Algorithms and Interfaces for Sketch-Based 3D Modeling

by
Olga Karpenko
B. S., Moscow State University, 1998
Sc. M., Brown University, 2001

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island
May 2007

This dissertation by Olga Karpenko is accepted in its present form by
the Department of Computer Science as satisfying the dissertation requirement
for the degree of Doctor of Philosophy.

Date _____ _____

John F. Hughes, Director

Recommended to the Graduate Council

Date _____ _____

Gabriel Taubin, Reader
(Department of Engineering and Computer Science)

Date _____ _____

Ramesh Raskar, Reader
(Mitsubishi Electric Research Laboratories)

Approved by the Graduate Council

Date _____ _____

Sheila Bonde
Dean of the Graduate School

Vita

Olga Karpenko was born on September 2nd, 1976 in Kazan, Russia. When she was 4 years old, her family moved to Chernogolovka, a small scientific town in the Moscow region of Russia. She went to Chernogolovka School 82 for both Middle and High school degrees. After finishing high school, she joined the department of Applied Mathematics and Cybernetics at Moscow State University where she got her Diploma in 1998. After spending a year as a Ph.D. student in the department of Applied Mathematics and Cybernetics at Moscow State University, Olga started her Ph.D. in Computer Science at Brown University. She got a Sc.M degree from Brown in May 2001. She spent the summer of 2001 working as an intern at Mitsubishi Electric Research Labs in Boston, MA.

Credits

This dissertation is based on several papers that were written jointly with John F. Hughes and Ramesh Raskar. Chapter 3 is based on the SIGGRAPH 2006 paper [56]. Chapter 4 is based on the Eurographics 2002 paper [54], and Chapter 5 – on the paper published at Eurographics SBIM-2004 workshop [55]. This research has been supported in part by a gift from Pixar Animation Studios.

Acknowledgements

This dissertation would not be possible without the support and help of my mentors, family and friends.

I have been very lucky to work with my advisor, John Hughes. His support and encouragement helped me get over the bumps on the long Ph.D. road. I have learned so much from him: how to do research, how to give clear talks, how to be a good teacher and how to be an ethical scientist. Our discussions about differential geometry and topology were so much fun, I will never forget them. From the bottom of my heart, I would like to thank John for everything.

I would also like to thank my committee members Gabriel Taubin and Ramesh Raskar for their insightful feedback about my research.

My advisor at Moscow State University, Yuri Bayakovskiy, introduced me to the beautiful field of computer graphics and I was hooked ever since.

I thank many Brown faculty members for creating a great atmosphere at the department, interesting classes, career advice and feedback about my research and teaching.

Thanks to my fellow graphics Ph.D. students who also have become good friends: Tomer Moscovich, Peter Sibley, and Liz Marai. I could not ask for a better officemate than Tomer - friendly, funny, and always ready to help. Thanks to you and Peter for many useful discussions about my research. Thanks to the friends at Brown without whom the Ph.D. would not be nearly as much fun - Nesime Tatbul, Melih Bitim, Lijuan Cai, Stefan Roth, Joe Lee, Gopal Pandurangan, Will Headden, Yotam Gingold, Daniel Acevedo, and Ying Xing.

I dedicate this dissertation to my family. My father, Anatoly Karpenko, who is a Professor at Bauman Moscow State Technical University, has inspired me in so many ways. From my childhood on, he spent hours and hours with me talking about exciting science ideas, explaining math, physics and CS concepts, discussing everything from philosophy to calculus. He encouraged me to enter the field of Applied Mathematics and Computer

Science. He supported me throughout my Ph.D. studies, providing me with insightful feedback and good suggestions. My mother has been very supportive throughout my studies, always giving moral support, believing more in me than I would myself. I would also like to thank my dear sister Daria, my parents-in-law Balbir and Santosh Ram, and brother-in-law Deepak. My dear husband Prabhat had to endure the ups and downs of my Ph.D. path, supporting me throughout, giving advice and living on the crazy graduate student schedule with me. Thank you for the love and support.

To my dear parents Anatoly and Raisa Karpenko,
my husband Prabhat and my sister Daria.

Contents

List of Figures **x**

1 Introduction **1**

1.1 Sketch-Based Modeling Interfaces 2

1.1.1 What is a sketch? 3

1.1.2 SmoothSketch: 3D free-from shapes from complex contours 4

1.1.3 Free-form sketching using variational implicit surfaces 9

1.1.4 Epipolar methods for multi-view sketching of generalized cylinders . 11

1.2 Contributions 12

2 Related Work **14**

2.1 Visual Perception of Shapes 14

2.2 Shape from Drawings 16

2.2.1 Creating CAD-like 3D models from 2D line drawings 16

2.2.2 Creating smooth 3D shapes from 2D line drawings 17

2.2.3 Creating 2.5-D representations from 2D line drawings 17

2.3 Contour Completion 18

2.4 Sketch-Based Modeling Interfaces 19

2.4.1 Sketching rectilinear objects 19

2.4.2 Sketching free-form objects 20

2.4.3 Sketching 3D curves 21

2.4.4 Sketch-based editing of existing meshes 21

2.4.5 Sketching pseudo-3D models 21

2.4.6 Specialized sketch-based modeling interfaces 23

2.5 Other Sketch-Based Interfaces 24

2.5.1 Mathematical sketching 24

2.5.2 Sketching animation . . . 24
2.6 Other Shape-from-X Techniques . . . 24
2.7 Mesh Optimization . . . 25
2.7.1 Remeshing . . . 25
2.7.2 Mesh smoothing . . . 26
2.7.3 Minimizing energy functionals for shape design . . . 26

3 SmoothSketch: 3D Free-form Shapes from Complex Contours 28
3.1 Overview and Background . . . 29
3.2 Notation and Problem Formulation . . . 32
3.3 The Flow of Operations in the System . . . 35
3.4 Figural Completion for Smooth Surfaces . . . 35
3.4.1 Preprocessing an input stroke . . . 37
3.4.2 Guessing T-points and cusps . . . 38
3.4.3 Pairwise completion . . . 38
3.4.4 Gluing segments and assigning Huffman's labels . . . 45
3.4.5 Results and Limitations of the figural completion algorithm . . . 46
3.4.6 Completable drawings . . . 48
3.5 From Drawing to Topological Embedding . . . 50
3.5.1 Creating 2D panels from a labeled drawing . . . 51
3.5.2 Triangulating the panels; the issue of two distinct points having the same 2D location . . . 52
3.5.3 Paneling construction . . . 53
3.6 Constructing a Topological Embedding . . . 56
3.6.1 Inflation of the objects where some strokes have no tees or cusps . . . 58
3.7 Smoothing the Embedding . . . 59
3.7.1 Using the mass-spring system . . . 59
3.7.2 Evolver: minimizing the discrete squared mean curvature of the mesh 60
3.8 Editing Gestures . . . 62
3.9 Discussion, Limitations, Conclusions . . . 65

4 Free-form Sketching using Variational Implicit Surfaces 67
4.1 Overview of Operations . . . 68
4.2 Surface Representation . . . 71
4.3 Implementation Details . . . 74

4.3.1 Inflation . . . 74
4.3.2 Hierarchy . . . 78
4.3.3 Merging . . . 80
4.3.4 Small modifications on the blobs . . . 83
4.4 Related Approaches . . . 85
4.5 Results, Limitations, and Future Work . . . 86
4.5.1 Future work . . . 87

5 Multi-View Sketching **88**
5.1 Epipolar Methods for Multi-view Sketching of Generalized Cylinders . . . 88
5.1.1 Epipolar geometry . . . 89
5.1.2 Multi-view sketching: the user perspective . . . 89
5.1.3 Interpolation between the Cross-sections . . . 93
5.1.4 Typical Results . . . 94
5.1.5 Problems and Limitations . . . 95
5.1.6 Simple Geometry Editing . . . 95
5.2 Discussion and Future Work . . . 97

6 Discussion and Conclusions **101**
6.1 Contributions . . . 102
6.1.1 Intuitive 3D modeling interfaces . . . 102
6.1.2 Contour completion as a step on the way to object recognition . . . 102
6.2 Limitations and Directions for Future Work . . . 103
6.2.1 Contour completion . . . 106
6.2.2 Inflation . . . 109

Bibliography **114**

List of Figures

1.1 Maya's interface for editing 3D curves [3]. 2

1.2 A snapshot of Teddy: a sketch-based interface for modeling free-form shapes [47]. Compare the simplicity of this interface to Maya's. *Image courtesy of Takeo Igarashi.* 3

1.3 This drawing of Spot the dog [41] shows how even simple contour drawings convey shape very well. ©*Eric Hill, 2006. All rights reserved. Reproduced by permission of Frederick Warne and Co., Ltd.* 4

1.4 The visible contour drawing and the 3D shape inferred by SmoothSketch. . 4

1.5 A drawing with the tee-points and cusps marked. 5

1.6 Contour components: T-junctions and cusps. 5

1.7 The problem of inferring a 3D shape from a 2D drawing is under-constrained. The circle could be an outline of any of these objects. 6

1.8 (a) The famous Penrose's triangle - impossible polyhedral object. (b) A smoothed version of the classic impossible triangle is actually the contour projection of the real surface (c), in contrast to the polyhedral case. 7

1.9 (First row) Huffman's labeling scheme for contours of generic smooth projections. Labels indicate the number of surfaces in front of the contour (visible contours have label zero); the surface on which the contour lies is to the left when you traverse it in the direction shown by the arrow. (Second row) For each case of Huffman's labeling we show the corresponding surfaces. 8

1.10 A constructive drawing technique is used for teaching children how to draw animals by first sketching ovals and circles, and then connecting them with guidance strokes to refine shapes [6]. 9

1.11 (a) A user sketches an oval stroke corresponding to the body of an animal, which (b) is "inflated" by the system into a 3D blob. (c) Two other strokes overlapping the body generate the legs attached to the body. 10

1.12 A user draws a curve on the image plane. When the virtual camera is rotated, epipolar lines appear as yellow segments at each point of the stroke. The user redraws the selected curve from a new point of view by specifying a green stroke. The green stroke is projected on the epipolar lines and the coordinates of the points of the original curve are updated to satisfy the new constraints. 11
1.13 A user clicks on the first segment of the 3D curve created in Figure **??** and a gray square shows up on which the user draws a green cross-section stroke. Then, the user sketches another cross-section stroke for the last segment of the curve - this step is not shown. The cross-section strokes are interpolated along the curve to create a generalized cylinder displayed in the center and right figures as viewed from different perspectives. 12

2.1 A famous optical illusion - Kanizsa's triangle. We perceive the edgeless triangle that appears to be brighter than the background. 16
2.2 The problem of contour completion. 18
2.3 A user oversketches a camel's lip by first, selecting the region of influence (shown in blue), and then drawing a yellow reference silhouette curve and a green target curve. The positional constraints are satisfied only approximately [78]. Image courtesy of Andrew Nealen. 21
2.4 (a) A sketch of a plaza is projected onto a viewer-centered unit sphere. (b) Two views generated by Tolba's system from this drawing [97]. Image courtesy of Julie Dorsey. 22
2.5 An example of artistic illustration; two views are generated from the same sketch [16]. Image courtesy of David Bourguignon. 22
2.6 From left to right: normal image illuminated by diffuse spheremap; the original cartoon drawing, and the final re-lighted cat image [50]. Image courtesy of Scott Johnston. 23

3.1 Huffman's labeling scheme for contours of generic smooth projections. Labels indicate the number of surfaces in front of the contour (visible contours have label zero); the surface on which the contour lies is to the left when you traverse it in the direction shown by the arrow. *Any* cusped knot diagram—i.e., collection of circles in the plane, possibly intersecting themselves and each other, and smoothly immersed except at finitely many cusp points, which are distinct from the crossing points—that can be so labeled corresponds to the projection of a smooth surface in 3-space (mostly proved by Williams), and all generic projections of smooth surfaces have this property (proved by Huffman). The first picture shows the surfaces corresponding to the first case of Huffman's labeling (the second picture). Figure **??** (left) corresponds to the last case. 30

3.2 None of these drawings can be extended by invisible contours to be the contour set of any good manifold projection. They exhibit two problems: in the first, a cusp appears in the outer region of the plane surrounding the figure. In the second, the outermost path around the drawing is clockwise. Although the third has neither of these problems, it is still not extendable. 31

3.3 How can we join the two tee-points on the left? With an optimal completion, as shown in the middle. Optimality is determined by choosing, among all C^1 random walks from p_1 to p_2, the most likely one, under a simple probabilistic model. Mumford shows such curves are elastica, which had been studied by Euler [29]. 31

3.4 The user draws the visible contours of a shape; our program infers the hidden contours, including hidden cusps, and then creates a fairly smooth 3D shape matching those contours. The 3D shape can be viewed from any direction. . 32

3.5 (Adapted from Williams' [107]) The contour, in blue, of a good surface embedded generically in 3-space projects to a contour drawing, in green; the visible contour (drawn bold) projects to the visible-contour drawing. A point where the projector is tangent to the contour projects to a cusp in the contour drawing. The restriction of the projection to just the contour is 1-1 except at finitely many points, where two contours cross in the drawing; these are called T-points. 33

