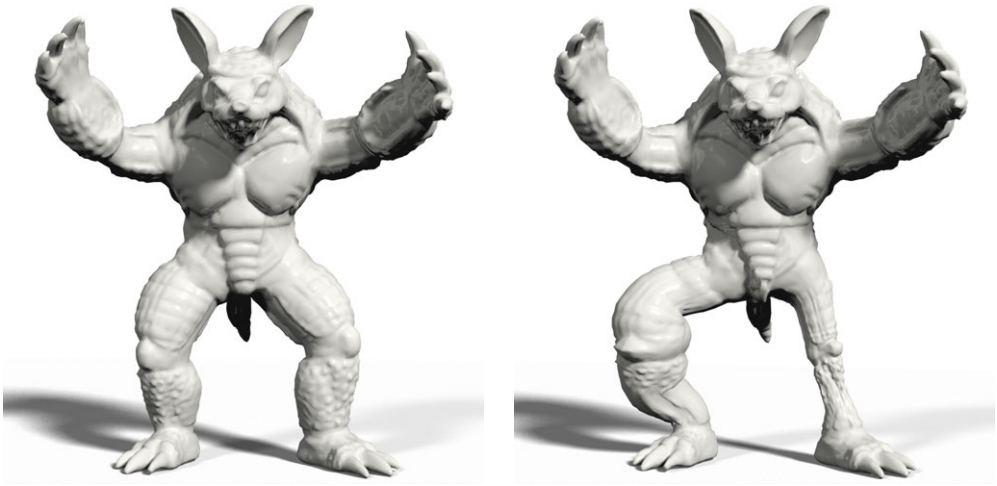


Figure 3.6: Key elements of the user-interface: the surface being reverse engineered, and the extracted edit-handles: the discovered sweep path and corresponding cross-sectional template, and a curve that controls the scale transformation applied to this template as it moves along the sweep path. The user can edit the surface by clicking and dragging any of the curves.

sweep. The user has drawn a small stroke on the left thigh to get the sweep started. The automatically extracted sweep path and cross-section template are shown in red. Both these curves can now be edited interactively and will immediately generate a new surface. The curve at the bottom right of Fig. 3.6 controls how the cross-section template varies as it moves along the sweep path. The y-coordinates of this curve indicates the scale factor as a function of the position along the sweep path; it can also be edited interactively. As the parameters of the sweep are modified, the surface is updated dynamically. The results can be seen in Fig. 3.7b: One leg of the Armadillo has been made thicker and more strongly bent; the other has become thinner and straight; but in both cases the original fine-structure has been preserved, because we maintain deviations between the surface and the sweep when we compute our surface-to-sweep correspondences.

3.3.1 Results

We tested our sweep extractor modules on a wide variety of input files. Figs. 3.3 and 3.4 demonstrate how the user can fit the same shape with very different sweeps. Fig. 3.5 shows the extraction of a progressive sweep along a rather long and highly curved path; in addition, its cross section does not simply follow a rotation-minimizing frame, but twists left and right while traveling from one lobe to the next. Figure 3.1 demonstrates that such sweeps can also be extracted from a tightly packed model or cluttered scene. Figs. 3.7 and 3.8 demonstrate that we can fit surfaces with complex details that deviate from the sweep primitive.



a b

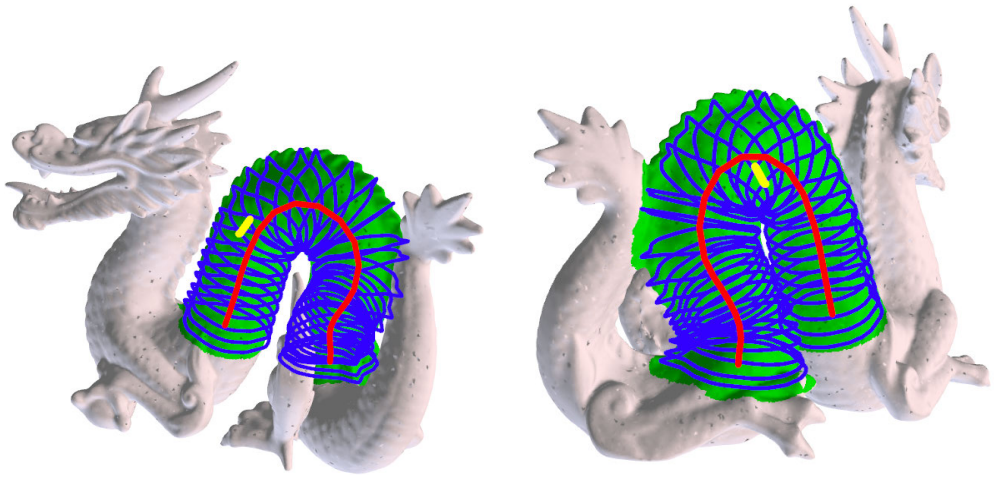
Figure 3.7: (a) The input Armadillo surface. (b) Modified surface after fitting sweeps to both legs and modifying the sweep paths and scale factors of the cross sections.

3.3.2 Performance

Table 3.1: Timings for progressive sweep module

Model	# Tris	Section Fit	# Segments	Time (ms)
"Pax Mundi" Sweep (Fig. 3.5)	5k	Whole surface	111	8531 ms
Teapot (Fig. 2.1)	25k	Spout	12	2578 ms
Armadillo (Fig. 3.7)	50k	One Leg	14	2593 ms
Trefoil Cluster (Fig. 3.1)	69k	One Knot	56	6531 ms
Dragon (Fig. 3.8)	870k	Main hump	20	5640 ms

To accelerate our sweep-fitting optimization, we pre-compute an adaptively sampled distance field [44] and use this to compute distances from the sweep to the data. Yin et al. showed that this pre-computation can be done in less than a second on a GPU for meshes with on the order of a million triangles [150]. After an adaptively-sampled distance field had been computed, all our examples took less than a minute of user interaction and computation time on our test machine – a laptop with 2GB RAM and a 2.4 GHz Core Duo processor. Fitting any single sweep generally took less than 10 seconds. Significant factors in the performance of the fitting algorithm were the number of segments required to fit, and the quality of the optimal fit (i.e., varying noise / bumpiness of the input surface that is not captured in the cross section). When a surface is noisy or has surface details not captured by the swept surface, like the spines of the dragon in Fig. 3.8, the optimizer will expend more effort trying to improve the fit by performing more iterations and expanding the number of segments over which the optimization is performed. The ends of the sweep also tend to require more



a b

Figure 3.8: From a stroke (in yellow) on the dragon mesh, we automatically extract the main hump of the dragon. Our default error thresholds stop here, though of course the user may manually ask the system to extract more. For clarity we show two views of the same fit; the initial stroke was made in view (a).

iterations, because the system sees these ends as high-error regions which it fails to fit (this gives a small advantage to looping sweeps which have no open ends).

To begin mesh editing with the extracted sweep parameters, a correspondence from original mesh vertices to the extracted model must be established; this requires time proportional to the size of the mesh, taking 1063 ms for the dragon mesh (Fig. 3.8 – our largest example, with 870k triangles). After the correspondence has been established, the mesh vertices can be updated quickly in response to any changes to the sweep parameters; this update is performed in a single pass over the mesh vertices, which takes just 16 ms for the dragon mesh.

Chapter 4

Quadric surface primitives

Quadric surfaces – ellipsoids, hyperboloids, paraboloids, cones, cylinders, and planes – are a common primitive for construction of man-made shapes, and thus another important class of primitive for our inverse modeling system to handle. By fitting quadrics to lower-level data, we arrive at a higher-level parameteric representation, and also enable other high-level shape editing methods that we discuss in subsequent chapters: notably, CSG-based editing (Chapter 7), and also editing with hierarchical structure – for example, constraining a sweep path to lie a quadric surface (Chapter 8).

In this chapter, we present a complete catalog of accurate, direct methods for fitting all types of quadric surfaces, and an interactive system for selecting such surfaces. We focus on type-specific fitting methods, which allow the user to optionally constrain or influence the choice of quadric type: If the user specifies a *desired quadric type* then our system uses these type-specific fitting methods to guarantee the fitted quadric has that type; otherwise, our system uses general quadric fitting to find the error-minimizing quadric of any type. These type-specific methods are useful in the common case that the users have some preference regarding the desired quadric type – either due to aesthetic sensibility, or because they know the true shape should have a specific type. When the desired type is known, general quadric fitting should not be relied upon because even a small amount of noise can easily alter the type of the best general quadric fit, as shown in Figs. 4.1, 4.6, and 4.8.

There are many applications where a quadric type is known a-priori: Allaire et al. give the example of fitting a bone joint that is known to be a hyperboloid [3]; many CAD parts are known to be cylinders or cones by construction; hyperbolic paraboloids show up in architecture, art, and Pringles potato chips. In some cases it is important that the quadric surface be bounded – for example, if the whole quadric surface is expected to correspond to some real surface – and in this case, the type must be an ellipsoid and not a hyperboloid. (Note that the best ellipsoid may be within ϵ of a paraboloid (Sec. 4.2.2), which is effectively not bounded, but we will also find a linear subspace of quadrics, in Sec. 4.2.1, which generally also contains much less eccentric, well-fitting ellipsoids, and is thus a good place to start a non-linear search for an ellipsoid of bounded eccentricity.) There are other cases where one knows the surface must have one of several types: Minimal surfaces ("soap film") have

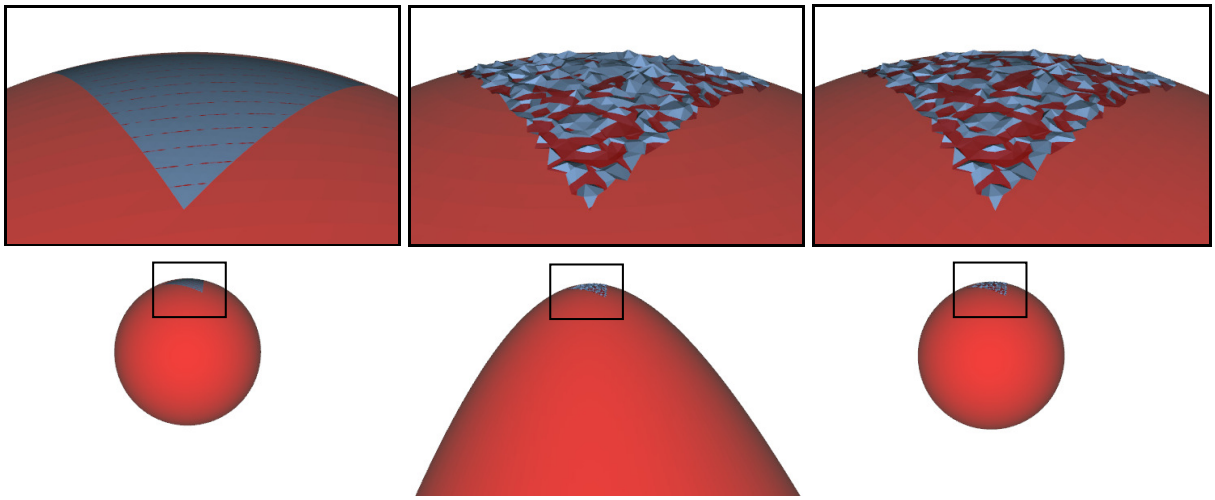


Figure 4.1: The difference between an ellipsoid and a hyperboloid can be very small in terms of local surface behavior, leading to ambiguity in the best-fitting quadric type. We show a small section of a sphere (blue) with no noise (a) and with small Gaussian noise ($\sigma = 1\%$ of bounding box size) (b and c) fit by a quadric (red). The top row shows a zoomed in view, while the bottom row is zoomed out; note the shapes are almost indistinguishable in the zoomed in view. For (a) and (b) we show the result of general quadric fitting (Sec. 4.1.2); for (c) we show the result of sphere-specific fitting (Sec 4.3.4).

negative Gaussian curvature everywhere, so they should be fit by hyperboloids of one sheet or related boundary types such as hyperbolic paraboloids. Developable surfaces must have zero Gaussian curvature everywhere, so they should be fit by general cylinders, cones, and planes.

Our fitting methods handle a wide range of quadric types with just two high-level strategies. In Sec. 4.2, we show how quadratic constraints and a closed-form line search in parameter space allow us to effectively fit hyperboloids, ellipsoids, and paraboloids. In Sec. 4.3, we show that transforming the problems to more convenient spaces and reducing the parameters used in fitting allows us to handle all remaining cases. In all cases we ensure good results by minimizing the nearly-unbiased linear error metric introduced by Taubin [137]. Finally, in Sec. 4.4 we show how our fitting method can be used in an interactive framework to perform fast segmentation and fitting of quadric surfaces.

4.1 Background

General quadric fitting has been well studied [103, 137, 149], but methods for type-specific quadric fitting are scattered throughout the literature: Some papers handle spheres, circular cones and cylinders [86]; a few others handle ellipsoids [80] or hyperboloids [3]. Non-circular

cones and general rotationally symmetric quadrics are not typically discussed. Our main contribution in this chapter is to present a complete catalog of type-specific quadric fitting methods to handle every quadric type, including new methods that handle neglected quadric types, and improvements to previously proposed methods for ellipsoid- and hyperboloid-specific fitting methods. Every method in our catalog is a fast direct method, based on the effective Taubin method [137], suitable for use in an interactive system and the inner loop of an iterative segmentation algorithm [149].

4.1.1 A Catalog of Quadric Types

For completeness, we give a brief overview of the different quadric types. Quadrics in their most general form are represented by a 10-parameter implicit function of the form

$$f(\mathbf{c},\mathbf{p}) = c_0 + c_1 p_x + c_2 p_y + c_3 p_z + c_4 p_x^2 + c_5 p_y^2 + c_6 p_z^2 + c_7 p_x p_y + c_8 p_x p_z + c_9 p_y p_z = 0. \quad (4.1)$$

When categorizing quadric types, it is more convenient to rotate to a canonical, axis-aligned form. In that form, the quadratic cross terms are eliminated, and we need only consider the signs of the remaining pure squared terms. To perform this rotation, we re-write the implicit quadric equation in matrix form:

$$\mathbf{p^T A p}+\mathbf{b^T p} + c_0 = 0 \text{ with } \mathbf{A} = \begin{pmatrix} c_4 & c_7/2 & c_8/2 \\ c_7/2 & c_5 & c_9/2 \\ c_8/2 & c_9/2 & c_6 \end{pmatrix}, \mathbf{b} = \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix}. \quad (4.2)$$

We can then rotate to the canonical form by way of an eigendecomposition, $\mathbf{A} = \mathbf{RDR^T}$. In the rotated space $\mathbf{p_r}=\mathbf{R^T p}$, the new quadric becomes $\mathbf{p_r^T D p_r}+(\mathbf{Rb})^\mathbf{T}\mathbf{p_r} + c_0$. In this rotated space, the new quadric equation without cross terms has the form

$$c_0 + c_1 p_x + c_2 p_y + c_3 p_z + c_4 p_x^2 + c_5 p_y^2 + c_6 p_z^2 = 0. \quad (4.3)$$

The new squared terms, c_4, c_5, c_6, are the eigenvalues of the original matrix of quadratic terms, $\mathbf{A}$. The signs of these eigenvalues determine the quadric type: If they are all positive, or all negative (i.e., if the original matrix of terms $\mathbf{A}$ was positive- or negative-definite) then the quadric is an ellipsoid. If they are a mix of positive and negative, then the quadric is a hyperboloid. If any are zero, then we have a quadric subtype that exists right on the border of the domain of all ellipsoids or hyperboloids.

When all the squared terms are non-zero, we can translate the quadric to center it at the origin, resulting in the further-simplified equation of the form

$$c_0 + c_4 p_x^2 + c_5 p_y^2 + c_6 p_z^2 = 0. \quad (4.4)$$

Depending on the sign of c_0 relative to the signs of the squared terms, hyperboloids can either have one sheet or two sheets. They become cones when $c_0 = 0$.

If one of the squared terms is zero (say the z-axis term, c_6; we can rotate the quadric so that this is the case), then there is no "center" along that axis and translation cannot zero the corresponding linear term. In this case, we have a paraboloid with a canonical equation of the form

$$c_0 + c_3 p_z + c_4 p_x^2 + c_5 p_y^2 = 0. \tag{4.5}$$

If the corresponding linear term also happens to be zero, we arrive at an equation without reference to the axis with zero coefficients:

$$c_0 + c_4 p_x^2 + c_5 p_y^2 = 0. \tag{4.6}$$

This is an elliptical, hyperbolic, or parabolic cylinder.

If two of the squared terms are zero, the result is either a parabolic cylinder or (if the corresponding linear terms are also both zero) a double plane, which has canonical form

$$c_0 + c_4 p_x^2 = 0. \tag{4.7}$$

Finally, if all of the squared terms are zero, then we have a plane equation.

Rotationally-symmetric quadric types (circular cylinders, circular cones, spheroids, and circular hyperboloids) have the same equations, with the added constraint that two squared terms have the same coefficient, i.e. $c_4 = c_5$. In the case of a sphere, all the squared terms are equal: $c_4 = c_5 = c_6$.

Note that all quadric types besides ellipsoids and hyperboloids exist right on the boundary of other, more general quadric types. For example, any elliptical cylinder (such as $p_x^2 + p_y^2 = 1$) is an infinitesimal parameter change ϵ away from becoming an ellipsoid (e.g. $p_x^2 + p_y^2 + \epsilon p_z^2 = 1$) or hyperboloid (e.g. $p_x^2 + p_y^2 - \epsilon p_z^2 = 1$) or paraboloid (e.g. $p_x^2 + p_y^2 + \epsilon p_z = 1$). We can always perturb the parameters of any elliptical cylinder by ϵ to arrive at an ellipsoid or hyperboloid, but the parameters of most ellipsoids and hyperboloids cannot be perturbed by ϵ to arrive at a cylinder. When fitting a quadric of a more general type, we allow the fitting result to include bordering more-constrained types: For example, an elliptical cylinder is a valid fitting result of ellipsoid-specific fitting. This simplifies our discussion, and avoids exclusion of, or bias against, quadrics near the boundary of their quadric type. Enforcing the "true" quadric type can always be done subsequently by perturbing the result by ϵ as needed. We illustrate these boundary relationships in Fig. 4.2.

4.1.2 Algebraic Direct Fitting Methods and Taubin's Method

Algebraic direct fitting methods are a standard class of methods commonly used for fitting quadric surfaces [111]. In this work, we focus primarily on Taubin's direct fitting method [137], which has been shown to be an effective, nearly-unbiased direct method for quadric fitting [24, 116, 41].

All algebraic fitting methods are based on the same simple idea: We can approximate a difficult, non-linear error (like the true orthogonal distance from data points to a quadric

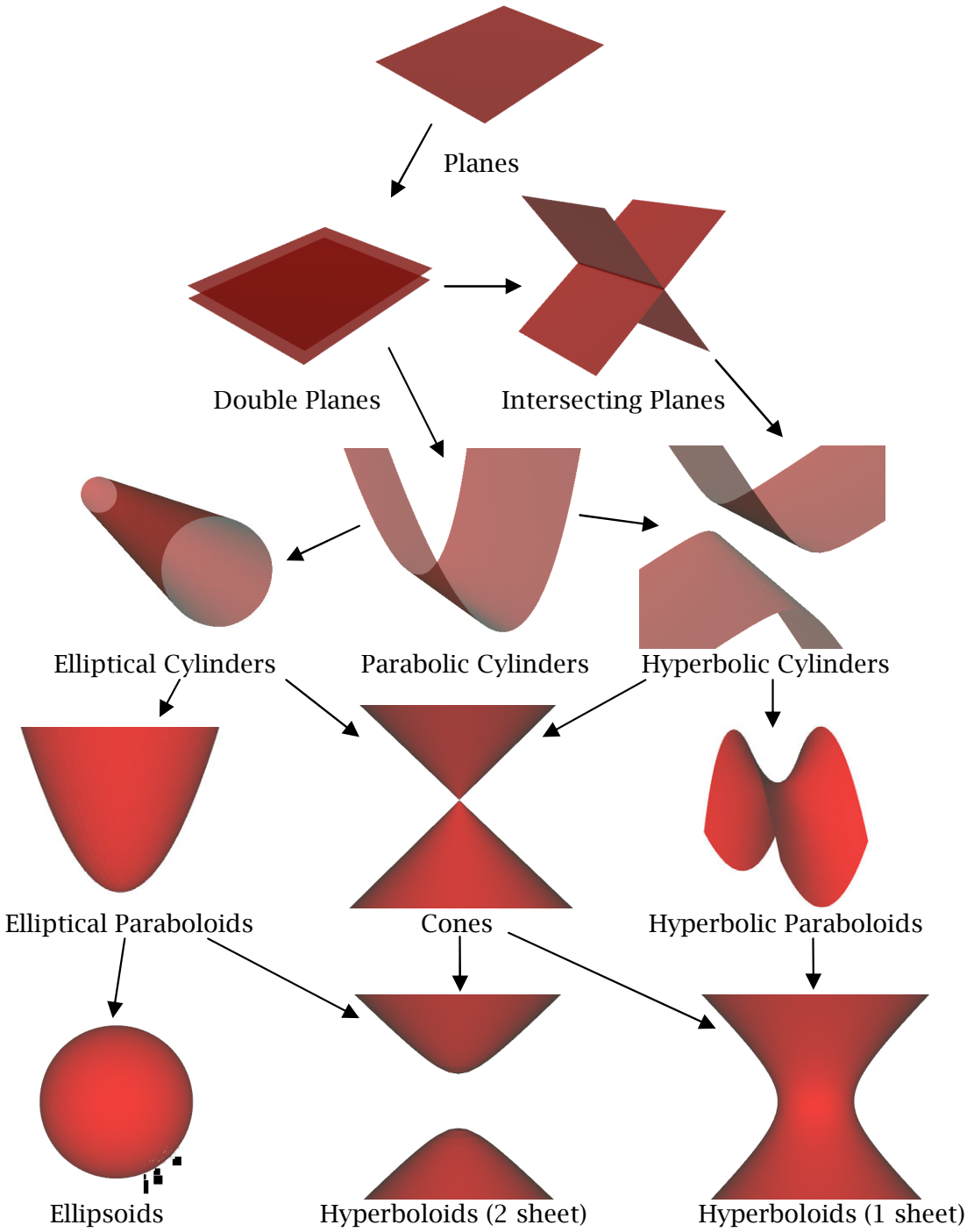


Figure 4.2: A chart of quadric types, not including rotationally symmetric subtypes. Arrows indicate "is on the boundary of" relationships: One quadric type is on the boundary of another if any quadric of the first type can be perturbed by ϵ to create a quadric of the second type. These relationships are transitive, so, for example, planes lie on the boundary of all other quadric types.

surface) by a simple, linear approximation: $\sum_{i=0}^{n} w_i f(\mathbf{c}, \mathbf{p}_i)^2$ where $f(\mathbf{c}, \mathbf{p})$ is an error metric that is linear in terms of the parameter vector $\mathbf{c}$, and w_i is a weighting per data point. Weights w_i are set propotional to the local surface area that point represents; we use Dunavant Gaussian quadrature rules [31] to set these weights, as we discuss in Sec. 4.5. For general quadrics, we use the algebraic function that implicitly defines the quadric surface, Eqn. 4.1, as our "algebraic" error metric, $f(\mathbf{c}, \mathbf{p})$.

The scale of that linear error metric is arbitrary: Scaling the vector $\mathbf{c}$ scales the error without changing the surface. All algebraic methods therefore normalize the error by some quadratic normalization function, $q(\mathbf{c})$, arriving at the error metric, $\left(\sum_{i=0}^{n} w_i f(\mathbf{c}, \mathbf{p}_i)^2\right) / q(\mathbf{c})$. This normalization function can also be viewed as a constraint function: It is equivalent to minimize $\left(\sum_{i=0}^{n} f(\mathbf{c}, \mathbf{p}_i)^2\right)$ subject to the constraint that $q(\mathbf{c}) = 1$. It is standard to require that both the squared error and normalization are quadratic functions, which permits an efficient solution: We define symmetric matrices $\mathbf{M}$ and $\mathbf{N}$ such that $\sum_{i=0}^{n} w_i f(\mathbf{c}, \mathbf{p}_i)^2 \equiv \mathbf{c^T M c}$ and $q(\mathbf{c}) \equiv \mathbf{c^T N c}$, and take the eigenvector $\mathbf{c}$ of the generalized eigenvalue problem $(\mathbf{M} - \lambda\mathbf{N})\mathbf{c} = 0$ with smallest eigenvalue λ as our solution.

Algebraic fitting methods are distinguished by their choice of quadratic normalization function $q(\mathbf{c})$. The choice of $q(\mathbf{c})$ dramatically affects the quality of the results: A good choice can result in a fit that is nearly as good as the best non-linear fit, while a poor one will exhibit clear biases against large parts of the solutions space. For example, an ellipse-specific normalization used for 2D conic fitting [39] biases against eccentric ellipses, leading to a fitting result that becomes less eccentric as noise increases.

The best choices of $q(\mathbf{c})$ in theory [116] and in practice [41] are Taubin's method [137], and the more recent HyperLS method [116]. HyperLS is marginally more accurate, but also more complex and less intuitive. For this thesis we explain and use Taubin's method.

Taubin's method is to choose $q(\mathbf{c}) = \sum_{i=0}^{n} w_i ||\nabla_{\mathbf{p}} f(\mathbf{c}, \mathbf{p}_i)||^2$. This is based on Sampson's error, which is a first-order approximation of the squared distance $d(\mathbf{c}, \mathbf{p})^2$ from a point $\mathbf{p}$ to a quadric $\mathbf{c}$: $d(\mathbf{c}, \mathbf{p})^2 \approx s(\mathbf{c}, \mathbf{p})^2 = \frac{f(\mathbf{c},\mathbf{p})^2}{||\nabla_{\mathbf{p}} f(\mathbf{c},\mathbf{p})||^2}$ [119]. We can view the algebraic error $f(\mathbf{c}, \mathbf{p})$ as a product, $f(\mathbf{c}, \mathbf{p}) = s(\mathbf{c}, \mathbf{p})||\nabla_{\mathbf{p}} f(\mathbf{c}, \mathbf{p})||$, of Sampson's error approximation multiplied by the (arbitrary) magnitude of the gradient of $f(\mathbf{c}, \mathbf{p})$. This arbitrary gradient magnitude can be seen as a bias term, weighting the algebraic error at each data point. If these bias weights at the data points are smaller overall for some shapes, then the algebraic error will be smaller for those shapes, leading to a bias toward those shapes. Taubin's idea – normalizing by the squared magnitude of these bias weights – counters the aggregate effect of this bias: For example, when the bias weights are smaller overall, Taubin's normalization $q(\mathbf{c})$ is likewise also smaller, countering the algebraic error's bias towards the shape. For a rigorous analysis of how well this approach removes bias, refer to the analysis of Rangarajan et al. [116].

For any given data set, we compute matrices $\mathbf{M}$ and $\mathbf{N}$ such that $\sum_{i=0}^{n} w_i f(\mathbf{c}, \mathbf{p}_i)^2 \equiv \mathbf{c^T M c}$ and $q(\mathbf{c}) = \sum_{i=0}^{n} w_i ||\nabla_{\mathbf{p}} f(\mathbf{c}, \mathbf{p}_i)||^2 \equiv \mathbf{c^T N c}$. To do so, we define $\mathbf{l}(\mathbf{p})$ such that $\mathbf{l}(\mathbf{p}) \cdot \mathbf{c} = f(\mathbf{c}, \mathbf{p})$ (we can always do so because $f(\mathbf{c}, \mathbf{p})$ is linear in $\mathbf{c}$), and define $\mathbf{l}_i(\mathbf{p})$ as the partial derivative of $\mathbf{l}(\mathbf{p})$ with respect to the i^{th} dimension. Then $\mathbf{M} = \sum_{i=0}^{n} w_i \mathbf{l}(\mathbf{p}_i)\mathbf{l}(\mathbf{p}_i)^{\mathbf{T}}$ and $\mathbf{N} = \sum_{i=0}^{n} w_i \sum_{j=1}^{3} \mathbf{l}_j(\mathbf{p}_i)\mathbf{l}_j(\mathbf{p}_i)^{\mathbf{T}}$. We compute Taubin's error metric as: $\frac{\mathbf{c^T M c}}{\mathbf{c^T N c}}$. Taubin's

method applies generally to many least squares fitting problems. We use it to fit customized quadric functions and also kinematic fields in Sec. 4.3 and in Chapter 2.

4.2 Fitting method for hyperboloids, ellipsoids, and paraboloids

Previous direct fitting methods for ellipsoids and hyperboloids have used algebraic fitting with a custom normalization function $q(\mathbf{c})$ to ensure that at least one of the resulting eigenvectors has the desired quadric type [3, 80]. These methods guarantee a hyperboloid or ellipsoid, but the result may not be a good fit: The type-constraining normalizations introduce more bias than Taubin's method, leading to poorer fitting results in the presence of noise, as shown in Fig. 4.3(a,c).

Others have proposed exploring the space of solutions returned by the fitting method: While the best fit may not have the desired type, algebraic fitting uses a generalized eigenvalue method that returns a basis of solutions, and one of the other eigenvectors might have the desired type [3]. However, these eigenvectors are not optimal in terms of fitting error – in fact, even the second-best eigenvector tends to correspond to a very poor fit, as we illustrate in Figs. 4.4 and 4.5.

Our approach is inspired by previous work in the domain of 2D conic fitting: Harker et al. [55] showed that an effective approach for ellipse- and hyperbola-specific fitting is to rely on the biased ellipse- or hyperbola-specific fitting methods to ensure the correct conic type is found, but then compensate for the bias by searching a linear subspace of conics for a better-fitting result.

In this section, we generalize the search method of Harker et al. to quadrics. We improve the method by more carefully considering which subspace we should search, and we simplify the search by introducing the observation that, when Taubin's method does not return the desired type, then the best fit is right on the boundary of that type; i.e. it is within ϵ of being a paraboloid. This result also leads us naturally to a method to fit paraboloids, and to constrain the number of sheets in the hyperboloid fit.

4.2.1 How to Define a Good Linear Subspace of Ellipsoids and Hyperboloids

Our goal is to search a linear subspace spanning both hyperboloids and ellipsoids for the best quadric of the desired type. First, we need to identify a good subspace to search: That is, we need to identify two parameter vectors $\mathbf{c}_\mathrm{a}, \mathbf{c}_\mathrm{b}$, such that $\mathbf{c}_\mathrm{a} + t\mathbf{c}_\mathrm{b}$ includes both ellipsoids and hyperboloids – ideally, well-fitting ones. For example, if we have one parameter vector $\mathbf{c}_\mathrm{e}$ that defines an ellipsoid, and another parameter vector $\mathbf{c}_\mathrm{h}$ that defines a hyperboloid, then $\mathbf{c}_\mathrm{e} + t\mathbf{c}_\mathrm{h}$ is an acceptable subspace spanning both hyperboloids and ellipsoids; at $t = 0$,

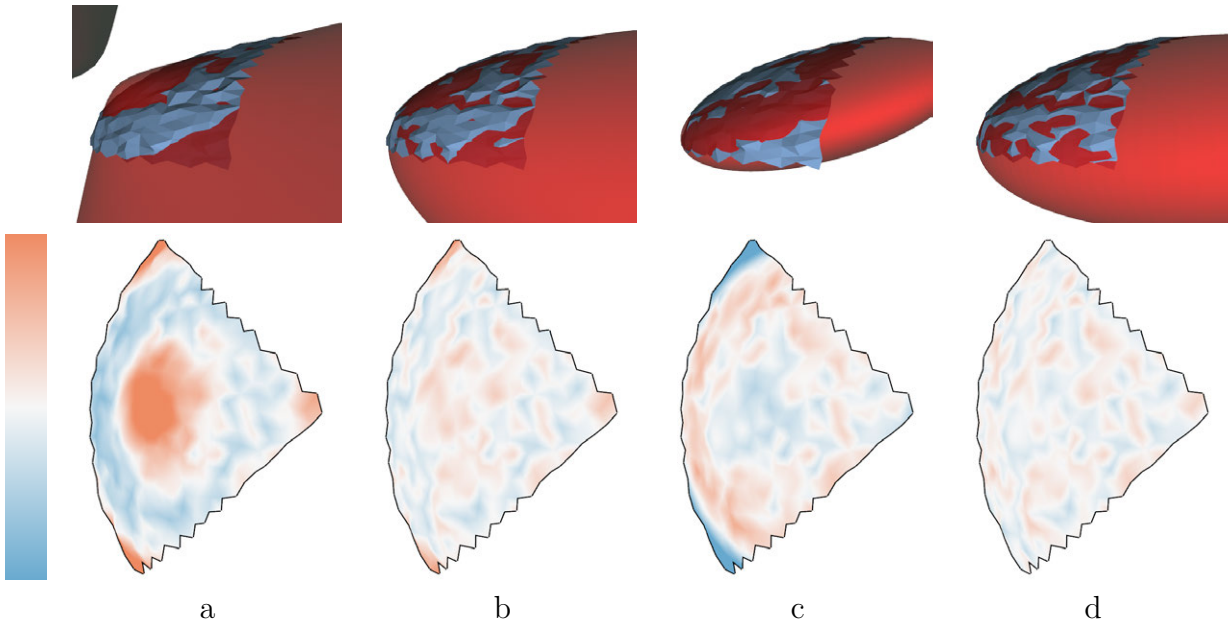


Figure 4.3: Ellipsoid- and hyperboloid-specific fitting results for an 8th of an ellipsoid with vertices perturbed by Gaussian noise ($\sigma = 0.5\%$ of bounding box size). We show the results of Allaire et al.'s hyperboloid- and ellipse-specific fitting methods (a) and (c) respectively, and our improved methods (b) and (d). Top row: Side view of mesh to be fit (blue) and quadric fitting result (red). Bottom row: Heat maps of error of fit over mesh surface, with error key on left. Errors are orthogonal distances from mesh surface to quadric, relative to bounding box size, ranging from 3% outside (blue, bottom of key) to 3% inside (red, top of key). Note the high-error regions in the center of (a) and corners of (c).

the original ellipsoid is reproduced, and as $t \to \infty$, the original hyperboloid is reproduced (because the scale of the quadric parameters has no effect on the shape).