3.6 (Left) The generic projection of a contour at a cusp reverses direction at the cusp. (Right) A drawing with the tee-points and cusps marked; hidden contours and hidden cusps that must be inferred are shown in dotted lines. 34

3.7 This cartoon-like illustration shows us how even the simplest drawings can have complex contours. 35
3.8 The flow of operations in SmoothSketch. Pink ovals marked with a letter U represent the places in the pipeline where the user can interfere. 36
3.9 SmoothSketch expects oriented contours as input — the user should draw contours so that the surface is to the left of the contour. Thus, to create a donut, the user draws a counter-clockwise outer stroke and a clockwise inner stroke (top row). If the inner stroke was drawn in a counter-clockwise manner, the system would create two blobs one in front of the other (bottom row). 37
3.10 (Left) Endpoints 1 and 2 are visible cusps, while points 3 and 4 are "pseudo-cusps": regular points that should be later glued together to lie on a single stroke. (Right) We compute a vector b as a vector connecting endpoints 3 and 4, vector a as connecting the point previous to 3 and 3, and vector c as connecting 4 and the point next to it in the stroke. We classify endpoints as "regular" if the distance between them is small, and the angles between vectors (b, a), and vectors (b, c) are less than epsilon. 38
3.11 The Figure shows how tangent directions and bezier completion curves are computed when two tee-points, two regular endpoints or two cusps are paired up. 39
3.12 The energy of the polyline approximating the Bézier spline is computed as a product of the sum of angle changes between the consecutive segments and the exponent of the sum of the segment lengths. 40
3.13 When we have a T-point and a cusp to match it to, we seek the location of a hidden cusp such that the two hidden contour parts joining our points to the hidden cusp have the highest probability. 40
3.14 (Left) Given a tee point T, a cusp C and tangent vectors at these points, we need to estimate the hidden cusp H and the tangent vector at H. (Right) Precomputing the table of hidden cusp locations for a number of sampled locations and directions. 42
3.15 The figure demonstrates the 'beam'-search procedure for choosing the best valid configuration on an example sketch that has six junctions. Two best configurations are shown in the red rectangles. 43

3.16 Topological consistency checks near the endpoints. Top row shows 4 Huffman's labeling cases, bottom row - the corresponding validity checks. (a)-(b) Consistency check at the T-point. Let T be the T-junction location, V_T - the tangent direction at the T, pointing away from the visible stroke, B - the Bezier point lying on the hidden contour closest to T, and $V_B = (T, B)$. Then, Rule 1 says that $dot(V_T, V_B) \geq 0$. (c)-(d) Let C be the visible cusp, B - the point on the hidden contour closest to C, V_C - the tangent direction at the cusp, and $V_B = (C, B)$. Then, Rule 2 says that the cross-product of V_C and V_B should be equal to $(0, 0, -1)$ (assuming the z-axis is perpendicular to the screen pointing away from the screen). Rule 3 says that $dot(V_B, V_B) \leq 0$. 45

3.17 Assigning depth labels and orientations to hidden contours is straightforward if hidden curves do not intersect each other or visible contours. In this case, hidden contours connecting a tee point with either another tee point or a hidden cusp are labeled 2, and hidden contours connecting two cusps (visible or hidden) are labeled 1. 46

3.18 Some of the results of the figural completion algorithm. Visible contours are shown in black, hidden contours with Huffman's depth 1 are shown in blue and with depth 2 in red. 47

3.19 Problem cases: our method can produce a contour completion which places the hidden cusps at impossible locations. This happens because our method only considers local probabilities, and not the shape of the remainder of the visible contour. 48

3.20 (a) The back-leg drawing case; (b),(c), and (d) show possible completions; our system produces completion (d). 48

3.21 The visible contours of the bean divide the plane into two regions; traversing a path slightly displaced from the boundaries of these regions gives the two red curves shown. The inner one passes the cusp and then the T-point (we don't count where it makes a turn at the T-point as "passing" it); the passages are marked with dots. The outer one passes no endpoints at all. 49

3.22 Adjacent cusp-T pairs can be joined with two arcs and a hidden cusp; adjacent T-T pairs can be joined with a single arc, as can adjacent cusp-cusp pairs; the side on which the arc lies, in this case, depends on the order in which the two cusps were encountered. 50

3.23 (a) shows the labeled drawing (the edges shown in black correspond to the true edges in the graph); (b) shows the graph with extra edges added (shown in red); (c) shows blue counter-clockwise cycles and (d) red clockwise cycles found in the graph. The final set of panels is shown in Figure (e). 51
3.24 The last segment of the incoming edge is shown in purple, the red and the black vectors show the first segments of the outgoing edges. The red one is chosen as the leftmost one. 53
3.25 (left) A panel for a bean shape. (right) There are actually two distinct points corresponding to the original red 2D vertex. 53
3.26 Schematic view of the disjoint union of panels that are glued to form the topological manifold homeomorphic to the bean. Each copy of each panel lies in a different layer; the union of all these copies is called U. The map π is "projection back to $\Re^2$ along z." The collection of all points that project to A (the red dots) is called the "stack above A". The magenta edges are the stack above the edge e. Each panel is indexed by its height in z, so all panels have different indices. 54
3.27 The panels, re-ordered for visibility; edges with the same colors are identified. This identifies *clusters* of vertices in each stack; vertices with the same color form a cluster. Note that the near vertex in the two large panels has been split into two copies. 55
3.28 (a) A contour-completed drawing of a leg attached to a body, with panels colored. (b) The two panels for the bottom of the leg, colored to show edge identifications and vertex clusters. Note that the top edges e and e' share endpoints but are not identified. (c) We add mid-edge vertices, sort, and cluster them as before. 56
3.29 (a) A drawing of a torus. (b) 2D vertices corresponding to tees and cusps are shown in blue. (c) An extra vertex shown in red is added to the outer stroke so that the depths could be assigned to all edges. (d) Extra length springs connecting the red vertex to other vertices are added to the mass-spring system that is used to assign depths to vertices. Here only one of them is shown. 58
3.30 The examples of the shapes created with our system from user drawings. The top row shows the shapes from the sketching viewpoint, and the bottom row shows them from a different view. 61

3.31 (a) The initial mesh of the bean shape. The visible contour is shown in green, the faces adjacent to the visible contour - in dark blue, and the target normals of the "silhouette" faces - in red. (b) The top view of the original mesh. (c) The top view of the bean mesh after 10 Evolver iterations, the weight of the normal energy is 5, the weight of the dihedral hook energy is 1. (d) The top view of the bean mesh after 100 Evolver iterations. (e) A different view of the final bean shape. The pictures were generated using Evolver's graphical interface. 63
3.32 Examples of shapes inflated using the Surface Evolver. Visible silhouette points are only allowed to move in z. The minimized energy consists of the "dihedral_hook energy" with weight 1 and the energy term for the normals with weight 5. (a)-(b) The target volume is set to 0.3. (c)-(d) The target volume is set to 0.1. 64
3.33 For each of the four sketches we show the default hidden contours inferred by the system in the main window and a smaller "suggestions window" with up to three alternative topological interpretations of the sketch. 66

4.1 A constructive drawing technique is used for teaching children how to draw animals by first sketching ovals and circles, and then connecting them with guidance strokes to refine shapes [6]. 67
4.2 (a) A single closed stroke above the ground-plane, which (b) is "inflated" to become the body. (c) Another stroke overlapping the body generates a leg attached to the body. (d) Another leg has been added, and the two near-side legs, their placement having been adjusted slightly, are duplicated with a "symmetrize" operation. (e) A different view. (f) The foreleg is merged with the body. 69
4.3 Merging with guidance strokes. In each case, the guidance stroke constraint points are shown as red dots, and the new surface is shown in pink. 71
4.4 Over-sketching: The upper left blob is modified by the user's drawing a stroke near the silhouette (upper right). The surface deforms to match the stroke (lower left). 72
4.5 Examples of hierarchies created automatically by the system. 72

4.6 Inflating a stroke: (a) the user's stroke is resampled and projected onto the $z = 0$ plane. (b) The points defining the stroke are used as zero-points for inflation (indicated by small circles), and points slightly offset along the normals are used as plus-points (indicated by plus-signs). 75
4.7 Placing extra plus constraints on the line passing through the center of mass of the contour parallel to the z-axis does not work for concave drawings like the one shown here, since the center of mass lies outside the stroke. 76
4.8 First we displace each point of the contour along the inward pointing normal, with the distance proportional to the signed 2D curvature of the contour at this point. Then, two copies of this resulting curve shown here in red could be translated in the positive and negative z directions, and zero constraints placed at these locations. 77
4.9 The underlying branch graph for a blob 79
4.10 Isosurface blending in 2D. The level set for f and the level set for g intersect. When points of these level sets are mapped to R^2 by H, they land at corresponding points in the right-hand figure. One can see that the pre-image of the positive axes consists of a merge of the "outside" parts of the two level sets. 81
4.11 Automatic merging: the hollow constraint points on the left are eliminated, leading to the new iso-set in purple on the right. 81
4.12 Merging with a guidance stroke: The region of influence is shown in pink and blue colors. 82
4.13 (Left) Blob and modification stroke. (Middle) Blue dots indicate zero points in the green region of influence. (Right) The blue points are eliminated and the new red zero points from the target stroke are added. 84
4.14 Two animal models created using our system. Each took about 5 minutes to create by the user familiar with the system. 86

5.1 Given an object point P, two pinhole cameras with the optical centers C_1 and C_2, and two image planes, we can define an epipolar plane (C_1, C_2, P), two epipoles E_1 and E_2 (the points of intersection of the line (C_1, C_2) with the image planes) and two epipolar lines (shown in orange). 89

5.2 Sketching a 3D curve using epipolar constraints: a user draws a red line on the left image plane to create a black 3D curve; then she can modify the shape of the black curve by drawing its green projection from a different point of view (on the right image plane). The green curve is projected onto epipolar lines (that are shown in orange and intersect at the epipole E_2). As a result, a blue 3D curve is created. The projection of this final blue 3D curve onto the left image plane remained unchanged because of the epipolar constraints. 90

5.3 (a) A user draws a curve on the image plane. (b) When the virtual camera is rotated, epipolar lines appear as yellow segments at each point of the stroke. (c) The user redraws the selected curve from a new point of view by inputing a new green stroke. (d) A new stroke is projected on the epipolar lines and the coordinates of the points of the original curve are updated to satisfy new constraints. (e) The resulting curve from a different angle - notice that new epipolar lines are calculated for each point of the new curve. 91

5.4 The Bishop framing along the curve. The blue and green lines are the two vectors of the Bishop frame; the red lines are in the direction of the tangent vectors. Notice that the Bishop frame is continuous even at the inflection between the two bends. 93

5.5 A generalized cylinder in a shape like that of Luxo, Jr.'s power cord; the small vertical bump is in a different plane from the main curvature of the cord. 95

5.6 Examples of the generated generalized cylinders. (a)-(b) The lizard's tail is generated from two cross-sections that are a morph between a circle and a triangle. (c)-(d) The mouse tail is created from two circular cross-sections. (e)-(f) This generalized cylinder shows off the interpolation between flower-shaped and circular cross-sections. (g) The monkey's tail has a simple circular cross-section. 96

5.7 (a) Assume that the following flat spiral curve was drawn in the bottom plane (xz-plane) and a user wants to add torsion to this flat curve in y. (b) When the virtual camera is rotated to the most intuitive view for "redraw" operation, the system cannot project a newly drawn stroke on the epipolar lines because it is unclear which point matches which epipolar line (c). . . . 97

5.8 (a) The user has drawn the back of a chair (as a rectangle) and its seat (as a parallelogram). (b) When the user views it from a new direction, it is clear that the back and seat are actually coplanar. (c)-(d) The user tries to modify the seat in this new view by using a shear operation to make them actually perpendicular, but (e)-(f) the results, seen from another view, show that, although the seat and the back of the chair are not coplanar anymore, they are still not at 90 degrees to one another. 98

6.1 A drawing that currently can not be handled by SmoothSketch since hidden contours intersect with each other and with visible contours. Currently, the system would reject such a completion. Generalizing our figural completion algorithm to handle such cases would let a user create much more interesting shapes like the one shown here. Here, we mark new tee-junctions in orange and show valid Huffman labels for hidden contour parts. All visible contours parts have label 0. 106

6.2 How can we modify the algorithm so that the 3D shape is modified whenever a user adds details to the original drawing? 107

6.3 (a) A bean drawing with hidden contours computed by SmoothSketch. (b) The user oversketches one of the hidden contours, the new contour is shown in red. (c) The second hidden contour is automatically modified by the system so that the tangents of two curves meeting at the hidden cusp are equal. . . 107

6.4 (a) A drawing with hidden contours computed by SmoothSketch. (b) The user removes hidden contours by either selecting them and pressing delete, or drawing a "cross" stroke. (c) Then the user pairs up two tee points shown in red by clicking on one of them and dragging the mouse till the other junction. The other two points can be automatically paired up by the system. 108

6.5 Regional contour editing gesture: a user draws a red stroke to specify a region of the drawing for which the contour completion should be recomputed. . . 109

6.6 (a) A contour generator γ of the kidney bean surface is shown in blue. The visible cusp is marked with C and the hidden cusp with H. (b) The 2D contour of the bean is shown together with the normals to the surface at each point. (c) The curves $\delta^{\pm}$ are obtained by displacing γ along the vector $S = \frac{\gamma'}{||\gamma'||} \times N$. (d) One of such curves, δ^{+}, is shown here dotted in gray. (e) γ is extruded slightly in the tangent plane to create a thin ribbon. (f) The depths and normals are interpolated to the interior of the bean. 110

6.7 Extending SmoothSketch to handle surfaces with creases (like the creases on the pants shown here) would allow a user to model a larger variety of shapes. 111
6.8 (a) A contour drawing, where the inner green stroke can be interpreted as either a contour or a crease. (b) The shape inferred from the sketch assuming that the inner green stroke is a crease (side view). (c) The shape inferred assuming that the inner green stroke is a counter clockwise contour (side view). 112
6.9 A piece of cloth is one example of surface with boundary. 112
6.10 From left to right: an inflated drawing of a dog's body with two legs; the user changes the camera view to look at the dog from the back; the user oversketches the silhouette of one of the legs to make it thinner. 113

Chapter 1

Introduction

There are two ways to approach computer graphics. The first one is a "sampling" approach, where the goal is to capture properties of real-world entities (such as shape and material reflection) and redisplay these entities under novel conditions. Research in image-based rendering, scanning and processing of 3D models, and high-dynamic-range imaging contributed to this approach in the past years.