We always let the first parameter vector $\mathbf{c}_\mathrm{a}$ be the best fitting quadric under Taubin's method, without type constraints; it will be within ϵ of being either an ellipsoid or a hyperboloid (because every quadric is, as shown in Sec. 4.1.1). For the second parameter vector, $\mathbf{c}_\mathrm{b}$, there are two natural strategies: We could use one of the remaining (not optimal) eigenvectors from Taubin's fit, or we could use one of the biased, constrained fitting methods. The subspace including the first- and second-best eigenvectors has the lowest maximum error of all subspaces, because the Taubin error is bounded between the corresponding eigenvalues. We find that when the data could be reasonably approximated by both ellipsoids and hyperboloids, this subspace tends to include both types and to span reasonable fitting results, as shown in Fig. 4.4. If the data is very far from an ellipsoid, however, the subspace may only include hyperboloids. A subspace generated from the output of biased fitting methods can span poorer fitting results, but guarantee the resulting quadric type. We use a hybrid strategy: First we search the space of the two best eigenvectors, and if this fails to include

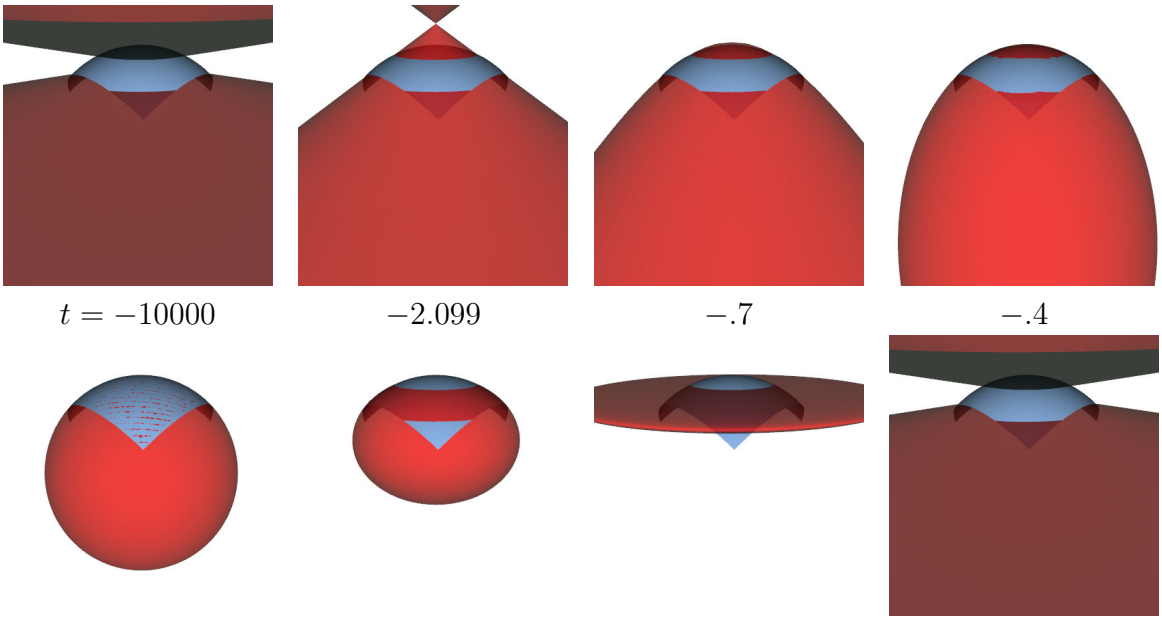


Figure 4.4: A sampling of the quadrics in a linear subspace $\mathbf{c}_\mathrm{a} + t\mathbf{c}_\mathrm{b}$, where $\mathbf{c}_\mathrm{a}$ and $\mathbf{c}_\mathrm{b}$ are the best and second-best eigenvectors of Taubin's fitting method applied to a section of a sphere. In each image the quadric is shown in red, and the data (a section of a sphere) is shown in blue. As parameter t goes from -10000 to 10000, the quadric shape goes through hyperboloids of one and two sheets, ellipsoids, cones, and paraboloids. When t is near zero, the quadrics approximate the data well.

the desired quadric type we fall back to using the biased fitting method to find the second vector $\mathbf{c}_\mathrm{b}$ by constraining it to have the type that $\mathbf{c}_\mathrm{a}$ does not have.

In the fall back case, where a biased fitting method is needed, we follow the method of Allaire et al. [3]. Allaire et al. build their constraints out of basis-invariant quadratic functions of matrix $\mathbf{A}$ from the matrix form of the quadric (Eqn. 4.2) to guarantee that the constraints themselves are basis invariant:

$$q_\mathrm{Allaire}(\mathbf{A}) = \alpha \sum \det{}_2^P(\mathbf{A}) + \eta \mathrm{tr}(\mathbf{A})^2 \ , \tag{4.8}$$

where $\sum \det_2^P(\mathbf{A})$ is the sum of the three principal second-order minors of $\mathbf{A}$, and $\mathrm{tr}(\mathbf{A})$ is the trace of $\mathbf{A}$. This constraint, applied to a general quadric equation using the standard algebraic fitting framework (Sec. 4.1.2), guarantees the fitting solution will include at least one ellipsoid if $\alpha = 4, \eta = -1$, and guarantees the solution will include hyperboloids if $\alpha = 0, \eta = -1$. Solving with this constraint will result in a set of eigenvectors; we evaluate each eigenvector and take the best under Taubin's error metric with the desired quadric type. To efficiently determine if a quadric is an ellipsoid, we check that the second leading principal minor of $\mathbf{A}$ is positive, and that the first and third leading principal minors of $\mathbf{A}$ have the

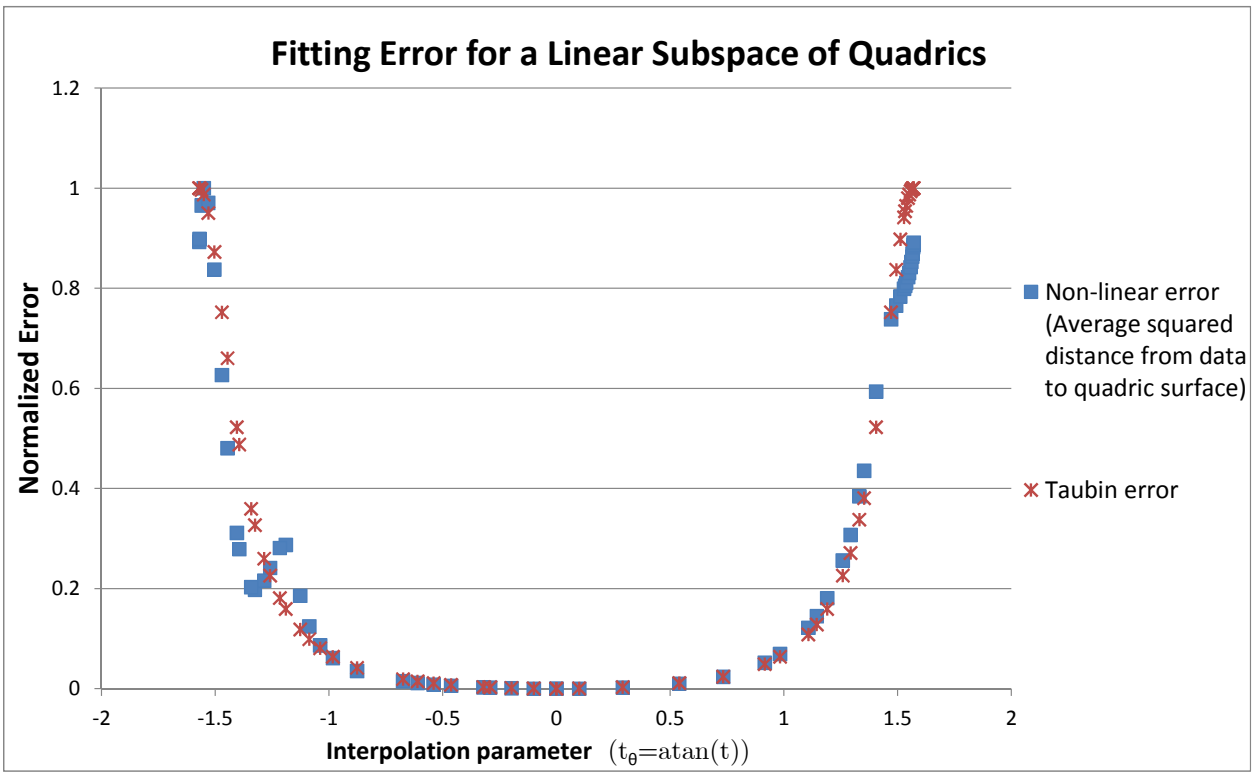


Figure 4.5: Fitting errors at samples of a linear subspace of quadrics formed from the best two eigenvalues of Taubin's method (visualized in Fig. 4.3 above). Interpolation parameter t has been transformed by $t_\theta = \text{atan(t)}$ to compress its range, and errors have been normalized by the error at the worst-fitting quadric. Taubin error (red crosses) approximates the true non-linear error (blue squares).

same sign. Note that previous work has suggested instead using the sign of the eigenvalue corresponding to each eigenvector to determine the quadric type [80], but (as noted by that work) doing so makes the fit unable to handle some shapes such as flattened ellipsoids. Our approach of directly testing the quadric type of each eigenvector avoids this limitation.

4.2.2 The Best Ellipsoid or Hyperboloid is Always a Paraboloid (if it isn't the Best Quadric Overall)

Assume for this discussion that the best quadric under Taubin's metric does not have the desired type: Otherwise, we would have simply returned the result of Taubin's method, without the need for any additional constraints or searches.

We now show that when constraints are needed, for example because the best fitting quadric is an ellipsoid and we want a hyperboloid, then the best quadric of the desired type (e.g. hyperboloid) must be right at the transition point between quadrics: at some

exact point t where the interpolated quadric becomes the quadric of the desired type. This transitional quadric is a paraboloid (or on the boundary of a paraboloid). To prove this, we first note that Harker et al. showed that the maxima and minima of Taubin's error for any linear subspace $\mathbf{c}_\mathrm{a} + t\mathbf{c}_\mathrm{b}$ can be found by solving the roots of a quadratic equation in the interpolation parameter t [55]. Because this error function is always non-negative, and quadratic equations have at most two roots, it can only have one minima and one maxima. Let $\mathbf{c}_\mathrm{a}^{'}$ and $\mathbf{c}_\mathrm{b}^{'}$ be the minima and maxima respectively, and consider the subspace $\mathbf{c}_\mathrm{a}^{'} + t\mathbf{c}_\mathrm{b}^{'}$. This is equivalent to searching the original subspace $\mathbf{c}_\mathrm{a} + t\mathbf{c}_\mathrm{b}$: Due to scale invariance, $\mathbf{c}_\mathrm{a} + t\mathbf{c}_\mathrm{b}$ spans the full planar subspace $s(\mathbf{c}_\mathrm{a} + t\mathbf{c}_\mathrm{b})$, and any two different quadrics from this planar subspace will span the same set of quadrics. In the limit as $t \to \infty$ or $t \to -\infty$ the interpolated quadric $\mathbf{c}_\mathrm{a}^{'} + t\mathbf{c}_\mathrm{b}^{'}$ becomes $\mathbf{c}_\mathrm{b}^{'}$, again due to scale invariance. Therefore the maximum error is at the limits $t \to \pm\infty$, and the error is monotonic in the ranges $t \in [0, +\infty)$ and $t \in [0, -\infty)$. When the minima does not have the desired type, it follows that within each monotonic range the best quadric of the desired type must be on the boundary of that type: If it were not, we could move t closer to 0 without changing the type and arrive at a quadric of the desired type with lower error, thanks to monotonicity. Therefore, the best quadric of the desired type overall is also on the boundary of that type – i.e., it is a paraboloid.

This result is not an artifact of our choice to restrict our search to a specific linear subspace of quadrics, since the reasoning applies to any linear subspace – including an optimal subspace spanning the true best ellipsoid and hyperboloid. It is therefore a fundamental property of Taubin's metric on quadrics that the best quadric of a specific type is either the best quadric of general type, or it is right on the boundary of the desired quadric type.

Note that the result is not specific to quadrics: Any linear subspace of parameters has at most one maxima and one minima under Taubin's error, regardless of the application, so the logic of this section also applies to any other cases where Taubin's error is used. For example, it follows for 2D conics that the best ellipse or hyperbola under Taubin's metric is a parabola (if it isn't the best conic overall).

4.2.3 Fitting Ellipsoids or Hyperboloids

If the desired type is returned by unconstrained Taubin fitting, then we just use that result. Otherwise, we know that the best quadric of the desired type is at a transition (Sec. 4.2.2), and we have selected a reasonable subspace of quadrics to search for that best quadric (Sec. 4.2.1). We now find all the transitional quadrics in our chosen subspace, and return the one with the desired type and lowest error.

To search the subspace $\mathbf{c}_t = \mathbf{c}_\mathrm{a} + t\mathbf{c}_\mathrm{b}$ for transitions between ellipsoids and hyperboloids, we refer to the matrix form of the quadric expression (2.2) to express the subspace of the quadratic coefficient matrices as an interpolated matrix $\mathbf{A}_t = \mathbf{A}_\mathrm{a} + t\mathbf{A}_\mathrm{b}$. We search for the points t where $\mathbf{A}_t$ could change its positive- or negative-definiteness. Because eigenvalues of any symmetric matrix are continuous with respect to the entries of the matrix [52], these are the points where one or more of the eigenvalues is 0; i.e., where the matrix determinant is 0.

To compute the determinant of $\mathbf{A}_t$ as a function of t, we express the determinant of $\mathbf{A}_t$ with column vectors $|\mathbf{A}_t| = \left|\mathbf{a}_1 + t\mathbf{b}_1 \quad \mathbf{a}_2 + t\mathbf{b}_2 \quad \mathbf{a}_3 + t\mathbf{b}_3\right|$ where $\mathbf{a}_i$ and $\mathbf{b}_i$ are the i^{th} column of $\mathbf{A}_\text{a}$ and $\mathbf{A}_\text{b}$, respectively. We use the linear independence of the columns to expand the determinant to a third degree polynomial:

$$\begin{aligned}|\mathbf{A}_t| = \left|\mathbf{a_1} \quad \mathbf{a_2} \quad \mathbf{a_3}\right| + \left(\left|\mathbf{b_1} \quad \mathbf{a_2} \quad \mathbf{a_3}\right| + \left|\mathbf{a_1} \quad \mathbf{b_2} \quad \mathbf{a_3}\right| + \left|\mathbf{a_1} \quad \mathbf{a_2} \quad \mathbf{b_3}\right|\right) t + \\ \left(\left|\mathbf{a_1} \quad \mathbf{b_2} \quad \mathbf{b_3}\right| + \left|\mathbf{b_1} \quad \mathbf{a_2} \quad \mathbf{b_3}\right| + \left|\mathbf{b_1} \quad \mathbf{b_2} \quad \mathbf{a_3}\right|\right) t^2 + \left|\mathbf{b_1} \quad \mathbf{b_2} \quad \mathbf{b_3}\right| t^3.\end{aligned} \tag{4.9}$$

The best quadric of the desired type is then (within ϵ of) one of the roots of this cubic polynomial. If we want a hyperboloid, then it is simply the root with the lowest error under Taubin's metric. For an ellipsoid, we also require all non-zero eigenvalues of $\mathbf{A}_t$ to have the same sign, to ensure it can be perturbed by ϵ to generate an ellipsoid.

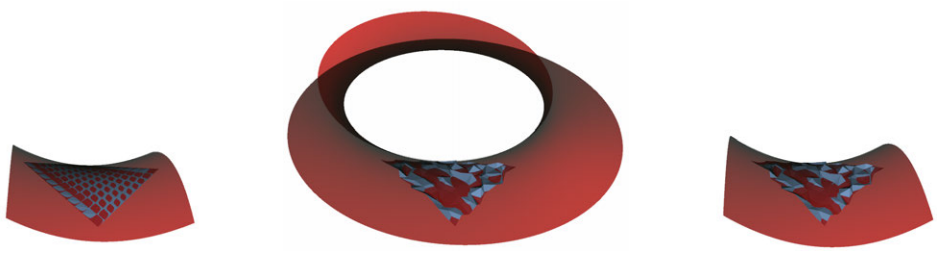
When fitting hyperboloids, we may also wish to constrain the number of sheets in the hyperboloid we fit. If the result of hyperboloid-specific fitting has the wrong number of sheets, then Sec. 4.2.2 shows that the best hyperboloid with the desired number of sheets must be at the boundary of the desired type. In this case, that means it must be (within ϵ of) either a paraboloid or a cone. Therefore, if the initial fit does not have the desired number of sheets, we separately fit a paraboloid (elliptical for two sheets, hyperbolic for one sheet; see Sec. 4.2.4) and cone (Sec. 4.3.3) to the data, and take whichever has the lowest Taubin error.

4.2.4 Fitting Paraboloids

Because the hyperboloid and ellipsoid fitting method above already finds well-fitting paraboloids, it seems natural to use the same approach for paraboloid-specific fitting. Any quadric is within ϵ of a paraboloid if its matrix of squared coefficients $\mathbf{A}$ is singular; i.e. if $|\mathbf{A}| = 0$. Therefore, the roots of Eqn. 4.9 are all the paraboloids in a linear subspace of quadrics. We use the linear subspace of quadrics formed by the best two eigenvectors of Taubin's method, and take the quadric corresponding to the root of Eqn. 4.9 with the lowest Taubin error as our paraboloid-specific fit.

To fit elliptical paraboloids specifically, we search the subspace including both ellipsoids and hyperboloids found in Sec. 4.2.1; this subspace must also include elliptical paraboloids because the quadric type on the boundary between hyperboloids and ellipsoids is the elliptical paraboloid (Fig. 4.1).

To fit hyperbolic paraboloids specifically, we first search the space of the best two eigenvectors, which appears to always include hyperbolic paraboloids in practice. However, since this is not a mathematical guarantee, in the event that no hyperbolic paraboloid is found, we suggest fitting a hyperbolic cylinder (Sec. 4.3.2) – the most general quadric type on the boundary of hyperbolic paraboloids (Fig. 4.1) – based on the intuition of Sec. 4.2.2. An example of hyperbolic paraboloid fitting is shown in Fig. 4.6.



a b c

Figure 4.6: Data from a hyperbolic paraboloid (blue) fit with a quadric (red). (a) Noise-free data. (b and c) Data with Gaussian noise ($\sigma = 2\%$ of bounding box size). (b) General quadric fitting finds a hyperboloid. (c) Paraboloid-specific fitting finds a hyperbolic paraboloid.

4.3 Fitting methods for lower-dimensional quadric types

Ellipsoids and hyperboloids exist in a high-dimensional parameter space with 9 degrees of freedom (having 10 unconstrained, but scale-invariant, parameters). The remaining quadric types all exist in simpler sub-spaces with fewer degrees of freedom [103]. The key to efficiently fitting these quadric types is to express the quadric with fewer parameters, such that only quadrics of the specified type can be generated, and then apply the standard algebraic fitting procedures of Sec. 4.1.2 on that parametrically-reduced form. For example, for planes and spheres we can simply drop and combine terms from the standard implicit quadric function to arrive at a plane- or sphere-specific function (e.g. $c_0+c_1p_x+c_2p_y+c_3p_z+c_4(p_x^2+p_y^2+p_z^2) = 0$ for spheres).

For most other lower-dimensional quadrics, the required low-dimensional space is more complicated: For example, there is no known linear least-squares method to fit circular cones and cylinders to a point cloud using just point positions [103]. However, there are linear least-squares methods for fitting such shapes to point clouds with normals [86]. For dense point clouds and polygonal meshes we can estimate normals (e.g. by local plane fitting, or averaging triangle normals), and then use a two-step process to fit the quadric. First, we estimate key parameters of the quadric using a direct "kinematic surface fitting" procedure that can determine properties such as a rotation symmetry axis (for a rotationally symmetric quadric), the direction in which the shape does not change (the axis of a cylindrical shape), or the central point of scaling (for a general cone) (Chapter 2). Second, we transform the data to a more convenient space and perform the standard algebraic fit in that space. In the transformed space, it is possible to reduce the quadric parameters as we did for planes and spheres. For example, to fit general cones we translate the data so that the cone apex is at the origin, and then fit a quadric with the linear and constant parameter terms dropped: $c_4p_x^2 + c_5p_y^2 + c_6p_z^2 + c_7p_xp_y + c_8p_xp_z + c_9p_yp_z = 0$ (Sec. 4.3.3).

4.3.1 Kinematic field fitting

Kinematic surface fitting is another type of fitting procedure, typically aimed at reconstructing surfaces of revolution, general cylinders, or other shapes that can be generated by sweeping a general profile curve along a simple, linear velocity field. We explain kinematic surface fitting in greater detail in Chapter 2, where we use it to fit stationary sweeps. It is performed in two stages: First, fitting a "kinematic field" that defines the parameters of the sweep motion, and second, using that field to transform the points into a shared plane in which the profile curve can be fit. For quadric fitting, we focus on the first stage – fitting a motion field – to find some property of the desired quadric type: the axis of invariance for the cylinder, or the axis of rotation for a spheroid, or the central point of the scaling for a general cone. We can then use that to transform the data to a space where the type-specific quadric fitting problem is easier to specify.

Kinematic motion fields are linear functions $\mathbf{v}(\mathbf{p})$ that define a velocity field everywhere in space. For our quadric fitting catalog we use three specific field types (Fig. 4.7): first, the simple general cylinder field due to [115], which defines a constant motion field:

$$\mathbf{v}(\mathbf{p}) = \mathbf{a}, \tag{4.10}$$

where $\mathbf{a}$ is a parameter vector defining the direction of the motion field. Second, we use the rotational field due to [108], which defines a rotational (or helical) motion:

$$\mathbf{v}(\mathbf{p}) = \mathbf{r} \times \mathbf{p} + \mathbf{a}, \tag{4.11}$$

where $\mathbf{r}$ and $\mathbf{a}$ are parameter vectors encoding the rotation axis and position of the axis (plus helical motion, if any). And finally, we use a scaling field, which defines a scaling motion:

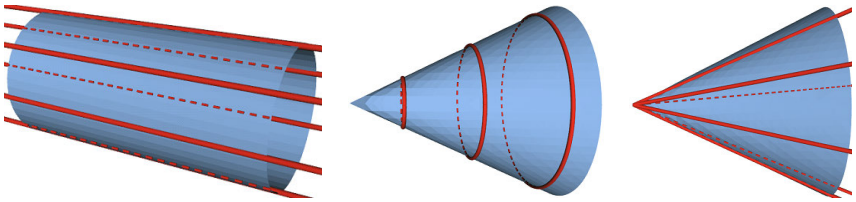
$$\mathbf{v}(\mathbf{p}) = \gamma\mathbf{p} + \mathbf{a}, \tag{4.12}$$

where γ is a parameter defining the scaling and $\mathbf{a}$ is a parameter vector encoding the center of the scaling.

To fit a field, we need data points that include both position and normal information. (Normals, if unavailable, can be approximated for dense point clouds [15]; for meshes we use area-weighted vertex normals.) Data points fit the field if they define a surface that is tangent to the field, which we characterize by the error function $f(\mathbf{p}, \mathbf{n}) = \mathbf{v}(\mathbf{p}) \cdot \mathbf{n}$. In Chapter 2, we show that an effective direct solution to this fitting problem for any field is to apply Taubin's method (Sec. 4.1.2), with $\sum_{i=0}^{n} w_i (\mathbf{v}(\mathbf{p}_i) \cdot \mathbf{n}_i)^2 \equiv \mathbf{c^T M c}$ and $q(\mathbf{c}) = \sum_{i=0}^{n} w_i ||\mathbf{v}(\mathbf{p}_i)||^2 \equiv \mathbf{c^T N c}$, where $\mathbf{c}$ is a vector concatenating all parameters defining the motion field (e.g. for scaling, $\mathbf{c} = [\gamma, a_x, a_y, a_z]^T$).

4.3.2 Fitting Cylinders

Cylinders of any subtype can be fit in two steps: First, we fit a general cylinder field to find the invariant axis of the cylinder, along which the cross section does not change. Next, we



a b c

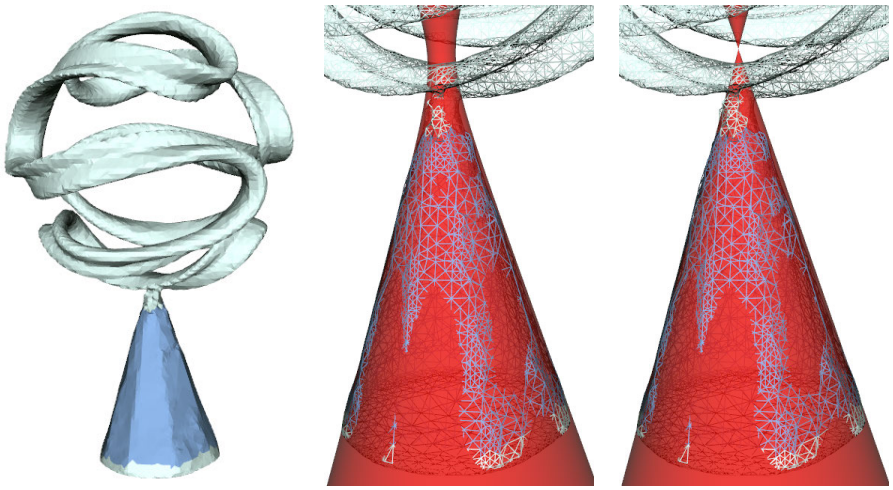
Figure 4.7: Red streamlines showing motion fields of (a) Eqn. 4.10, (b) Eqn. 4.11, and (c) Eqn. 4.12, fit to mesh data.

rotate all points so that this axis is aligned with the z-axis. We then fit a conic of the desired type to the points: i.e., for a circular cylinder, we fit a circle; for an elliptical cylinder, we fit an ellipse. For circle fitting, we fit the equation $c_0 + c_1 p_x + c_2 p_y + c_4(p_x^2 + p_y^2) = 0$ directly using Taubin's method. For other types, we use the methods from Sec. 4.2 (adapted from Harker et al. [55]) for type-specific conic fitting, as explained below. The resulting conic equation, evaluated in 3D, is a cylinder along the z-axis. We finally rotate the cylinder back to the original basis.

In full detail, the type-specific conic fitting method adapted from Harker et al. [55] and Sec. 4.2 proceeds as follows. First, we fit using Taubin's method, and check the type of the best fit result. If it is the correct type, we return that and stop. Otherwise, we use biased fitting to find some (sub-optimal) conic of the desired type; specifically, we use the constraint function $q_{\text{Fitzgibbon}}(\mathbf{c}) = 4c_4c_5 - c_7^2$ [39]. We search the subspace between the best fit conic using Taubin's method, $\mathbf{c}_\text{a}$, and the biased fit of the desired type, $\mathbf{c}_\text{b}$, for an improved fit of the desired type. Now we define $\mathbf{A}_\text{a}$ and $\mathbf{A}_\text{b}$ as 2×2 matrices of quadratic terms terms, of the form $\mathbf{A} = \begin{pmatrix} c_4 & c_7/2 \\ c_7/2 & c_5 \end{pmatrix}$ (analogous to the 3×3 of Eqn. (2.2)), and define $\mathbf{A}_t \equiv \mathbf{A}_\text{a} + t\mathbf{A}_\text{b}$. Sec. 3.2 shows that a well-fitting conic of the desired type should be at the transitional parabola where the conic changes type. This is a point where $|\mathbf{A}_t| = 0$. With $\mathbf{a}_i$ and $\mathbf{b}_i$ defining the i^{th} columns of $\mathbf{A}_\text{a}$ and $\mathbf{A}_\text{b}$ respectively, we solve for the roots $|\mathbf{A}_t| = \left|\mathbf{a}_1 \quad \mathbf{a}_2\right| + \left(\left|\mathbf{b}_1 \quad \mathbf{a}_2\right| + \left|\mathbf{a}_1 \quad \mathbf{b}_2\right|\right) t + \left|\mathbf{b}_1 \quad \mathbf{b}_2\right| t^2 = 0$, and take the root with lowest Taubin error.

4.3.3 Fitting General Cones

To fit a cone, we first find the center point of scaling by fitting the kinematic scaling field $\mathbf{v}(\mathbf{p}) = \gamma\mathbf{p} + \mathbf{a}$. The center of scaling is $-\mathbf{a}/\gamma$. Note that if $|\gamma|$ is very small, then division by γ becomes unstable. In the limit as γ approaches zero, the center of scaling moves infinitely far from the origin, so the cone becomes a cylinder. We therefore define a threshold $t_\text{cyl} = 10^{-6}$, and if $|\gamma| < t_\text{cyl}$, we consider the best cone to be a cylinder. If a cylinder is an acceptable result of cone-fitting, then we can fit an elliptical cylinder instead of a cone (see Sec. 4.3.2). Alternatively, if a cylinder is not an acceptable result, we can instead take the



a b c

Figure 4.8: An image-based reconstruction of a sculpture has a noisy cone as its base (a). We select the base (blue) and fit quadrics (red). (b) Without type specific-fitting, the result is a hyperboloid with a non-zero neck at the top. (c) With cone-specific fitting, the result is an exact cone.

best eigenvector returned by algebraic fitting with $|\gamma| > t_{\text{cyl}}$.

Once some (non-infinite) cone center is chosen, we can translate the points so that the cone center is at the origin, and fit a centered cone quadric equation $c_4 p_x^2 + c_5 p_y^2 + c_6 p_z^2 + c_7 p_x p_y + c_8 p_x p_z + c_9 p_y p_z = 0$ using Taubin's method. Finally, we translate the resulting quadric back to the original space.

4.3.4 Fitting Rotationally Symmetric Quadrics

To fit a rotationally symmetric quadric, we first find the axis of rotational symmetry by fitting the rotational field $\mathbf{v}(\mathbf{p}) = \mathbf{r} \times \mathbf{p} + \mathbf{a}$. We normalize the resulting parameters by dividing both $\mathbf{a}$ and $\mathbf{r}$ by $||\mathbf{r}||$, and then find a point on the axis of the rotational field $\mathbf{p}_r = \mathbf{r} \times (\mathbf{a} - (\mathbf{a} \cdot \mathbf{r})\mathbf{r})$. If $||\mathbf{r}||$ is very small, then normalizing by $||\mathbf{r}||$ is unstable. When $||\mathbf{r}|| = 0$, the field becomes a pure translation and therefore has no rotation axis. To handle small values of $||\mathbf{r}||$, we use a threshold $t_{\text{cyl}} = 10^{-6}$: If $||\mathbf{r} < t_{\text{cyl}}||$ then we assume the best motion is approximately a pure translation, i.e. a cylinder, so we fit a circular cylinder (see Sec. 4.3.2). If a cylinder is not desired, we can alternatively take the best eigenvector with $||\mathbf{r}|| > t_{\text{cyl}}$. Once the axis of rotation is known, we transform the data so that the z-axis is aligned with the rotation axis, and the point $\mathbf{p}_r$ is at the origin. We then fit the equation $c_0 + c_3 p_z + c_4(p_x^2 + p_y^2) + c_6 p_z^2 = 0$ for a general quadric with rotational symmetry, and finally transform the result back to the original basis. We can directly apply the methods of Section 3 to further constrain the quadric type.

For circular cones we either find the center of scaling and make that $\mathbf{p}_r$, then fit the

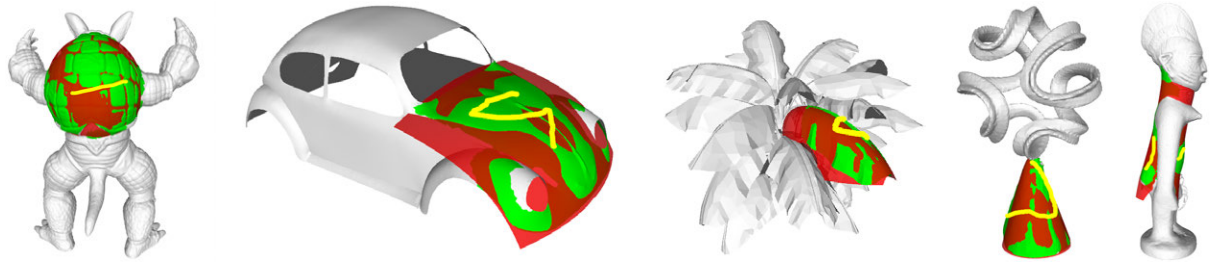


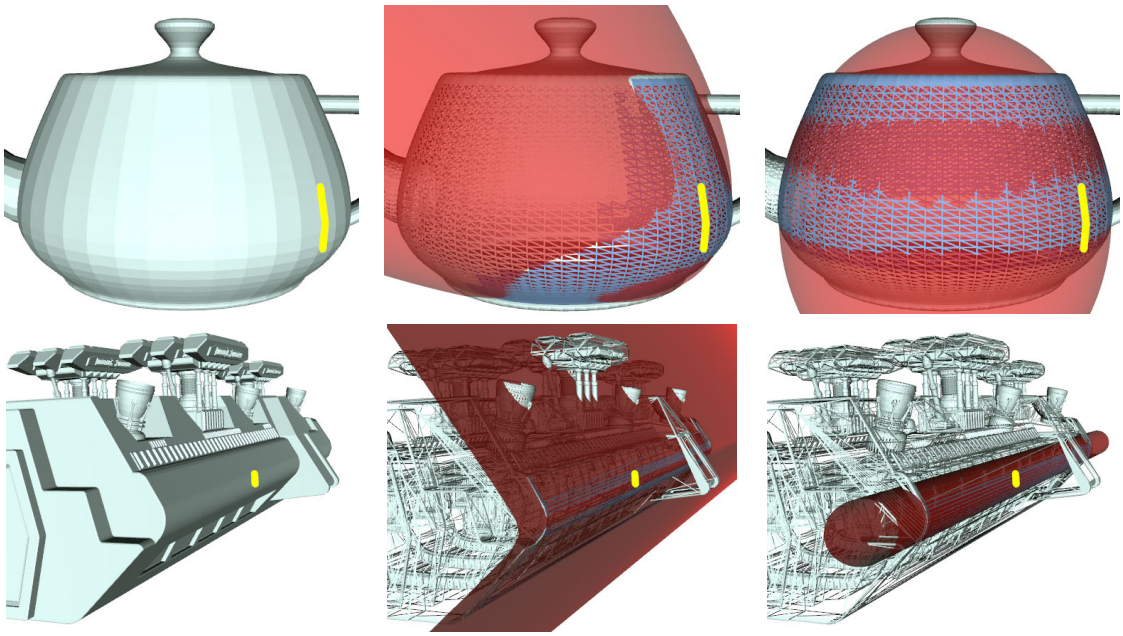
Figure 4.9: Strokes (in yellow) are used to initialize various quadric fits on a variety of shapes. The quadric is shown in red, and the given input triangles are shown in green.

simpler equation $c_4(p_x^2 + p_y^2) + c_6 p_z^2 = 0$, or we can alternatively rotate the data points to a shared plane and fit a 2D line to find the angle of the cone. Note that quadric cones are in fact two cones connected at the tip, and this second method ignores the possibility that the data points may come from both cones; but it may be desirable in the common case where only one of the two cones is desired in the fit.

For spheres we directly fit the equation $c_0 + c_1 p_x + c_2 p_y + c_3 p_z + c_4(p_x^2 + p_y^2 + p_z^2) = 0$ [2]. For planes, we translate the data to its centroid and fit the equation $c_1 p_x + c_2 p_y + c_3 p_z = 0$ [101]. We handle double planes and intersecting planes by just applying multiple plane fits, because we always prefer single-plane solutions.

4.4 User-Guided Segmentation and Fitting

We use a simple, user-guided system to allow the user to select and fit a quadric surface. The method is essentially the same as the stationary sweep selection method in Chapter 2. The user makes a stroke on the display to designate a surface region for extraction, and presses the "select quadric" button. The data under the stroke is selected, and then the system iteratively (a) fits a quadric to the selected data points, and (b) locally grows the selection to any neighboring data points that are as close to the fitted quadric as the originally-selected data points are. For our flood-fill process, we use two metrics to test whether a given input point and normal is a match with the current quadric surface: the distance to the quadric surface, and how parallel surface normal is to the gradient of the quadric: $|\frac{\nabla f(\mathbf{p})}{||\nabla f(\mathbf{p})||} \cdot \mathbf{n}| - 1$. We only grow the selection to a new point if it is closer than a threshold to the quadric by both metrics; we set this threshold as the maximum distance of the user-selected points to the fitted quadric. This simple method provides a quick, effective method for selecting quadrics on many different surfaces; we show a number of results using general quadric fitting in Fig. 4.9. As with the user interface in Chapter 2, the user can make additional strokes to expand the selection, and these additional strokes also allow the user to fit disconnected parts of the surface with the same quadric. Further refinement of the selection is possible using the general sketch-based segmentation techniques discussed in Chapter 6.



a b c

Figure 4.10: (a) A user has made a small stroke (yellow) on a mesh surface (top row: teapot; bottom row: moonbus), to select a quadric surface under the stroke using our simple region growing method. In (b) and (c), we show the fit quadric (red) and wireframe views of the mesh (selection in blue); (b) shows the result of general quadric fitting; (c) shows the result when a preference for rotationally symmetric quadrics is added.

Quadric type-preferences expressed by the user, in addition to guaranteeing the quadric type, have the additional benefit that they can help disambiguate the quadric selection process. Our convenient, simple selection method faces inherent ambiguity both in the best directions to grow the selection (Fig. 4.10, top row) and how much error should be accepted in order to fit a larger region (Fig. 4.10, bottom row). A user's preference for a quadric of a specific type or with a specific property, like rotational symmetry, can reduce that ambiguity (Fig. 4.10c).

Preferences for quadric types can either be absolute, or heuristic: If the user's type-preference is absolute, we simply replace general quadric fitting with the appropriate type-specific fitting method. Otherwise, the user can specify how much additional error they will accept in exchange for a simpler quadric type; the system will fit both the simpler type and the general type, and use the simpler fit only if its error is within the user's additional error threshold of the general fit.

4.5 Implementation Details

Each direct fitting method we describe involves simply solving generalized eigenvalue problems, and in some cases finding the roots of a cubic or quadratic equation. Generalized eigenvalue problems can be solved with standard linear algebra packages (we use LAPACK [4]), but some care must be taken to ensure good results.

First, care must be taken to avoid numerical instability. We observed some cases in which floating point error resulted in an incorrect ordering of the eigenvalues of a generalized eigenvector problem $\mathbf{c^T Mc} = \lambda \mathbf{c^T Nc}$ for clean data with low noise. To ensure that the best eigenvector is chosen, we disregard the computed eigenvalues and instead directly compute Taubin's error metric for each eigenvector. We then choose the eigenvector with lowest error (ignoring eigenvectors of the wrong quadric type, if the constraint matrix N was intended to constrain the quadric type). We also use double precision floating point numbers for computation, and center and re-scale the data points (to a unit-sized bounding box) before fitting.

Another important implementation detail is the method of sampling of the original surface. When fitting segments of a polyhedral mesh, we have two reasonable options: We could sample the error at vertices, or we could integrate the error over the surface of the mesh. Sampling error at vertices makes it easier to smooth noise implicitly, for example by using averaged vertex normals. It also leads to more intuitive results if the vertices represent true samples from an original surface, which is often the case. However, integrating the error over the surface of the mesh can greatly reduce ambiguity in the case of sparse, non-uniform sampling, as shown in Fig. 4.11. In our tests, we found that the reduced ambiguity can greatly improve robustness when fitting implicit quadric equations.

For ease of implementation, we perform integration over a polyhedral mesh by trian-

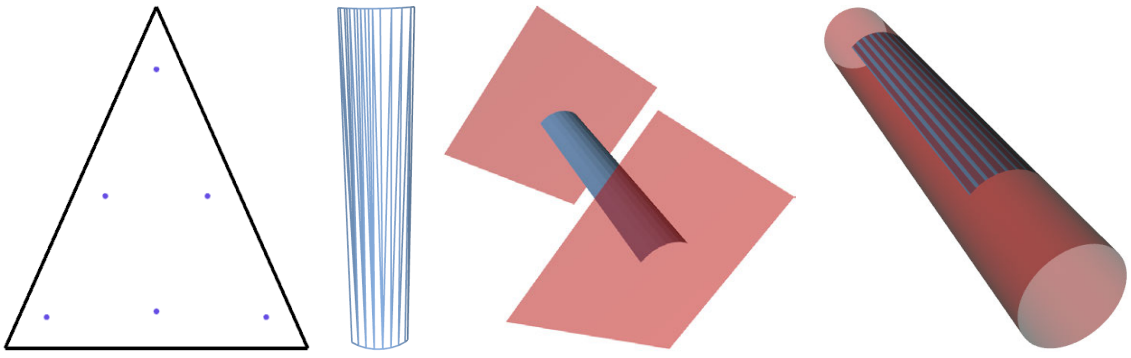


Figure 4.11: (a) Dunavant's integration points. (b) A wireframe view of a mesh with sparse, non-uniformly distributed vertices. (c) Fitting result when error is measured only at vertices: A double-plane quadric that exactly interpolates the vertices. (d) Fitting result when error is integrated over mesh surface.

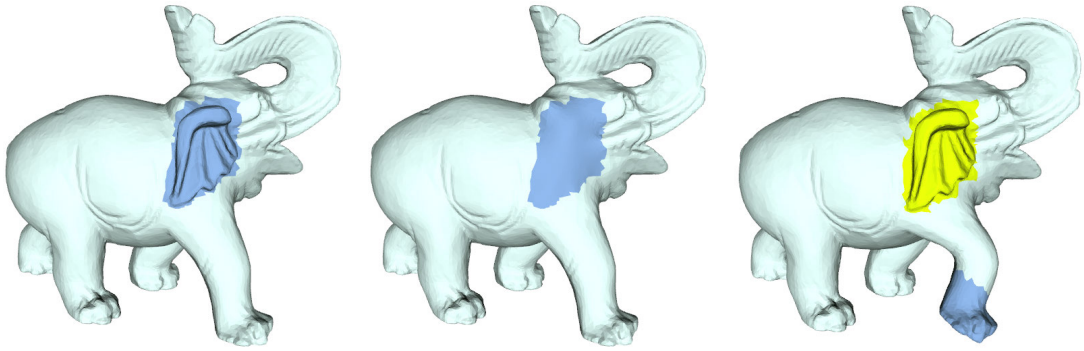
gulating the mesh and using Dunavant's Gaussian quadrature rules [31]. This is simple to implement because it is exactly like fitting with discrete points: We just generate six sample points per triangle (at barycentric co-ordinates given by Dunavant) and weight each point according to Dunavant's rules. Because the error function (a squared quadratic) is a quartic polynomial in terms of position, Dunavant's six-point rule correctly integrates the error without approximation.

Chapter 5

Smooth Surface Primitives

Smooth surface primitives are very general, and can fit any surface. In our framework, smooth surfaces primitives serve three main purposes: (1) to surpress detail in a user-selected area (Fig. 5.1b), (2) to create smooth blends between parts (see Chapter 10), and finally (3) with detail-preservation enabled, to act as a generic default primitive. The third mode, as a generic default primitive fit to the unselected portions of the surface, allows the user to edit other parts of the shape while maintaining continuity across the detail-preserving smooth surface, as shown in Figs. 5.1c and 5.4.

All our smooth surface operations can be done with existing methods [12, 132]. In this chapter, we review the existing smooth surface primitive methods, and explain the trade-offs between these methods in the context of our system. We detail the method we chose to use [12] for our prototype system, and present results of this method in the context of our framework.



a b c

Figure 5.1: (a) The elephant's ear region is selected; (b) when optimized as a thin-plate spline, all surface details get smoothed away. (c) The ear is fixed in place as a constraint, and the foot marked by the blue patch is moved; continuity between the blue and yellow patches and the rest of the shape (gray) is preserved by Laplacian-preserving surface editing.

5.1 Background

There are many types of smooth surface primitive: for example, NURBS patches, subdivision surfaces, and PDE surfaces [141, 12]. We can categorize these methods based on a few important distinctions: (a) Linear vs. non-linear surfaces; (b) methods where the result is strongly tied to a mesh/network topology, or independent of it; and finally (c) methods that preserve surface detail by storing displacements, or by storing surface curvatures or Laplacians. In this section, we explain the tradeoffs inherent in these distinctions, and why these tradeoffs lead us to favor mesh-based PDE surfaces for our system.

5.1.1 Linear vs. Non-Linear Surfaces

Linear smooth surfaces define a smooth surface as a linear combination of some basis functions, so that the position of any point on the final smooth surface can be expressed as a linear combination of some control points. The weights may be determined by a polynomial (for linear spline patches), recursive subdivision rules (for subdivision surfaces), or a linear system (for PDE surfaces). Weights may be defined over the surface, generally over space [30], or in reference to some enclosing "cage" surface [122, 64]. These weights can be computed quickly, and provide predictable control: Control point movements always move the surface in the same direction as the control points, and no further than the control points move. However, the resulting surface may not always be smooth: The surface can have sharp creases in it. For example, if the control points of a 3D shape all lie in a flat plane, then any linear combination of those points (and thus any point on the smooth surface) will also lie in that flat plane; for any shape with a front and back (e.g., for a sphere, but not a disc with open boundary), this will cause a sharp, "fold over" crease at the transition between the front and back of the shape [97].

Non-linear smooth surfaces typically define a surface by means of a geometric "functional" that is minimized by the surface: For example, the surface may be defined to minimize bending energy, or variation of curvature, or any of a number of other options [65]. In contrast to linear smooth surfaces, this can guarantee that the surface is smooth everywhere. However, these surfaces are typically much more expensive to compute, and thus very difficult to include in an interactive system. Furthermore, they can be unpredictable to control – in the worst case, the surface can potentially diverge to an infinitely large surface. We are not aware of an interactive, non-linear smooth surface solver that always converges: We found that the only solver that works at interactive rates, "FiberMesh" [96], fails to converge on a number of simple examples shown in Fig. 5.2. For any user-guided system, we believe stable behavior and fast performance are extremely important, and non-linear smooth surface solvers are still not quite up to the task.

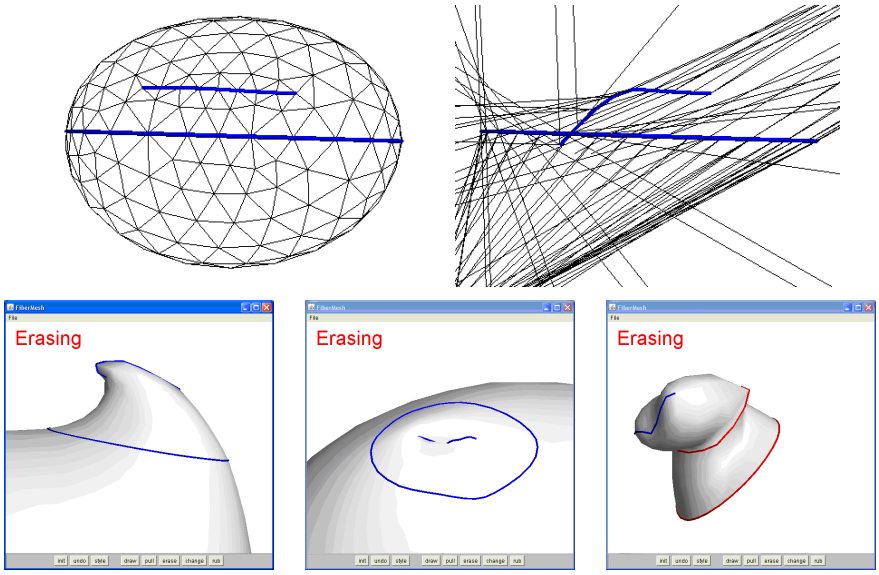


Figure 5.2: Non-linear formulations for smooth surfaces are very difficult to make fast and stable. We show simple cases where a state-of-the-art non-linear smooth surface method, FiberMesh [96], fails to converge in a number of simple configurations. In this figure, the thick blue or red curves are constraint curves: The FiberMesh surface interpolates these curves, and users edit the surface by adding or moving these curves. The top row shows an example where horizontally dragging a constraint curve drawn on an ellipsoid causes the surface to diverge to infinity. The bottom row shows configurations in which the FiberMesh system produces an oscillating surface that never converges.

5.1.2 Topology-Dependence

Linear smooth surfaces based on spline or subdivision surfaces are defined by a patch network (for splines), or base mesh (for subdivision surfaces), that may need to be fairly dense to accurately capture a complex shape. The vertices of this network or mesh are then the control points used to edit the smooth surface. The exact structure of this network or mesh affects the quality of the resulting surface, and the degrees of freedom available to edit that surface. Automatic systems for reverse engineering a shape to subdivision or B-Spline surfaces [60, 32, 84] tend to give networks that are not ideal for shape redesign – either because they do not provide the desired degrees of freedom, or because they are overly dense and therefore tedious to edit. Krishnamurthy and Levoy [75] developed an artist-centered system for fitting a B-Spline network to a dense triangle mesh, and found that the artists wanted to completely control the spline network layout by specifying it up-front – a task that requires substantial user interaction and expertise.

PDE surfaces on meshes [12], as well non-linear smooth surface methods [96], and methods that define deformations over space [30] or in reference to a separate control cage [122], in contrast, can separate the degrees of freedom of the surface from the control points used to

edit the surface: One can increase the degrees of freedom by increasing the mesh resolution independently of the number of control points exposed to the user. Furthermore, the user has the ability to pick sparse, arbitrary control points based on their intent, without needing to form any coherent mesh topology connecting those control points. This approach completely avoids the need for any reverse-engineering system to guess the degrees of freedom and control points desired by the user. These advantages come at a higher computational cost, and less accuracy: Spline and subdivision surfaces can be evaluated quickly and exactly at any point, while mesh-based PDE surfaces are evaluated by the finite element method, which requires the solution of a large sparse matrix, and is an approximation. Non-linear smooth surfaces are likewise typically found by an optimization algorithm that relies on some approximation.

For our prototype system, we prefer the simplicity of a topology-independent approach. Mesh-based PDE surfaces, while slower to evaluate than traditional spline surfaces, are still typically fast enough for interactive use. The primary weakness of this approach is accuracy: We were not looking in depth at applications where exact solutions are critical, but if we were, it likely would have been worth exploring a subdivision surface approach as well.

5.1.3 Detail-Preservation Methods

Space warp approaches like free-form deformation [122], which define smooth deformations over space, automatically preserve detail. In contrast, surface-based approaches often decompose a surface into a smooth part and a separate encoding of surface detail. There are two common, high-level approaches to preserving fine-scale details alongside a smooth surface representation: One can store a displacement map that records details as local offsets in the direction of the surface normal, or one can store a map of local bending (or Laplacians) of the surface. Of surface-based approaches, the displacement map approach is perhaps conceptually simpler, and very easy to compute on the fly (one can even implement it on the GPU as a shader). However, it is more limited in the kind of detail it can preserve: The detail must effectively be a "height field" with respect to the smooth surface, i.e., a displacement map cannot faithfully preserve details when the angle between the smooth surface and the corresponding detail surface exceeds 90 degrees. Methods that preserve local bending of a surface are more general in the type of detail they can preserve.

Preserving "local bending" or a surface Laplacian can be done in several ways. The easiest way, "Laplacian surface editing" [132], is to preserve the discrete Laplacian vector at each vertex of a mesh. This method is relatively easy to compute – only requiring the solution to a sparse linear system – but it preserves details with respect to an absolute, not surface-relative, orientation, which leads to unintuitive results when the control points are rotated. Preserving the Laplacians in a surface-relative orientation is harder to get right, and doing so without artifacts turns out to be fundamentally a non-linear problem [13]. One simple, iterative approach is the "as-rigid-as-possible" surface deformation method, which iteratively (a) solves for the surface with Laplacian vectors in an absolute orientation, and (b) finds a locally-rigid rotation to re-orient the Laplacian vectors [131]. Unlike non-linear smooth surfaces, this iterative solution is proven to always converge towards the correct

solution – and though it may take an arbitrarily large number of steps to do so in theory, in practice researchers have reported that these systems converge fast enough to be interactive for surfaces with tens of thousands of vertices [14]. For our prototype system, for simplicity we just use the basic Laplacian surface editing method, but a more complete system should ultimately use a non-linear method to correct the Laplacian orientations.

There is one inherent limitation for both of these approaches: The smooth surface must topologically match the detailed one, without any additional holes or tunnels. To preserve the details of a completely arbitrary surface, while deforming the surface analogously to a smooth surface, one needs to establish a more global smooth mapping between the smooth surface and nearby space. One practical method to do this has been described in detail by Peng et al. [102].

5.2 Implementation

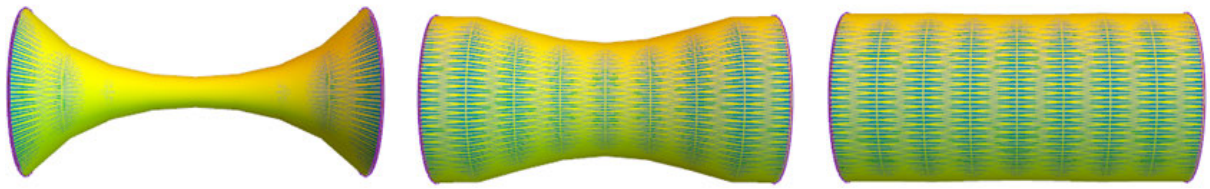
Because our prototype system is focused on surface mesh inputs, the mesh-based PDE surface primitive of [12] was a natural fit. This method is implemented as follows: Given some selection of the mesh that we wish to treat as a smooth surface, we create the desired smooth surface by triangulating the selection and solving for the surface where each free vertex satisfies $\Delta^k(\mathbf{p}) = 0$ for k typically equal to 2 or 3, where Δ is a discrete Laplacian operator. Specifically, the higher-order Laplacian is defined recursively at a mesh vertex as

$$\Delta^k(\mathbf{u}) = \sum_i w_i(\Delta^{k-1}(\mathbf{u}) - \Delta^{k-1}(\mathbf{v_i})) \ , \tag{5.1}$$

where $\mathbf{v_i}$ are the one-ring neighbors of vertex $\mathbf{u}$, $\Delta^0(\mathbf{u}) = \mathbf{u}$ and w_i are the cotangent weights [104] scaled by the inverse Voronoi area the vertex (as defined by Meyer et al. [92]). Cotangent weights are computed in a fixed initial domain and held constant to linearize the system – resulting in a surface that smoothly interpolates positions of constrained or boundary vertices, using the original mesh as a base domain. (Avoiding this linearization would result in a non-linear surface with the performance and convergence problems discussed in Sec. 5.1.1.) By formulating this equation for all unconstrained vertices in a mesh, we obtain a sparse, symmetric positive-definite linear system of the form $\mathbf{Ax} = \mathbf{b}$, which we factor and solve with a sparse direct Cholesky method [21].

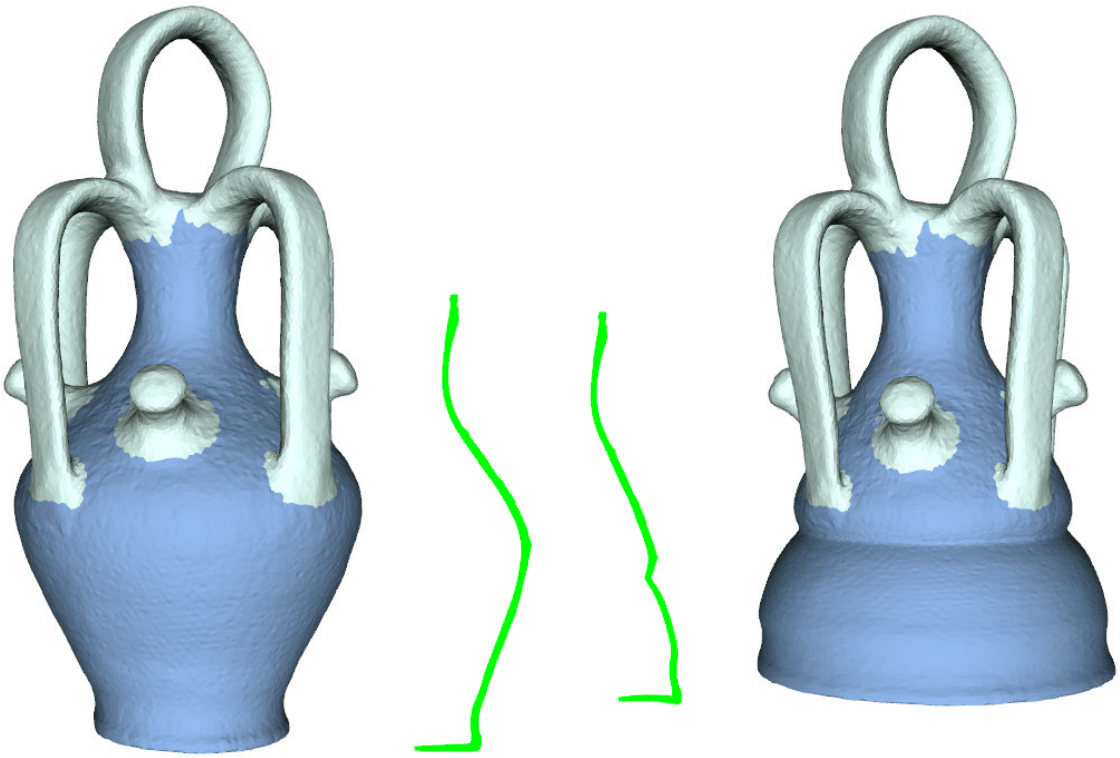
To preserve detail, we preserve the surface Laplacians [132]. For simplicity, in our current implementation we do not rotate the Laplacian vector, so we can preserve Laplacians with a simple extension of the mesh-based PDE surface: Instead of solving for the surface where $\Delta^k(\mathbf{p}) = 0$, we solve for the surface where $\Delta^k(\mathbf{p}) = \Delta^k(\mathbf{p}_{\text{orig}})$, where $\Delta^k(\mathbf{p}_{\text{orig}})$ is the k^{th}-order Laplacian of the original, detailed surface.

The order of the Laplacian, k, trades off two considerations: (1) As k increases, the linear system defining the surface becomes more dense and has a larger condition number, thus requires more time to solve and is more likely to face issues with numerical instability, but (2) for smaller k, the surface is less smooth at constraint boundaries: The surface will



a b c

Figure 5.3: A surface that was originally a cylinder is solved as a smooth surface without detail preservation, with Laplacian order (a) $k = 1$, (b) $k = 2$, and (c) $k = 3$. Laplacian order $k = 1$ results in a linearized minimum-area surface, which cannot guarantee continuity with boundaries and tends to collapse tubes; Laplacian order $k = 2$ is a linearization of minimizing bending energy, which still tends to collapse tubes slightly; Laplacian order $k = 3$ is a linearization of minimizing curvature variation, and tends not to collapse tubes.



a b c d

Figure 5.4: (a) A botijo model is approximated as a surface of revolution; (b) the corresponding profile; (c) edited profile, and (d) the resulting shape. The handles, modeled as detail-preserving smooth surfaces, move with the changes in the body.

maintain C^{k-1} smoothness at boundaries, so for $k = 0$ there may be sharp creases at the boundary. In addition, surfaces with $k < 3$ tend to collapse tube-like shapes, as we illustrate in Fig. 5.3. This effect is mitigated somewhat by detail-preservation: The bending necessary to keep the tube straight will be preserved in the Laplacian detail vector, ensuring the tube does not collapse.

When detail preservation is not used, our smooth surface solver can be used to suppress details on the input shape. For example, in Fig. 5.1a, the user selects the portion of the mesh that covers the elephant's ear (blue). The user clicks the "smooth surface" button, and our system re-models the ear with our smooth surface solver, removing all the details and resulting in Fig. 5.1b.

When detail-preservation is enabled, the smooth surface solver allows us to edit parts of a shape while automatically maintaining continuity with adjacent parts. Fig. 5.1c shows how this capability can be used to preserve continuity and smoothness in the right fore-leg of the elephant when the foot (marked by the blue patch) is moved, while the yellow surface serves as an anchor. This is especially useful in combination with other primitive-editing methods: For example, in Fig. 5.4 the user edits the body of a Botijo model using the stationary sweep module, and the handles are automatically updated by our system using a detail-preserving smooth surface, so that their connection points to the body are maintained.

Chapter 6

General User-Guided Segmentation

In Chapters 2 to 4, we discussed segmentation in the context of finding selections of a surface that fit specific primitive types. In this chapter, we discuss the general user-guided segmentation problem, to allow us to tackle cases in which the segmentation is not closely tied to a specific modeling primitive. This is useful when a user wants to edit a surface with a primitive type that does not fit the surface closely, as in Fig. 2.15, or when the user wants to use the smooth surface solver to smooth or move an arbitrary part of the surface, as in Fig. 5.1.

Efficient methods for user-guided segmentation have been studied primarily in the context of "mesh cutting" – selecting parts of a mesh to pull off and re-attach to another surface, resulting in novel shapes [45, 37]. Rather than focus on primitive fitting, typically these methods aim to segment "natural" parts of a model by aligning segment boundaries with mesh features such as creases on the mesh surface. In this chapter, we give a brief overview of recent work in this field, and highlight an existing method [63] that we believe is a good match for user-guided inverse 3D modeling.

6.1 Overview of Methods

We categorize methods for user-guided segmentation by the method of user interaction. To evaluate the trade-offs between these methods, we rely on a pair of user studies [89, 37] evaluating best-of-class algorithms for each interaction type on a number of metrics, including the speed at which users can perform segmentation, the accuracy with which they perform segmentation, and the users' own senses of how easy the methods are to use. We use these study results as a guide to evaluate the different segmentation methods in the context of our own inverse 3D modeling system, and in doing so identify a technique well-suited to our system.

The method of user interaction in all these systems is that users either (a) mark what is inside and (optionally) outside the desired segment [63, 36], or (b) mark where the segmentation boundary should be placed [154, 90, 155]. Methods for marking the boundary include

stroking along the boundary [90], across the boundary [154], or simply clicking a point on the boundary [155]. A recent user study comparing these approaches [37] found that marking the boundary was faster, and (when stroking along the boundary) perceived as easier to use, but less accurate than marking inside (and outside) the segment by a wide margin – the boundary markings gave a reasonable initial segmentation, but users found it more difficult to adjust the segmentation with additional markings.

For our purposes, a problem with boundary marking systems is that they achieve usability by making large assumptions about the user intent: The user can stray quite far from the "natural" segment boundary, and the system will still snap back to that "natural" boundary. This is valuable because it avoids the need for the user to make highly precise inputs with an imprecise tool like a mouse, but it also makes the tool potentially frustrating to use if the system's idea of a "natural" boundary doesn't match the user's design goal. The user study comparing these segmentation methods [37] specifically focused on part-like segmentations selected from a Princeton ground truth segmentation database [20], and therefore assumed that all users want to make selections of "natural" object parts. However, in some cases a user may prefer a different type of segmentation, depending on their design goal. Some systems use a bi-modal approach, allowing the user to choose if they are selecting "parts" or "patches" (flatter segments bounded by creases) [154, 36], but in our context the user's selection may not correspond cleanly to either concept – the user selection is simply a section of the model that the user wants to redesign with one of our user-guided inverse modeling modules. In constrast to boundary marking systems, systems that use inside/outside markings typically do not second-guess the user input: It is hard to mark a boundary exactly, but much easier to ensure a stroke stays inside (or outside) the desired segment, so systems that use inside/outside markings can typically respect the user's input exactly.

Of the methods that work by the user marking inside (and optionally outside) the desired selection, the results of two user studies indicate that responsiveness to user input is the most important feature. The first user study [89] examined only systems that required both inside-selection and outside-selection strokes [63, 76, 148, 16], and found the best tool was "Easy Mesh Cutting" [63]: a very simple method, that – thanks to its simplicity – was also the fastest and therefore most responsive. The second study [37] compared "Easy Mesh Cutting" with a new system, "Paint Mesh Cutting" [36], which gave immediate feedback without requiring any outside-selection strokes, and found that "Paint Mesh Cutting" performed even better still. (Note that "Paint Mesh Cutting," as originally presented, does not allow outside-selection strokes to indicate where the selection should not go, but one could easily add support for such strokes – in the simplest approach, by simply switching to Easy Mesh Cutting as soon as any such strokes are made.)

In the context of our system, general user-guided segmentation is largely a fall-back to handle cases for which primitive-specific segmentation is insufficient to meet the user's needs (e.g., Fig. 6.1). For these cases, the user will already have drawn inside-selection strokes, and will want to reduce an over-inclusive segmentation. This is a scenario where requiring outside-selection strokes seems very natural, so "Easy Mesh Cutting" seems like the natural paradigm.

6.2 Easy Mesh Cutting

We have chosen "Easy Mesh Cutting" [63] as an effective, simple algorithm that fits well with our system. For completeness, we give the details of this algorithm here, and then show how it handles examples where the primitive-specific segmentation has trouble due to ambiguities.

Easy Mesh Cutting is a simple, greedy region-growing algorithm. It starts from two sets of user strokes: The first set is comprised of "foreground," or the inside-selection strokes that mark points that must be included in the selection; the second set contains "background," or the outside-selection strokes that mark points that must not be included in the selection. All vertices of triangles crossed by these strokes are marked accordingly as foreground or background. Then, neighboring vertices, paired with the label of their neighbor, are added to a priority queue ordered by a feature-sensitive metric of the distance to the initial vertices. The closest vertices are popped from the queue, labelled as foreground or background according to the neighbor's label that they were paired with in the queue, and their unlabelled neighbors added to the queue (paired with their new label), until the queue is empty.

The feature-sensitive distance metric used by Easy Mesh Cutting is what lets the algorithm place segment boundaries at "natural" boundaries. This distance metric is a combination of Euclidean distance, angular difference (change in surface normal), and a curvature-based distance,

$$d(p,q) = ||\mathbf{p} - \mathbf{q}|| + w_n||\mathbf{n_p} - \mathbf{n_q}|| + w_k||k(\mathbf{p},\mathbf{q})||,$$

where w_n and w_k are weights of the normal-difference and curvature-difference terms, which guide the feature-sensitivity of the metric, $\mathbf{n_p}$ and $\mathbf{n_q}$ are surface normals, and $k(\mathbf{p},\mathbf{q})$ is a function of curvature along the surface in the direction from $\mathbf{p}$ to $\mathbf{q}$. Ji et al. use $w_n = w_k = 5$ for all examples [63]. The curvature function treats positive and negative curvature differently, because natural part boundaries tend to more often fall at points with negative curvature:

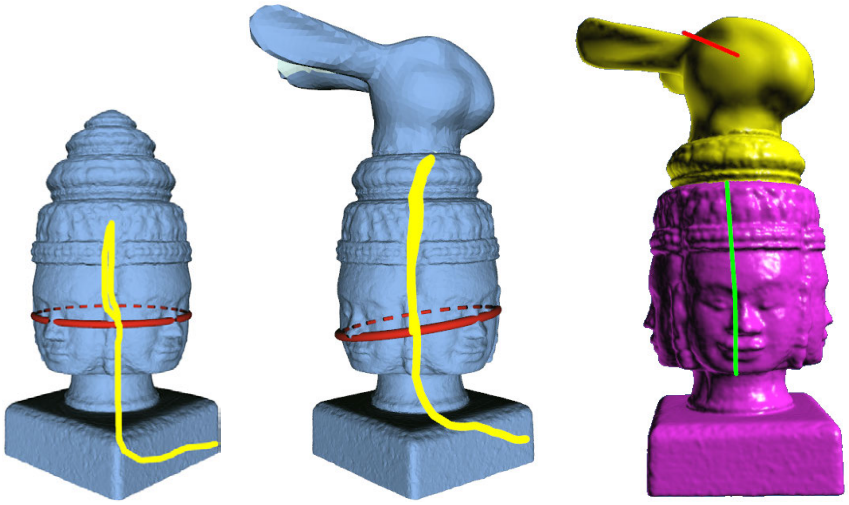
$$k(\mathbf{p},\mathbf{q}) = \begin{cases} k_{\mathbf{pq}}, & k_{\mathbf{pq}} > 0 \\ e^{-k_{\mathbf{pq}}} - 1 & k_{\mathbf{pq}} < 0 \end{cases},$$

where $k_{\mathbf{pq}}$ is the curvature along the line from $\mathbf{p}$ to $\mathbf{q}$. Note that the relative weighting of the terms of this distance metric is scale dependent; so to create a consistent metric across all examples, the mesh is first scaled to a unit box before processing.

In the original algorithm, after region growing has generated an initial segmentation, the segmentation boundary is smoothed by a greedy optimization that cuts across faces, instead of following face borders. This is important for mesh cutting, but not for our application: We want the segmentation to be made on a per-face basis, so we do not need this step in the context of inverse 3D modeling.

6.2.1 Examples

We now show practical examples where general user-guided segmentation offers advantages over primitive-specific segmentation. A prime example where this segmentation helps us is



a b c

Figure 6.1: (a) The user fit a kinematic surface (a surface of revolution, indicated by the red streamline) to a sculpture, using a large stroke to ensure even the poorly-fitting pedestal is included. The fitting result is a sweep around the primary axis of symmetry. (b) We add a bunny's head to the top of the sculpture, and with the same selection stroke the system also selects most of the bunny head, because the error threshold has been raised so high. This leads to a less desirable fitting result, indicated by the red streamline. (c) Easy mesh cutting [63] can be used to directly segment the faces from the bunny: The user strokes the the sculpture base as foreground (green), and the bunny head as background (red), and the system automatically finds a reasonable segmentation boundary. Additional strokes could be used to refine this boundary.

shown in Fig. 6.1: The user wants to select a rotationally symmetric statue, and interpret it as a surface of revolution despite details that do not match the surface of revolution model well. This works well if only the rotationally-symmetric part is selected (Fig. 6.1a), but when an undesired part (the bunny head) is adjacent, the error threshold to select the lower half plus pedestal of the statue is so high that the undesired part will also be included (Fig. 6.1b). By making a stroke where we do not want the selection to go, and relying on Easy Mesh Cutting, we can separate the undesired part (Fig. 6.1c). Note that the method does not choose to place the segment boundary right at the boundary of the non-symmetric part, but it does make it easy to control the placement of the boundary, and additional strokes will move the boundary if desired.