The second approach, discussed in this thesis, is to *synthesize* new digital shapes from scratch through user-computer interaction. Professional designers, artists and architects need to create 3D shapes as a part of their jobs. Recently, there has also emerged a need for non-expert users to create 3D models. One example of that is communication: if one wanted to explain a concept to a colleague that involved a 3D shape, it would be nice if one could just sketch the shape on the computer screen the same way one would sketch it on paper. Another application area is computer games. Interaction is a very important aspect of computer games, and being able to create your own character is going to increase interactivity of the game. Currently, many game characters (created by professional artists) while very complex and detailed, tend to look somewhat alike and lack personality. If a user could create his or her own character fast, the character would probably look less polished, but would have a unique look, conveying the user's personality and taste. Finally, there have been great advances in search systems over the recent years — the systems let users search text, images, videos and 3D models. As online 3D mesh repositories grow, so does the need to search among these models. A 3D shape search engine developed by Funkhouser *et. al* [32] has an interface that lets a user sketch a 3D shape and then use it to search the mesh repository.

Commercial modeling software tools (Maya, 3D Studio, etc.) are very powerful and let

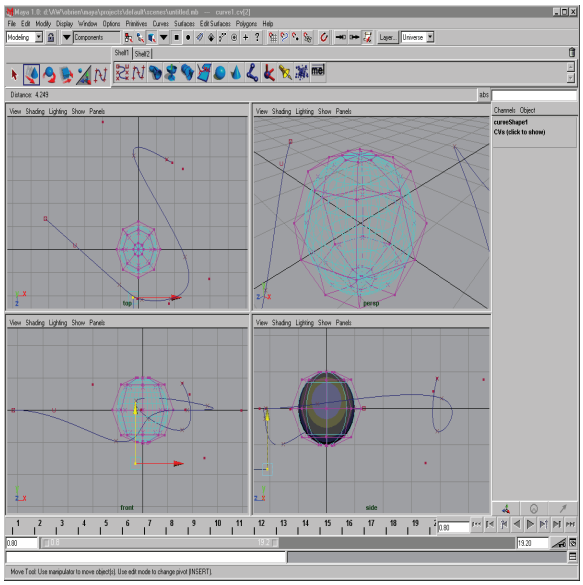


Figure 1.1: Maya's interface for editing 3D curves [3].

users create fine, detailed models, but are hard to learn and use and not very well suited for the stage of initial conceptual design. Figure 1.1, for instance, shows Maya's interface for editing 3D curves. Because the interface is complex, it is unsuitable for non-expert users and hard to use even for professionals.

By contrast, we seek to develop simple interfaces that rely on perceptually significant aspects of models for input — letting a user sketch parts hierarchies, or contours, for example, to indicate shape.

1.1 Sketch-Based Modeling Interfaces

In the recent years, several researchers [112] [47] [110] [92] [56] have proposed such simple sketch-based interfaces as an alternative approach to the kind of 3D modeling provided by system like Maya. Sketch-based modeling interfaces let users create rough, informal models in seconds or minutes (see Figure 1.2). Paper sketches are a great way to communicate ideas, they are quick to draw and easy to discard and correct (which is very important for creativity), and have been used by professional designers, artists and architects for years. Sketch-based modeling interfaces combine people's ability to sketch shapes well with the computer's ability to infer 3D shapes from sketches, giving users a great tool for quickly modeling digital shapes.

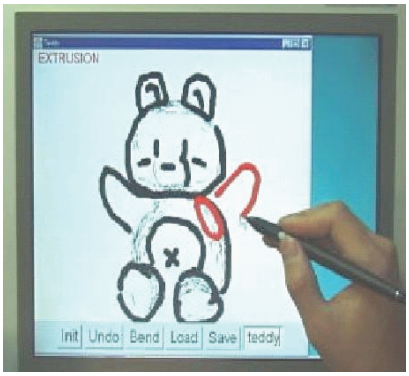


Figure 1.2: A snapshot of Teddy: a sketch-based interface for modeling free-form shapes [47]. Compare the simplicity of this interface to Maya's. *Image courtesy of Takeo Igarashi.*

We propose the following application areas for such sketching interfaces:

- Initial design of free-form 3D models for character animation and computer games,
- Rough industrial design of free-form objects (vases, toys, etc.),
- Educational and recreational use.

In this thesis we describe algorithms that can be used for building sketch-based modeling interfaces.

1.1.1 What is a sketch?

The nature of what constitutes a sketch depends on the medium. A sketch of a novel is a rough description of the story with many details omitted. A sketch of a cat is a quick, informal drawing of it (see Figure 1.10). A sketch is usually something unfinished and rough, that captures the essence of the object. When we talk about sketch-based modeling interfaces, by "sketch" we mean something that is produced via user-computer interaction. It is the user's informal description of a shape to the computer, a description that may contain contours of shapes from one or many points of view, as well as various gestural marks. An example of such sketch is shown in Figure 1.2.

If we have in mind a good idea of the shape we want to create, how do we communicate this idea to the computer? It depends on what is being modeled, and is a choice of the designer. In this thesis we present three different approaches.

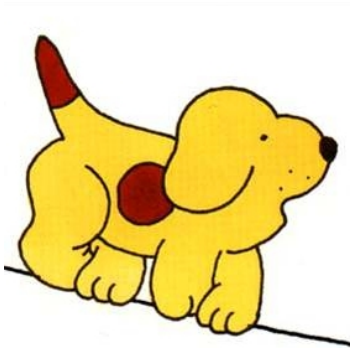
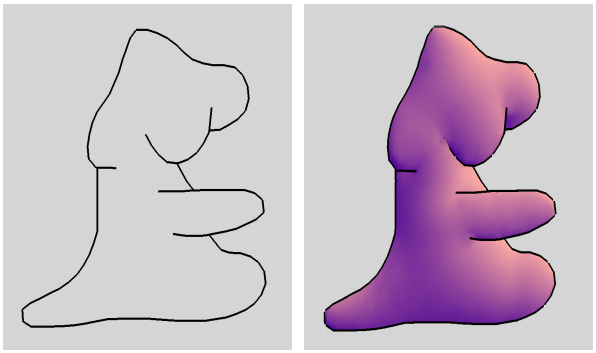


Figure 1.3: This drawing of Spot the dog [41] shows how even simple contour drawings convey shape very well.



(a) (b)

Figure 1.4: The visible contour drawing and the 3D shape inferred by SmoothSketch.

1.1.2 SmoothSketch: 3D free-from shapes from complex contours

The first approach leverages our ability to draw contours of shapes well. Not only are people good at drawing contours (artists more so than others), they are also good at perceiving shape from contours. As one can see in Figure 1.3, even simple contour drawings convey shape very well. User studies [11] show that people are just as fast at recognizing shapes from edge-based representation as from color photographs. Our system, SmoothSketch, takes a contour drawing of some object and returns a 3D model that matches a given contour and is one of the plausible interpretations of the drawing (see Figure 1.4). At a high level, this is a kind of a shape-from-contour problem that has been studied in computer vision. This is a very hard problem and so far has been solved only for restricted classes of drawings.

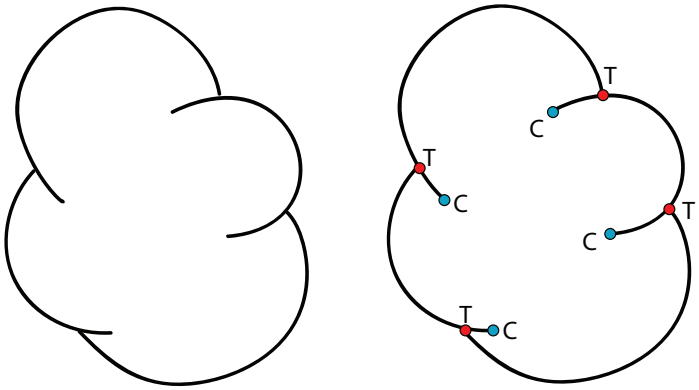


Figure 1.5: A drawing with the tee-points and cusps marked.

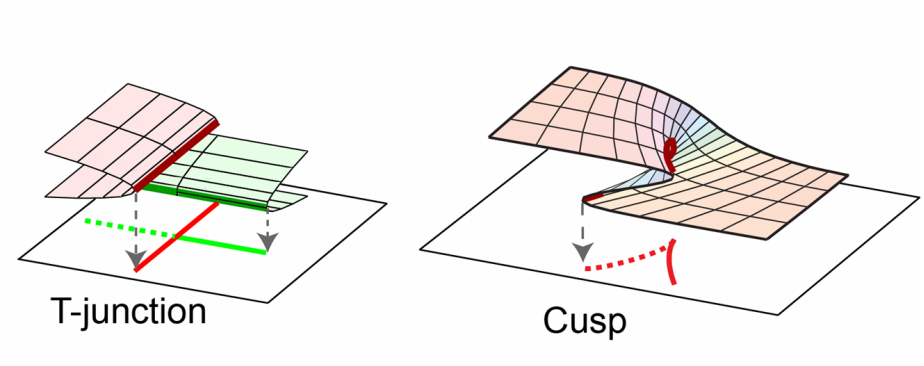


Figure 1.6: Contour components: T-junctions and cusps.

In the area of sketch-based modeling interfaces, the previous systems like Teddy [47] could handle only simple closed strokes. Our work makes inflation possible for the more complex drawings that contain various junctions.

We propose an interactive, mixed-initiative process, in which the user draws contours, and the computer makes shape inferences based on the user input and also allows the user to either select a different topological interpretation from the list of suggestions or edit the existing intermediate shape representation: hidden contours.

We are interested in the contours of smooth closed surfaces. We assume that the projection is *generic*, that is the view is not accidental and no probability-zero events occur. The contours can be quite complex: in particular, they can contain two types of singularities: T-points and cusps. Informally, a T-point is where one contour disappears behind another piece of surface. A cusp is where a contour disappears behind its own surface[1]. T-points

[1]For a formal definition of a T-point and a cusp please see Chapter 3.

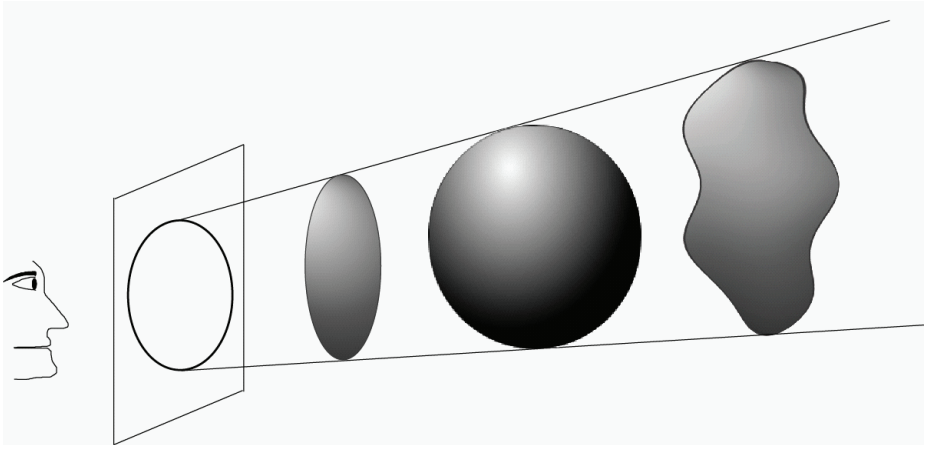


Figure 1.7: The problem of inferring a 3D shape from a 2D drawing is under-constrained. The circle could be an outline of any of these objects.

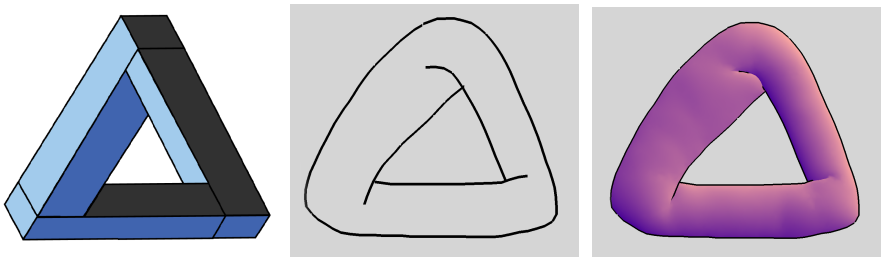
and cusps are the only types of singularities that occur in an arbitrary generic contour of a smooth surface (see Figure 1.5 and Figure 1.6).

The ambiguity of the sketch

The problem of inferring a 3D shape from a 2D contour is inherently ambiguous. For instance, Figure 1.7 shows multiple shapes that project to the same contour. Yet when people look at the drawings, most see the same shape. The explanation of this ability of our visual system can be found in research on visual intelligence and perception [43] [82]. When we perceive a 3D world around us, we create it from 2D images on our retina and face the same ambiguity problem every time we look at something. Yet, the worlds that we create are consistent, and we all construct the same worlds [43]. Bayesian models of this inference process characterize the variety of different image interpretations by a probability distribution, which is determined by the prior distribution on 3D shapes and the likelihood of a particular image given the interpretation [59]. This is the approach that motivates our algorithms.

Separate from this, but consistent with it, Koenderink [61] proposed an explanation of how one can construct smooth 3D shapes from their silhouettes. Several of his observations have been informally summarized by Hoffman [43]:

- Interpret each convex point on a 2D contour as a convex point on the 3D contour and each concave point as a saddle point on a 3D contour.



(a) (b) (c)

Figure 1.8: (a) The famous Penrose's triangle - impossible polyhedral object. (b) A smoothed version of the classic impossible triangle is actually the contour projection of the real surface (c), in contrast to the polyhedral case.

- When possible, interpret a T-junction in an image as a point where the 3D contour conceals itself - the cap conceals the stem (this rule actually follows from a theorem proved by Whitney [104]).
- If possible, create 3D surfaces that are smooth.

We would like to point out that an image input is very different from an input sketch that is given to the computer via user interaction. The very nature of interaction tells us more — the orientation of contours, the sequence in which the strokes are drawn, the opportunity to constantly edit the sketch adding details and gestural marks. On the other hand, images have other cues such as shading, color and texture that help us infer shapes. So, our problem is different from the one faced by the visual system and the solution to our problem does not necessarily mean imitating what the visual system does. Nevertheless, such understanding can give us good ideas and insights about our shape-from-sketch problem.

Huffman labeling scheme

Not only is any contour drawing ambiguous as we discussed earlier, there are contour drawings that do not correspond to any physically-realizable 3D objects. Figure 1.8(a) shows one of the famous impossible polyhedral objects — Penrose's triangle. Whether the drawing corresponds to any reasonable 3D shape or not depends on the assumptions made about the shape. A smoothed version of Penrose's triangle corresponds to a valid 3D shape in the class of smooth free-form shapes (see Figure 1.8(b) and Figure 1.8(c)). Impossible objects have been of interest to cognitive scientists for a long time: we refer the reader to Kanizsa's book [52] for further details.

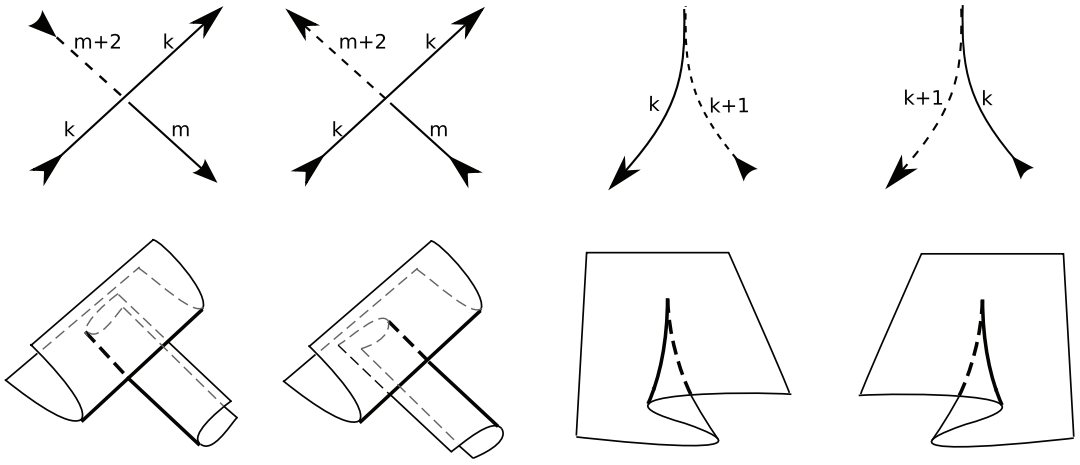


Figure 1.9: (First row) Huffman's labeling scheme for contours of generic smooth projections. Labels indicate the number of surfaces in front of the contour (visible contours have label zero); the surface on which the contour lies is to the left when you traverse it in the direction shown by the arrow. (Second row) For each case of Huffman's labeling we show the corresponding surfaces.

Given a user contour drawing, the system interpreting it must first answer the question of whether there is a 3D shape that matches this drawing. Huffman [44] developed labeling schemes for both polyhedral objects and smooth objects that can be used to determine if the drawing is realizable or not. These labeling schemes contain some relative depth information and serve as an intermediate level between a 2D input drawing and a 3D shape. Figure 1.9 shows Huffman's labeling for smooth surfaces.

Huffman labelings are for complete contour projections — the projections of both the visible and invisible parts of an object's contours. Given a drawing of the visible parts of a contour, we must infer where the invisible parts lie (this process is called *figural completion*) before we can apply Huffman's rules.

Building on top of Huffman's work, Williams [106] proposed the following framework for going from a visible contour drawing to a smooth 3D shape:

1. *Complete* the drawing by inferring the hidden contours, and provide a Huffman-labeling for this completion [44].

2. *Convert* the completed-drawing to an abstract topological surface, and map this surface to $\Re^2$ so that the "folds" of the mapping match the contours of the drawing.

3. *Lift* this mapping to a smooth embedding in $\Re^3$ whose projection is the mapping to $\Re^2$.

If we were faithful to the Bayesian approach mentioned earlier, in step 3 we would compute the resulting 3D shape by optimizing some smoothness measure like total squared curvature, over a space of all shapes that have a given abstract topological representation (step 2). This ideal solution would produce rounded smooth shapes without sharp corners. Instead, we came up with two workable solutions that have characteristics of an ideal solution described above. In the first approach, to compute the desired shape, we construct and relax the mass-spring system that produces reasonably smooth low-curvature shapes. In the second approach, the shape is computed by minimizing a certain energy functional that depends on the dihedral angles of the adjacent triangles in the mesh and the lengths of the edges.

In Chapter 3, we explain each of these steps in detail (including some corrections to Williams' results); describe how we extended step 1 to handle contours containing cusps; and how we carry out step 3 for which Williams did not propose a solution.

1.1.3 Free-form sketching using variational implicit surfaces

Our second approach to shape sketching lets a user create complex 3D shapes by first creating a hierarchy of basic shapes (which we call **blobs**) by sketching their outlines and then refining the shape by drawing **connecting** strokes. The motivation for this approach is twofold. One comes from the research on visual perception: cognitive scientists believe that as a part of object recognition, our "visual system decomposes the shape into a hierarchy of parts" [42]. So when we choose an intermediate representation of a shape for a sketch-based modeling system, it is reasonable to consider the one that our visual system uses.

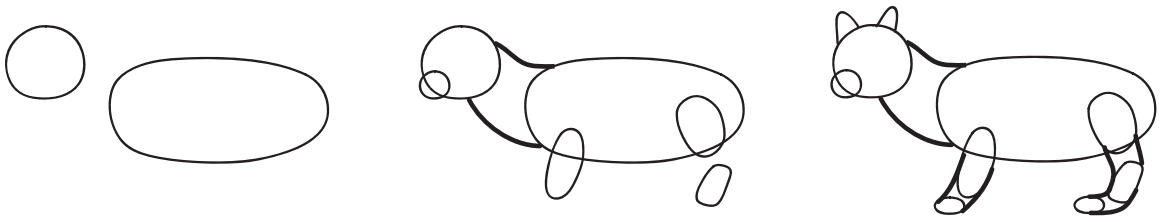
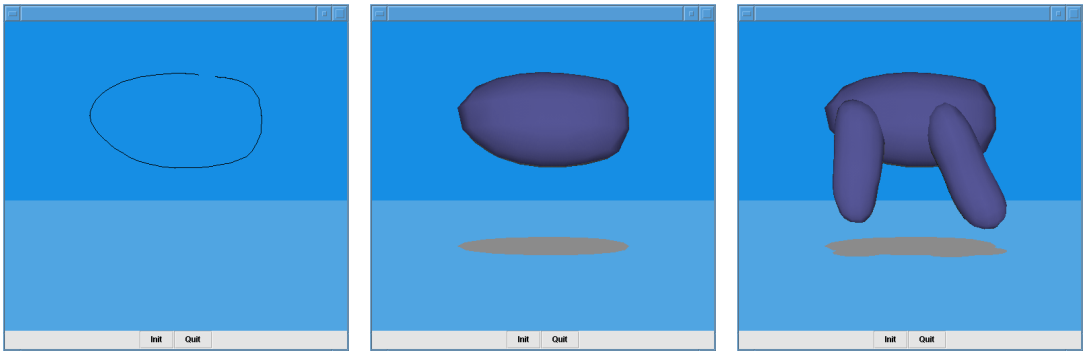


Figure 1.10: A constructive drawing technique is used for teaching children how to draw animals by first sketching ovals and circles, and then connecting them with guidance strokes to refine shapes [6].

The second motivation comes from a 2D technique that is used in children's drawings books [6] [96] (see Figure 1.10) and was previously studied by Crulli [25]. This technique is called "constructive drawing" and is used to draw free-form shapes such as animals and



(a) (b) (c)

Figure 1.11: (a) A user sketches an oval stroke corresponding to the body of an animal, which (b) is "inflated" by the system into a 3D blob. (c) Two other strokes overlapping the body generate the legs attached to the body.

cartoon characters. It consists of the following steps [6] [96]:

1. Drawing the "foundation" that consists of basic shapes (called *sectional parts*) such as circles, ovals and squares. Each of the sectional parts should have the correct proportion, and the parts should be related to one another.

2. Refining the overall shape by cleaning up the old strokes and adding new strokes - in particular, strokes connecting the basic shapes with one another.

For example, if one wants to draw a cat with this method, one would first draw an oval for the body and a circle representing the head, then ovals for the legs, then merge these basic shapes with one another using connecting strokes; and continue to make refinements as necessary (see Figure 1.10).

Tilton [96] suggests that an artist should observe different objects and try to mentally break them into the basic shapes before drawing.

In Chapter 4 we show how this idea can be extended to 3D nicely (see Figure 1.11): a user sketches outlines of basic shapes and the system creates 3D blobs matching them; as one draws, the shapes are automatically related to each other in a hierarchy, and then more definition can be added to the shape via over-sketching and new strokes.

Surface representation

In contrast to previous approaches like Teddy [47] that use a polygonal surface representation, we use implicit functions [14] to represent the basic shapes in the hierarchy. This

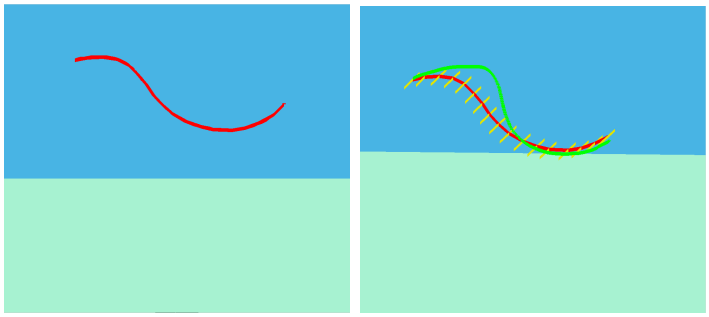


Figure 1.12: A user draws a curve on the image plane. When the virtual camera is rotated, epipolar lines appear as yellow segments at each point of the stroke. The user redraws the selected curve from a new point of view by specifying a green stroke. The green stroke is projected on the epipolar lines and the coordinates of the points of the original curve are updated to satisfy the new constraints.

allows us to introduce new operations easily while providing nice smoothness characteristics automatically. In Chapter 4 we talk about a particular representation for implicit surfaces — the *variational implicit surfaces* [88] [99] — and show how it can be used to support a collection of free-form modeling operations.