Another example where we may want to use general user-guided segmentation is shown in Fig. 6.2. This is an example where primitive-specific segmentation almost works: It finds a good axis of revolution, but it includes some extraneous regions in the segmentation that are not desired despite having low error-of-fit. In Chapter 2 we point out that this is easy to fix with primitive-specific methods: We can simply mark the undesired parts in a 2D view

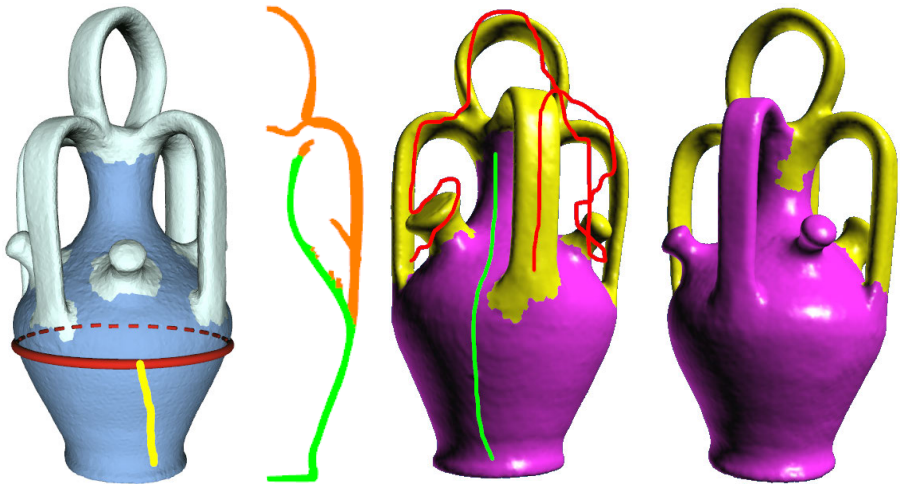


Figure 6.2: A comparison of segmentation approaches. When using the primitive-specific segmentation method of Chapter 2, we use a single stroke on the body (a), and some addition selection in the 2D "profile" view (b) to finalize the selection. When using a foreground-background method, scribbling on the body (green) and the handles (red) gives an initial segmentation (c), although one will need to make more markings in additional views (d) to fully separate the previously-invisible handle from the body.

(Fig. 6.2b). However, we can also use Easy Mesh Cutting to perform the segmentation, as shown in Fig. 6.2c. In this case, Easy Mesh Cutting requires a bit of additional work as well: Parts not seen in the original view may need some correction later, as shown in Fig. 6.2d.

Chapter 7

Fitting CSG Structure

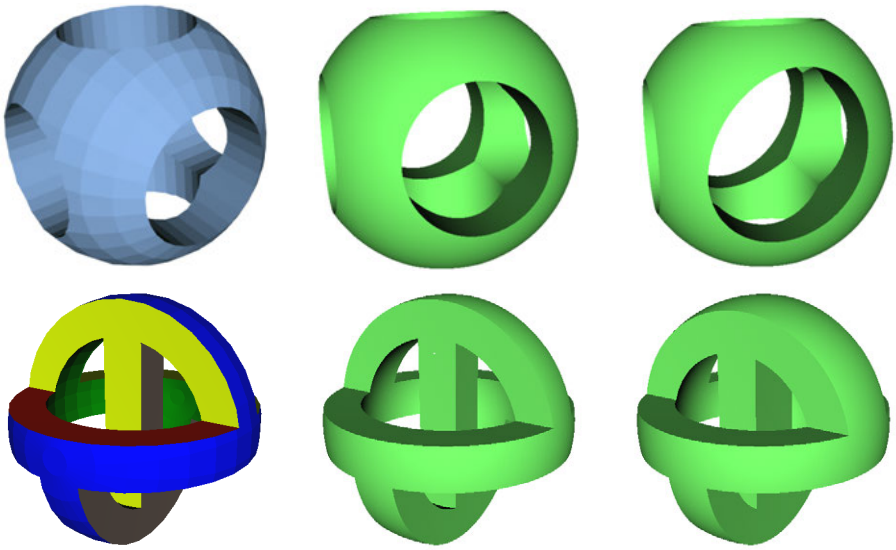
Constructive solid geometry (CSG) is a traditional, powerful modeling method for shape design: It defines a shape using Boolean operations (union, intersection, and subtraction) to combine solid primitives. CSG combination of primitives such as solid sweeps and quadric surfaces can very naturally define the shape of many man-made objects, as shown in Fig. 7.1. CSG-based editing is also a convenient way to redesign those same shapes, enabling design changes like those shown in Fig. 7.2.

In our context, it is common that any CSG structure used to design a shape has been lost, so we must use boundary-to-CSG reverse engineering methods [126] to construct a new CSG representation if we want to use CSG-based editing to redesign the shape. CSG structures are lost for a number of reasons: For example, if the shape was exported, stored or distributed in a file format that does not include this information, or if the user never explicitly stored the Boolean structure in the first place because the original designer performed Boolean operations "in place" on boundary representations. Even when the structure is kept, it may be inconsistent with the model after another modeling method has been used to edit the shape, forcing a new structure to be imposed for any further CSG-based editing [113].

Boundary-to-CSG representation conversion is a fundamentally ambiguous problem: Many CSG expressions result in the same boundary representation, but generate very different shapes once the user begins editing using the CSG primitives (see Figs. 7.9, 7.10,

a b c d

Figure 7.1: A selection of shapes easily defined by CSG operations.



a b c

Figure 7.2: Simple examples of shape editing operations enabled by boundary-to-CSG conversion. (a) Two boundary representations of shapes that were originally generated by CSG. (b) A CSG-based version of both shapes is reconstructed by fitting quadric primitives to the surfaces, and then recovering matching CSG structures with our system. (c) The shape after the user has edited the primitives defining the CSG shape: On top, the user selected and scaled a cylinder; on bottom, the user moved a plane (colored yellow in the original boundary representation) forward.

and 7.11). Fully-automatic boundary-to-CSG conversion methods [126, 125, 127] attempt to minimize the size of the CSG expression, but do not explicitly consider whether the CSG expression matches the user's intent. Additionally, these methods are limited to surfaces that are entirely composed of quadric surface patches. In this chapter, we present a preliminary exploration into how lightweight user input can help to disambiguate the problem, giving more control over the results of boundary-to-CSG-based shape edits. In addition, we demonstrate how boundary-to-CSG conversion can fit in the larger framework of inverse 3D modeling: Specifically, we discuss how *partial* CSG structures can be extracted and used for editing as soon as any primitive structures are fit, and finally how we can apply CSG-based editing to inexact input that deviates significantly from fitting primitives.

Note that this chapter focuses on boundary-to-CSG conversion to edit existing CSG-like structures in a shape. We don't always need this conversion to perform CSG-like editing of a shape: For example, one could add or subtract a cylinder from an arbitrary surface by simply selecting a circular patch on that surface and extruding. However, as that cylinder intersects other surfaces, we will typically want some way to resolve those intersections: Will the cylinder destroy the surface it intersects, or will the surface cut into the cylin-

der? Boundary-to-CSG conversion is one formal approach to ensure we always have logical, consistent answers to those questions, which is especially valuable when editing a complex existing structure formed out of intersecting CSG primitives. A drawback of this formal process is that it requires that our shape have a well-defined inside and outside: i.e., our input boundary representation should be a watertight solid. If an input shape is not watertight, we must therefore pre-process it so that (1) within some region-of-interest where we want to apply CSG-based editing, we define a watertight solid, and (2) any non-watertight surfaces that we do not need to edit with CSG are simply marked as such, and ignored by the CSG-based editing process. To preprocess a shape (or the region of interest on that shape) so that it is watertight, we can use the mesh cleanup methods for solid shapes described in Chapter 10. Any surfaces added to ensure the shape is watertight could be optionally removed after CSG-based editing is completed, so this preprocess need not leave any artifacts in the final result. For the remainder of this chapter, we assume that the input has been pre-processed to be watertight.

7.1 Automatic Boundary-to-CSG Conversion

A robust, fully-automatic algorithm for converting quadric-bounded surfaces to a compact CSG representation was introduced in a series of papers by Shapiro and Vossler [126, 125, 127]. Although other approaches for 3D boundary-to-CSG conversion have been proposed, most are either not robust and will in fact fail on some simple shapes [83], or are not designed to generate a compact output for shape editing [136]. We base our approach on that of Shapiro and Vossler [127]; in this section we explain the ideas behind their approach, as well as our own tweaks to those ideas that will allow us to handle more general inputs, and to generate a more designer-friendly result. This automatic system will create an initial result that the user can then refine to suit their needs, using interactions that we discuss in Sec. 7.2.

The approach of Shapiro and Vossler has three steps:

1. First, determine the exact primitives (and their parameters) to be used in the CSG structure.

2. Second, compute an initial Boolean expression by a union of all the primitives' "fundamental products" (defined below) that are inside the shape.

3. Finally, optimize this expression to be concise.

7.1.1 Determining the Primitive Set

Our system models a shape by a set of primitives that were defined by the fitting modules described in Chapters 2-4: This is a collection of swept surfaces and quadrics describing the boundaries of the shape. However, this set of primitives may not be sufficient to describe

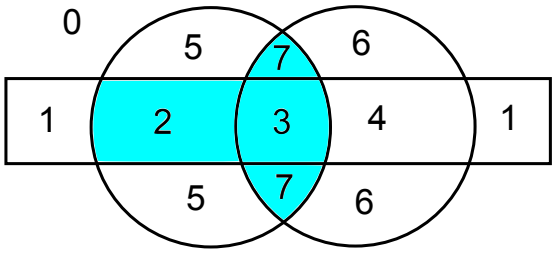


Figure 7.3: Example of the "cells" that can define a given shape (cyan). Here we have two circles and a box as our primitives, used to describe the cyan shape in the center of the diagram. Any region of the diagram with a different in/out classification with respect to any of the primitives, as well as the input shape, is numbered as a distinct "cell." Note that individual cells need not be contiguous, and the combination of cells is a disjunctive covering of the entire space (cell 0 is an unbounded cell covering everything not in any primitive).

the target shape with CSG. The key concept needed to analyze the sufficiency of a given set of primitives is the "fundamental product" of the primitives.

Fundamental products of a set of primitives are intersections that include each primitive (or its complement) exactly once. Together, these products define a kind of "Venn diagram" of the primitives – a cellular decomposition of space, which is disjunctive and covers each Boolean combination of primitives present in space. The cells of this decomposition (Fig. 7.3) are the smallest elements that CSG can produce with the given primitives, and are therefore the building blocks from which the shape must be constructed: Shapiro and Vossler proved that if any cell is partly both inside and outside the shape we want to reconstruct, then the set of primitives is not sufficient to represent the shape [126], as illustrated in Figs. 7.4 and 7.5. Otherwise, the union of cells inside the desired shape is a CSG expression of the shape, as illustrated in Fig. 7.3.

Automatically generating a sufficient set of primitives is difficult in the general case, especially if we want these primitives to be useful to the artist re-designing the shape. For surfaces bounded by quadrics, Shapiro and Vossler show that a sufficient set of primitives can be acquired by adding an exhaustive set of planar half-spaces, called "separators," which are chosen to isolate each curved face [127]. However, for more general surface types, complicated non-planar separator primitives may be required. In our case, we have a wide range of parametric surface types and typically have at least some surfaces that the user has not (yet) labeled with any particular primitive type (for example, because the unlabeled surfaces are not the part of the shape the user intends to redesign). Furthermore, the choice of primitives will affect how the user's subsequent edits using the CSG system will affect the shape, so the user will likely want some control over these choices.

Rather than require our system to generate a sufficient set of primitives, we modify Shapiro and Vossler's method to allow for *partial* CSG decompositions: Parts that are not representable by CSG with the given primitive set are simply marked in red and preserved

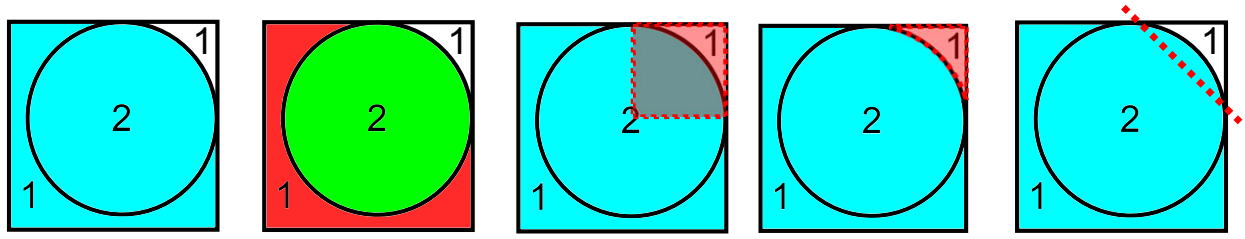


Figure 7.4: A box with a rounded corner (cyan) is not representable by the shown primitives – a circle and a box – because cell 1 is both inside and outside the shape (a). Cell 1 is intersected with the cyan shape to create the "non-representable" part of the CSG expression, shown in red; the representable portion is shown in green (b). Adding an additional shape can make the shape representable: For example, an axis-aligned bounding box of the part of the circle present in the original shape's boundary (c), the portion of cell 1 outside the original shape (d), or a minimal separator between the inside and outside parts of Cell 1 (e).

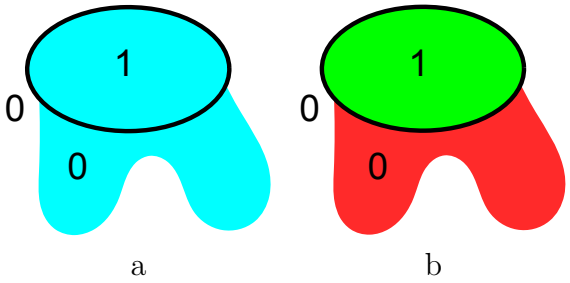
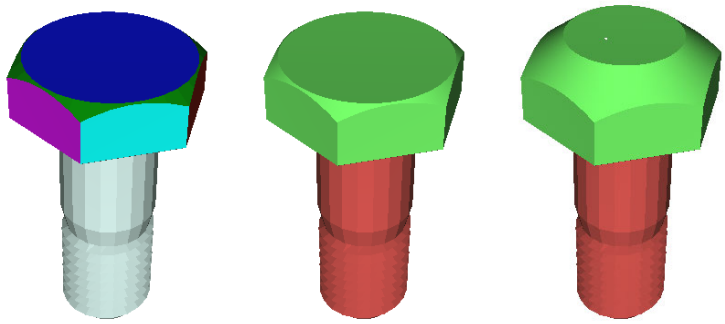


Figure 7.5: (a) A cyan shape consists of an ellipse and appendages to which no primitives have been fit. (b) This leads to the partial CSG representation: The ellipse is representable, while the rest is not because parts of cell 0 are both inside and outside of the target shape. In constrast to Fig. 7.4, there are no simple separator primitives to suggest in this case: Minimal separators and bounding boxes do not help, and the portion of cell 0 outside the shape encompasses all empty space.

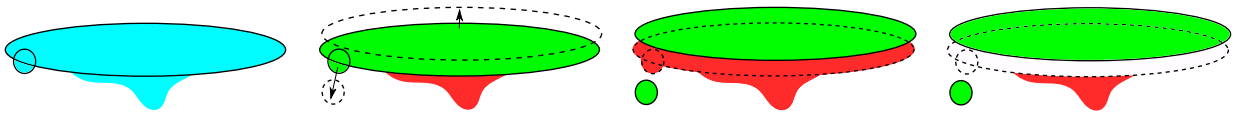
as-is, while the CSG-representable parts of the shape are marked in green and exposed as editable primitives to the user (Fig. 7.6). Addition of "separator" primitives to make more of the shape representable with CSG can be performed as-needed in a subsequent user-driven step, detailed in Sec. 7.2, in which the user is able to choose between possible separator primitive options like those in Fig. 7.4c-e. Note that, thanks to the more general range of surface types we handle, it may be impossible to make the full shape representable with planar separators [127] and other simple primitives, as illustrated by Fig. 7.5.

The fundamental product analysis of a shape naturally provides a formal definition of "non-representable" parts: Non-representable parts are the intersection of the target shape itself with the cells that are partially both inside and outside of the target shape. Including



a b c

Figure 7.6: (a) A screw model where the surfaces on top have been labelled as quadric primitives (a cone and a number of planes), but the bottom is left unlabeled. (b) A partial CSG representation, with the non-representable part left in red, and the representable part in green. (c) The user edits the green part by moving the top plane primitive upward.



a b c d

Figure 7.7: A shape composed of a circle, ellipse and some additional unlabeled geometry is shown (a). This decomposes into two representable parts (the circle and ellipse) and a non-representable part (red). The user edits the shape by moving the ellipse up and the circle down (b). We show two possible results of this edit (c,d). If we do not subtract non-editable copies of the circle and ellipse from the non-representable part, then corresponding areas of the non-representable part will remain as those primitives are moved (c). We find this growth non-intuitive, and prefer the result with non-editable primitive copies (d).

non-representable parts in our expression is equivalent to adding the target shape itself to the primitive set, ensuring everything is representable, but using the target shape as a primitive in as few places as possible. Non-representable parts can be edited to a limited extent: Subtractions from these parts work normally, but allowing extension of these parts can give unintuitive results as shown in Fig. 7.7c. We disallow such extensions by adding static, non-editable copies of each primitive to the Boolean expression defining a non-representable part, as illustrated in Fig. 7.7d.

7.1.2 Computing an Initial Boolean Expression

Shapiro and Vossler [126] show that an initial Boolean CSG expression can be computed as the union of fundamental products that are inside the target shape. These fundamental

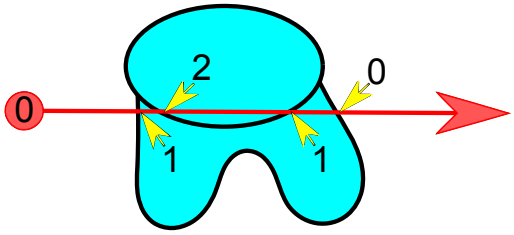


Figure 7.8: The ray casting approach to finding all fundamental product cells for a given set of primitives: A ray (red) is cast through the primitives to identify cells. The start of the ray is identified as the first cell, and the yellow arrows show where the ray crosses a primitive boundary, potentially creating a new cell.

products define a cellular decomposition of space; computing the initial CSG expression is a matter of finding all the cells that (a) exist with non-zero volume, and (b) are inside the target shape. The fundamental products are defined with respect to some primitive set; to extend our system to handle otherwise non-representable shapes, we add the target shape itself to the primitive set when computing cells. Our initial CSG expression is then the union of cells inside the target shape, and each cell is marked as *representable* if there is no otherwise-equivalent cell outside the target shape, and *non-representable* otherwise. We leave the target shape out of the Boolean expression for representable cells.

There are several ways to find all the existing fundamental product cells of a set of primitives. Shapiro and Vossler [127] introduce a robust but slow approach based on intersecting offset surfaces of all primitives; this requires computation of offset surfaces for all primitives, which is more difficult in our case where the primitive set includes arbitrarily complex surfaces. Hartquist [57] proposed instead using a ray-casting approach, in which rays are intersected with each primitive to find the transition points between cells (Fig. 7.8). This approach is easier to optimize – and can even likely be generalized to run on the GPU by a Layered Depth Image approach [153] – but could possibly miss small cells, depending on the distribution of the rays cast. For our initial prototype, we simply used an ad-hoc, brute force sampling approach that can miss cells: We regularly sampled points in space, performed inside-outside tests on each point with respect to each primitive, and stored the results of these tests in a hash set to arrive at a set of unique cells present at the sampled points. A more complete, robust system would instead use a ray-casting approach.

7.1.3 Optimize the Boolean Expression

The initial Boolean expression we generate is a valid CSG description, but (a) it is much longer than necessary, and thus probably unweildly, and (b) each fundamental product cell is defined by an intersection including every primitive (or their complements), including distant primitives, which the user may view as unrelated to the cell, and this leads to unintuitive results as we show in Fig. 7.9. Shapiro and Vossler [126] solve the first issue by applying generic

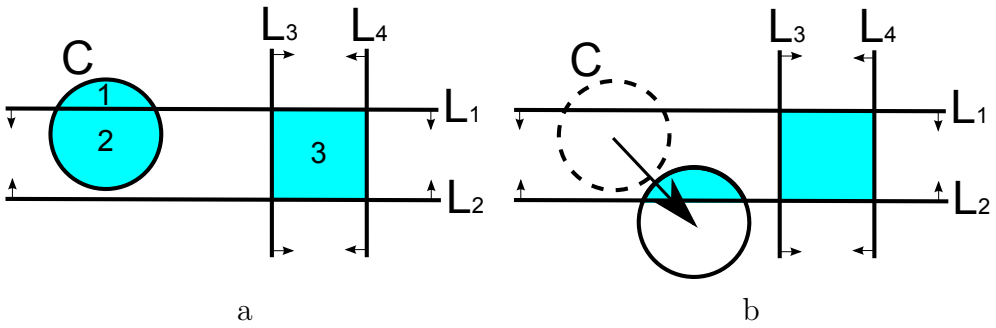


Figure 7.9: (a) The cyan circle and box are reconstructed with a circle primitive and four linear halfspace primitives (arrows indicate the side of the linear halfspaces that is "inside"). (b) When the user moves the circle down, the initial CSG expression of Sec. 7.1.2 generates the solids shown on right. Note that the bottom of the circle is missing because no cells in the original model included an intersection of the bottom linear half space and the circle.

Boolean expression optimization, which can typically find much shorter equivalent expressions. Here, we focus primarily on the second issue, which is to our knowledge unaddressed in previous work. Our approach does reduce the CSG expression size as well.

To optimize our expressions, we use a simple greedy approach to remove un-needed primitives from the fundamental product expressions for each cell defining our shape. For each cell inside the shape, we iterate through each primitive intersection in the definition of that cell, and see if we can remove the primitive from the expression without creating a shape that goes outside the target shape. The test of whether a primitive can be removed is listed as Algorithm 3 below, and the complete algorithm to construct an optimized expression is listed as Algorithm 4.

To check if a cell can be removed, Algorithm 3 simply checks each possible cell that removing a primitive could add, to see if that cell exists outside the target shape. Note that this is an $O(2^n)$ procedure, where n is the number of primitives. It is effective in practice because our CSG trees tend to be fairly small. To scale to larger examples, it will be important to cull these cell tests.

To see how this optimization works, it is instructive to step through the example given in Fig. 7.9. This figure shows a circle defined by a circle primitive, $\mathbf{C}$, and a box defined by the intersection of four linear half-spaces, $(\mathbf{L}_1 \cap \mathbf{L}_2 \cap \mathbf{L}_3 \cap \mathbf{L}_4)$. Due to the initial positions of these primitives, our initial CSG reconstruction process finds three separate cells: the top of the circle, $\overline{\mathbf{L}_1} \cap \mathbf{L}_2 \cap \overline{\mathbf{L}_3} \cap \mathbf{L}_4 \cap \mathbf{C}$, the bottom of the circle, $\mathbf{L}_1 \cap \mathbf{L}_2 \cap \overline{\mathbf{L}_3} \cap \mathbf{L}_4 \cap \mathbf{C}$, and the box, $\mathbf{L}_1 \cap \mathbf{L}_2 \cap \mathbf{L}_3 \cap \mathbf{L}_4 \cap \overline{\mathbf{C}}$. The initial CSG expression is simply the union of these cells. As the circle is moved down (Fig. 7.9b), the expression remains the same: Cell 1 is still evaluated but is now empty, and the bottom of the circle is missing because there was no cell including the intersection $\overline{\mathbf{L}_2} \cap \mathbf{C}$ in the initial configuration.

Our optimization proceeds as follows: Cell 1, $\overline{\mathbf{L}_1} \cap \mathbf{L}_2 \cap \overline{\mathbf{L}_3} \cap \mathbf{L}_4 \cap \mathbf{C}$, becomes simply $\mathbf{C}$, since all the other half-spaces can be removed without changing the shape. Cell 2, the

bottom of the circle, is now covered by the expanded cell 1, so is discarded. Cell 3 becomes $\mathbf{L}_1 \cap \mathbf{L}_2 \cap \mathbf{L}_3 \cap \mathbf{L}_4$, since $\mathbf{C}$ can be removed without changing the shape. The complete, optimized expression is therefore: $\mathbf{C} \cup (\mathbf{L}_1 \cap \mathbf{L}_2 \cap \mathbf{L}_3 \cap \mathbf{L}_4)$. Now the circle can be moved anywhere without losing its integrity.

Algorithm 3 CanRemove(Boolean expression $\mathbf{E}$, Primitive $\mathbf{P}_i$):
This function returns true if primitive $\mathbf{P}_i$ can be removed from the Boolean expression $\mathbf{E}$ without creating a shape that goes outside the target shape $\mathbf{T}$. It also returns what cells that removal (if allowed) would cause the expression to additionally cover.

Require: This function can access the target shape $\mathbf{T}$, the set of all primitives $\{\mathbf{P}\}$, and the set of fundamental product cells $\{\mathbf{S}\}$ as found in Sec. 7.1.2
Require: Primitive $\mathbf{P}_i$ or $\overline{\mathbf{P}_i}$ is in $\mathbf{E}$
Require: Boolean expression $\mathbf{E}$ only contains intersections
Initialize an empty set of cells: $\mathbf{C}$
if there is some primitive $\mathbf{P}_j$ in $\{\mathbf{P}\}$ but not referenced by $\mathbf{E}$ **then**
  if not CanRemove($\mathbf{E} \cap \mathbf{P}_j$, $\mathbf{P}_i$) **then**
    return false
  end if
  if not CanRemove($\mathbf{E} \cap \overline{\mathbf{P}_j}$, $\mathbf{P}_i$) **then**
    return false
  end if
  Add to $\mathbf{C}$ all cells returned by these two calls to CanRemove
else
  Switch $\mathbf{P}_i$ for its complement in $\mathbf{E}$ (or vice versa if its complement is already in $\mathbf{E}$)
  if any fundamental product cell matches the primitives now in $\mathbf{E}$, but outside $\mathbf{T}$ **then**
    return false
  else
    Add this cell to $\mathbf{C}$
    return true, and $\mathbf{C}$
  end if
end if
return true, and $\mathbf{C}$

Our optimization method aggressively removes intersections terms, leading the resulting shape to be as large as possible when edited, since as little as possible of the shape is "cut away" by intersection with other primitives. For distant primitives, and for best performance, this is generally desirable. But for nearby primitives that appear conceptually related, there is more ambiguity: For example, the un-optimized result in Fig. 7.10b may seem more intuitive than the optimized result in Fig. 7.10c, depending on the user's intent. In these cases we still prefer our optimization's result as the default choice, because it makes it easier for the user to clarify their intent when needed: The optimization always errs on the side of having excess geometry, and it seems easier for a user to identify and select excess geometry

Algorithm 4 Compute an optimized Boolean CSG expression

Require: This algorithm can access the target shape $\mathbf{T}$, the set of all primitives $\{\mathbf{P}\}$, the set of fundamental product cells $\{\mathbf{S}\}$ as found in Sec. 7.1.2, and a set $\{\mathbf{C}\}$ that contains those cells of $\{\mathbf{S}\}$ that are in $\mathbf{T}$

Initialize Boolean expression $\mathbf{E}$ as an empty expression
Initialize a set of removed cells $\mathbf{R}$ as an empty set
for all cells $\mathbf{C}_i$ in $\{\mathbf{C}\}$ **do**
  if $\mathbf{C}_i \notin \mathbf{R}$ **then**
    Initialize a Boolean expression to represent the cell: $\mathbf{E}_c := \varnothing$
    Initialize flag representableCell := true
    if $\{\mathbf{S}\}$ contains a cell outside $\mathbf{T}$ that is otherwise-equivalent to $\mathbf{C}_i$ **then**
      $\mathbf{E}_c := \mathbf{E}_c \cap \mathbf{T}$
      representableCell := true
    end if
    for all primitives $\mathbf{P}_i \in \{\mathbf{P}\}$ **do**
      Active primitive $\mathbf{A} := \mathbf{P}_i$
      if $\mathbf{C}_i \notin \mathbf{P}_i$ **then**
        $\mathbf{A} := \overline{\mathbf{P}_i}$
      end if
      $\mathbf{E}_c := \mathbf{E}_c \cap \mathbf{A}$
      if not representableCell **and** there is an otherwise-equivalent cell in $\{\mathbf{S}\}$ with opposite inside/outside classification with respect to primitive $\mathbf{P_i}$ **then**
        Define $\mathbf{A}_s$ as a static, non-editable copy of A
        $\mathbf{E}_c := \mathbf{E}_c \cap \mathbf{A}_s$
      end if
    end for
    if representableCell **then**
      for all $\mathbf{P_i} \in \{\mathbf{P}\}$ **do**
        if CanRemove($\mathbf{E}_c$, $\mathbf{P}_i$) **then**
          Remove reference to $\mathbf{P}_i$ from $\mathbf{E}_c$
          Add the cells returned by CanRemove() to set $\mathbf{R}$
        end if
      end for
    end if
    $\mathbf{E} := \mathbf{E} \cup \mathbf{E}_c$
  end if
end for
return optimized CSG expression $\mathbf{E}$

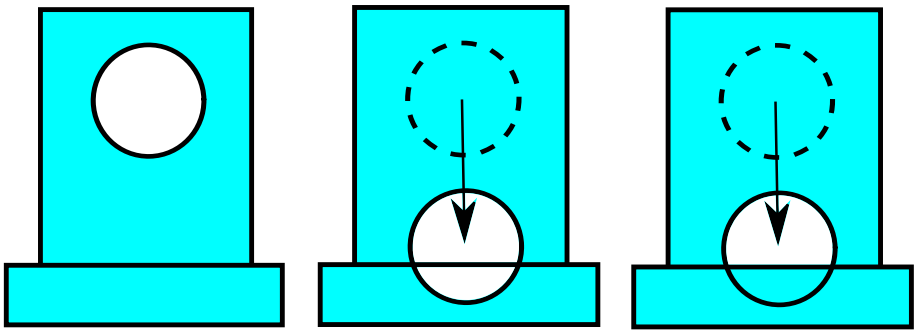


Figure 7.10: A cyan shape with two boxes and a circular hole (a). If we reconstruct the shape and move the hole down in our system, the result without optimization extends the hole into the bottom box (b), while the result with optimization stops the hole at the second box, since the hole has been removed from the expression defining the second box (c).

to remove than it is for a user to select "missing" geometry to add. The interface for removing such excess geometry is presented next.

7.2 Interactive Refinements of the CSG Structure

Once an initial optimized CSG representation has been computed, users may immediately begin selecting and modifying the parameters of the primitives defining this shape. They may also wish to modify the CSG representation itself to better suit their redesign goals. In this section we focus on modifications tailored to address ambiguities inherent in the reconstruction process. These ambiguities arise because many different CSG expressions reconstruct the same target shape, but generate different shapes after the primitives are edited.

We identify two types of ambiguity in our reconstruction that users may wish to modify: (1) As the primitives are modified the user may wish to add back interactions between primitives that the optimization removed, for example in the scenario illustrated in Fig. 7.10. And (2), if the CSG representation was partial, they may wish to add separator primitives to gain more editing control over the parts of the shape that were left as "non-representable" by CSG.

The first type of modification, adding back interactions between primitives, is supported by a straightforward interface: The user first clicks to select the portion of the current shape that they would like another primitive to subtract or intersect, and then clicks the primitive that they would like to do the subtraction or intersection. The system will automatically add the appropriate intersection or subtraction to remove the initially-clicked part. This interaction is illustrated in Fig. 7.11.

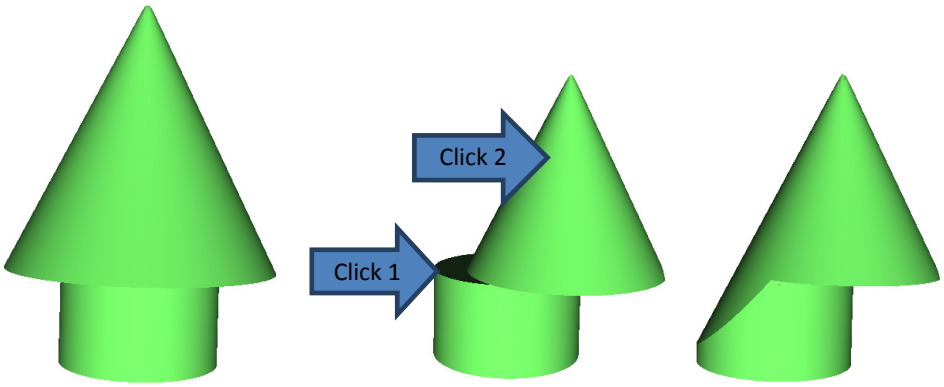


Figure 7.11: (a) An arrow is represented as a CSG decomposition of an infinite cone, an infinite cylinder, and two planar half spaces. (b) The CSG decomposition arising from optimization leaves the cylinder at the base and the cone at the top as separate primitives. The user can click (click 1) the portion of the cylinder outside the cone, and then the cone (click 2), to add an intersection with the cone to the cylinder and arrive at shape (c).

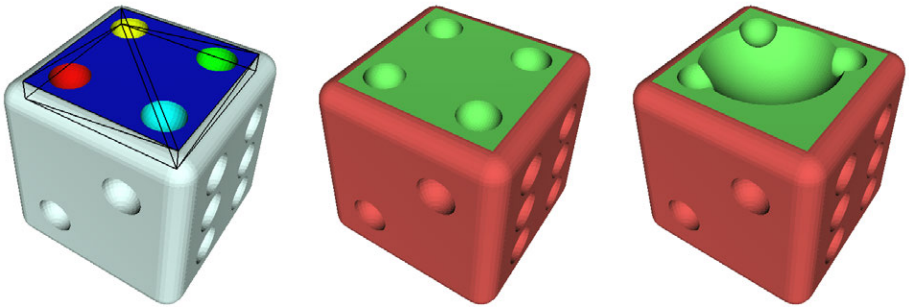
The second type of modification, adding separator primitives, is more open-ended: There are a number of separators that could be added to any primitive set to make the shape representable, as we illustrate in Fig. 7.4. Therefore, when the user selects a non-representable part, we show a list of possible separators that they may wish to add to that part. Upon the user's selection, we re-compute the CSG representation for that non-representable part.

Recall that non-representable cells have two parts – the part inside the target shape, and the part outside – and both can be generated by CSG expressions involving the target shape (intersection with the target shape for the inside part, with the complement of the target shape for the outside part). The purpose of the separator primitives is to separate these two parts. We identify several candidates as potentially-useful separator primitives:

1. The inside or outside parts themselves. We use the inside part automatically in our initial partial solution, which works well if the user does not wish to edit that part of the shape. The outside part can alternatively be subtracted, which works well when there are irregular cavities or other negative spaces that the user does not wish to edit.

2. The bounding box of the input triangles to which the user has fit primitives. We call this a "region of interest" separator, because it can help to separate the part that the user is interested in from the rest of the shape – avoiding the need for excessive primitive fitting in regions that the user does not want to edit, as illustrated in Fig. 7.12.

3. The bounding box of either the outside part or the inside part. This works well in some common cases, including rolling ball-blended corners and edges, and cases where



a b c

Figure 7.12: (a) A boundary representation of a die, with a plane fit to one side (blue) and spheres fit to the corresponding dots (red, yellow, green, cyan). A "region of interest" separator bounding box, bounding all colored faces, is also added (shown in black wireframe). (b) A partial CSG representation, with the non-representable portion left in red, and the representable portion in green. Without the region of interest separator, the whole shape would be non-representable. (c) The user edits the green part by moving and scaling one of the spheres.

the exact separator need not be precise (for example, if the cells are well separated). When applicable, it is attractive for its simplicity.

4. A single "maximum margin" separating primitive. Separating the two parts with a single primitive is equivalent to a support vector machine (SVM) classification problem [19]. SVM classifiers take two sets of points and find a plane separating those sets of points. By transforming the points using the so-called "kernel trick," SVM classifiers can alternatively find a quadric separator, or any higher-order implicit primitive separator – for example, a radial basis function could be used to define a smooth separator primitive by a few control points.

5. For quadric surfaces, Shapiro and Vossler propose an exhaustive set of planar half-spaces as separators [127]; although we may have non-quadric primitives, we can also propose these half-spaces as additional separators.