1.1.4 Epipolar methods for multi-view sketching of generalized cylinders

Single view sketch-based modelers like Teddy can be powerful tools, but sometimes their inferences are inadequate: in Teddy, for example, if one draws an animal shape, the centerline of the tail will always lie in a plane rather than being curved in 3D. In these cases, multiple-view sketching seems like a reasonable solution: a single sketch gets viewed from a new direction, in which further sketching modifies the shape in a way that's consistent with the original sketch.

We have explored the idea of multi-view sketching for the case of generalized cylinders. In computer-aided design, it is common to describe a generalized cylinder by a shape that varies along the backbone curve. The approach we propose lets a user first draw such backbone curve from several views where epipolar constraints serve as guides for the user. Epipolar constraints arise from two simultaneous views of the same object, thus superimposing them guarantees that the two consecutive views of a 3D curve are consistent (see Figure 1.12). This technique is formally introduced in Chapter 5. Once a 3D backbone curve is defined, the user sketches several cross-sections perpendicular to this curve; the cross-section strokes are later interpolated and connected smoothly to create a 3D shape (see Figure 1.13). This interface was added to the system for drawing free-form animals described above, and

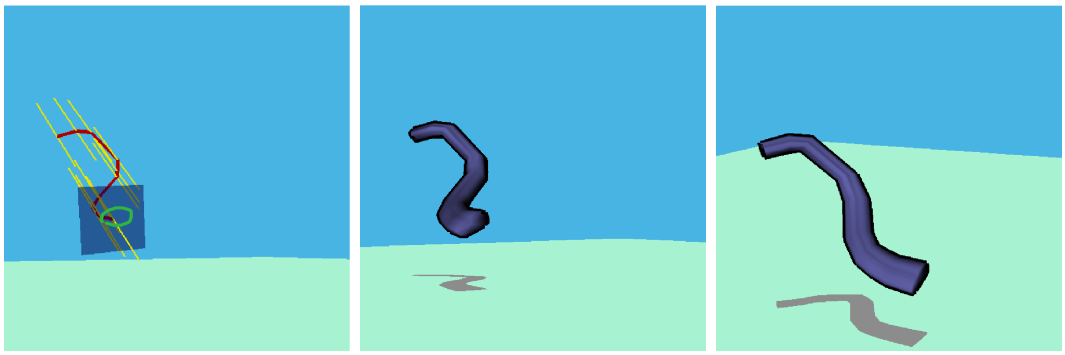


Figure 1.13: A user clicks on the first segment of the 3D curve created in Figure 1.12 and a gray square shows up on which the user draws a green cross-section stroke. Then, the user sketches another cross-section stroke for the last segment of the curve - this step is not shown. The cross-section strokes are interpolated along the curve to create a generalized cylinder displayed in the center and right figures as viewed from different perspectives.

proved to be useful in creating a variety of animal tails and other objects whose shape can be represented by generalized cylinders.

1.2 Contributions

This thesis presents several technical and interface solutions for creating sketch-based modeling interfaces. In particular,

- We present a complete framework that enables us to go from a complex visible contour drawing of a free-form shape to a 3D model matching the drawing. Unlike previous work, our system can handle contours containing tee-junctions and cusps.
- We extend the figural completion algorithms proposed by Nitzberg [79] and Williams [108] to the contours containing cusps in addition to tee-points.
- We provide editing gestures that let users change the inferred shape by editing the corresponding hidden contours in 2.5D interpretation of the sketch or select different topological interpretations of the drawing from a list of suggestions.
- We present a system for creating free-form models that uses a novel interface for modeling shapes: it lets a user create a complex object by first modeling a hierarchy of basic shapes, and then refining this model via modification and merging strokes. Our choice of modeling operations allows artists to construct shapes in a way that closely parallels the way that they draw, starting from basic forms followed by refinement.

- We propose a novel inflation technique that is on based variational implicit surfaces as opposed to triangular meshes. The advantage of the implicit surface representation is that the natural modeling operations — inflation, merging, stroke-based merging — are all easy to implement in this context.
- We introduce the idea of multi-view sketching of generalized cylinders based on epipolar constraints. This is potentially useful for any sketch-based modeling interface.

Chapter 2

Related Work

There has been a lot of research on 3D shapes. Human perception of shape and the problem of shape inference have been studied by cognitive scientists, psychologists and mathematicians. Computer vision researchers were interested in inferring it from one or several images, contour drawings, texture and shading. Algorithms were also developed for modeling different classes of 3D models using interactive gestural interfaces. We use a categorical organization of the papers, where some approaches fall into several categories.

2.1 Visual Perception of Shapes

Shape perception is one of the oldest fields and has been extensively studied. We only mention the most relevant papers and refer the reader to Peters's survey [85] for further details.

Our visual system is an amazing mechanism — we hardly ever think of the complexity of the problem it solves every time we look at something. Although an unlimited number of 3D interpretations of each 2D image are possible, the visual system constructs only one of them, *the mostly likely one*.

In his seminal work, Marr [70] suggested that the following intermediate representations are involved in the perception of shape: first, a visual system reconstructs a *primal sketch* from an image; the primal sketch contains image edges that are grouped into clusters. Then, a 2.5-D sketch of the scene is produced, which contains visible surface orientation and depth information. Finally, a volumetric 3D shape is reconstructed from the 2.5-D sketch.

The Bayesian inference approaches to the problem of visual perception have become popular in recent years [59]. The idea dates back to the 70s, when Grenander [35] proposed a very general approach, *Pattern Theory*, whose goal is to reconstruct the processes that

produce the patterns generated by the world and predict these patterns. Helmholtz [39] was the first to describe perception as unconscious inference from sensory data and past knowledge. Bayesian inference approach characterizes the variety of different image interpretations by a probability distribution, which is determined by the prior on 3D shapes and the likelihood of a particular image given the interpretation.

Hoffman [43] postulates that our perception of shape, color and motion is guided by a set of innate rules that he calls the *rules of universal vision*. There are four main rules: the rule of generic view, the rule of projection, the rule of symmetry and simplicity, and the rule of proximity. The first two of them are the most powerful and produce a number of smaller rules. The rules, most closely related to the interpretation of contours are the following:

1. Straight lines in an image are interpreted as straight lines in 3D.
2. A corner formed by two intersecting line segments in an image is interpreted as two segments sharing the same point in 3D.
3. Collinear lines in an image are interpreted as collinear lines in 3D.
4. Smooth curves in an image are interpreted as smooth curves in 3D.
5. Curves in an image are interpreted as contours of some 3D objects whenever possible.
6. When a 2D contour forms a junction in the form of letter T, it is interpreted as a point where the contour conceals itself: the cap of the letter T is closer to the viewer than the stem.
7. Convex points of the 2D contour are interpreted as convex points on the 3D contour of a shape; concave points — as saddle points in 3D.
8. Construct smooth surfaces if possible.

The last three rules were discovered by Koenderink [61] [62] and Whitney [104]. The rules are so powerful that they can override our "common sense" and lead to inference of impossible objects discussed in the introduction.

Kanizsa and other researchers have studied visual illusions [52] [34] ("errors of perception") and showed that they can provide insight into the processes of visual inference. A classical example of the visual illusion, Kanizsa's triangle, is shown in Figure 2.1.

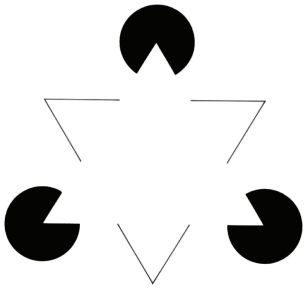


Figure 2.1: A famous optical illusion - Kanizsa's triangle. We perceive the edgeless triangle that appears to be brighter than the background.

2.2 Shape from Drawings

The problem of inferring 3D shape from 2D drawings has been studied in a great many forms; if one extends it to include determining drawings from images as a first step, it occupies much of the computer vision literature. We only describe the work most closely related to this thesis.

Witkin [109] proposed a statistical model for estimating the orientation of planar and curved surfaces given the image contours.

Pentland and Kuo [83] presented a system that infers simple 3D curves from 2D contour drawings. They assume that the view is generic and that 3D interpretation is smooth. They represent a curve as a set of connected nodes whose z-positions are adjusted to minimize the total energy of the system (they minimized changes in length and curvature). Such 1D dynamic models are called *snakes* [57].

2.2.1 Creating CAD-like 3D models from 2D line drawings

Much early shape-from-drawing work applied to blueprint-like drawings of machined surfaces. The important features of such shapes are sharp bends, like the edges of a cube – and their trihedral intersections. Some of the methods assume the drawings have both visible and hidden lines drawn, and others do not. The two most common approaches to these problems are using compliance functions and line labeling, the first being more popular in the recent publications. The first approach (optimization-based reconstruction) extracts implicit spatial information from a 2D line drawing by using some heuristics like image regularities. Examples of image regularities are parallelism of lines, isometry, skewed

symmetry, and skewed orthogonality. Regularities are mathematically expressed using compliance functions, whose combination is later optimized by moving the z-coordinates of the vertices in the sketch. Because this process leads to non-linear multi-dimensional optimization, it can be slow depending on what regularities are used. Another limitation is that most heuristics work on the drawing consisting of straight lines, rather than on general curved faces. It is also hard to find a set of regularities that works for all cases.

An overview of work on the line labeling approach is presented in Varley's thesis [101].

Lipson and Shpitalni [65] introduced a system in which a user sketches both visible and hidden contours and boundaries of a rectilinear CAD-like geometric object, and the system infers a shape. Their approach is based on correlations among arrangements of lines in the drawing and lines in space. For instance, by doing experiments, they notice that "the more two lines are parallel in the sketch plane, the more they are likely to represent parallel lines in space" [65]. They rely on a geometric correlation hypothesis that claims that we could evaluate the likelihood of human understanding of the sketch by the joint probability of all correlations in the sketch.

2.2.2 Creating smooth 3D shapes from 2D line drawings

Prasad and Fitzgibbon [86] reconstruct a smooth surface from the contours containing multilocal singularities by optimizing the thin-plate energy function subject to contour and inflation constraints.

2.2.3 Creating 2.5-D representations from 2D line drawings

A classic paper in this area is by Huffman [44], who developed two labeling schemes—one for objects made from planar surfaces, one for smooth objects—and proved that their complete contour drawings must have the corresponding sorts of labeling. This schema can be considered a 2.5-D representation of the drawing, as it contains some relative depth information. This representation can serve as an intermediate level between a 2D input drawing and a 3D shape.

Williams [106] suggested a framework for going from a 2D contour drawing of a smooth solid object to its topological representation. The reconstruction Williams describes is in the form of a set of panels with relative depths assigned and parts to be glued specified. His approach consists of three steps: completion of hidden contours ("figural completion"), topological reconstruction of the shape ("paneling construction"), and smooth embedding by relaxation. We describe the details and limitations of this work in the next section.

Bellettini *et al.* [10] have also been working on a variational model for reconstructing a smooth surface from its contour containing self-occlusions[1].

2.3 Contour Completion

In computer vision, there is a long history of work in contour completion, which is a process of guessing hidden (invisible) contours in a drawing. We will just mention some of this work here. Williams gives a more detailed overview of the work in this area [106].

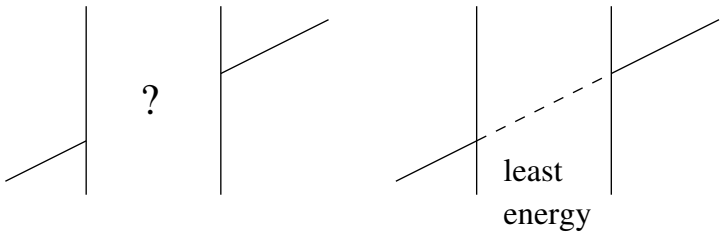


Figure 2.2: The problem of contour completion.

Huffman labelings are for complete contour projections — the projections of both the visible and invisible parts of an object's contours. Given a drawing of the visible parts of a contour, we must infer where the invisible parts lie. Kanizsa's work [52] on contour completion (and its relationship to the mechanisms of the human visual system) forms the basis for much of the later work in the area.

Let us consider a drawing which describes a typical problem in contour completion shown in Figure 2.2. The question Mumford [76] asks for the first drawing in Figure 2.2 is "How can we measure the relative likelihood of two disappearing edges being matched up by an invisible edge"? He showed that theoretically, elastica curves (curves of least energy for a particular 'energy' functional) are the best solution. A solution proposed first by Grenander [35] was to use a stochastic process to model the space of all possible edges. Mumford proves that the elastica curves could be modeled by white noise stochastic process and gives needed formulas. Williams [108] approximates the solution by considering a sampling of the space of all random walks (with varying $\Delta\theta$ - the direction of the walk) starting from the first point with the first direction and coming to the second point with the second direction, and taking the random walk with the highest probability as the best path connecting two edges.