In our initial prototype system, we only implement the first two options; we are interested in exploring more separators in future work. Note that even separators that fail to fully separate the inside and outside parts may still be useful, because they may reduce the amount of the shape that is not representable. We plan to present separators in a ranked list based on two criteria: First, how effectively the separators reduce the size of the non-representable part, and second, how simple the separator primitives are.

7.3 Handling Inexact Representations

The approach so far has operated under the assumption that the primitives fit the target shape exactly, without deviation. However, in practice our input may be noisy, it may have additional details, or it may simply have minor errors due to tessellation. Such differences between the primitives and the target shape, if treated naively, will lead to errors in the cellular classification process and in the creation of static cells. In addition, some of these details may be desirable features to preserve, but they will be lost in the CSG reconstruction.

A related concern arises if the CSG operations themselves are not computed exactly: Even if the primitive matches the surface exactly, an inexact CSG operation may leave behind portions of the surfaces on subtraction. For example, with an image-based CSG method, the ordering of primitives is only determined to the resolution of the depth buffer, and is decided arbitrarily for primitives closer together than that.

7.3.1 Handling Inexact Matches Between Primitive and Target Geometry

There are two ways to handle such inexact matches between a primitive and the target geometry: Either change the geometries to exactly match, or characterize the maximum size of the deviations between matching geometries and use this as a threshold to ignore tiny cells that are artifacts of these deviations. These methods are illustrated in Fig. 7.13.

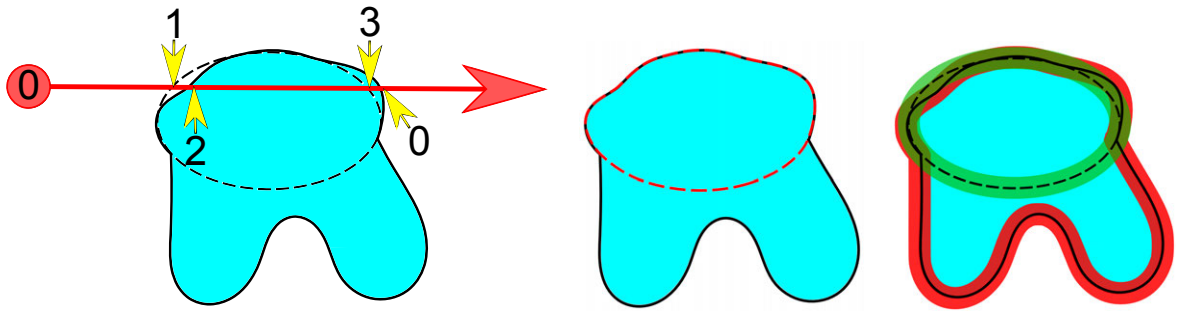


Figure 7.13: The target shape (cyan) is fit with an ellipse primitive, but the ellipse doesn't match the target shape exactly. This causes our analysis to find cells 1 and 3, which will make the entire shape non-representable (a). We propose two ways to eliminate these cells. The first option is to make the ellipse and the target shape match exactly (b). The second option is to compute the maximum deviation between the ellipse and the corresponding part of the target shape, and reject cells that are within that distance of both the matched primitive and the target shape. We show this in (c) with green and red curves following the ellipse and target shape respectively, with thickness equal to the maximum deviation: Cells 1 and 3 are discarded because they are completely under both the green and the red curves.

Changing the geometries to match exactly is the more direct solution. For simple cases, a mesh merging approach, such as the one proposed by Takayama et al. [135], may be used to copy matched geometry from the target to the primitive (i.e., removing the part of the primitive that corresponds to the target surface, and putting the target surface geometry in its place, as illustrated in Fig. 7.13b) or vice-versa, ensuring a match. However, the solution of Takayama et al. only applies if the "matching" geometries are topologically disks; cylindrical selections, selections with holes, or selections with different topology require a different approach. In more complicated cases, changing the geometries to match becomes similar to a general hole-filling problem that we discuss in Chapter 10; however, general hole-filling approaches can be slow, especially if they need to be applied for every primitive. We have yet to find a method that is both fast and general enough to handle every possible case.

The threshold method, which discards small cells near the target shape's surface, is less ideal: It could miss small, user-intended cells near the target shape surface in addition to cells that are just artifacts of the inexact match. However, it is simple to implement, and robust regardless of the differences (topological or geometric) between the corresponded surfaces. It is performed in two steps: First, compute the maximum distance between the two surfaces, d. Then, discard any cells for which all points are are within that distance d of the two surfaces – for example, in Fig. 7.13c we would discard the two cells that are entirely underneath the thick red and green lines. To do so with the ray-casting method, one could intersect each ray with a thin offset volume of distance d from the inside and outside of each surface (the thick red and green lines in Fig. 7.13c), and if the ray is inside both corresponding offset volumes for the whole span in which the ray is inside a cell, then do not add that cell.

These methods are not mutually exclusive: One can try to change the geometry to match, detect cases where it fails (for example, where the corresponded surfaces are not topological matches), then fall back on the threshold method for those cases. A related option is to match the geometry in a simpler, inexact way, before applying the threshold method: We project the vertices of the target shape to their corresponding ideal primitive surfaces, and then apply the threshold method using this projected geometry. This is a simple, fast way to reduce (but not eliminate) the mismatch between the surfaces, thus reducing the size of the small real cells that the threshold method can miss.

7.3.2 Handling Inexact CSG Operations

Even when the target shape exactly matches the primitive geometry, artifacts can appear if the CSG operations themselves are performed inexactly: A primitive subtracted from an exactly matched part may leave behind geometry if the CSG operation is not performed at high enough resolution to recognize the exact match as illustrated in Fig. 7.14. While an exact CSG algorithm such as the one provided by CGAL [17] could solve this problem, it is not practical for an interactive session: Exact CSG computations can take minutes on examples that are rendered interactively by the inexact method. Therefore, we use a more immediate solution along with inexact CSG during interactive shape-editing sessions: Our

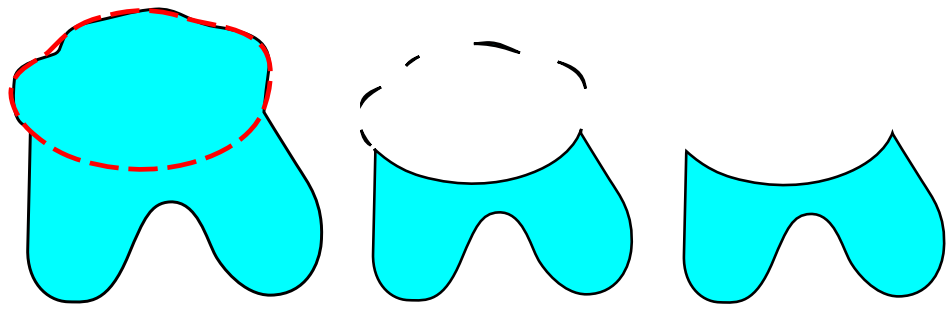


Figure 7.14: (a) A primitive (red dotted line) has been modified to match a target shape (cyan). (b) If the shape and primitive do not match exactly, or if the CSG operation is not performed exactly, then subtracting the primitive from the shape will leave some artifacts near the boundary of the primitive, whereever the target primitive is found to be slightly outside the subtracting primitive. (c) If we simply push the surface of the target shape to be slightly inside the subtracting primitive, we can eliminate these artifacts.

system moves the target shape's surface slightly inside the shape that is being subtracted – just enough so that the CSG operation is disambiguated (Fig. 7.14c). This removes the artifacts while only changing geometry that will never be seen, because it is deleted by the CSG operation.

7.3.3 Preserving Surface Details on CSG Primitives

Some deviations between ideal primitive surfaces and the target shape may include surface details that we would like to preserve during editing, without encoding these details as additional primitives in the CSG expression. We can do so by transferring these details to the primitive surface. A number of methods have been developed for transferring details between two surfaces, ranging from simply encoding those details in a bump or displacement map [75], to transferring the mesh geometry explicitly [135]; any such method will work as long as the resulting primitive remains a watertight, manifold solid. Note that preservation of these details is orthogonal to the problem of generating an initial CSG representation: Because these details are not included in the CSG expression itself, we can add or remove these details without re-computing the CSG expression. As a detail-preserving primitive is transformed, the details must be transformed as well; we discuss methods to do so for sweeps and quadric surfaces in the chapters on fitting those primitives (Chapters 2-4).

7.4 Implementation

To enable CSG-based editing of arbitrary shapes with numerous surface types (sweeps, quadrics, arbitrary polygon meshes), we convert all primitives to the "least common de-

nominator" format of watertight triangle meshes, and use a mesh-based CSG method. To ensure the system can be used interactively and robustly, we use OpenCSG [72], an image-based CSG library that uses the GPU depth buffer to compute CSG between closed solids.

OpenCSG requires that all primitives, including open half-spaces such as planes and paraboloids, must be closed. To do so, we add the bounding box of the target shape as an additional primitive. We extend the open primitives so they completely cover this box, and then close off the primitives (by making planar half-spaces into boxes, and adding end-caps as-needed to open ended primitives including cones, paraboloids, and sweeps). Primitives that extend outside the target shape's bounding box are intersected with the box. The users can extend the bounding box during editing whenever they wish to enlarge the shape.

7.5 Limitations and Future Work

We present a prototype, proof-of-concept system that shows how CSG-based editing can be integrated in our overall system, and we demonstrate that system working with several examples (Figs. 7.2, 7.6, 7.11, 7.12). However, as we discuss throughout the chapter, there is much more work to be done to make this a scalable system that allows more effective user control. Specifically, the main research questions we that remain open are: (1) How do we best create a faster algorithm for reducing the CSG expressions, as we discuss in Sec. 7.1.3, so that the algorithm scales to large examples? And (2) what are the best separator primitives of those discussed in Sec. 7.2, and what is the best way to help the user choose separator primitives in a general setting like ours?

Chapter 8

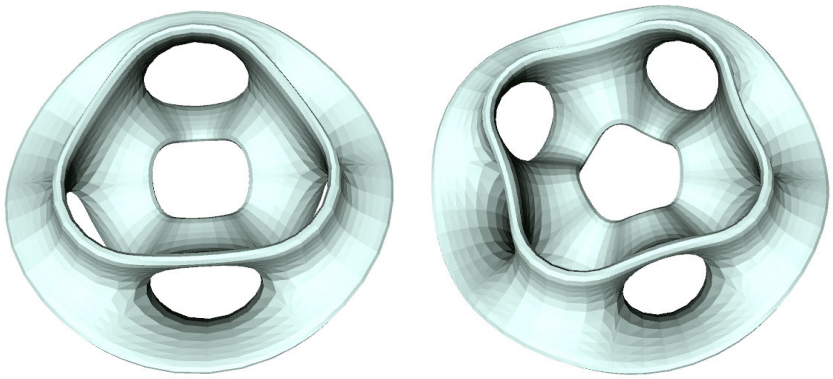
Higher Level Structure and Interaction Between Primitives

In this chapter, we explore higher-level structures that we can find in an object, when we go beyond focusing on a single segment of the object and a single primitive-fitting module. We discuss structure we can identify on the object as a whole, such as symmetry. We also look for relationships between primitive fits, structures we can fit recursively to previously-fitted primitives, and transformations between primitive structures. We show examples that demonstrate the benefits of integrating many different primitive- and structure-fitting modules in a shared system – allowing editing operations that would not be possible from a single module alone.

8.1 Symmetries

Approximate symmetries, readily apparent to the designer, can be specified as constraints and can then be either used either to fill in domains in the prototype description that may be missing in the given data set, or to average and smooth the available input data in order to generate a truly symmetrical description. The high-level description of such a shape relies on multiple instances of one extracted and averaged segment. Thus we can readily change the symmetry of such an object by changing the number of instances after the prototype segment has been properly modified, e.g., rotationally compressed in a cylindrical coordinate system (Fig. 8.1).

A large body of recent work on automatically identifying complete and partial symmetries in a given shape has made automatic symmetry detection practical [94]. For our prototype system, we do not research new methods, and simply assume the symmetries are found by one of these systems. However, we do note that it should be very simple to accelerate symmetry-finding algorithms with a small amount of user input: One of the more successful partial symmetry detection algorithms [93] starts by finding approximate point correspondences, and then searches locally to verify these correspondences. Thus a user could readily provide



a b

Figure 8.1: Given a symmetric object, our system can edit the symmetry, for example by changing the 3-fold rotational symmetry in (a) to a 4-fold rotational symmetry in (b). We do this by scaling a 120° slice of the original model into a 90° wedge and instancing it four times.

such approximate correspondences. Once a correspondence has been identified, it simply needs to be verified and re-aligned – a step that takes less than ten seconds even for large examples (> 100k triangles) where the full process would have taken several minutes.

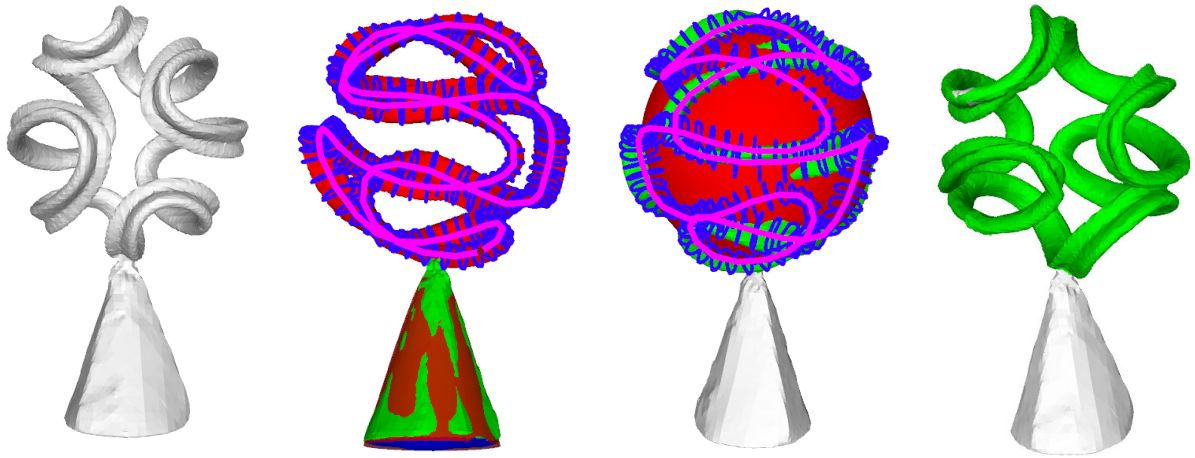
In addition to the symmetries of a part, we can also identify symmetry in fitted primitive parameters: For example, we might recognize a sweep path as symmetric (Fig. 8.2).

8.2 Interactions Between Fitting Primitives

Combining many structure- and primitive-fitting modules into a unified system allows these modules to interact in interesting, useful ways. We identify some of these interactions, and show examples of some relevant cases.

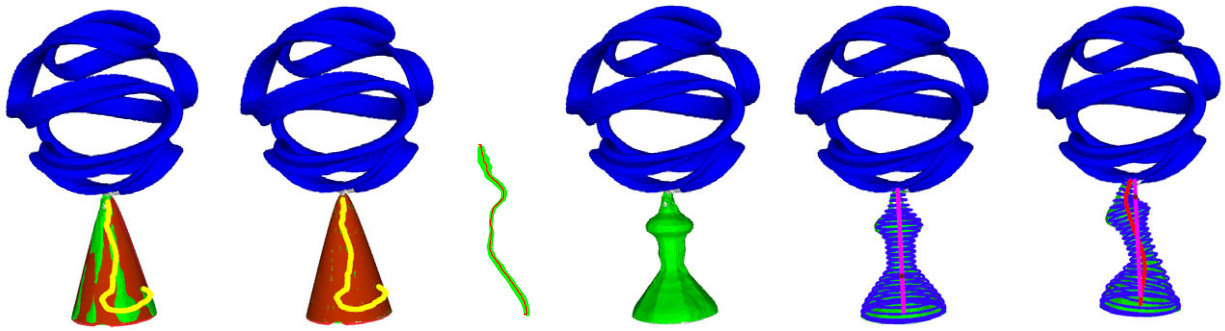
As the user specifies various modeling primitives to approximate different portions of a given shape, the covered surface areas will tend to grow into one another. There are several ways to handle this overlap. A first approach uses segmentation to find a natural boundary between the areas modeled by different primitives and then restricts the growth of the covered surface area to one side of that boundary. Alternatively, a hierarchical distinction can be used. Several primitives can apply to the same geometry, but at different levels of abstraction. For example, the overall structure could be an ellipsoid, but small details could be modeled as thin plate splines.

The extracted higher-level structures themselves can also be fit by an additional structure- or primitive-fitting module. For instance, a progressive 3D sweep path could be fit against a quadric surface. This might reveal that the sweep path of the sculpture in Fig. 8.2 lies on a sphere, and this property could then be locked in during a subsequent shape editing session. Moreover the symmetry of the extracted path can be extracted and enforced during editing.



a b c d

Figure 8.2: The sculpture model (a) is fit with multiple primitives (b): a progressive sweep on top, a cone in the lower half, and a plane at the bottom. The sweep path itself can be fit onto a sphere (c). This spherical structure can be retained along with any specified symmetry (here D_2) while editing the sweep path (d).



a b c d e f

Figure 8.3: An example of progressive editing: The base (green) of a sculpture (reconstructed from a visual hull) is first approximated by a quadric surface shown in red (a). Projecting the base mesh to the quadric surface gives a cleaner result (b). The mesh is then converted to a surface of revolution and modified by editing the profile curve (c, d). Finally, the base is converted to a progressive sweep (e), and its sweep path (magenta) is edited to further deform the pedestal (f).

Some fitting modules can also be applied to extracted 2D structures: For example, one can edit 2D curves (e.g., extracted cross-sections or profile curves) with Laplacian curve editing [132], a natural extension of the smooth surface editing system to 1-manifolds. Or one can edit the same 2D curves by fitting 2D conics – for example, to ensure a profile curve remains elliptical – using methods we have already implemented for cylinder-specific fitting (Sec. 4.3.2).

Finally, we can also transform a primitive fit to a more general fit. Currently we allow quadrics to be converted to surfaces of revolution (with an additional scaling factor), and surfaces of revolution to be converted to progressive sweeps, as demonstrated in Fig. 8.3. One could also allow rotational symmetries to be converted to surfaces of revolution – e.g., after finding the rotational symmetry of Fig. 1.2(top), one could convert to a surface of revolution to allow edits like the one in Fig. 1.2b. These conversions allow the user to easily add degrees of freedom to an existing fit.

8.2.1 Transformations Between Primitives Types

Tranforming one primitive into a more general primitive is not always completely straightforward, so we give the details of these tranformations here.

To convert a surface of revolution to a progressive sweep, we place a straight-line sweep path along the axis of revolution and use a circular cross section that scales to match the profile curve. In cases where the profile curve folds back on itself, and multiple cross section scales would be needed simultaneously, we can break the sweep into separate segments that join at the turn-around points.

To convert an arbitrary quadric to a surface of revolution, we first rotate the quadric into a canonical space that is convenient for analysis. To do so, we follow the method of [40] and express the quadratic terms in matrix form:

$$\mathbf{p}^T\mathbf{A}\mathbf{p} + \mathbf{b}^T\mathbf{p} + C_0 \text{ with } \mathbf{A} = \begin{bmatrix} C_7 & C_4/2 & C_5/2 \\ C_4/2 & C_8 & C_6/2 \\ C_5/2 & C_6/2 & C_9 \end{bmatrix}, \mathbf{b} = \begin{bmatrix} C_1 \\ C_2 \\ C_3 \end{bmatrix}. \tag{8.1}$$

We can rotate the quadric to eliminate the cross-terms by using an eigendecomposition, $\mathbf{A} = \mathbf{R}\mathbf{D}\mathbf{R}^T$; then in the rotated space $\mathbf{p_r} = \mathbf{R}^T\mathbf{p}$ the new quadric becomes $\mathbf{p_r}^T\mathbf{D}\mathbf{p_r} + (\mathbf{R}\mathbf{b})^T\mathbf{p_r} + C_0$. The columns of the rotation matrix $\mathbf{R}$ are the axes of the quadric. A point $\mathbf{p_a}$ on the axes can then be found as $\mathbf{p_a} = \frac{1}{2}\mathbf{R}\mathbf{D}^{-\mathbf{P}}\mathbf{R}^T\mathbf{b}$ where $\mathbf{D}^{-\mathbf{P}}$ is a Moore-Penrose pseudo-inverse [52] because $\mathbf{D}$ could be singular. The terms of the diagonal matrix $\mathbf{D}$ also tells us how space is scaled along each axis. If all the terms have the same sign, we pick the column of $\mathbf{R}$ corresponding to the smallest term as the axis of revolution $\mathbf{r_1}$; otherwise, we choose the axis corresponding to the term with the opposite sign. The remaining two axes $\mathbf{r_2}$ and $\mathbf{r_3}$ then describe the major and minor axes of an ellipse scaled by the corresponding scale terms D_{22} and D_{33}. By scaling space with the scale matrix $\mathbf{S} = (1 - \frac{\mathbf{D_{22}}}{\mathbf{D_{33}}})(\mathbf{I} - \mathbf{r}_3 \otimes \mathbf{r}_3)$, this ellipse becomes a circle, and the scaled space can then be edited as a surface of revolution. (In the degenerate case where one or both of D_{22} and D_{33} are zero, the quadric

surface is planar and can be handled separately as a plane.) The stationary sweep velocity field which fits the surface is finally

$$\mathbf{v}(\mathbf{p}) = \mathbf{S^{-1}}(\mathbf{r_1} \times \mathbf{Sp} - \mathbf{r_1} \times \mathbf{p_a}). \tag{8.2}$$

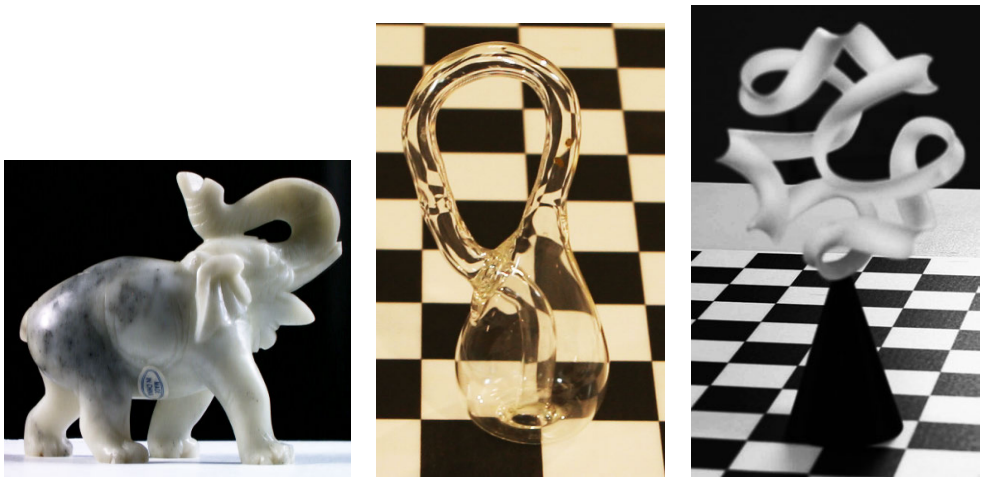
Chapter 9

Additional Input Sources

Although the bulk of this thesis focuses on polygonal mesh inputs, the concept of user-guided inverse 3D modeling can be applied to other data sources, such as scans, photographs, or even videos. This could be done indirectly, by using a fully automatic reconstruction method to aquire a polygonal mesh representation from the original scan or photograph data, but such automatic techniques often require certain assumptions to be met about the material properties and illumination of the shape and the number and distribution of images. Researchers have repeatedly demonstrated that by bringing a user into the loop of the reconstruction process itself, a clean, easy-to-edit model can often be made with less raw data and fewer assumptions about the material properties of the shape [27, 147, 58, 130].

However, these systems tend to introduce a new assumption: that the shape can be modeled easily by the modeling primitives exposed to the user. For example, [147] assume that the shape has strong curvilinear features that form a full network over the surface of the shape. [27, 130] assume that the shape is modeled by planar surfaces, while [58] allow planar surfaces and networks of feature curves in combination. [58] also provide tube sweep primitives, but these are kept separate – not informing or informed by the other primitives. These assumptions restrict the domain of feasible shapes that can be tackled by interactive, image-based modeling: None of the above systems could easily model the shapes in Fig. 9.1, because they lie outside the assumed domain for those systems.

In this chapter, we explore how a system that is aware of many different modeling primitives simultaneously could expand the domain of shapes that such user-guided shape reconstruction systems can handle. We focus specifically on reconstructing shapes from photographs by combining the information from five primitive types: visual hulls, CSG primitives, swept curves (or generalized cylinders), mesh-based thin-plate splines (as a generic, explicit smooth surface), and surface feature curves. While none of these primitives are new concepts in interactive image-based modeling, our methods for combining information from each modeling primitive into a single unified shape enables us to model geometries that would be cumbersome or impossible to model with just one of those primitives, or with a fully automated system. Our work in this area is preliminary, and we discuss remaining challenges and limitations in applying these approaches.



a b c

Figure 9.1: (a) An example of an organic shape without an obvious closed network of feature curves. (b, c) Examples of more sculptural shapes that also lack obvious closed networks of feature curves, (B) having internal structure and material properties that could stymie automatic methods relying on photo consistency.

9.1 Related Work

Previous interactive systems for modeling objects from multiple photographs or from a video sequence have typically focused on directly constructing a set of surface patches that will result in the final boundary representation. For example, [147] ask the user to draw closed boundaries from multi-view feature curves, and they then fill those boundaries by thin-plate spline interpolation of the enclosed sparse data points. [58] proceed similarly, but use NURBS or planar patches. [27, 130] focus on planar patches exclusively. In contrast, we start with a rough, approximate, volumetric reconstruction based on the visual hull, then fit parameterized modeling primitives, and finally fine-tune the resulting surface to be compatible with all the depicted features.

Silhouette and occluding-contour curves have been used extensively in single-view modeling systems. Such systems tend to assume the object is a smooth shape that is orthogonal to the image plane at the silhouette contours, giving lumpy, pillow-like results [70, 110]. Other single view modeling systems only attempt to reconstruct the visible geometry [152]. Silhouettes from three orthogonal views were used in a recent from-scratch modeling system [117]. This could be combined with previous techniques to create a silhouette-based orthogonal-view system for modeling from photographs [139]. However, the ability of this system to reconstruct smooth shapes is somewhat limited, since it uses overly simplistic assumptions to place the silhouette generators (rims) of the shape; and for complex shapes its smoothed surfaces may overshoot beyond the visual hull. In a more general, multi-view context, silhouettes have also been used effectively for automatic reconstruction systems that

use these features as the starting point for further shape optimization [34, 47]. Silhouettes have also been used in combination with specialized models for specific domains such as animating human motion [144]; we adapt a number of key ideas from [144] to our more general context.

Simple sweep primitives were used in the *VideoTrace* system to define tubular shapes by drawing the tube path in two views and by indicating the tube radius with a separate stroke [58]. No optimization was used to ensure an accurate fit, since the swept tube was thin enough to avoid ambiguity in their example. Sweep primitives have also proven effective for modeling organic creatures [50]. In both cases, the reconstructed surfaces were smooth and featureless, since no additional step was taken to add features on top of the swept surface.

The problem of automatically identifying skeleton curves from incomplete point cloud data has been approached recently by Tagliasacchi et al. [134]; but this is a more difficult task in the context of multiple 2D views. Swept surfaces with non-circular cross sections have not yet been used successfully as a modeling primitive in multi-view modeling systems, since it is typically difficult to devise what the cross section curve might look like, as it is normally not directly observable.

9.2 Technical Approach

9.2.1 Overview

Our approach is to add information from various types of modeling primitives in a logical sequence, illustrated in Fig. 9.2. First, we start with the silhouette curves: These primitives are unambiguous and provide both an initial shape estimate in the form of a visual hull, and constraints that will inform the use of all subsequent modeling primitives. Next, we introduce higher-level, parameterized modeling primitives; in our current system these are generalized sweeps and CSG (constructive solids geometry) primitives. Initially some (usually imprecise) user input places these primitives into the context of the model, indicating which parts should be represented in what particular way. Information from the visual hull is then used to constrain and fine-tune the placement and parameter values of these primitives. A modified surface reconstruction algorithm is introduced to combine the geometry of these primitives and of the visual hull into a single boundary representation that follows the primitives where available and uses the shape of the visual hull elsewhere. Finally, overall surface optimization is performed to bring this boundary representation in agreement with any other constraints, such as local surface feature curves produced by sharp edges or by abrupt texture or color discontinuities.

Our contribution focuses on the geometric reconstruction phase of image-based modeling. We thus make use of many common components found in other interactive modeling systems. Specifically, we assume that our images or video frames come from some set-up for which the intrinsic and extrinsic camera parameters are known, and that our work is done once

we have a boundary representation of the 3D geometry. Thus texturing and re-lighting are outside the scope of our investigation.

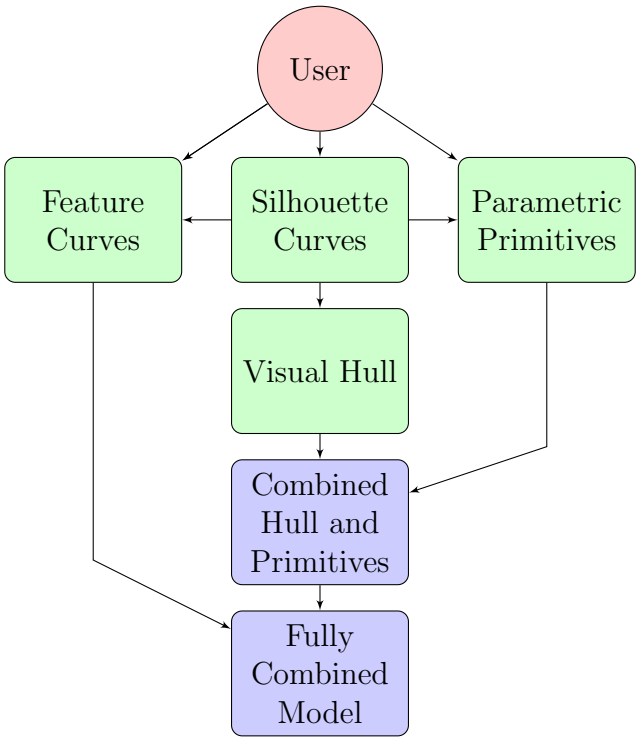


Figure 9.2: Overview of how our modeling primitives are combined. Starting from the silhouette curves defining a visual hull, we add in parametric primitives and feature curves, and finally combine these primitives into a unified model, using a series of optimization steps.

9.2.2 Modeling with Silhouette Curves

Silhouette curves are our initial modeling primitive, because they are typically easy to extract and can define both a reasonable initial approximate shape and a good guide for constraining and fine-tuning the parameters of additional modeling primitives. To extract the curves, any standard object selection algorithm such as GrabCut [118] or Lazy Snapping [82] can be used. Similar algorithms have been extended to video [25]. Silhouette curves are then projected from the camera to form cones, the intersection of which yields the *visual hull* – an incomplete but fairly precise shape description, which in many computer vision applications is used as a good-enough approximation of the actual shape [133, 77]. Recently the use of visual hulls as powerful modeling primitives was demonstrated in a constrained context of orthogonal orthographic views [117].

Some shapes can be modeled well by the silhouette curves alone. For example, we show a Klein bottle reconstructed from eight silhouette views in Fig. 9.3a. By allowing the user to

a b c

Figure 9.3: The standard visual hull of the Klein bottle (a) does not capture internal structure and has jagged edges. Allowing the user to draw silhouettes of negative space lets us capture internal structure (b), and a mesh smoothing optimization that takes silhouettes into account can smooth out unwanted sharp features from the hull (c).

mark also the silhouettes of the negative space, we can capture the internal structure of this glass bottle. Similarly, by allowing the user to manually place a ground plane, implemented as a negative space, we can give the shape a clean bottom even when views from that direction are impossible to capture – a common case for large objects on solid ground. This results in the reconstructed bottle of Fig. 9.3b, rendered to show its internal structure. Finally, a silhouette-aware mesh optimization (introduced in Sec. 9.2.4) can remove the characteristic jagged hull structure from the surface, giving a smooth, clean surface (Fig. 9.3c).

Our other two example shapes will prove more difficult to model with the silhouettes alone; however we will still make good use of the information from the silhouettes in subsequent modeling phases.

9.2.2.1 Visual Hull Surface Reconstruction

We use a publicly available octree marching cubes algorithm [71] to construct our visual hull, employing a method similar to that of [33]. This allows us to efficiently model the hull to the desired accuracy, with a higher density of vertices in areas with much detail or with high curvature. The basic idea of the octree method is simply to subdivide the octree cells until every leaf cell is either entirely inside or outside the hull, or a maximum depth is reached, or some adaptive sampling criteria (like curvature of the local faces, or distance to the surface being less than a "local feature size" measure) is satisfied. To test if an octree cell is on the boundary of the hull, we use the test of [133]: Project the cube defining the cell to each silhouette view, and test the 2D bounding box of that cell's projected shape in

the corresponding silhouette image.

The main advantage of this method over non-volumetric methods is that we can easily adapt it to include additional modeling primitives in a single watertight reconstruction, using the method discussed in Sec. 9.2.3.4.

9.2.2.2 Approximate Distance Queries on the Visual Hull

The efficient calculation of approximate signed distances to the visual hull is crucial not only for efficient root finding in the marching cubes algorithm that constructs the visual hull as an actual boundary representation, but is also invaluable for the optimizations that will be performed in subsequent phases of shape reconstruction. Therefore we use the methods of [33, 144] to compute signed distance estimates efficiently for any point in space.

We first precompute an approximate 2D distance field for each silhouette image on the pixel-raster grid of the original images. This can be computed by the fast marching method [124] or by Dijkstra's algorithm [28]. To compute the distance of a given point p to the 3D visual hull surface, we project p to each available 2D image, look up the signed distance d_2 in the precalculated 2D distance field, and transform that distance back to a 3D distance d_3 to the 3D "silhouette cone" based on the distance of p from the eye point. If p projects outside one of the 2D images (for example, see point p_2 and camera c_3 in Fig. 9.4), then it is outside the range for which we have precomputed distances; in this case we find the closest point p_c on the corresponding image boundary, and approximate the distance from p as the precomputed distance at p_c (red dotted line in Fig. 9.4) plus the distance from p_c to p (blue dotted line in Fig. 9.4). The maximum of all observed d_3 values in the various images works in practice as a reasonable approximation of the distance field. For points lying outside the visual hull, it is the maximum distance to any silhouette cone surface; for points lying inside, it is the actual closest-to-zero distance. This distance approximation algorithm is shown in Fig. 9.4. For negative silhouettes (the silhouettes of tunnels through the object) we simply compute the distance to the negative hull using this same method, then combine distance fields by taking the maximum of the ordinary distance field value and the opposite of the negative distance field value, as suggested (albeit for an oppositely-signed distance field) by Frisken et al. [43].