Guy and Medioni [38] proposed an approach to the completion problem based on the

[1]The authors have sent us a preprint copy of the recently submitted paper.

"Extension Field" - a voting pattern similar to Hough transform. First, each pixel collects 'orientation' and strength votes from image segments. The votes are combined to find sites of high saliency that can then be linked together.

Nitzberg *et al.* [79] solve the figural completion problem for the contours of the scenes composed of paper cutouts arranged in depth. The only types of contour junctions that can arise in generic views of such scenes are T-junctions. They use a greedy search to find the best pairing of junctions, where the minimized energy depends on the length and curvature of the contours, as well as orientations of tangents at the endpoints. The method can not be applied to natural images.

Masnou and Morel [71] show how completion for natural images can be computed using a level lines structure.

Although contour completion is such a well-studied research topic, many problems are still open; in Section 3.2 we propose our solution to a problem of finding a hidden cusp for a cusp-contour completion case. This algorithm was partially inspired by the work of Williams, Mumford and Nitzberg.

2.4 Sketch-Based Modeling Interfaces

Several gestural interfaces for sketching three-dimensional shapes have been developed for different classes of models.

2.4.1 Sketching rectilinear objects

For rectilinear objects, the Sketch system described by Zeleznik *et al.* [112] lets a user create and edit models through gestural interface, where geometric aspects of gestures determine numerical parameters of the objects; a cuboid is created by drawing three lines meeting at a point; the lengths of the lines and position of the point determine the geometry of the cuboid.

These ideas were extended by several research groups [92] [84], and appear in the SketchUp [1] architectural design software.

Igarashi [45] presented an idea of *suggestive* modeling interface, where, given the current user input, a number of possible operations are suggested to the user by the system.

2.4.2 Sketching free-form objects

For free-form objects, Igarashi's Teddy [47] was the first interface for free-form modeling via sketching. In it, a user inputs a simple closed curve and the system creates a shape matching this contour. Then the user can add details by editing the mesh with operations like extrusion, cutting and bending, all done gesturally. The Smooth Teddy [46] system extended this by adding algorithms for beautification and mesh refinement, as well as organizing the shapes into a hierarchy.

Wyvill *et al.* [110] describe the blob-tree, a CSG-like hierarchy of implicit models in which shapes are combined by a rich collection of operators, including various deformation operators.

Karpenko *et al.* [54] describe a system for creating shapes from free-form sketches in which the primitive objects are variational implicit surfaces, which facilitates operations like surface blending.

ShapeShop [89] uses hierarchical implicit volume models to let a user interactively edit complex models via a sketching interface.

Alexe *et al.* [5] extract the skeleton from the sketch and then construct a convolution surface.

None of the above systems handle complex strokes containing tees and cusps.

Karpenko and Hughes [53] demonstrated a method for inferring certain free-form shapes from sketches by detecting 'templates' in the sketches and building a part of the 3D surface from a standard recipe for each template.

Recently, Cordier and Seo [23] also explored the problem of inferring free-form shapes of drawings of occluding contours. They restricted themselves to contours containing only tee-junctions, but allowed multiple objects. They first compute the 2D skeleton of the sketch and then solve the constrained optimization problem to find the corresponding 3D skeleton. Finally, the 3D shape is created using the work of Alexe *et al.* [5]. Their approach guarantees that the surfaces do not intersect. Only objects with circular cross-sections can be reconstructed with their system; the examples shown in the paper demonstrate the use of the system for modeling tubular shapes.

Cherlin *et al.* [20] presented a sketch-based modeling system inspired by spiral, scribble and bending illustration techniques. The user first draws constructive curves to specify simple geometry, and then refines it using cross-sectional and deformation strokes.

Nealen *et al.* [77] recently introduced FiberMesh – a system for modeling surfaces with 3D curves.

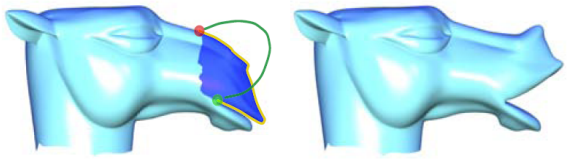


Figure 2.3: A user oversketches a camel's lip by first, selecting the region of influence (shown in blue), and then drawing a yellow reference silhouette curve and a green target curve. The positional constraints are satisfied only approximately [78]. Image courtesy of Andrew Nealen.

2.4.3 Sketching 3D curves

Cohen *et al.* [22] proposed a system for modeling 3D curves by sketching the curve and its shadow.

Karpenko *et al.* [55] built a simple multi-view sketching system to support the construction of curves ("backbones") along which generalized cylinders are extruded. The user draws the backbone from one point of view, redraws it from another, and the constraints of epipolar geometry are used to find the curve matching both sketches as closely as possible. Then the user proceeds to select and draw a couple of cross-section strokes, and the generalized cylinder is created by interpolating the cross-section strokes along the backbone curve.

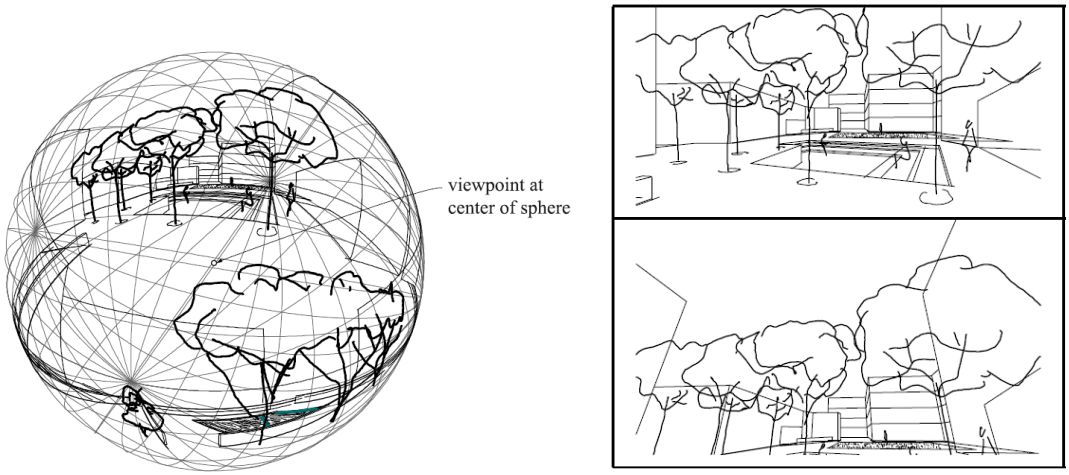
2.4.4 Sketch-based editing of existing meshes

Nealen *et al.* [78] presented a sketch-based interface for laplacian mesh editing where a user draws reference and target curves on the mesh to specify the mesh deformation (see Figure 2.3).

Kho and Garland [58] developed a similar sketch-based interface for posing 3D characters, in particular, bodies and limbs.

2.4.5 Sketching pseudo-3D models

Tolba *et al.* [97] describe a system that lets a user draw a scene with 2D strokes and then view it from several new locations as if a 3D scene had been created. This is done by projecting the 2D strokes on a sphere centered at the eye point and then viewing them in perspective (see Figure 2.4). Their system is a tool for perspective drawing and does not construct a 3D model.



(a) (b)

Figure 2.4: (a) A sketch of a plaza is projected onto a viewer-centered unit sphere. (b) Two views generated by Tolba's system from this drawing [97]. Image courtesy of Julie Dorsey.

Bourguignon *et al.* [16], describe a system that takes a set of 3D strokes representing contours and creates a small piece of surface near each stroke whose contour is the given stroke. Contours of this surface, seen from nearby viewpoints, give the appearance of a full-fledged 3D model, although in distant viewpoints the illusion is lost (see Figure 2.5). Applications of such systems include artistic illustration and 3D scene annotation.

Figure 2.5: An example of artistic illustration; two views are generated from the same sketch [16]. Image courtesy of David Bourguignon.

Williams [105] generates interesting 3D shading effects on 2D images without reconstructing a 3D geometry by applying a variety of complex shading techniques.

Johnston [50] computes lighting on 2D drawings without reconstructing 3D geometry by estimating surface normals from the drawing (see Figure 2.6).

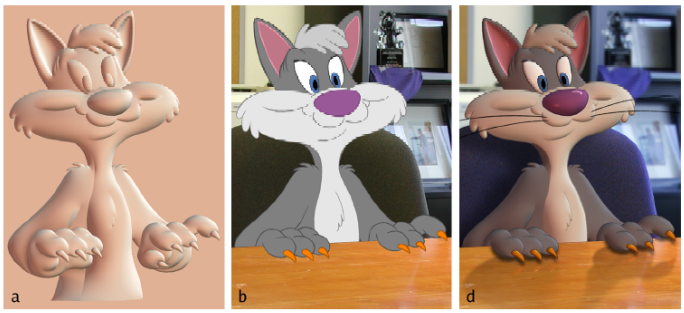


Figure 2.6: From left to right: normal image illuminated by diffuse spheremap; the original cartoon drawing, and the final re-lighted cat image [50]. Image courtesy of Scott Johnston.

2.4.6 Specialized sketch-based modeling interfaces

As we mentioned in the introduction, every sketch-based modeling system has to resolve the inherent ambiguity of a 2D sketch. This task is greatly simplified if we restrict ourselves to certain classes of models or to modeling in a certain context.

Ulupinar *et al.* [30] solve the "shape-from-contour" problem for images by considering only a special class of symmetrical 3D shapes: straight homogeneous - and constant cross section generalized cylinders.

Ijiri *et al.* [48] developed a simple intuitive sketch-based interface for modeling flowers. First, a user describes the structure of a flower using a floral diagram editor and an inflorescence editor. A floral diagram describes the relative positions of flower parts such as petals and stamens with respect to the receptacle. An inflorescence diagram depicts how multiple flowers are arranged on a branch. Then, a user edits the geometry of different flower parts via sketching interface.

Okabe *et al.* [80] presented a system for modeling trees. First, the system infers a 3D model of a tree from a 2D sketch, and then a user adjusts the branches and leaves via gesture-based editing operations.

The system developed by Turquin *et al.* [100] is used to create simple garments for virtual characters by sketching the contours of the garments directly on the virtual actor.

Finally, Malik's interface [68] allows users to create and edit 3D hair styles using free-form strokes to specify cutting, curling, combing and other hair-styling operations.

2.5 Other Sketch-Based Interfaces

Recently there has been a lot of research on creating accessible graphics software for pen-based devices such as tablet PCs. In the previous sections we talked about sketch-based 3D modeling. There are other application areas that can benefit from pen-based interfaces.

2.5.1 Mathematical sketching

One such area is mathematical sketching. In MathPad [64] the user can handwrite mathematical formulas and sketch diagrams, and then form associations between them. The system can then show an animated diagram, where the simulation is described by the given formulas. For example, the system could show an animated pendulum, if the user writes the formulas describing the physics behind it.

2.5.2 Sketching animation

Davis *et al.* [26] have developed a sketching interface for animating an articulated figure. Moscovich *et al.* [75] created a system for sketching simple 2D animations. In their system, a user sketches a number of simple 2D objects, and then selects and drags an object with the mouse; the system records the motion and then plays it back. The authors also address the issue of coordinating the motions of several objects.

A good survey on sketch-based interfaces can be found in [63].

2.6 Other Shape-from-X Techniques

Apart from inferring the shape from the contour, researchers have long tried to infer shape from texture, shading, stereo and other cues.

Shading is one of the powerful perception cues artists have successfully used for a long time to convey shape. Shape-from-shading methods try to recover the shape of an object from image brightness using some assumptions about the surface (such as smoothness) and the lighting model. We refer the reader to the shape-from-shading survey by Zhang *et al.* [113].

Texture is another cue which humans use to infer structure from images. Systems have been proposed that mimic this human ability and recover a surface shape based on texture deformation from point to point; for example, texture patches are foreshortened differently depending on the patch orientation. Some of the methods assume that the texture model is

known, others make weaker assumptions. Re-texturing is one of the applications of shape-from-texture algorithms. Loh's Ph.D. thesis [66] contains a detailed summary of the related work in this field.

As for recovering shape from multiple images/views, the work mostly relevant to ours is by Cipolla and Giblin [21]. They were interested in recovering 3D surfaces (and/or camera motion) from multiple silhouettes. Since the silhouettes are view-dependent, it is harder to establish a correspondence between points on the silhouettes in two different views (since they came from different space curves). Cipolla and Giblin studied dynamic properties of silhouettes and showed how epipolar parameterization of silhouettes, as well as known viewer motion can be used to recover depth and surface curvature.

All of the above cues can be combined to produce a more reliable solution. For example, Ikeuchi *et al.* [49] propose an iterative method for inferring shape from both occluding contour and shading information. Further details on shape-from-X methods can be found in [31].

2.7 Mesh Optimization

Our system SmoothSketch uses triangular meshes to represent surfaces. The performance of both the mass-spring system relaxation algorithm (described in Section 3.7.1) and the squared curvature minimization algorithm (described in Section 3.7.2) depend on the quality of mesh triangulation. The goal is to represent the surface well with a mesh that has a reasonable polygon-count and well-shaped triangles. Thus, the research on remeshing and mesh smoothing is relevant for our work.

Optimization of some surface energy functional has been long considered a good way to model smooth free-form surfaces [74] [102] [103] [51]. Since smoothness is related to the variation of curvature, the most commonly used functional is the integral of the squared mean curvature. One of the key research issues is how to find good discrete approximations of differential properties of surfaces, such as mean curvature.

In this section, we describe several algorithms for mesh optimization that are most relevant to our work. An excellent survey on mesh processing can be found in [15].

2.7.1 Remeshing

The input to the remeshing procedure is a manifold triangular mesh, while the output is a mesh that has higher quality, where *quality* is defined by the user and may refer to things

like vertex distribution, shape of the triangles, regular connectivity etc. Another goal of remeshing is to simplify the original mesh (a process called *mesh decimation*).

Kobbelt *et al.* [60] perform a number of simple operations on the mesh to improve its quality. They define the quality measure as a set of the following requirements: the length of each edge should lie in a given range; the connectivity of the mesh should be as regular as possible and skinny triangles should be avoided. The remeshing operations they suggest include collapsing short edges, subdividing long edges, and flipping edges to increase the number of vertices with a valence value of six.

Turk [98] proposed a system that remeshes the polygonal surface in two steps. In the first step, a number of new vertices (given by the user) are randomly placed on the mesh, and then, each vertex repels its neighbors to produce a more uniform distribution. In the second step, the old vertices are removed and the new vertices are connected into a remeshed surface.

2.7.2 Mesh smoothing

Taubin [94] introduced a λ/μ mesh smoothing technique that is based on low-pass filtering and is linear in time and space. The algorithm does not change the topology of the mesh. In the first step, Gaussian smoothing is applied to the mesh; each vertex is moved according to the formula: $v_i' = v_i + \lambda \sum_{j \in N(i)} w_{ij} \cdot (v_j - v_i)$, where v_i is the old position of vertex i, $N(i)$ is the set of indices corresponding to the first ring neighbors of v_i, w_{ij} are the weights, and λ is the positive scale factor. The first step causes mesh shrinkage - a well-known problem with Gaussian filtering. The second step "unshrinks" the mesh; each vertex is moved again according to the formula: $v_i' = v_i + \mu \sum_{j \in N(i)} w_{ij} \cdot (v_j - v_i)$, where μ is the negative scale factor.

There have also been proposed algorithms that optimize a fairness functional on the vertices of the mesh. We mention some of them in the next section.

2.7.3 Minimizing energy functionals for shape design

Witkin and Welch [103] describe an interactive modeler where a user can design and edit 3D free-form shapes represented by the triangular meshes. In their system, the user specifies the needed boundary constraints and the system minimizes the integral of the squared principle curvatures over the remaining vertices to create the resulting smooth shape.

The 'Surface Evolver' software developed by Brakke [18] minimizes the energy of the triangular mesh given positional or normal constraints on mesh vertices or faces . A user can

either choose to minimize one of the predefined energies of the mesh such as surface tension or discrete squared mean curvature, or specify desired energy as an integrated quantity. We describe Evolver in more detail in Section 3.7.2.

Chapter 3

SmoothSketch: 3D Free-form Shapes from Complex Contours

Our visual system, presented with a line-drawing, makes nearly-instant inferences about the shape that the drawing represents. Some aspects of this inference mechanism are surely based on expectation — when we see something that looks like an ear, it's easy to infer that the nearby protrusion must be a nose, for instance. But other aspects depend on local cues — a contour that ends, or that disappears behind another — and a gestalt view that helps us integrate these local cues into a coherent whole [43].

Dual to this recognition ability is our ability to learn to draw contours of objects in a way that lets us communicate their shape to others. While drawing *well* can be difficult, even children can draw easily recognizable shapes. On the other hand, while drawing outlines or contours is relatively easy, we know few people who can reliably draw the hidden contours of even simple shapes.

When we seek to create shapes with a computer, however, there are few interfaces based directly on drawing; inferring a shape from a complex contour-sketch has generally proved too difficult. The value in doing so, from the sketching point of view, is that it allows a user to draw what he or she is thinking of directly. Teddy's [47] inflation algorithm is a good step, but limited to simple closed curve contours. Our work extends this substantially, although it is by no means a final answer. Such a final answer may never be found, though—it's easy to draw contour sets that are so complicated that different viewers make different inferences about them. The best one can hope for is to create plausible shapes for a fairly large class of contours on which users agree on the interpretation. That is what our work does.

SmoothSketch is not a *system* like Teddy [47]; it's an *inflation component* that can be

used in a free-form-sketching interface like Teddy. We believe that a sketching program should let the user and the computer share the work, each doing what it does best. The computer can infer a plausible shape from a moderately complex contour like the ones shown in this chapter. We believe we have succeeded in this step, and thus provided a new starting-point for sketch-based systems. Then, to create more complex objects (or to, say, modify the thickness of the inflated models), a user would use various gestures like the ones available in Teddy and other sketch-based systems. Currently, SmoothSketch supports a limited number of editing gestures that are described in section 3.8.

3.1 Overview and Background

Our system takes a user's contour-drawing of a smooth, compact, oriented, embedded surface-without-boundary (which we'll call a *good* surface) and determines a 3D surface whose contours match those that the user drew. Figure 3.4 shows an example of a typical user drawing and the shape inferred. Because this is an under-determined problem—is that circle a contour drawing of a pancake? a sphere? a cigar viewed end-on?—we aim to generate surfaces that have generally low curvatures, to the degree allowed by the constraints of the contours. The contours must be *oriented*, i.e., drawn so the surface lies on the left. Thus to draw a torus, a user would draw a counter-clockwise outer stroke and a clockwise inner stroke. This assumption helps us to resolve some of the ambiguities in the drawing (is it a belly or a a pocket?).

Williams' thesis [106] and subsequent work lay out a plan for finding a surface fitting a given collection of visible contours[1]. The steps involved are:

1. *Complete* the drawing by inferring the hidden contours, and provide a Huffman-labeling for it [44].

2. *Convert* the completed-drawing to an abstract topological surface, and map this surface to $\Re^2$ so that the "folds" of the mapping match the contours of the drawing.

3. *Lift* this mapping to a smooth embedding in $\Re^3$ whose projection is the mapping to $\Re^2$.

Having laid out this scheme, Williams then completes several significant parts: he completes step 1 for "anterior surfaces" – roughly the front-facing parts of scenes, which generically have no cusps, and does step 2 for both these *and* for drawings of smooth surfaces.

[1]The reader interested in implementing the ideas of this chapter will need first to become acquainted with Williams' work.

For our purposes, we need step 1 for good surfaces, and we need step 3, which we do in two steps: first we lift to a topological embedding in $\Re^3$, and then we smooth that. We cannot claim to produce a *smooth embedded* manifold; with our current tuning constants, our results are usually immersed (i.e., have self intersections) rather than embedded. Furthermore, the contours of the surfaces we construct cannot in general project exactly to the input drawing, because, for example, the projection of the contour near a cusp always has infinite curvature at the cusp [62], while our user's input may not satisfy that constraint. We produce a fairly smooth mapping of the manifold into 3-space, whose visible-contour projection nearly matches the user's drawing.

In carrying out step 2, Williams (a) observes that the projection of the complete contours of a generic view of a good surface onto an image plane gives a knot-drawing-with-cusps which can be labeled by Huffman's labeling scheme for smooth surfaces (see Figure 3.1) [44], and (b) gives a method (the *paneling construction*) [36] to build an abstract manifold M that can be embedded so that its projection has contours matching the knot-drawing. (He does not actually construct an embedding $e : M \to \Re^3$, but instead describes a map $f : M \to \Re^2$ with the property that if such an embedding e exists, and if P is projection onto the drawing plane, then $f = P \circ e$.)

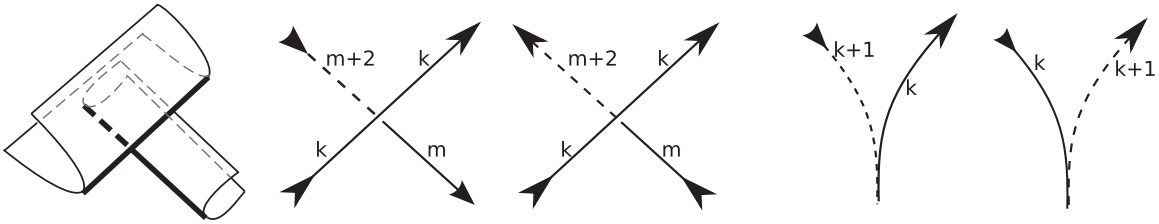


Figure 3.1: Huffman's labeling scheme for contours of generic smooth projections. Labels indicate the number of surfaces in front of the contour (visible contours have label zero); the surface on which the contour lies is to the left when you traverse it in the direction shown by the arrow. *Any* cusped knot diagram—i.e., collection of circles in the plane, possibly intersecting themselves and each other, and smoothly immersed except at finitely many cusp points, which are distinct from the crossing points—that can be so labeled corresponds to the projection of a smooth surface in 3-space (mostly proved by Williams), and all generic projections of smooth surfaces have this property (proved by Huffman). The first picture shows the surfaces corresponding to the first case of Huffman's labeling (the second picture). Figure 3.6 (left) corresponds to the last case.

To carry out step 1 for good surfaces, we must establish which drawings of visible contours can be extended to complete contour drawings; not all can, as Figure 3.2 shows. We

partially solve this problem by exhibiting a large class of drawings that admit such extensions; the general problem of characterizing extendable visible-contour drawings remains open, however.

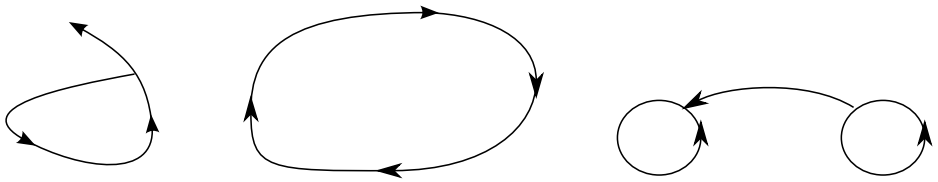


Figure 3.2: None of these drawings can be extended by invisible contours to be the contour set of any good manifold projection. They exhibit two problems: in the first, a cusp appears in the outer region of the plane surrounding the figure. In the second, the outermost path around the drawing is clockwise. Although the third has neither of these problems, it is still not extendable.

For the anterior-surface case, Williams and Jacobs [108] (and Mumford [76]) describe an approach to completing the hidden contours, which generically join tee-points in the drawing (see Figure 3.3). To join a pair of tees, they consider all C^1 random walks (i.e., random walks in which the tangent direction θ changes by an amount X at each point, where X is a Gaussian random variable) starting at the first tee, headed in the right direction, and ending at the second, and assign to each a probability based on the product of the probabilities of each angle-change and $e^{-\lambda}$, where λ is the length of the curve. They posit that the maximum-likelihood random walk is a good candidate for the completion; when multiple pairs of tees might be joined, they compute which pairings have largest likelihoods and choose those.

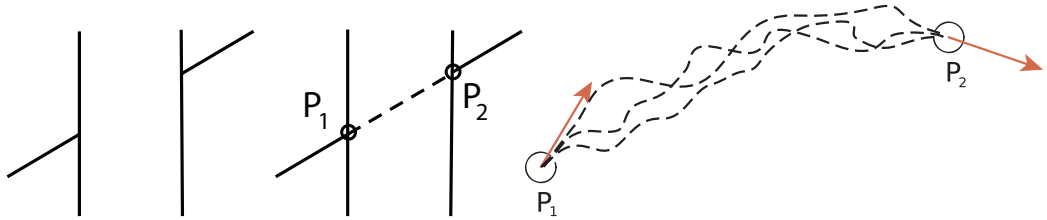


Figure 3.3: How can we join the two tee-points on the left? With an optimal completion, as shown in the middle. Optimality is determined by choosing, among all C^1 random walks from p_1 to p_2, the most likely one, under a simple probabilistic model. Mumford shows such curves are elastica, which had been studied by Euler [29].

We extend this approach, in Section 3.4, to the cases where a T-point must be joined to a cusp, or two cusps must be joined. To determine which visible endpoints (tees or cusps)

should be joined to which, we use a greedy search similar to Nitzberg *et al.* [79].