Because these distance estimates tend to be conservative, we can also use them to do efficient ray-casting on the visual hull by distance field ray marching [56]. To cast a ray from p in direction $\hat{d}$, we sample the distance field at p and move p by $(D(p) + \epsilon)\hat{d}$, where $D(p)$ is the approximate distance at point p and ϵ is some small constant to avoid excessively tiny steps. This process stops when the sign of the distance field changes, indicating we've crossed through the surface, or when we exit the bounding box of the space where the object could be, indicating that the ray doesn't intersect the visual hull. This ray cast query will be useful in several of the optimization steps discussed below.

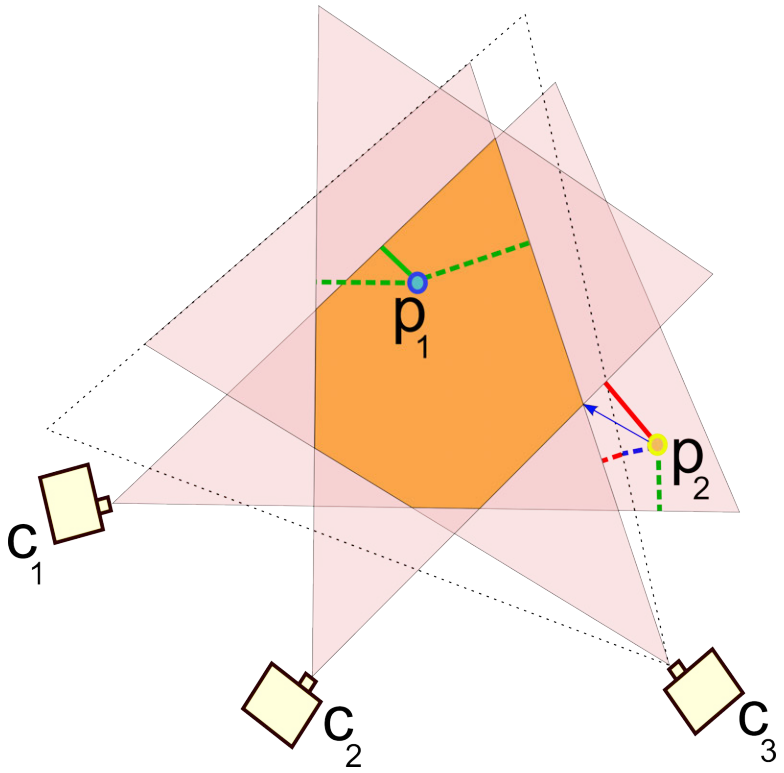


Figure 9.4: A 2D example of a visual hull (in orange) obtained by intersecting the (pink) object silhouette cones from three cameras. We approximate the distance to the hull surface as a maximum of signed distances to the cone edges (with negative distances inside each cone). For point p_1 (inside hull) this gives the true distance to the hull, in solid green. For point p_2 (outside hull) the true distance is marked with a blue arrow, and the approximate distance is shown as a solid red line. Camera c_3 is shown with an additional cone (with dotted black outline) indicating the region that is visible in the image taken from that camera. We only precompute distances in the region of the input image, so since p_2 is outside this cone, we approximate the distance from p_2 to the c_3 cone by first projecting to the dotted cone (blue dotted line) and then projecting from there to the silhoutte cone (red dotted line).

9.2.3 Modeling with Parametric Primitives

Once we have constructed a visual hull based on all silhouette constraints, we can introduce parametric modeling primitives commonly used in the design of man-made parts – specifically: swept surfaces and common CSG primitive shapes. While the visual hull tends to be noisy and difficult for the user to edit (except the highly controlled case of orthogonal orthographic views), the parametric primitives are precise and designed specifically for shape editing. Where applicable, generalized sweeps in particular are highly desirable, flexible representations, if the user intends to subsequently edit the reconstructed shape.

The chosen parameterized primitives could be placed and scaled by the user to obtain the

desired fit. However, 3D-positioning of objects and especially of complex curves is cumbersome. Thus we introduce some optimization processes that can help to fit these primitives within the silhouette constraints. We do not fully automate the process, but just reduce the workload for the user to simple, typically single-view, interactions. We herewith introduce a method to reconstruct a combined surface model that uses the parametric primitives where applicable, and falls back to the visual hull in other regions.

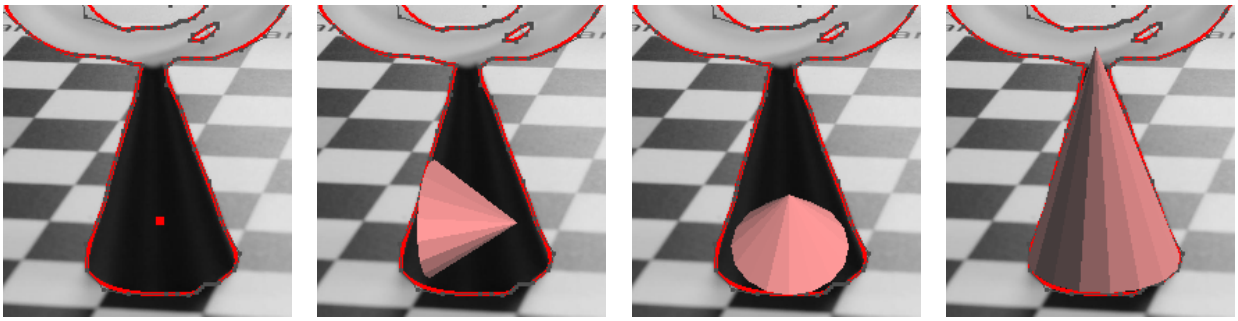
The details of our fitting process depend on the type of the parameterized primitive. However, the high-level concept is always the same: First the user selects an image with a good view of the object, and sketches some input indicating the location of the desired primitive. The system casts rays from the position of the camera that took that image, through the user's input. It marches along these rays to find a set of points inside the hull that are candidates for the position of the primitive, and then uses a discrete optimization to choose the best of those points as the initial position of the primitive. Finally the system uses non-linear optimization to refine the primitive's parameters, by penalizing generated sample points on the surface of the modeling primitive for being outside the visual hull. To ensure the primitive "fills out" the visual hull as much as possible, we also penalize the primitive for being too small, or too far inside the hull. We believe that this basic approach is general enough that it can be extended to a variety of other modeling primitives beyond the types that we demonstrate in this chapter.

9.2.3.1 Placing Simple CSG Primitives

Simple primitives such as ellipsoids, cubes, or cones may be applied by asking the user to click in the rough center of the desired shape (Fig. 9.5a). A ray cast places the object inside the hull (Fig. 9.5b), and the user can use a crystal-ball interface to orient the shape to roughly align with the desired result (Fig. 9.5c). We parameterize the shape by a translation $\mathbf{t}$ and an unconstrained 3×3 transformation matrix $\mathbf{M}$, which allows an input quadric shape to deform to match any quadric in the same class, and we optimize those parameters to minimize the error function:

$$\sum_i \left(\frac{D(\mathbf{p}_i)}{\sqrt[3]{|\mathbf{M}|}}\right)^2 , \tag{9.1}$$

where each $\mathbf{p}_i$ is a sample on the surface of the primitive, and $D()$ estimates the signed distance to the visual hull. This penalizes samples for their distance to the visual hull surface, while penalizing downscaling to avoid e.g., the solution where the shape is shrunk to a single point on the hull surface. The resulting shape should fit more snugly inside the visual hull than the original placement by the user, as demonstrated in Fig. 9.5d. The function minimization can be performed by any non-linear optimization tool. We use the Levenberg-Marquardt algorithm [87].



a b c d

Figure 9.5: An illustration of our simple primitive placement system. (a) The user clicks a point (red) where they would like to place a simple primitive. (b) The user chooses a cone primitive, and it is placed such that its centroid is directly under the user's point, in the part of the visual hull furthest from the surface. Its scale is proportional to the distance from that centroid point to the hull surface. (c) The user adjusts the cone orientation manually to roughly align with the desired cone. (d) An optimization deforms and translates the cone to conform to the visual hull.

9.2.3.2 Placing Swept Curve Primitives

We also support placement of a limited form of progressive sweep: The sweeps we fit here have fewer degrees of freedom compared to the progressive sweeps of Chapter 3, because the guidance provided by the visual hull is much more limited than that provided by an input 3D model. Specifically, we only support sweeps that set their cross-section orientation by a fixed global scheme (i.e., a rotation minimizing frame or Frenet frame) without local twists. Extending to more general sweeps is an important area for future work, as we discuss in Sec. 9.4, but our current, limited sweep primitive already gives promising results – allowing us to handle the complicated ribbon-sweep of Fig. 9.1c. In this section, we detail our current approach to fitting sweep primitives.

To fit our sweep primitive, we begin by asking the user to to draw an approximate sweep path on a single image. We let the user draw this path as a compound Bézier curve. Optionally the user may supplement this drawing with additional sweep paths, or portions of such, in other images. We then introduce an optimization procedure to locate this sweep in 3D space. This optimization requires two steps: an initialization step in which we generate a set of candidate points and use a discrete optimization to choose a good initial sweep path through those candidate points, and a subsequent continuous optimization that adjusts all sweep parameters to find the best fit in the local neighborhood of that initial estimate.

Our initial discrete optimization observes the following constraints: (1) The curve must be continuous in space, (2) it should stay within the visual hull or at least very close to it. If the same curve, with nearly the same start and end points (e.g., the handle or spout of the Utah teapot), can be drawn in two views, a number of methods can be used to approximate the curve, such as the method of Videotrace [58]. However, in some cases, such

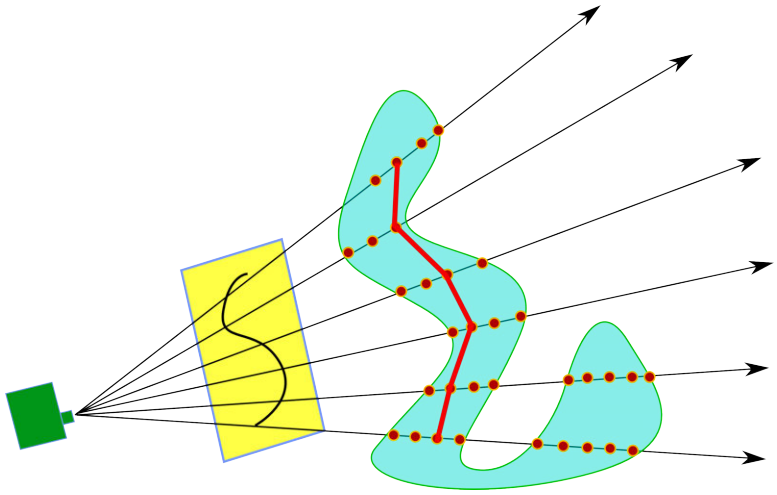


Figure 9.6: Visualization of the curve fitting algorithm: Given a curve on an image (yellow), the curve is sampled and rays are shot through the image samples from the camera (green). These rays are intersected with the visual hull (blue), and candidate points inside the hull are allocated for each ray (spaced evenly – at a distance of a hundredth of the length of the diagonal bounding box of the visual hull, for scale). We then use dynamic programming to find a shortest path through these candidate points in the order of the rays, with some bias towards the center of the hull.

as Fig. 9.1c, drawing such corresponding curves segments – or even two complete curves – is difficult. In addition, these methods are not guaranteed to respect goal (2): There can be large ambiguities in a 2-view matching, which may result in a curve that leaves the visual hull. Therefore, we introduce a simple optimization that works for a curve drawn in only a single view.

Our discrete optimization works as follows. We first create a set of possible points: We sample the 2D curve, and for each 2D sample point on the curve we cast a ray from the camera through the visual hull (Fig. 9.6). When that ray is inside the visual hull or within a few pixels of its surface, we sample 3D candidate points at a fixed spacing along the ray. Our goal is to then select, for each 2D sample point in order, a 3D point that satisfies our optimization objectives. We use dynamic programming to solve this optimization problem, where the incremental cost of adding a point $\mathbf{p}_i$ to a path with previous point $\mathbf{p}_{i-1}$ is computed as

$$w_1 D_p(\mathbf{p}_{i-1}, \mathbf{p}_i) + w_2 \frac{1.0}{|D_h(\mathbf{p}_i + \epsilon)|} \ , \tag{9.2}$$

where w_1 is a weight controlling the importance of the curve continuity, while w_2 is the weight controlling the importance of falling inside the hull, favoring the center. D_p is the distance between two points, while D_h is the approximate distance from a point to the visual hull (Sec. 9.2.2.2).

The output of our discrete optimization is a set of 3D points in space, which we then fit with a uniform quartic B-spline to get a smooth spine curve. We distribute B-spline control points to match the Bézier control points used to specify the original 2D curve, with the intent that our resulting curve have similar degrees of freedom as the input and thus will give the user the expected flexibility in control. This also allows us to place C1 discontinuities in the spline where they have been placed in the original Bézier drawing, as an initial guess at the locations of non-smooth junctions.

Given this approximate spine curve, we must now estimate the initial cross-section curve and its scale. We currently always initialize the cross section's shape to a circle, and turn to the user to estimate its scale: Following the method of [58], we let users specify cross section scale locally along the curve by (1) clicking the point on the curve where they would like to specify a scale, and then (2) clicking a point away from the curve that the cross-section curve should go through, when it is projected to the user's current view. Using this system, the user can quickly place approximate scales along the curve, which we interpolate with a Catmull-Rom spline.

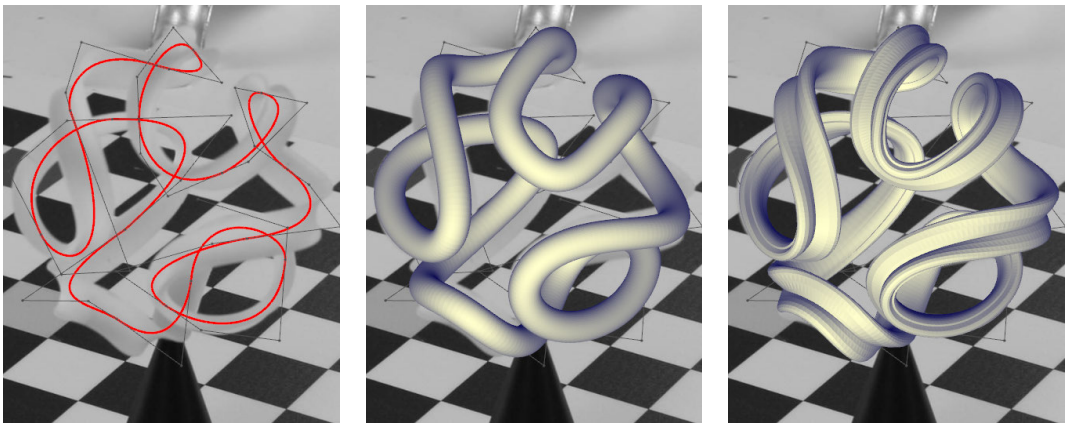
Finally, we introduce a non-linear optimization to refine both the spine curve and the cross-section shape simultaneously. Note that, in theory, a generalized sweep primitive may have arbitrary degrees of freedom in the cross section: The cross section may have an arbitrary shape, orientation, and may even change shape and orientation arbitrarily as it progresses. However, allowing too many degrees of freedom makes the optimization meaningless: Any shape at all can be perfectly fit by any spine curve simply by aggressively morphing the cross section. Therefore it is up to the user to specify the limited degrees of freedom that they actually want the system to use. We give the user several choices for restricted degrees of freedom that are common in practice: We can limit morphing to just scaling, or optionally no morphing at all; we can limit the shape to a circle; and we can limit the rotation of the cross section to a standard choice of either the Frenet frame or the rotation minimizing frame [10]. For example, the Klein bottle (Fig. 9.1b) has a circular cross section for which morphing is limited to pure scaling, while the sculptural sweep (Fig. 9.1c) has a fixed size cross section that derives its orientation from the Frenet frame. To further simplify our optimization and avoid the need for strong smoothness constraints on the cross section, we model the cross section as a polar function sampled at a fixed number of angles – thus limiting our cross sections to star shapes.

Given a user-specified set of limitations of the cross section and framing, we then optimize the sweep by minimizing the following energy:

$$w_1 \sum_i (\max(0, D(\mathbf{p}_i)))^2 + w_2 (\frac{1.0}{S+\epsilon})^2 \ , \tag{9.3}$$

where w_1 is the weight controlling the importance of staying inside the hull, and w_2 penalizes shrinking the cross section. $D()$ estimates an approximate distance to the visual hull, and S is the cross section area, while ϵ is a small value used to avoid division by zero.

Note that this optimization is different from the one used for the simple CSG primitives, because the sweep primitive has many more degrees of freedom and is thus more sensitive



a b c

Figure 9.7: From a user-drawn compound Bézier curve in one view (a), a curve optimization taking into account 8 views and their silhouettes produces an initial 3D sweep curve. After the user has indicated a cross section scale, a tubular sweep is produced (b). Finally an optimization of the sweep spine control points and of the sweep cross section yields a final fitted sweep shape (c).

to inaccuracies in the error function. Errors computed for points outside the visual hull always correspond to real problems with the fit, but errors computed for points inside the visual hull may arise simply because the visual hull does not match the true surface very well (a common occurence for complex sweeps; see Fig. 9.10b). Therefore, our error function penalizes the sweep only for points that are *outside* the visual hull. However, we then also need a new way to ensure the optimization does not shrink the primitive away to nothing; for this we adopt the slightly less elegant approach of adding a separate term to penalize shrinking solutions. We use Levenberg-Marquardt to perform this optimization [87].

The sweep fitting process is illustrated in Fig. 9.7, which shows an initial 2D curve first being reconstructed as a simple tube sweep, and then optimized to have a better sweep path and a curved cross section like the photographed sculpture (Fig. 9.1c).

9.2.3.3 Skeletal Swept Curve Primitives for Organic Shapes

We can adopt the swept primitive optimization to a much looser setting for organic shapes, where the goal is not to exactly model the shape, but rather to define the rough structure and topology in the same vein as the single-view creature modeling system of [50]. This is most beneficial when a relatively small number of views is available. There may then be significant artifacts in the visual hull that can be removed by a swept shape approximation and thus yield a much better starting point for a subsequent mesh optimization step (Sec. 9.2.4). Fig. 9.8 shows an example of a elephant with a noisy hull constructed from only five silhouettes. Note that the sweep curves were defined by the single-view curve optimization described in

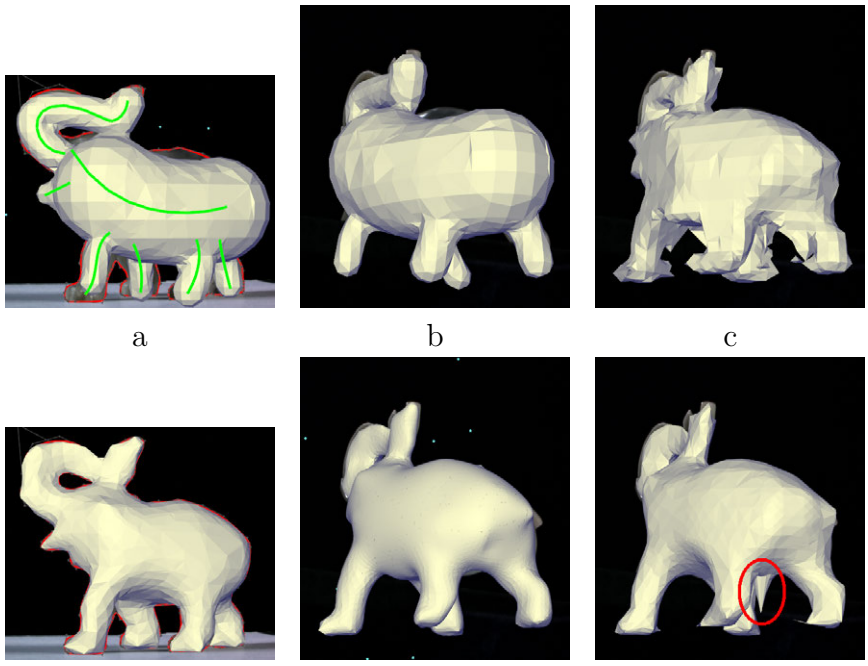


Figure 9.8: The single-view spine curve fitting process can also be used to fit tube sweeps to the elephant (a,b), quickly defining a smooth approximate shape with the correct topology, which can then be optimized to fit the visual hull (d,e) using the smooth surface optimization of Sec. 9.2.4. In contrast, starting from the noisy geometry of the visual hull itself (c) will be much more difficult for the smooth surface optimization (f) – the mesh surface optimization cannot discard some large ghost geometry, and could not ever correct a topological error.

the previous section, using the green curves in Fig. 9.8a. For these sweeps we used a default circular cross section with spherical end-caps.

9.2.3.4 Combined Reconstruction Method

In many cases, parametric surfaces will only cover a portion of the model being reconstructed. In these cases, we still want to reconstruct the full surface represented by the visual hull, but using the (presumably more accurate or more meaningful) parametric surface(s) where applicable. In this section, we introduce a new method that allows us to combine visual hull geometry with parametric primitive geometry – using the visual hull geometry only on portions not defined by the parametric primitives.

To combine all primitives and the hull together, we first represent the primitives in the same implicit octree as the visual hull. Our simple CSG primitives typically have analytic distance functions that can be computed easily, allowing for efficient reconstruction with any standard implicit meshing method. For the reconstruction of sweeps we use a method

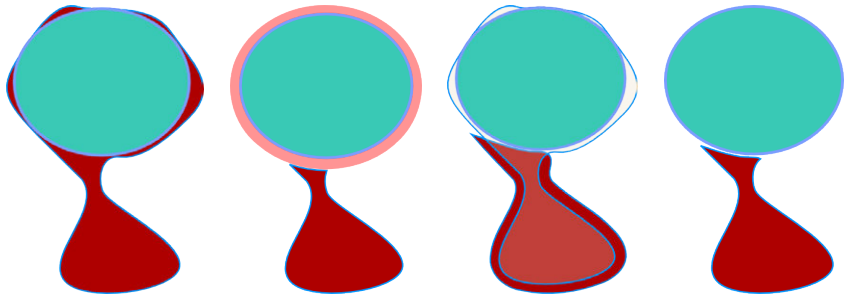


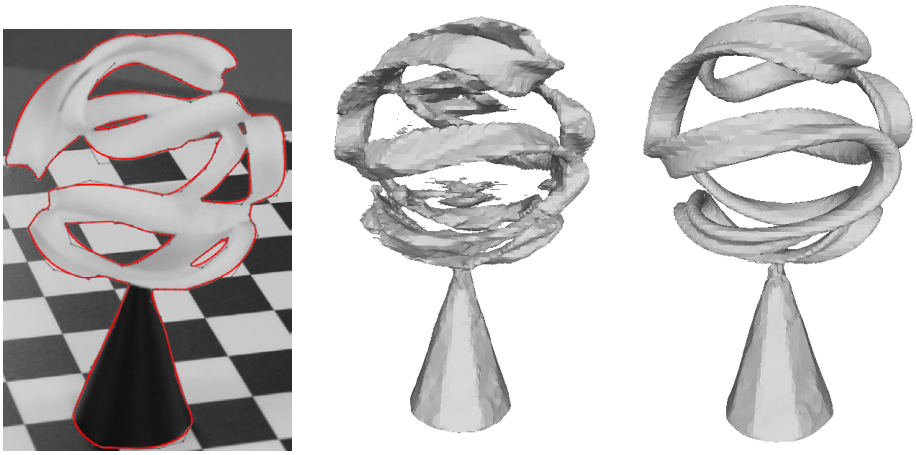
Figure 9.9: Visualization of the hull-primitive combination algorithm, from left to right: (a) We start with an overlapping hull (red) and parametric primitive (green), and then (b) subtract from the original hull the "hull" of the primitive, grown by some fixed radius (pink). We then grow any remaining visual hull geometry (c), and add back in the intersection of the grown hull and the original (d).

similar to that of [121].

We introduce a heuristic that lets us estimate whether or not a point in the visual hull is also represented by a known parametric surface. We define a visual hull of the parametric surface by projecting an explicit model of the parametric surface into each view for which we have silhouette curves indicating the visual hull of the full surface, thus getting a set of silhouettes for the parametric surface. To account for disparity between the models, we grow each parametric model silhouette by some fixed T pixels, thus growing the corresponding hull of the parametric surface. If a point is inside this hull, we tentatively consider it owned by the parametric surface.

An exception is made near the boundary of the parametric surface. There will often be some geometry connecting the non-parametric hull geometry to the parametric surface, and this connecting geometry must not be deleted. Therefore, for any point inside both the original hull and the parametric surface hull, but outside the actual parametric surface, we check the distance to the surface defined by the Boolean subtraction of (original hull − parametric hull). If this distance is equal to the distance to the parametric hull, we assume that we are in the geometry connecting us to the non-parametric hull geometry, which is owned by the original hull. Unfortunately, our efficient distance approximations (Sec. 9.2.2.2) are especially inaccurate near the border of a Boolean subtraction, so a more expensive distance calculation is typically required: As a next-best-heuristic we use a ray cast in the direction of the parametric hull gradient, and check if the distance to the surface along that ray meets our criteria.

Once a point's ownership is determined, we use the owner's distance field estimate for surface reconstruction purposes. An overview of the full process is illustrated in Fig. 9.9, and an example of a successful sweep-hull merge is shown in Fig. 9.10.



a b c

Figure 9.10: A small deviation from a hull view (a) reveals large "ghost" geometry in the sweep half of the model, but the cone at the bottom looks reasonable (b). Here we combine a fitted sweep primitive for the top with the hull on the bottom, while still keeping the two parts connected at the join point (c).

9.2.4 Thin-plate Splines and Surface Feature Curves

Once we have some initial reasonable surface mesh from our hull and/or parametric primitives, we can introduce a thin-plate spline primitive [12]. Thin plate splines have been used previously in image-based modeling to connect networks of feature curves [147], and are especially attractive as a smooth surface representation because they avoid the topological constraints of NURBS patches and even the need for a connected network curves at all. We will use these primitives to smooth our visual hull surfaces, and to integrate surface feature curves in a free form manner.

The linearized thin plate surface itself can be found by a simple, linear, least squares optimization on the extracted hull or parametric primitive mesh surface. We minimize an energy on the vertex positions V:

$$w_L||\Delta V||^2 + w_C||CV - P||^2 \ , \tag{9.4}$$

where w_L is the weight of the smoothness term, and w_C is the weight of control points P defining where the surface should go. Matrix C contains the barycentric coordinates of the points on the mesh that should be moved to meet control points P. Δ is the area-weighted cotangent Laplacian operator [73]. This minimization results in a sparse, linear, least squares problem, which we solve via the normal equations using a sparse Cholesky factorization using the CHOLMOD library [21].

The key is to find a good correspondence between points on the mesh surface (CV) and our control points (P). We do so in two ways: (1) We find a correspondence between samples of 2D silhouette curves and the mesh surface, which we use to enforce the silhouette

constraints, and (2) we find a correspondence between samples of 3D space curves defining surface features and the mesh surface. In addition to these control points, we add (with much smaller weight) an extra "control point" for each vertex that just pulls the vertex back to its original position on the visual hull: This regularizes the solution, keeping it from shrinking away too excessively in regions without other constraints. In a complete system, one could allow the user to control the relative weighting of this regularization term more directly – allowing the user to interpolate between the visual hull and the smoothed surface – but in our prototype system we just used a small fixed weight over the whole surface, so that the smoothness terms and curve-specific control points largely dominate the solution.

9.2.4.1 Silhouette to Surface Correspondence

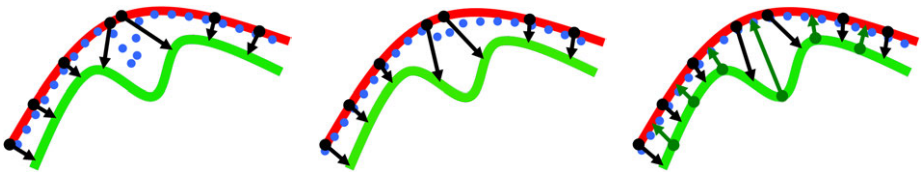
The general silhouette to surface correspondence problem has seen a number of similar recent approaches [144, 74, 117, 47]. Each method attempts in some way to find the silhouette *generators* – the points on the mesh that project back to the silhouette curve. Of these methods we chose to pursue [144], because it easily applies to meshes that were not generated by the hull (unlike [47, 117]) and does not expect continuity of the silhouettes on the surface (unlike [74], which was designed for occluding contours). We find that the method of [144] works well in practice; for completeness we summarize it here.

The approach is to sample the 2D silhouette curves and project a ray through each sample point from the silhouette curve camera. Then we try to find a point on the surface that, if moved to the silhouette ray, is likely to reproduce that silhouette. If the ray misses the mesh, we just use the point on the mesh closest to the ray. If the silhouette ray intersects the mesh, we find the point on the ray that is deepest inside the mesh, and use the point on the mesh closest to that point. In both cases, we find the 2D normal of the sample on the silhouette curve, project that direction to 3D, and bias the closest point query in the direction of the silhouette normal to avoid "mesh folding," as shown in Fig. 9.11b. To bias the query, we scale distances in the bias direction by a factor of 0.25, which is the constant factor suggested in [144]. We perform this matching and re-solve for the surface four times, increasing the weight of the silhouette constraints by a factor of two each time, to reduce the likelihood of artifacts due to a "bad" or conflicting correspondence; the smoothness constraints dominate in the first few iterations and help prevent folding.

The end result of this process is a mesh that respects the visual hull constraints, but which is otherwise smooth. Thus the process can be thought of as a mesh smoothing algorithm that is specially tailored to use the visual hull primitives, capable of acting like Laplacian smoothing away from the contour generators, while continuing to fill out the provided silhouette curves.

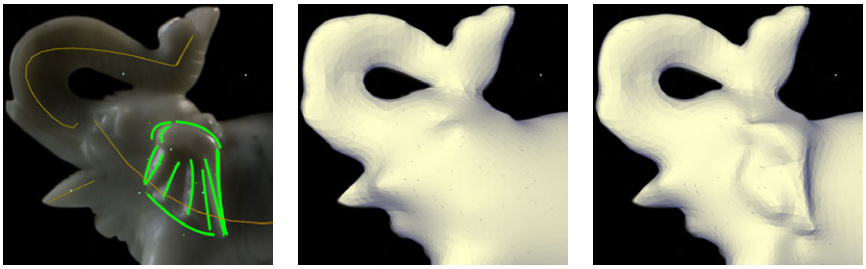
9.2.4.2 Feature Curve to Surface Correspondence

Our final modeling primitive is the surface feature curve. Previous interactive image-based modeling systems have relied heavily on networks of such curves to define a smooth surface



a b c

Figure 9.11: Different methods for finding the curve-surface correspondence, illustrated on a simple 2D example: The curve is drawn in red, the surface in green, and the correspondences in black and green arrows. The surface after optimization is shown in dotted blue. (a) The naive approach of simply using the closest point. (b) The result of biasing the distance measure in a suitable direction. (c) The result of adding intermediate curve points, thus avoiding the generation of folding artifacts.



a b c

Figure 9.12: A set of 3D curves (in green) capture details of the elephant ear (a) constructed from lines drawn by the user in 3 of the 5 initial images. When these 3D curves are used in conjunction with the thin-plate spline reconstruction, the ear details appear on the elephant mesh (c). Without using these curves we would get the smooth, earless shape in (b).

topology [147, 58]. We instead simply add these curves as freeform constraints on the surface of the thin-plate spline, thus integrating them naturally with the spline surface and any primitives that initially generated that surface. Our process is: (1) the user reconstructs the surface curve from multiple views, using any of the many available methods [147, 58, 35], (2) the curves are dynamically mapped to the surface of the mesh and included in surface optimization along with the silhouette constraints.

To perform the mapping from 3D space curve to mesh surface, we adapt the above silhouette correspondence approach: From samples on the space curve, we find closest points on the mesh surface using a biased distance metric which scales distances in a "bias direction" by a factor of 0.25 (the same as was used for the silhouette curves). In this case, we choose the bias direction based on the camera views in which the 3D curve was originally found: Specifically, we take the average direction from those cameras to the current sample point on the space curve as the bias direction. We can completely avoid the "surface folding" issue raised by [144] for the 3D curve case by creating extra sample points on the surface where

the geodesic path between subsequent initial correspondence points crosses any mesh edges. These additional sample points are mapped back to the appropriate curve interval to create additional correspondence pairs, illustrated by the green arrows in Fig. 9.11c. To allow the mesh to bend along feature curves, we also temporarily, locally re-tessellate the mesh so that we have a continuous path of edges following each path.

We demonstrate the effect of our feature curves by adding an elephant ear to the smooth mesh we previously extracted (Fig. 9.12).

9.3 Results and Discussion

We evaluated our methods on the three data sets illustrated in Fig. 9.1. Each data set came with camera information already provided, and we used the "QuickSelect tool" in Adobe Photoshop to extract silhouettes. Other curve data was extracted within our own system. Our results are shown in Figs. 9.3, 9.5, 9.7, 9.8, 9.10, and 9.12. For each example we typically had a large number of photos available (at least 25) but we chose to use only a fixed subset of five to ten photos for reconstruction: This smaller set more accurately simulates a casual modeling context in which the user may not have taken more than a few photos of an object (from unique angles) – either because the object in the wild was not visible from more angles, or the user did not have the knowledge, forethought or time to collect a large data set, or the user may even be using images from another person or time when 3D reconstruction from photos was not a consideration. Table 9.1 lists the number of views used in each set. We have demonstrated different features of our procedure for different shapes as appropriate: The Klein bottle demonstrates the features of the visual hull primitive and thin plate spline, the sculpture demonstrates the parametric primitives and combination with the visual hull, and the elephant demonstrates the skeleton primitives, thin plate spline and surface feature curves. In each case, a combination of at least two techniques was required for a good reconstruction that could be used as the start of a redesign effort, emphasizing the value of a combined approach.

9.3.1 Performance

While our focus was not on optimizing the method to be as fast as possible, we do want it to fit into the context of an interactive modeling system. As an initial guideline we have attempted to avoid optimizations taking more than a minute of user time. Note that the thin plate spline method takes time roughly proportional to the mesh size, so it could take longer if we required a higher resolution mesh – for our examples we chose what we felt to be reasonably "high-quality" settings (30-60k triangles).

Timings for each data set are listed in Table 9.1. We performed all tests on a laptop with a 2.4 GHz Core Duo processor and 2 GB of RAM. Timings are only listed for the optimizations new to this thesis: Efficient tessellation of an implicit surface and fitting curves from multiple views are already known to be efficient from previous work [33, 58].

Table 9.1: Timings for each optimization

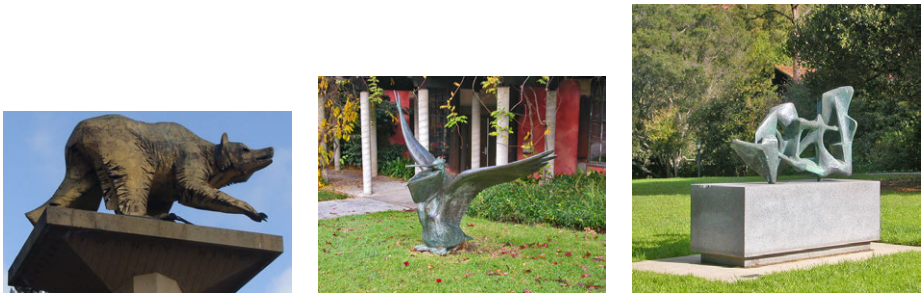
Model	# Views	Curve (ms)	Prim. (ms)	Sweep (ms)	Thin Plate (ms)	# Tris.
Fig. 9.1a	5	47	N/A	N/A	10328	30366
Fig. 9.1b	8	N/A	N/A	N/A	22125	64864
Fig. 9.1c	9	594	406	5297	N/A	17600

9.3.2 Current Scope

With just the addition of two parameterized primitives – a simple sweep (with constant cross-section) and a CSG element (cone) – we are able to reconstruct decent models from just a few photographs of objects that would impose great difficulties on earlier image-based reverse-engineering systems: We are able to deal with transparent objects with internal structure, complex geometry wrapping around a large void, and organic shapes with a non-distinct, noisy visual hull.

9.4 Limitations and Future Work

Reconstructing from photographs is a very challenging problem, and in this chapter we have focused on only a slice of that problem: We assume camera calibration and positions are known from the start, and then focus on reconstructing geometry from that information. To do so, we use photographs collected in a lab setting, where calibration is known and positions can be reconstructed in reference to a pattern on which the reconstructed object was placed. In the future, we would of course like our system to work with arbitrary photographs taken in uncontrolled settings, which raises new challenges and new opportunties for user interaction. We performed a preliminary test in which we used a popular solution for finding camera calibrations and positions automatically [146], to find camera positions for collections of 22, 23, and 40 photos of the objects in Fig. 9.13a-c respectively, and we found that the system did not really work well enough for us to use the results: It could only find relative



a b c

Figure 9.13: Objects that we may attempt to handle with our modeling system.

camera positions for small subsets of the images in each set (the subsets contained only 2-3 photos typically, and at most 8 photos but with very low accuracy in that case), and these subsets tended to include only images taken from very similar views. Our approach of visual hull-based model fitting really needs views of multiple sides of an object to work well, so this was not enough to work with. While taking more photos would likely help this system work better, we thought our collections of photos were already quite large and covered all angles of the objects well – therefore, it seems unlikely that casual users would provide better data. We then tested a semi-automatic system [1] on the same data, and this gave much more promising results – leading to some partial reconstructions that could use a much wider range of views – but this semi-automatic approach required a long session of manually finding and correcting point correspondences between the images. We are interested to see what higher-level structure-labelling could bring to this aspect of the problem as well. We hope that in the future we can push such systems to work with much smaller sets of photographs, on the order of 4-5 photos, which seems like a more reasonable amount for a casual user to take. Enabling our system to work well outside of a controlled lab setting will be an important step in making our techniques useful for a much wider range of examples.

In addition to expanding the scope where our system can apply, we also want to improve our fitting methods to handle more general primitives. For example, the sweep primitives we fit in this chapter do not allow for arbitrary rotation of the cross section, and instead require that the sweep maintain a torsion-minimizing frame or a frenet frame. Generalizing our sweep primitives to have the same degrees of freedom as progressive sweeps in Chapter 3 is challenging because fitting a visual hull is much more ambiguous than fitting an actual surface: The visual hull may have significant amounts of "ghost" geometry (see Fig. 9.10b), and in regions near this ghost geometry there may be many cross section orientations that appear to fit the visual hull equally well. To disambiguate the solution in these cases, we will need to add additional, optional user hinting: The user should be able to specify (1) the rough shape of the cross section, and (2) the positions and orientations of cross section slices at arbitrary points along the curve. We plan to solve for the sweep parameters with these inputs as approximate constraints to guide the optimization.

This chapter has focused primarily on reconstruction, but one of the primary goals of our primitive-fitting approach is that it is also useful in subsequent redesign of the shape. Portions of a shape that were generated by parameteric primitives can be easily matched to the generating primitive, so we can use the generating primitive parameters to edit the shape. Likewise feature curves can be edited, and the smooth surface can be updated to match. However, two interesting questions arise: (A) What editing can be done with the silhouette curves, and (B) what do we do about the *interaction* between primitives?

Silhouettes have been shown to be intuitive handles for shape editing in the special cases of a single view [157] or orthogonal, orthographic views [117]. However, in the case of arbitrary views, editing one silhouette will typically change the silhouettes in neighboring views. We plan to explore whether there is any intuitive and efficient way to edit a silhouette while respecting the silhouettes of nearby views – leaving them as "un-damaged" as possible. If not, the initial silhouettes may not be useful for redesign, and in this case silhouette-based

editing could be achieved by some adaptation of the single-view methods [157].

For the problem of interaction between primitives, we propose that the user could indicate for each primitive (A) if it is hierarchically related to another primitive, so that when it moves the other primitive should move with it, and (B) if its structure should be preserved while edits to other primitives occur. For example, the user may want a portion of a reconstructed surface to always maintain its original sweep-like structure – if so, this structure should be imposed as a constraint on any subsequent shape edits. This would likely require that mesh optimization be broken into phases: First, primitive parameters would be optimized to match the edited constraints; second, any child primitives would be updated to move along with their parent(s); and, finally, mesh optimization would be run on the remaining surfaces.

Chapter 10

Cleaning Results for Export

In the process of freely editing a shape, users can easily create a number of undesirable artifacts in the surface, such as self-intersections and holes – for example, if the users bend a sweep too sharply, or clone the handle of a pot and "re-attach" it by simply letting it intersect with the body. In addition, many artifacts may already exist in surfaces that the user wants to redesign: Degenerate triangles, non-manifold geometry, self-intersections, and holes are all common problems in existing scanned and CAD models. These artifacts must be fixed in order to properly support downstream applications like 3D printing and finite element simulation. If we can export our high-level representation with fitted primitive (e.g., to a CAD tool that supports such higher level primitives), we may want to also clean up the higher-level representation: We may want to ensure, e.g., that the higher-level representation segments the input surface cleanly into logical parts without holes, and with smooth boundaries.

In this chapter, we give an overview of the common problems that arise in our input and output, and the methods available to clean such output. These problems are already well-studied, so we highlight the solutions best-suited to cleaning our models, and discuss issues with and tradeoffs between these solutions.

10.1 Cleaning General Polygon Mesh Surfaces

Our most basic level of cleaning starts with an arbitrary polygon mesh – either scanned or modeled manually – that may have been further edited in our inverse modeling system. A number of artifacts can be present in such a surface: For example, there may be small gaps, either from the scanning process or from poor stitching of CAD surfaces; there may be overlapping surface components; faces may be oriented inconsistently; there may be non-manifold edges which are shared by more than two polygons. Repairing artifacts of these types is a well-studied problem, and recently well-covered by an extensive literature review [6]. Here we give a brief overview of the techniques we find most useful, as well as pointers to existing software that is effective for cleaning these artifacts.

a b

Figure 10.1: Examples of surfaces that may be defined in a CAD file with open boundaries, self-intersections, and/or non-orientable components: (a) An open, symmetrical form of Girl's Cap [53]. (b) A Klein bottle. Although these surfaces can also be defined as solids, it is often easier to represent them as thin sheets, especially if the user intends to continue editing the shape.

Note that in this section we do not assume in our initial cleanup phase that the surface is intended to define a watertight solid: We want our inverse 3D modeling system to be able to handle abstract mathematical shapes such as those in Fig. 10.1, which may be non-orientable, may have open boundaries and may have self-intersections. We discuss methods to clean surfaces that are supposed to be watertight below, in Sec. 10.3.

10.1.1 Correcting Face Orientations

In some badly-constructed CAD models, faces may be arbitrarily oriented; often, users prefer that mesh faces have a consistent orientation. Enforcing a consistent orientation of mesh faces is not possible for a non-orientable surface (e.g., a Klein Bottle or Moebius Strip), but it is easy to detect when consistent face orientation is possible, and to enforce it in these cases [5]. To enforce consistent face orientation, we can pick a face and propagate its orientation to all neighbors – flipping any oppositely-oriented neighbor faces. Then continue to propagate this orientation to all neighbor's neighbors, recursively, until the entire connected component has the same orientation. As we traverse the neighbors, we can mark which faces have already been verified to have a consistent orientation, and not revisit those faces. If we ever see a face that we have already visited and its orientation corrected, but which still must be flipped to maintain a consistent orientation with another neighbor, then this component of the mesh is not orientable. Consistent orientation algorithms have been implemented in a number of existing mesh cleanup packages [5, 91]. When each component of the mesh is consistently oriented, it is also easier for the user to correct components that are facing the incorrect direction: The user can select and flip the whole connected component at once.

10.1.2 Enforcing a Manifold Surface

Typically we want our surfaces to be manifold, meaning that edges are incident to at most two faces, and vertices are not shared by multiple rings of faces. Gueziec et al. [54] showed how to fix non-manifold elements by duplicating vertices and edges that are shared by too many faces, and re-assigning faces to the new vertices and edges as needed. They also showed how to optionally pull such elements apart, if duplicate identical vertices and edges are also undesirable. These methods are implemented in freely available open source software [5, 91]. Note that this approach is designed to respect all the geometry, and simply ensure that it is represented as manifold; in some cases, however, non-manifold portions of a surface are small local artifacts due to the failure of an earlier mesh processing algorithm (for example, the low-res meshes of the Stanford Bunny model [138] have tiny non-manifold details, added by the software used to simplify them). When non-manifold details should be deleted, one option (implemented in the MeshMixer software [120]) is to simply delete the triangles in the local neighborhood of the non-manifold elements, and then treat the resulting hole as a small gap to be filled.

10.1.3 Closing Undesired Gaps

Some holes in a surface may be artifacts that arise when a mesh is aquired by scanning – some geometry cannot be easily seen by the scanner – or may be left inadvertently by an artist in a CAD model, or arise as cracks between parametric patches in some modeling programs. They may also arise from errors in mesh processing – for example, a naive implementation of marching cubes [98] can leave holes in a surface in certain cases. Therefore, it is useful to have an a semi-automated way to fill holes in a mesh surface. Of course, some holes may be left intentionally – for example, in the visualization of the Girl's Cap surface in Fig. 10.1a, the holes help clearly visualize how the true surface interpenetrates. The process of cleaning holes in surfaces therefore has two parts: (1) Determining whether a hole should be filled or not, and (2) if it should be filled, generating the hole-fill geometry.

The simplest approaches to determine whether holes should be filled are (a) ask the user about every hole (implemented in MeshMixer [120]) or (b) fill all holes smaller than a threshold size (implemented in MeshLab [91]). It would be interesting to explore more sophisticated heuristics for deciding whether a hole should be filled or not: For example, if the data is scanned (i.e., a scan of a thin sheet with holes), then a hole is likely intentional if the scanner had a clear view of some part of the hole, and it still left the hole. Also, if the hole fill surface would either intersect other parts of the existing surface, or otherwise be relatively much larger than the boundary of the original hole (as would be the case for the holes in a thin-sheet version of Girl's Cap in Fig. 10.1a), then the holes are likely intentional. However, simply asking the user whether or not to fill all holes larger than a threshold is not an overly-burdensome approach for most surfaces.

Once we've identified a hole to fill, there are a number of approaches to filling it. In the simplest case, a hole is defined by a single open boundary loop – this is the common case

handled easily by most freely available mesh cleaning software [5, 91, 120]. While closing open boundary loops works well for simple holes, for more complex holes this approach can create self-intersecting geometry, and cannot properly handle holes with disconnected "islands" of geometry that should be incorporated into the hole fill surface [26]. In these general cases, a more robust approach is to locally segment space into "inside" and "outside" components [26, 105]. Segmenting space into inside and outside components is most commonly used when the whole model is known to be a solid model, so a globally coherent notion of what is inside and outside the shape applies, but the same approach could be applied to open surfaces as well, as long as we can define some local notion of what is inside and outside of the surface, and then restrict the volume of space we process to that local region.

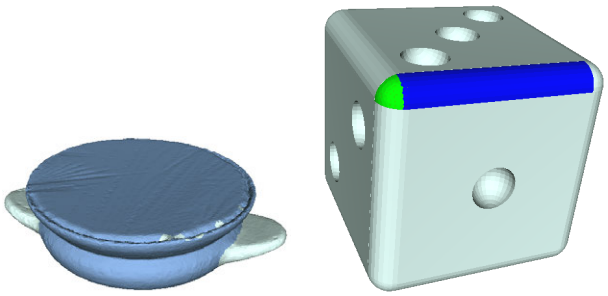
10.1.4 Merging Parts

Often artists will "merge" two parts of a model by simply moving the parts together, leaving them intersecting. As long as the viewer does not have an inside view of the surfaces, the rendered surface will appear to be the union of the two surfaces. When cleaning a model for export, we may want to resolve these intersections and explicitly merge the two parts – removing intersections and making a continuous surface that includes both parts. Merging parts in this way is a well-studied [45, 135, 120] special case of resolving holes and intersections in surfaces, and effective tools to do so are now freely available [120]. Note that if the surface is known to be solid, this is not as necessary – we can apply more general methods to globally resolve all self-intersections and holes – but in the non-solid case this special-case cleanup is common enough to be valuable.

10.2 Cleaning the High Level Structure and Segmentation

Our higher-level surface primitives are typically derived by fitting surfaces to an underlying polygon mesh surface. To faithfully represent all features of the underlying polygon mesh surface, we would have to transfer holes to the higher-level primitive surface, for example by using trim curves. Removing holes is then a matter of simply not transferring them. Alternatively, if undesired holes are patched in the underlying polygon surface using the machinery above, then there will be no undesired holes to transfer to the higher-level surfaces. The higher-level primitives could also make filling holes on the polygon mesh easier, by guiding where the hole-fill surfaces go [88].

In addition to gaps in the underlying geometry, we may have gaps in the segmentation: Small regions of unselected faces within a larger segment can arise if there are small regions of noise or detail that cause the surface to deviate more than usual from the fitted surface, as shown in Fig. 10.2a. Cleaning such gaps in a segmentation can be performed with a morphological approach similar to the one described in Chapter 2 (Fig. 2.13b). Specifically, we can use the "opening" operator, which selects unselected elements touching the boundary



a b

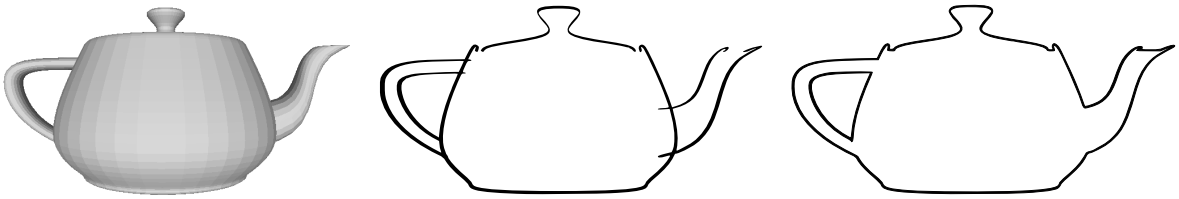
Figure 10.2: Examples of different artifacts in the high-level structure that a user may want to clean before export: (a) Segments like the one shown in blue may have small gaps where the surface deviated from the fitted primitive. (b) The cylinder's segment (blue) includes a small region that would ideally only belong to the sphere's segment (green), demonstrating how neighboring segments may not initially have a clean boundary at the ideal transition point between the two shapes.

of the selection, followed by the "closing" operator, which deselects selected elements touching the boundary of the selection. The size of the selection and deselection by the opening and closing operators respectively controls the size of segmentation gap that the algorithm will fill. The selection size can be defined in terms of the number of polygons away from the boundary that are selected, or in terms of a geodesic distance – in which case we would select any polygons that are entirely within some threshold geodesic distance d of the boundary.

Another common issue is that the boundaries between two segments may not placed at the ideal curve of intersection between the two primitives (Fig. 10.2b). Because the primitive surfaces and the input surfaces generally do not match exactly, the true curve of intersection could be arbitrarily far from the input surface – or may not even exist. Therefore, the boundaries between segments are typically refined by locally optimizing the segmentation boundaries on the input surface. One effective method in practice is to use a graph cut to find a shortest-path curve that separates the two surfaces and that stays within the region that fits both primitives [149].

10.3 Guaranteeing a Solid Model

If the surface we are attempting to clean is intended to define a solid model, then we know that the surface must be orientable, and that all self-intersections and holes in the surface are defects that need to be fixed. Therefore, it should be possible to more effectively automate the cleanup process – ideally resulting in a global, guaranteed algorithm for fixing holes and self-intersections as illustrated in Fig. 10.3c. In addition to our primary goal of guaranteeing that our output is a valid solid model, we also want to respect the structure of the input surface as much as possible: The polygons of 3D meshes are often carefully placed to best-



a b c

Figure 10.3: The Utah teapot (a) is a standard example of a surface with self-intersections and holes, as can be seen in a cross-section view (b). In cleaning up shapes like this, we aim to remove intersections and close gaps (c).

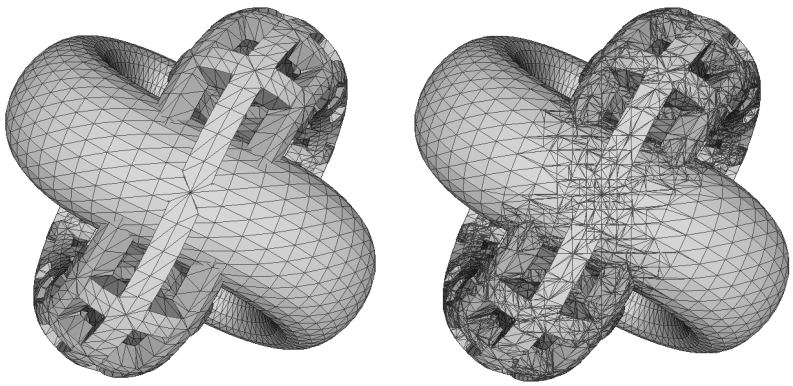
approximate the desired surface, so we don't want to change those polygons if we don't need to. In this section we give an overview of global solid-repairing algorithms, and discuss the tradeoffs and limitations in the best approaches developed so far. The algorithms typically work by segmenting space into "inside" and "outside" partitions, and differ in two key ways: the choice of representation of volumes in space, and the choice of criteria for segmenting space into inside and outside partitions.

10.3.1 Volumetric Representations

Most algorithms for partitioning space into solid "inside" and empty "outside" regions were initially defined in terms of a specific volumetric representation (such as regular voxels, BSP trees, or tetrahedra), but could be adapted to run on any volumetric representation. The choice of volumetric representation trades ease of implementation for exactness of representation and quality of output mesh. In this section we give the details of important trade-offs made by this choice.

The most popular choices to represent volumes in space are regular grids of voxels [99] and implicit surfaces [128], because these approaches are relatively easy to implement in a way that guarantees a correct output. The main difficulty with these approaches is their accuracy in preserving the input surface, and the economy of their representation: Naive implementations will destroy sharp details and small features, and more sophisticated, hierarchical implementations can still miss some very tiny features and will still tend to create a shape with many more polygons than the original input. The mesh could be simplified to reduce this problem, but if the input was an artist-crafted mesh with carefully placed polygons, it's difficult for any automatic simplification algorithm to recover an as-good result. It is often preferrable to preserve those original polygons directly. This issue can be greatly minimized by using a hybrid approach [9] that uses the original surface except in the neighborhood of areas that required repair. However, this adds a great deal of implementation complexity and still "damages" more of the surface than is strictly necessary.

To preserve all sharp edges, tiny features, and the exactly input polygons, we can use a volumetric representation that respects the input geometry exactly, such as a BSP tree or



a b

Figure 10.4: We attempt to resolve intersections in a complicated self-intersecting triangle mesh (16k triangles) (a). Robust BSP-based code [18] resolves most major self-intersections, but produces a result with an order of magnitude more triangles (142k triangles), many near-degenerate sliver triangles and still has 2k pairs of self-intersecting triangles due to floating point error in the output vertex positions (b). The excessive additional triangles are co-planar, so could largely be removed by mesh simplification, but the self-intersections may be more difficult to correct.

a tetrahedralization. BSP-based approaches still create many more polygons in the output than the original input (as we show in Fig. 10.4), but they exactly represent the input polygons, and all additional polygons should be exactly co-planar and thus easier to simplify away. One new problem with BSP-based approaches is that the new geometry computed at the intersection of previously self-intersecting facets may require exact computation to represent. Converting exactly-computed intersection coordinates to finite-precision vertex coordinates (a necessary step to export a shape in most file formats) can then introduce new self-intersections, which are difficult to remove in a robust, guaranteed-correct way. We find that state of the art exact BSP-based approaches to resolving self-intersections [18] do indeed create new self-intersections in practice on examples such as the one shown in Fig. 10.4. Finally, the volumes that result from a BSP-based decomposition of space are somewhat arbitrary, and since holes will be filled by surfaces that follow the boundaries of those volumes, the resulting hole-fill geometry can be oddly shaped as well.

Tetrahedralization of an input surface faces many of the same numerical problems as BSP-trees, and worse: All self-intersections must be resolved *before* attempting to tetrahedralize an input, by another method that simply breaks up intersecting polygons with explicit mesh edges along the intersections [62], and this process is as difficult to do in a guaranteed, robust fashion as computing an intersection-free BSP-tree. But in addition to this problem, even after all self-intersections are resolved, currently the state of the art in efficient, robust tetrahedralization [129] still cannot tetrahedralize many challenging inputs. Jacobson et al. [62] recently found that it is necessary to instead approximately tetrahedral-

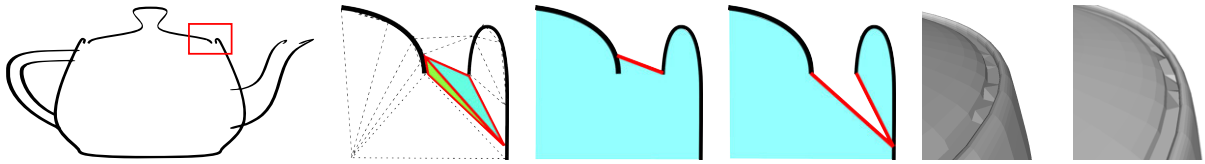
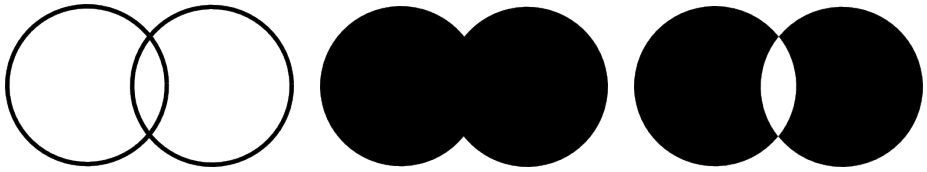


Figure 10.5: There is a gap between the top and body of the Utah teapot (a) that should be easily filled by connecting the boundary edges of the teapot top and body. However, a per-element inside/outside segmentation of some tetrahedralizations may be *incapable* of connecting these boundary edges, because the necessary edges or facets simply aren't present in that tetrahedralization. We illustrate this problem in 2D with a triangulation (b): In this example, we can at best either cover some of the teapot's top, by marking the green and blue triangles as "inside" (c) or connect the teapot top to the inside of the teapot body, by marking the green and blue triangles as "outside" (d). The equivalent problem occurs in some 3D constrained Delaunay tetrahedralizations of the Utah teapot: We show the result of covering some of the teapot top (e) or connecting the teapot top to the inside of the body (f) for one such example.

ize the input, meaning that some tetrahedra will cut through polygons of the input surface. They minimize the problems this causes by locally subdividing the problematic polygons, so that typically only a small fraction of each problematic polygon will not be preserved in the output tetrahedralization. Once a tetrahedralization has been computed, there are some advantages over a BSP tree: (1) The decomposition of space is less arbitrary, and tends to result in more natural hole-fill boundaries in practice, (2) the tetrahedra can be useful for other downstream applications like finite element simulation, and (3) tetrahedralizations do not tend to introduce nearly as many un-needed new triangles in the output surface as BSP-trees do.

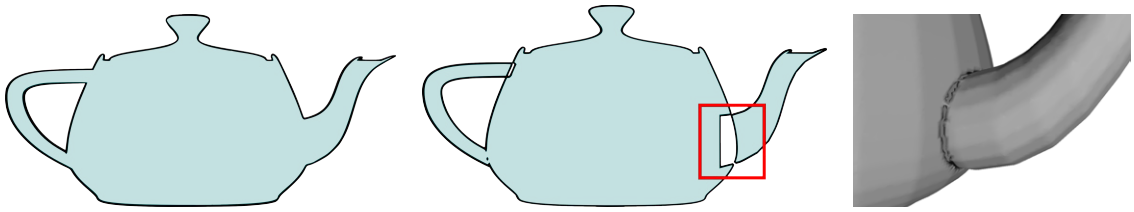
Both BSP- and tetrahedra-based approaches assume that a good output surface will lie on the boundaries of the volumetric elements (BSP-leaves or tetrahedra). Unfortunately, this is not always the case: Some BSP-trees and tetrahedralizations will make an ideal inside/outside segmentation impossible, because the ideal boundaries simply do not exist in their decomposition of space, as demonstrated in Fig. 10.5. This problem has been addressed partially by Podolak et al. [105] in the context of hole filling: They subdivide space in areas that could otherwise be too coarsely tessellated to include their ideal hole-fill surface. This idea has not yet been generalized to algorithms that repair self-intersections as well as holes; thus it is a promising direction for future work. An orthogonal idea is to create a hybrid approach: We could tetrahedralize the input so that all details of the input are captured exactly by tetrahedra boundaries, and then perform a marching-tetrahedra isosurface extraction step so that hole-fill surfaces can cross freely through tetrahedra.

10.3.2 Criteria for Inside/Outside Segmentation



a b c

Figure 10.6: When two or more intersecting shapes (a) are repaired by the mesh cleanup algorithm, we typically want the region of the intersection to be solid (b), and not hollow (c). Parity-based mesh repair strategies that try to ensure that the shape changes from solid to hollow across facets of the input mesh tend to create hollow intersections.



a b c

Figure 10.7: (a) A 2D slice of a good inside/outside segmentation of the Utah teapot, with inside colored blue. (b) A 2D slice of Polymender's segmentation using a parity-based criteria for inside/outside segmentation [66], with inside colored blue. A 3D view of the region in the red box is shown in (c) – note the small tunnels connecting the spout to the body.

There are few common high-level criteria used to help decide what parts of a shape should be inside or outside, which we refer to as (1) parity-based approaches, (2) visibility/outer hull-based approaches, and (3) winding number-based approaches. Parity-based approaches are based on the assumption that the segmentation should tend to switch between inside and outside across facets of the input. Murali and Funkhouser [95] and later Nooruddin and Turk [99] and Ju [66] introduced methods to approximately satisfy a parity-like criteria for arbitrary inputs. Unfortunately, parity-like approaches tend to generate undesirable results for intersecting volumes: For example, the intersection of two closed volumes will be labelled as outside, creating an undesired void as illustrated in Fig. 10.6c. This can lead to bizarre results in practice even for simple examples like the Utah teapot, as we show in Fig. 10.7. Visibility or outer hull-based approaches [26, 99, 18] avoid these problems by instead attempting to only take the outer-most surfaces of the volumes – for surfaces with holes, this is approximated by preserving only the surfaces that are visible from outside the bounding box of the input shape. This deletes all internal structure of the shape – including structures the user may have wanted to preserve. Finally, winding number-based approaches

avoid the problems of parity-based approaches while still preserving internal structure, but assume the orientations of all input faces are correct. The standard concept of winding number is only defined for closed, oriented 2D polygons, so until recently, winding number-based approaches would need to close holes in a model first by some other method, and then resolve self-intersections by taking the winding number of the polygons in 2D planar slices of the new hole-free model. However, recently Jacobson et al. [62] generalized the concept of winding numbers to open, 3D surfaces. Their approach diffuses winding number across open holes, effectively combining winding number with diffision-based hole filling [26].

The best of these methods depends on whether the shape has complex internal structure that should be preserved: If so, then repairing incorrect or inconsistent surface orientations is a relatively easy mesh-repair task (see Sec. 10.1.1), so we suggest repairing surface orientation first, then using a winding number-based approach. However, if there is no internal structure to represent – as is the case for scanned shapes, where such a structure could not have been captured by the scanner – then it is easier to use a visibility/outer-hull approach.

Chapter 11

Summary

We have presented the idea of user-guided inverse 3D modeling, an approach to shape redesign that gives users simple, flexible control of a fast reverse-engineering process. Our goal is to enable users to quickly express a shape with exactly the degrees of freedom they need to achieve the redesigns they want. To demonstrate the effectiveness of this approach, we presented a number of useful inverse 3D modeling modules, and we presented a number of shape redesigns achieved in practice with prototype implementations of these modules.

In the first half of this thesis, we presented a collection of surface-fitting modules that let users quickly fit shape modeling primitives – sweeps, quadrics and smooth surfaces – to an input surface mesh. The modules we described work from minimalistic input – typically a few user strokes, and a choice of desired primitive type. The modules return segmentation and fitting results to users in an immediately-editable form, so users can begin using their fitted model to edit the shape as soon as possible. We identified ambiguities that can cause initial segmentation and fitting results to not reflect the users' intent, and we presented simple additional inputs that can disambiguate the desired fit. We identified efficient methods to guarantee that subsequent fitting would respect those additional inputs. In addition to in-depth analysis of how users can guide the primitive-fitting process, these chapters also presented improvements to the state of the art in efficient fitting methods: We showed that previous methods used for kinematic surface fitting have significant failure cases and then showed how to fix those failures, and we also presented direct, type-specific quadric fitting methods that find lower-error solutions than previous methods.

Next, we explored how multiple primitive fitting modules can be used in combination to enable an even wider range of shape redesigns. We discussed a boundary-to-CSG conversion process that combines primitives in a Boolean expression to define the target shape (or as much of it as possible), allowing CSG-based redesigns. We discussed the fundamental ambiguity that many different CSG trees can describe any given shape, and we introduced simple additional inputs to disambiguate cases where those different CSG trees would give different editing results. In addition to CSG-based editing, we also explored how different primitive fits can be combined hierarchically, allowing multiple levels of structure to be enforced or edited. Finally, we discussed transformations between primitive fits, allowing

users to quickly change the degrees of freedom available to them when editing a shape.

We then proposed that combining multiple shape modeling primitives into a unified system is also a powerful idea in the domain of image-based modeling. We demonstrated this with a study of three example shapes that we believe would be very challenging to reconstruct with previous methods, using only five to ten photos of each shape. We presented a series of optimizations to combine data from visual hulls, simple CSG primitives, swept primitives, curves, and smooth surfaces into a unified model; by using these optimizations, we were able to reconstruct reasonable models for each of our three challenging examples. Much work is left to build a truly practical system based on this approach: In particular, our examples all used photos taken in a controlled setting with very accurate camera calibration and position data, while an ideal system would work from small sets of photos taken in an uncontrolled setting.

Finally, we discussed methods to clean up redesigned shapes for export to other systems. Artifacts such as holes, self-intersections, inconsistent face orientations, and more are common in CAD models and poorly-processed scanned models, and our tools for shape redesign may create even more artifacts – such as additional self-intersections if a sweep path is bent sharply. An ideal inverse 3D modeling system would include the tools to correct such problems, ensuring that a clean representation of the shape is available to downstream applications. This kind of model repair is a well studied problem, so we provided a guide to existing approaches – highlighting methods that work well, and giving a breakdown of the tradeoffs between methods in cases where no method was ideal.

Overall, this thesis has presented a thorough guide to the problems and opportunities presented by the idea of user-guided inverse 3D modeling. The promising results of our many prototype inverse 3D modeling modules make the case that this approach to shape redesign is both powerful and computationally reasonable, as most of our fitting modules return useful results in a matter of seconds. Throughout this thesis we have highlighted a number of interesting avenues for future work: New velocity fields could be explored to fit new types of stationary sweep; some of the modules, especially boundary-to-CSG conversion, could be optimized further to handle more complex examples; new separators for boundary-to-CSG conversion could be explored; the sweep-fitting approach in our image-based modeling prototype module could be improved by integrating more user guidance. Another key area for future research and development is the integration of these prototypes into a full shape-modeling software package, with a user-interface that is accessible and convenient for novice and professional users alike. We hope that this thesis inspires other researchers to join us in tackling these wide-ranging, interesting problems, and that our work ultimately culminates in the availability of effective user-guided inverse 3D modeling tool sets in professional 3D shape editing software.

Bibliography

[1] *123D Catch*. 2012. URL: http://www.123dapp.com/catch.

[2] A. Al-Sharadqah and N. Chernov. "Error analysis for circle fitting algorithms". EN. In: *Electronic Journal of Statistics* 3 (2009), pp. 886–911. ISSN: 1935-7524. URL: http://projecteuclid.org/euclid.ejs/1251119958.

[3] S. Allaire, J.-J. Jacq, V. Burdin, C. Roux, and C. Couture. "Type-Constrained Robust Fitting of Quadrics with Application to the 3D Morphological Characterization of Saddle-Shaped Articular Surfaces". In: *ICCV*. IEEE, 2007, pp. 1–8. ISBN: 978-1-4244-1630-1. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?reload=true\&arnumber=4409163.

[4] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. 3rd ed. Society for Industrial and Applied Mathematics, 1999.

[5] M. Attene and B. Falcidieno. "ReMESH: An Interactive Environment to Edit and Repair Triangle Meshes". In: *IEEE International Conference on Shape Modeling and Applications 2006 (SMI'06)*. IEEE, 2006, pp. 41–41. ISBN: 0-7695-2591-1. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1631218.

[6] M. Attene, M. Campen, and L. Kobbelt. "Polygon Mesh Repairing: An Application Perspective". In: *ACM Computing Surveys* 45.2 (Feb. 2013), pp. 1–33. ISSN: 03600300. URL: http://dl.acm.org/citation.cfm?id=2431211.2431214.

[7] P. Benko, R. R. Martin, and T. Vrady. "Algorithms for reverse engineering boundary representation models". In: *Computer-Aided Design* 33.11 (2001), pp. 839–851.

[8] M. Bergou, M. Wardetzky, S. Robinson, B. Audoly, and E. Grinspun. "Discrete elastic rods". In: *ACM SIGGRAPH 2008 papers*. SIGGRAPH '08. New York, NY, USA: ACM, 2008, 63:1–63:12. ISBN: 978-1-4503-0112-1. URL: http://doi.acm.org/10.1145/1399504.1360662.

[9] S. Bischoff and L. Kobbelt. "Structure Preserving CAD Model Repair". In: *Computer Graphics Forum* 24.3 (Sept. 2005), pp. 527–536. ISSN: 01677055. URL: http://doi.wiley.com/10.1111/j.1467-8659.2005.00878.x.

[10] R. L. Bishop. "There is More than One Way to Frame a Curve". In: *The American Mathematical Monthly* 82.3 (1975), pp. 246–251. ISSN: 00029890.

[11] M. Bokeloh, M. Wand, and H.-P. Seidel. "A Connection between Partial Symmetry and Inverse Procedural Modeling". In: *ACM Siggraph*. 2010.

[12] M. Botsch and L. Kobbelt. "An intuitive framework for real-time freeform modeling". In: *ACM Trans. Graph.* 23.3 (Aug. 2004), pp. 630–634. ISSN: 0730-0301.

[13] M. Botsch and O. Sorkine. "On linear variational surface deformation methods." In: *IEEE Transactions on Visualization and Computer Graphics* 14.1 (2008), pp. 213–230. URL: http://www.ncbi.nlm.nih.gov/pubmed/17993714.

[14] S. Bouaziz, M. Deuss, Y. Schwartzburg, T. Weise, and M. Pauly. "Shape-Up: Shaping Discrete Geometry with Projections". In: *Computer Graphics Forum* 31.5 (Aug. 2012), pp. 1657–1667. ISSN: 01677055. URL: http://dl.acm.org/citation.cfm?id=2346796.2346802.

[15] A. Boulch and R. Marlet. "Fast and Robust Normal Estimation for Point Clouds with Sharp Features". In: *Computer Graphics Forum* 31.5 (Aug. 2012), pp. 1765–1774. ISSN: 01677055. URL: http://doi.wiley.com/10.1111/j.1467-8659.2012.03181.x.

[16] S. Brown, B. Morse, and W. Barrett. "Interactive part selection for mesh and point models using hierarchical graph-cut partitioning". In: *Proceedings of Graphics Interface 2009*. GI '09. 2009, pp. 23–30.

[17] *CGAL, Computational Geometry Algorithms Library*. 2013. URL: http://www.cgal.org.

[18] M. Campen and L. Kobbelt. "Exact and Robust (Self-)Intersections for Polygonal Meshes". In: *Computer Graphics Forum* 29.2 (June 2010), pp. 397–406. ISSN: 01677055. URL: http://doi.wiley.com/10.1111/j.1467-8659.2009.01609.x.

[19] C.-C. Chang and C.-J. Lin. "LIBSVM". In: *ACM Transactions on Intelligent Systems and Technology* 2.3 (Apr. 2011), pp. 1–27. ISSN: 21576904. URL: http://dl.acm.org/citation.cfm?id=1961189.1961199.

[20] X. Chen, A. Golovinskiy, and T. Funkhouser. "A benchmark for 3D mesh segmentation". In: *ACM Transactions on Graphics* 28.3 (July 2009), p. 1. ISSN: 07300301. URL: http://dl.acm.org/citation.cfm?id=1531326.1531379.

[21] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam. "Algorithm 887: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate". In: *ACM Trans. Math. Softw.* 35.3 (Oct. 2008), 22:1–22:14. ISSN: 0098-3500. URL: http://doi.acm.org/10.1145/1391989.1391995.

[22] Y. Chen, Z.-Q. Cheng, J. Li, R. R. Martin, and Y.-Z. Wang. "Relief extraction and editing". In: *Comput. Aided Des.* 43.12 (Dec. 2011), pp. 1674–1682. ISSN: 0010-4485.

[23] N. Chernov. "On the Convergence of Fitting Algorithms in Computer Vision". In: *J. Math. Imaging Vis.* 27.3 (Apr. 2007), pp. 231–239. ISSN: 0924-9907.

[24] N. Chernov and H. Ma. "Least squares fitting of quadratic curves and surfaces". In: *Computer Vision*. 2011, pp. 285–302.

[25] D. Corrigan, S. Robinson, and A. Kokaram. "Video Matting Using Motion Extended GrabCut". In: *IET European Conference on Visual Media Production (CVMP)*. London, UK, 2008.

[26] J. Davis, S. Marschner, M. Garr, and M. Levoy. "Filling holes in complex surfaces using volumetric diffusion". English. In: *Proceedings. First International Symposium on 3D Data Processing Visualization and Transmission*. IEEE Comput. Soc, 2002, pp. 428–861. ISBN: 0-7695-1521-5. URL: `http://ieeexplore.ieee.org/xpls/abs\_all.jsp?arnumber=1024098'escapeXml='false'/>`.

[27] P. E. Debevec, C. J. Taylor, and J. Malik. "Modeling and rendering architecture from photographs: a hybrid geometry- and image-based approach". In: *SIGGRAPH '96*. New York, NY, USA: ACM, 1996, pp. 11–20. ISBN: 0-89791-746-4. URL: `http://doi.acm.org/10.1145/237170.237191`.

[28] E. W. Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische Mathematik* 1 (1959), pp. 269–271.

[29] J. Dion D., D. Laurendeau, and R. Bergevin. "Generalized cylinders extraction in a range image". In: *3-D Digital Imaging and Modeling*. 1997, pp. 141–147.

[30] J. Dorman and A. Rockwood. "Surface design using hand motion with smoothing". In: *Computer-Aided Design* 33.5 (Apr. 2001), pp. 389–402. ISSN: 00104485. URL: `http://dx.doi.org/10.1016/S0010-4485(00)00130-5`.

[31] D. Dunavant. "High Degree Efficient Symmetrical Gaussian Quadrature Rules for the Triangle". In: *International Journal for Numerical Methods in Engineering* 21 (1985), pp. 1129–1148.

[32] M. Eck and H. Hoppe. "Automatic reconstruction of B-spline surfaces of arbitrary topological type". In: *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques - SIGGRAPH '96*. New York, New York, USA: ACM Press, Aug. 1996, pp. 325–334. ISBN: 0897917464. URL: `http://dl.acm.org/citation.cfm?id=237170.237271`.

[33] A. Erol, G. Bebis, R. D. Boyle, and M. Nicolescu. "Visual Hull Construction Using Adaptive Sampling". In: *Applications of Computer Vision and the IEEE Workshop on Motion and Video Computing, IEEE Workshop on* 1 (2005), pp. 234–241.

[34] C. H. Esteban and F. Schmitt. "Silhouette and stereo fusion for 3D object modeling." In: *Computer Vision and Image Understanding* (2004), pp. 367–392.

[35] R. Fabbri and B. B. Kimia. "{3D} Curve Sketch: Flexible Curve-Based Stereo Reconstruction and Calibration". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. San Francisco, California, USA: IEEE Computer Society Press, 2010.

[36] L. Fan, L. Lic, and K. Liu. "Paint Mesh Cutting". In: *Computer Graphics Forum* 30.2 (Apr. 2011), pp. 603–612. ISSN: 01677055. URL: http://doi.wiley.com/10.1111/j.1467-8659.2011.01895.x.

[37] L. Fan, M. Meng, and L. Liu. "Sketch-based mesh cutting: A comparative study". In: *Graphical Models* 74.6 (Nov. 2012), pp. 292–301. ISSN: 15240703. URL: http://dx.doi.org/10.1016/j.gmod.2012.03.001.

[38] M. A. Fischler and R. C. Bolles. "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography". In: *Commun. ACM* 24.6 (June 1981), pp. 381–395. ISSN: 0001-0782.

[39] A. Fitzgibbon, M. Pilu, and R. Fisher. "Direct least square fitting of ellipses". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 21.5 (May 1999), pp. 476–480. ISSN: 01628828. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=765658.

[40] A. W. Fitzgibbon, D. W. Eggert, and R. B. Fisher. "High-level CAD Model Acquisition from Range Images". In: *Computer-Aided Design* 29 (1997), pp. 321–330.

[41] A. Fitzgibbon and R. B. Fisher. "A Buyer's Guide to Conic Fitting". In: *British Machine Vision Conference*. 1995, pp. 513–522. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.36.695.

[42] M. S. Floater. "Mean value coordinates". In: *Computer Aided Geometric Design* 20.1 (Mar. 2003), pp. 19–27. ISSN: 01678396. URL: http://dx.doi.org/10.1016/S0167-8396(03)00002-5.

[43] S. F. Frisken and R. N. Perry. "Designing with distance fields". In: *ACM SIGGRAPH 2006 Courses*. SIGGRAPH '06. New York, NY, USA: ACM, 2006, pp. 60–66.

[44] S. F. Frisken, R. N. Perry, A. P. Rockwood, and T. R. Jones. "Adaptively sampled distance fields: a general representation of shape for computer graphics". In: SIGGRAPH '00. 2000, pp. 249–254.

[45] T. Funkhouser, M. Kazhdan, P. Shilane, P. Min, W. Kiefer, A. Tal, S. Rusinkiewicz, and D. Dobkin. "Modeling by example". In: *ACM SIGGRAPH 2004 Papers on - SIGGRAPH '04*. Vol. 23. 3. New York, New York, USA: ACM Press, Aug. 2004, p. 652. URL: http://dl.acm.org/citation.cfm?id=1186562.1015775.

[46] Y. Furukawa and J. Ponce. "Accurate, Dense, and Robust Multiview Stereopsis". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 32.8 (2010), pp. 1362–1376. ISSN: 0162-8828.

[47] Y. Furukawa and J. Ponce. "Carved Visual Hulls for Image-Based Modeling". In: *Int. J. Comput. Vision* 81.1 (2009), pp. 53–67. ISSN: 0920-5691.

[48] N. Gelfand and L. Guibas. "Shape Segmentation Using Local Slippage Analysis". In: *Eurographics Sympoium on Geometry Processing*. 2004.

[49] A. Gfrerrer, J. Lang, A. Harrich, M. Hirz, and J. Mayr. "Car side window kinematics". In: *Computer-Aided Design* 43.4 (Apr. 2011), pp. 410–416. ISSN: 00104485. URL: http://dx.doi.org/10.1016/j.cad.2011.01.009.

[50] Y. Gingold, T. Igarashi, and D. Zorin. "Structured Annotations for 2D-to-3D Modeling". In: *ACM Transactions on Graphics (TOG)* 28.5 (2009), p. 148.

[51] Y. Gingold and D. Zorin. "Shading-based surface editing". In: *ACM SIGGRAPH 2008 papers*. SIGGRAPH '08. New York, NY, USA: ACM, 2008, 95:1–95:9. ISBN: 978-1-4503-0112-1.

[52] G. H. Golub and C. F. Van Loan. *Matrix computations (3rd ed.)* Baltimore, MD, USA: Johns Hopkins University Press, 1996. ISBN: 0-8018-5414-8.

[53] S. Goodman, A. Mellnik, and C. H. Séquin. "Girl's Surface". In: *Bridges*. Enschede, The Netherlands, 2013.

[54] A. Gueziec, G. Taubin, F. Lazarus, and B. Horn. "Cutting and stitching: converting sets of polygons to manifold surfaces". In: *IEEE Transactions on Visualization and Computer Graphics* 7.2 (2001), pp. 136–151. ISSN: 10772626. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=928166.

[55] M. Harker, P. OLeary, and P. Zsombor-Murray. "Direct type-specific conic fitting and eigenvalue bias correction". In: *Image and Vision Computing* 26.3 (Mar. 2008), pp. 372–381. ISSN: 02628856. URL: http://dx.doi.org/10.1016/j.imavis.2006.12.006.

[56] J. C. Hart, D. J. Sandin, and L. H. Kauffman. "Ray tracing deterministic 3-D fractals". In: *SIGGRAPH Comput. Graph.* 23.3 (July 1989), pp. 289–296. ISSN: 0097-8930. URL: http://doi.acm.org/10.1145/74334.74363.

[57] E. E. Hartquist. *BCSG-1.0: A Practical Implementation of Boundary to CSG Conversion*. 1994.

[58] A. van den Hengel and A. Dick. "Image based modelling with VideoTrace". In: *SIGGRAPH Comput. Graph.* 42.2 (2008), pp. 1–8. ISSN: 0097-8930.

[59] M. Hofer, B. Odehnal, H. Pottmann, T. Steiner, and J. Wallner. "3D Shape Recognition and Reconstruction Based on Line Element Geometry". In: *Proceedings of the Tenth IEEE International Conference on Computer Vision - Volume 2*. ICCV '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 1532–1538. ISBN: 0-7695-2334-X-02.

[60] H. Hoppe, T. DeRose, T. Duchamp, M. Halstead, H. Jin, J. McDonald, J. Schweitzer, and W. Stuetzle. "Piecewise smooth surface reconstruction". In: SIGGRAPH '94. 1994, pp. 295–302.

[61] P. J. Huber. *Robust Statistics*. New York: John Wiley and Sons, 1981.

[62] A. Jacobson, L. Kavan, and O. Sorkine-Hornung. "Robust Inside-Outside Segmentation using Generalized Winding Numbers". In: *SIGGRAPH*. 2013. URL: http://igl.ethz.ch/projects/winding-number/.

[63] Z. Ji, L. Liu, Z. Chen, and G. Wang. "Easy Mesh Cutting". In: *Computer Graphics Forum* 25.3 (Sept. 2006), pp. 283–291. ISSN: 0167-7055. URL: http://doi.wiley.com/10.1111/j.1467-8659.2006.00947.x.

[64] P. Joshi, M. Meyer, T. DeRose, B. Green, and T. Sanocki. "Harmonic coordinates for character articulation". In: *ACM Siggraph* (2007).

[65] P. Joshi and C. Sequin. "Energy Minimizers for Curvature-Based Surface Functionals". In: *Computer-Aided Design & Applications* 4.5 (2007), pp. 607–617.

[66] T. Ju. "Robust repair of polygonal models". In: *ACM SIGGRAPH 2004 Papers on - SIGGRAPH '04*. Vol. 23. 3. New York, New York, USA: ACM Press, Aug. 2004, p. 888. URL: http://dl.acm.org/citation.cfm?id=1186562.1015815.

[67] K. Kanatani. "Further improving geometric fitting". In: *Proc. 5th Int. Conf. 3-D Digital Imaging and Modeling*. 2005, pp. 2–13.

[68] K. Kanatani and N. Ohta. "Comparing optimal three-dimensional reconstruction for finite motion and optical flow." In: *J. Electronic Imaging* 12.3 (2003), pp. 478–488.

[69] L. B. Kara, C. M. D'Eramo, and K. Shimada. "Pen-based styling design of 3D geometry using concept sketches and template models". In: *Proceedings of the 2006 ACM symposium on Solid and physical modeling*. SPM '06. New York, NY, USA: ACM, 2006, pp. 149–160. ISBN: 1-59593-358-1. URL: http://doi.acm.org/10.1145/1128888.1128909.

[70] O. A. Karpenko and J. F. Hughes. "Smoothsketch: 3D free-form shapes from complex sketches". In: *ACM Trans. Graph* 25 (2006), pp. 589–598.

[71] M. Kazhdan, A. Klein, K. Dalal, and H. Hoppe. "Unconstrained isosurface extraction on arbitrary octrees". In: *Proceedings of the fifth Eurographics symposium on Geometry processing*. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2007, pp. 125–133. ISBN: 978-3-905673-46-3. URL: http://portal.acm.org/citation.cfm?id=1281991.1282009.

[72] F. Kirsch and J. Döllner. "OpenCSG: a library for image-based CSG rendering". In: *Proceedings of the FREENIX / Open Source Track, 2005 USENIX Annual Technical Conference*. Apr. 2005, pp. 129–140. URL: http://dl.acm.org/citation.cfm?id=1247360.1247409.

[73] L. Kobbelt, S. Campagna, J. Vorsatz, and H.-P. Seidel. "Interactive multi-resolution modeling on arbitrary meshes". In: *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*. SIGGRAPH '98. New York, NY, USA: ACM, 1998, pp. 105–114. ISBN: 0-89791-999-8. URL: http://doi.acm.org/10.1145/280814.280831.

[74] V. Kraevoy, A. Sheffer, and M. van de Panne. "Modeling from contour drawings". In: *Eurographics Symposium on Sketch-Based Interfaces and Modeling*. SBIM '09. New York, NY, USA: ACM, 2009, pp. 37–44. ISBN: 978-1-60558-602-1.

[75] V. Krishnamurthy and M. Levoy. "Fitting smooth surfaces to dense polygon meshes". In: SIGGRAPH '96. 1996, pp. 313–324.

[76] Y.-K. Lai, S.-M. Hu, R. R. Martin, and P. L. Rosin. "Fast mesh segmentation using random walks". In: *Proceedings of the 2008 ACM symposium on Solid and physical modeling - SPM '08*. New York, New York, USA: ACM Press, June 2008, p. 183. ISBN: 9781605581062. URL: http://dl.acm.org/citation.cfm?id=1364901.1364927.

[77] A. Laurentini. "The Visual Hull Concept for Silhouette-Based Image Understanding". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 16.2 (1994), pp. 150–162. ISSN: 0162-8828.

[78] Y. Leedan and P. Meer. "Heteroscedastic Regression in Computer Vision: Problems with Bilinear Constraint". In: *Int. J. Comput. Vision* 37.2 (June 2000), pp. 127–150. ISSN: 0920-5691.

[79] M. Li, F. C. Langbein, and R. R. Martin. "Detecting design intent in approximate CAD models using symmetry". In: *Computer-Aided Design* 42.3 (2010), pp. 183–201. ISSN: 0010-4485.

[80] Q. Li and J. Griffiths. "Least squares ellipsoid specific fitting". In: *Geometric Modeling and Processing*. IEEE, 2004, pp. 335–340. ISBN: 0-7695-2078-2. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1290055.

[81] Y. Li, X. Wu, Y. Chrysathou, A. Sharf, D. Cohen-Or, and N. J. Mitra. "GlobFit: consistently fitting primitives by discovering global relations". In: *ACM Trans. Graph.* 30.4 (July 2011), 52:1–52:12. ISSN: 0730-0301.

[82] Y. Li, J. Sun, C.-K. Tang, and H.-Y. Shum. "Lazy snapping". In: *ACM SIGGRAPH 2004 Papers*. SIGGRAPH '04. New York, NY, USA: ACM, 2004, pp. 303–308. URL: http://doi.acm.org/10.1145/1186562.1015719.

[83] W.-C. Lin and T.-W. Chen. "CSG-based object recognition using range images". In: *9th International Conference on Pattern Recognition*. IEEE Comput. Soc. Press, 1988, pp. 99–103. ISBN: 0-8186-0878-1. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=28180.

[84] R. Ling, W. Wang, and D. Yan. "Fitting Sharp Features with Loop Subdivision Surfaces". In: *Computer Graphics Forum* 27.5 (July 2008), pp. 1383–1391. ISSN: 01677055. URL: http://doi.wiley.com/10.1111/j.1467-8659.2008.01278.x.

[85] Y. Liu, H. Pottmann, and W. Wang. "Constrained 3D shape reconstruction using a combination of surface fitting and registration". In: *Computer-Aided Design* 38.6 (2006), pp. 572–583.

[86] G. Lukács, R. Martin, and D. Marshall. "Faithful Least-Squares Fitting of Spheres, Cylinders, Cones and Tori for Reliable Segmentation". In: *ECCV*. June 1998, pp. 671–686. ISBN: 3-540-64569-1. URL: http://dl.acm.org/citation.cfm?id=645311.649078.

[87] K. Madsen, H. B. Nielsen, and O. Tingleff. *Methods for Non-Linear Least Squares Problems (2nd ed.)* 2004.

[88] T. Masuda. "Filling the signed distance field by fitting local quadrics". English. In: *Proceedings. 2nd International Symposium on 3D Data Processing, Visualization and Transmission, 2004. 3DPVT 2004.* IEEE, 2004, pp. 1003–1010. ISBN: 0-7695-2223-8. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1335425'escapeXml='false'/>.

[89] M. Meng, L. Fan, and L. Liu. "A comparative evaluation of foreground/background sketch-based mesh segmentation algorithms". In: *Computers & Graphics* 35.3 (June 2011), pp. 650–660. ISSN: 00978493. URL: http://dx.doi.org/10.1016/j.cag.2011.03.038.

[90] M. Meng, L. Fan, and L. Liu. "iCutter: a direct cut-out tool for 3D shapes". In: *Computer Animation and Virtual Worlds* 22.4 (July 2011), pp. 335–342. ISSN: 15464261. URL: http://doi.wiley.com/10.1002/cav.422.

[91] *MeshLab*. 2013. URL: http://meshlab.sourceforge.net/.

[92] M. Meyer, M. Desbrun, P. Schröder, and A. H. Barr. "Discrete differential-geometry operators for triangulated 2-manifolds". In: *Visualization and Mathematics III* (2003), pp. 35–57.

[93] N. J. Mitra, L. J. Guibas, and M. Pauly. "Partial and approximate symmetry detection for 3D geometry". In: *ACM Transactions on Graphics* 25.3 (July 2006), p. 560. ISSN: 07300301. URL: http://dl.acm.org/citation.cfm?id=1141911.1141924.

[94] N. J. Mitra, M. Pauly, M. Wand, and D. Ceylan. "Symmetry in 3D Geometry: Extraction and Applications". In: *Computer Graphics Forum* (Feb. 2013), no–no. ISSN: 01677055. URL: http://doi.wiley.com/10.1111/cgf.12010.

[95] T. M. Murali and T. A. Funkhouser. "Consistent solid and boundary representations from arbitrary polygonal data". In: *Proceedings of the 1997 symposium on Interactive 3D graphics - SI3D '97*. New York, New York, USA: ACM Press, Apr. 1997, 155–ff. ISBN: 0897918843. URL: http://dl.acm.org/citation.cfm?id=253284.253326.

[96] A. Nealen, T. Igarashi, O. Sorkine, and M. Alexa. "FiberMesh: designing freeform surfaces with 3D curves". In: *ACM Trans. Graph.* 26.3 (2007), p. 41. ISSN: 0730-0301.

[97] A. Nealen and O. Sorkine. *A Note on Boundary Constraints for Linear Variational Surface Design*. Tech. rep. TU Berlin, 2007.

[98] G. M. Nielson and B. Hamann. "The asymptotic decider: resolving the ambiguity in marching cubes". In: (Oct. 1991), pp. 83–91. URL: http://dl.acm.org/citation.cfm?id=949607.949621.

[99] F. Nooruddin and G. Turk. "Simplification and repair of polygonal models using volumetric techniques". In: *IEEE Transactions on Visualization and Computer Graphics* 9.2 (Apr. 2003), pp. 191–205. ISSN: 1077-2626. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1196006.

[100] M. Pauly, N. J. Mitra, J. Wallner, H. Pottmann, and L. J. Guibas. "Discovering structural regularity in 3D geometry". In: *ACM Trans. Graph.* 27.3 (Aug. 2008), 43:1–43:11. ISSN: 0730-0301. URL: http://doi.acm.org/10.1145/1360612.1360642.

[101] K. Pearson. "On lines and planes of closest fit to systems of points in space". In: *Philosophical Magazine*. 6th ser. 2.6 (1901), pp. 559–572. ISSN: 19415982. URL: http://stat.smmu.edu.cn/history/pearson1901.pdf.

[102] J. Peng, D. Kristjansson, and D. Zorin. "Interactive modeling of topologically complex geometric detail". In: *ACM Trans. Graph.* 23.3 (Aug. 2004), pp. 635–643. ISSN: 0730-0301.

[103] S. Petitjean. "A survey of methods for recovering quadrics in triangle meshes". In: *ACM Computing Surveys* 34.2 (June 2002), pp. 211–262. ISSN: 03600300. URL: http://dl.acm.org/citation.cfm?id=508352.508354.

[104] U. Pinkall and K. Polthier. "Computing Discrete Minimal Surfaces and Their Conjugates". In: *Experimental Mathematics* 2.1 (1993), pp. 15–36.

[105] J. Podolak and S. Rusinkiewicz. "Atomic volumes for mesh completion". In: (July 2005), p. 33. URL: http://dl.acm.org/citation.cfm?id=1281920.1281926.

[106] H. Pottmann, H.-Y. Chen, and I. K. Lee. "Approximation by Profile Surfaces". In: *The Mathematics of Surfaces VIII* (1998), pp. 17–36.

[107] H. Pottmann, I.-K. Lee, and T. Randrup. "Reconstruction of kinematic surface from scattered data". In: *Proceedings of Symposium on Geodesy for Geotechnical and Structural Engineering* (1998), pp. 483–488.

[108] H. Pottmann and T. Randrup. "Rotational and helical surface approximation for reverse engineering". In: *Computing* 60.4 (June 1998), pp. 307–322.

[109] H. Pottmann and J. Wallner. *Computational Line Geometry*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2001. ISBN: 3540420584.

[110] M. Prasad, A. Zisserman, and A. W. Fitzgibbon. "Fast and Controllable 3D Modelling from Silhouettes". In: *Annual Conference of the European Association for Computer Graphics (Eurographics)*. 2005, pp. 9–12. URL: http://marcade.robots.ox.ac.uk:8080/~vgg/publications/2005/Prasad05.

[111] V. Pratt. "Direct least-squares fitting of algebraic surfaces". In: *SIGGRAPH Comput. Graph.* 21.4 (Aug. 1987), pp. 145–152. ISSN: 0097-8930.

[112] A. I. Protopsaltis and I. Fudos. "A feature-based approach to re-engineering CAD models from cross sections". In: *Computer-Aided Design and Applications* 7.5 (2010), pp. 739–757.

[113] S. Raghothama and V. Shapiro. "Consistent updates in dual representation systems". In: *Computer-Aided Design* 32.8-9 (Aug. 2000), pp. 463–477. ISSN: 00104485. URL: http://dx.doi.org/10.1016/S0010-4485(00)00036-1.

[114] R. Ramamoorthi and J. Arvo. "Creating Generative Models from Range Images". In: *ACM Siggraph*. 1999.

[115] T. Randrup. "Approximation by cylinder surfaces". In: *Computer Aided Design* 30 (1998), pp. 807–812.

[116] P. Rangarajan, K. Kanatani, H. Niitsuma, and Y. Sugaya. "Hyper Least Squares and Its Applications". In: *2010 20th International Conference on Pattern Recognition*. IEEE, Aug. 2010, pp. 5–8. ISBN: 978-1-4244-7542-1. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5597662.

[117] A. Rivers, F. Durand, and T. Igarashi. "3D modeling with silhouettes". In: *ACM Trans. Graph.* 29.4 (2010), pp. 1–8. ISSN: 0730-0301.

[118] C. Rother, V. Kolmogorov, and A. Blake. ""GrabCut": interactive foreground extraction using iterated graph cuts". In: *ACM SIGGRAPH 2004 Papers*. SIGGRAPH '04. New York, NY, USA: ACM, 2004, pp. 309–314. URL: http://doi.acm.org/10.1145/1186562.1015720.

[119] P. D. Sampson. "Fitting conic sections to very scattered data: An iterative refinement of the Bookstein algorithm". In: *Computer Graphics and Image Processing* 18.1 (Jan. 1982), pp. 97–108. ISSN: 0146664X. URL: http://dx.doi.org/10.1016/0146-664X(82)90101-0.

[120] R. Schmidt. *MeshMixer*. 2013. URL: http://www.meshmixer.com/.

[121] R. Schmidt and B. Wyvill. "Generalized sweep templates for implicit modeling". In: *Proeedings of GRAPHITE 2005*. GRAPHITE '05. New York, NY, USA: ACM, 2005, pp. 187–196. ISBN: 1-59593-201-1.

[122] T. W. Sederberg and S. R. Parry. "Free-form deformation of solid geometric models". In: *ACM SIGGRAPH Computer Graphics* 20.4 (Aug. 1986), pp. 151–160. ISSN: 00978930. URL: http://dl.acm.org/citation.cfm?id=15886.15903.

[123] C. Séquin. "Virtual Prototyping of Scherk-Collins Saddle Rings". In: *Leonardo* 30.2 (1997), pp. 89–96.

[124] J. A. Sethian. "A Fast Marching Level Set Method for Monotonically Advancing Fronts". In: *Proc. Nat. Acad. Sci.* 93.4 (1996), pp. 1591–1595.

[125] V. Shapiro and D. L. Vossler. "Efficient CSG Representations of Two-Dimensional Solids". In: *Journal of Mechanical Design* 113.3 (Sept. 1991), p. 292. ISSN: 10500472. URL: http://mechanicaldesign.asmedigitalcollection.asme.org/article.aspx?articleid=1443058.

[126] V. Shapiro and D. L. Vossler. "Construction and optimization of CSG representations". In: *Computer-Aided Design* 23.1 (Jan. 1991), pp. 4–20. ISSN: 00104485. URL: http://dl.acm.org/citation.cfm?id=115604.115605.

[127] V. Shapiro and D. L. Vossler. "Separation for boundary to CSG conversion". In: *ACM Transactions on Graphics* 12.1 (Jan. 1993), pp. 35–55. ISSN: 07300301. URL: http://dl.acm.org/citation.cfm?id=169728.169723.

[128] C. Shen, J. F. O'Brien, and J. R. Shewchuk. "Interpolating and approximating implicit surfaces from polygon soup". In: *ACM SIGGRAPH 2004 Papers*. Vol. 23. 3. New York, New York, USA: ACM Press, Aug. 2004, p. 896. URL: http://dl.acm.org/citation.cfm?id=1186562.1015816.

[129] H. Si. *TetGen*. Berlin, Germany, 2011. URL: http://tetgen.org.

[130] S. N. Sinha, D. Steedly, R. Szeliski, M. Agrawala, and M. Pollefeys. "Interactive 3D architectural modeling from unordered photo collections". In: *ACM Trans. Graph.* 27.5 (2008), pp. 1–10. ISSN: 0730-0301.

[131] O. Sorkine and M. Alexa. "As-rigid-as-possible surface modeling". In: (July 2007), pp. 109–116. URL: http://dl.acm.org/citation.cfm?id=1281991.1282006.

[132] O. Sorkine, D. Cohen-Or, Y. Lipman, M. Alexa, C. Rössl, and H.-P. Seidel. "Laplacian Surface Editing". In: *Proceedings of the Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*. 2004, pp. 179–188.

[133] R. Szeliski. "Rapid Octree Construction from image sequences". In: *CVGIP: Image Understanding* 58.1 (1993), pp. 149–156.

[134] A. Tagliasacchi, H. Zhang, and D. Cohen-Or. "Curve skeleton extraction from incomplete point cloud". In: *ACM Trans. Graph.* 28.3 (July 2009), pp. 1–9. ISSN: 0730-0301.

[135] K. Takayama, R. Schmidt, K. Singh, T. Igarashi, T. Boubekeur, and O. Sorkine. "GeoBrush: Interactive Mesh Geometry Cloning". In: *Computer Graphics Forum* 30.2 (Apr. 2011), pp. 613–622. ISSN: 1467-8659. URL: http://doi.wiley.com/10.1111/j.1467-8659.2011.01883.x.

[136] K. Tang and T. Woo. "Algorithmic aspects of alternating sum of volumes. Part 1: Data structure and difference operation". In: *Computer-Aided Design* 23.5 (June 1991), pp. 357–366. ISSN: 00104485. URL: http://dx.doi.org/10.1016/0010-4485(91)90029-V.

[137] G. Taubin. "Estimation Of Planar Curves, Surfaces And Nonplanar Space Curves Defined By Implicit Equations, With Applications To Edge And Range Image Segmentation". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 13 (Nov. 1991), pp. 1115–1138.

[138] *The Stanford 3D Scanning Repository*. 2011. URL: `http://graphics.stanford.edu/data/3Dscanrep/` (visited on 2013).

[139] T. Thormählen and H.-P. Seidel. "3D-modeling by ortho-image generation from image sequences". In: *SIGGRAPH '08*. New York, NY, USA: ACM, 2008, pp. 1–5. ISBN: 978-1-4503-0112-1.

[140] W.-D. Ueng, J.-Y. Lai, and J.-L. Doong. "Sweep-surface reconstruction from Three-Dimensional Measured Data". In: *Computer-Aided Design* 30.10 (1998), pp. 791–805.

[141] H. Ugail, M. I. G. Bloor, and M. J. Wilson. "Techniques for interactive design using the PDE method". In: *ACM Trans. Graph.* 18.2 (1999), pp. 195–212.

[142] T. Várady, M. A. Facello, and Z. Terék. "Automatic extraction of surface structures in digital shape reconstruction". In: *Comput. Aided Des.* 39.5 (May 2007), pp. 379–388. ISSN: 0010-4485.

[143] T. Várady, R. Martin, and J. Cox. "Reverse Engineering of Geometric Models - An Introduction". In: *Computer Aided Design* 29 (1997), pp. 255–268.

[144] D. Vlasic, I. Baran, W. Matusik, and J. Popović. "Articulated mesh animation from multi-view silhouettes". In: *ACM Trans. Graph.* 27.3 (2008), pp. 1–9. ISSN: 0730-0301.

[145] W. Wang, B. Jüttler, D. Zheng, and Y. Liu. "Computation of rotation minimizing frames". In: *ACM Trans. Graph.* 27.1 (Mar. 2008), 2:1–2:18. ISSN: 0730-0301.

[146] C. Wu. *VisualSFM: A Visual Structure from Motion System*. 2011. URL: `http://homes.cs.washington.edu/~ccwu/vsfm/`.

[147] H. Wu and Y. Yu. "Photogrammetric reconstruction of free-form objects with curvilinear structures". In: *The Visual Computer* 21.4 (2005), pp. 203–216. ISSN: 0178-2789. URL: `http://dx.doi.org/10.1007/s00371-005-0281-7`.

[148] C. Xiao, H. Fu, and C.-L. Tai. "Hierarchical aggregation for efficient shape extraction". In: *The Visual Computer* 25.3 (Apr. 2008), pp. 267–278. ISSN: 0178-2789. URL: `http://link.springer.com/10.1007/s00371-008-0226-z`.

[149] D.-M. Yan, Y. Liu, and W. Wang. "Quadric surface extraction by variational shape approximation". In: *Proceedings of the 4th international conference on Geometric Modeling and Processing*. GMP'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 73–86. ISBN: 3-540-36711-X, 978-3-540-36711-6.

[150] K. Yin, Y. Liu, and E. Wu. "Fast Computing Adaptively Sampled Distance Field on GPU". In: *Pacific Graphics 2011*. 2011, pp. 25–30. ISBN: 978-3-905673-84-5. URL: `http://diglib.eg.org/EG/DL/PE/PG/PG2011short/025-030.pdf.abstract.pdf;internal\&action=action.digitallibrary.ShowPaperAbstract`.

[151] S.-H. Yoon and M.-S. Kim. "Sweep-based Freeform Deformations". In: *Computer Graphics Forum* 25.3 (2006), pp. 487–496. ISSN: 1467-8659.

[152] L. Zhang, G. Dugas-phocion, J.-s. Samson, and S. M. Seitz. "Single View Modeling of Free-Form Scenes". In: *In Proc. of CVPR*. 2002, pp. 990–997.

[153] H. Zhao, C. C. L. Wang, Y. Chen, and X. Jin. "Parallel and efficient Boolean on polygonal solids". In: *The Visual Computer* 27.6-8 (Apr. 2011), pp. 507–517. ISSN: 0178-2789. URL: http://link.springer.com/10.1007/s00371-011-0571-1.

[154] Y. Zheng and C.-L. Tai. "Mesh Decomposition with Cross-Boundary Brushes". In: *Computer Graphics Forum* 29.2 (June 2010), pp. 527–535. ISSN: 01677055. URL: http://doi.wiley.com/10.1111/j.1467-8659.2009.01622.x.

[155] Y. Zheng, C.-L. Tai, and O. K.-C. Au. "Dot scissor: a single-click interface for mesh segmentation." In: *IEEE transactions on visualization and computer graphics* 18.8 (Aug. 2012), pp. 1304–12. ISSN: 1941-0506. URL: http://www.computer.org/csdl/trans/tg/2012/08/05989803-abs.html.

[156] J. Zimmermann, A. Nealen, and M. Alexa. "SilSketch: automated sketch-based editing of surface meshes". In: *Proceedings of the 4th Eurographics workshop on Sketch-based interfaces and modeling*. SBIM '07. New York, NY, USA: ACM, 2007, pp. 23–30. ISBN: 978-1-59593-915-9.

[157] J. Zimmermann, A. Nealen, and M. Alexa. "Sketching contours." In: *Computers and Graphics* (2008).