To carry out step 3, we take the results of Williams' paneling construction — an abstract manifold and a continuous mapping f of it to $\Re^2$ and "lift" it to a mapping e into $\Re^3$ whose projection is f. We construct e a dimension at a time, first placing the vertices of the paneling construction, then embedding the edges, and finally each panel. This algorithm is described in Section 3.6. The result is a topological embedding (i.e., a 1-1 continuous map from the surface into $\Re^3$). Finally, in Section 3.7.1 we talk about smoothing out the creases in this topological embedding by a mass-spring system to produce the desired fairly smooth mesh in 3-space. In section 3.7.2 we present an alternative approach based on minimizing the total squared mean curvature and discuss some of its failures. Figure 3.4 shows an input user drawing, an intermediate representation that has hidden contours marked, and a final 3D shape from two points of view.

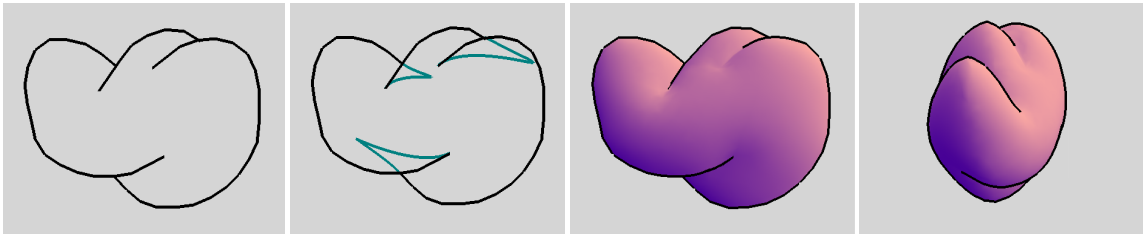


Figure 3.4: The user draws the visible contours of a shape; our program infers the hidden contours, including hidden cusps, and then creates a fairly smooth 3D shape matching those contours. The 3D shape can be viewed from any direction.

3.2 Notation and Problem Formulation

Much of the material that follows relies on ideas from differential geometry and combinatorial and differential topology. We refer the reader to the books of Guillemin and Pollack [37] and Koenderink [62] for clear expositions of the necessary background.

Suppose that S is a smooth, closed, compact, orientable surface-without-boundary (i.e., a *good* surface) embedded in the $z > 0$ halfspace of $\Re^3$. The orthogonal projection of S onto the $z = 0$ plane will have a compact image. Following Williams and Mumford, we will assume that the embedding and this projection are *generic*, i.e., that no probability-zero events occur, e.g., no projector meets three contours, no cusp projects to a point on another contour, etc. If the projector through the point $s \in S$ lies in the tangent plane at s, then s is called a *contour point*; if the projector first meets S at s, then s is a *visible* contour point (see Figure 3.5). For a generic projection, the set of all contour points forms a compact

1-manifold-without-boundary C in S, i.e., a collection of disjoint topological circles in S. The set of visible contour points form a compact 1-manifold-with-boundary, V in S, i.e., a collection of disjoint topological circles and line-segments. The projection of C to the $z = 0$ plane is the *contour drawing* of S; the projection of V to the $z = 0$ plane is the *visible contour drawing* of S.

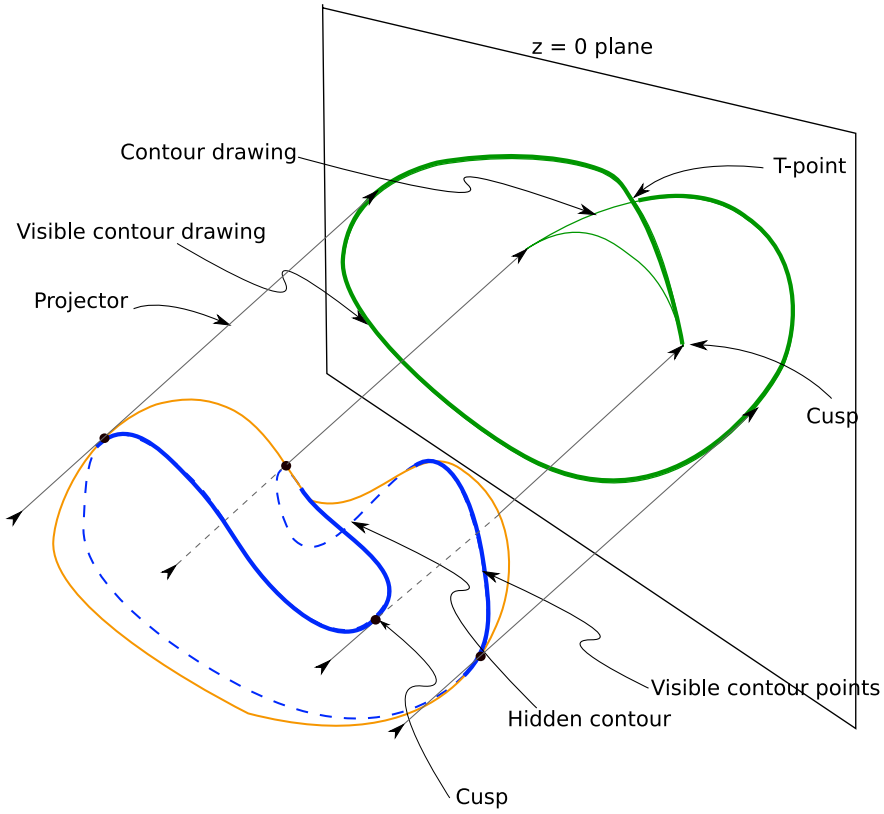


Figure 3.5: (Adapted from Williams' [107]) The contour, in blue, of a good surface embedded generically in 3-space projects to a contour drawing, in green; the visible contour (drawn bold) projects to the visible-contour drawing. A point where the projector is tangent to the contour projects to a cusp in the contour drawing. The restriction of the projection to just the contour is 1-1 except at finitely many points, where two contours cross in the drawing; these are called T-points.

The projection from the contour to the contour drawing is an embedding at most points; the exceptions are *crossings*, where two contours meet, and *cusps*. A cusp is a point $s \in S$ where the projector through s is tangent to C at s. The projection of a cusp appears as a point where the contour drawing "reverses direction" (see Figure 3.6, left). When an arc of the visible contour drawing reaches a crossing, it appears as a *T-point*: one part of the contour becomes invisible there.

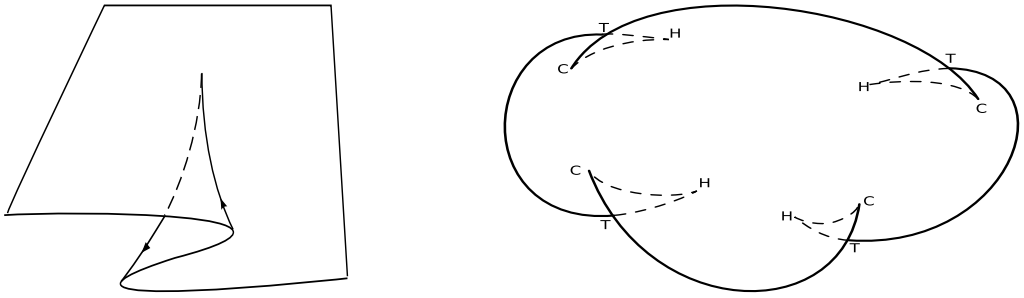


Figure 3.6: (Left) The generic projection of a contour at a cusp reverses direction at the cusp. (Right) A drawing with the tee-points and cusps marked; hidden contours and hidden cusps that must be inferred are shown in dotted lines.

For a generic smooth surface and viewpoint, tees and cusps of the contour will be isolated, as will curvature zeroes of the contour; this guarantees a unique osculating plane at a cusp, which means the projected contour must reverse direction rather than emanating from the cusp in any other direction (see Figure 3.6, left).

The "bean" example (Figure 3.5) is something of an archetype for the method described in this chapter, in the sense that it's the simplest shape that has a cusp; the way that this single cusp is processed is the key to processing more general drawings, hence we use the bean as an example throughout.

In Figure 3.6 (right) we show in solid lines a typical input drawing; in dotted lines are the projections of invisible contours. Certain hidden contour points are also cusp-points; the visible cusps are marked with a "C" while the hidden cusps are marked with an "H".

Note that the user input is the part of the contour drawn in solid lines. Everything marked by a dashed line is a part of a *hidden* contour and needs to be inferred by our program.

With this terminology, our goal is to take a user-provided directed visible-contour drawing of a good surface as above and to determine a surface S whose visible contours match the given drawing. Note that we do not seek to reconstruct exactly the surface that the user was drawing; the map from surfaces to drawings is many-to-one, hence non-invertible. Note too that we require that the drawing arise from *some* surface, so that the problem has at least one solution; a drawing consisting of a single line segment, for instance, cannot be the projection of the visible contours of any surface.

Our system currently produces *a* surface consistent with the user's drawing, and one which we generally find to be plausible. Eventually, we would like to solve a more general problem: we want not only to produce *one* of the reasonable-looking shapes, we would

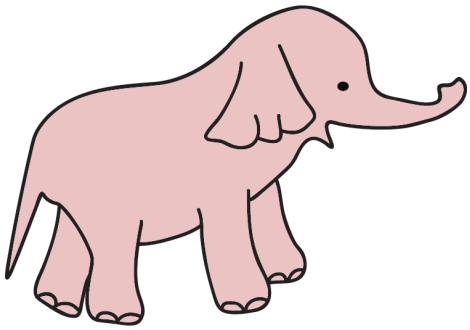


Figure 3.7: This cartoon-like illustration shows us how even the simplest drawings can have complex contours.

like to return *the most natural shape*. Of course, "most natural" can be very subjective and depend on a user's preferences, but experience shows that people generally agree about what a drawing conveys. We could take cartoon illustrations (see Figure 3.7) as an example. These pictures vary from simple to very sophisticated, but their expressiveness is such that people interpret them immediately. Such a degree of "naturalness" (or indeed, any way of measuring it) appears to be a very long-range goal.

3.3 The Flow of Operations in the System

The flow of operations in SmoothSketch [56] is shown in Figure 3.8. The boxes contain the implementation details of three major steps of the algorithm: figural completion, paneling construction and smooth embedding. In the next sections we describe the details of each of these steps.

3.4 Figural Completion for Smooth Surfaces

Given the visible contour drawing, in the $z = 0$ plane, for a good surface in $\Re^3$, we describe an approach to completing the drawing, i.e., adding hidden contours so that the resulting drawing can be Huffman-labeled. The approach works in a large number of cases, although not all. The existence of this algorithm raises a question: what drawings are completable? The complete answer is currently unknown, but we discuss our preliminary results in Section 3.4.6.

We assume that visible contours are oriented, that is, a user always draws contours so that the surface is to the left of the contour (see Figure 3.9). We also assume that the

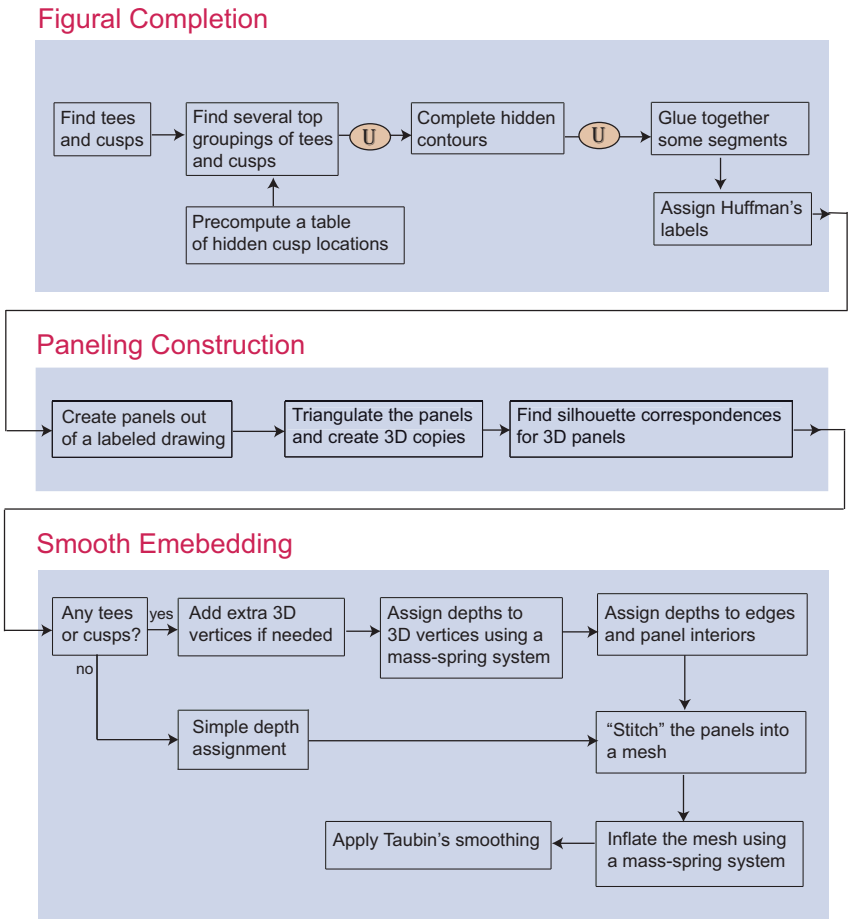


Figure 3.8: The flow of operations in SmoothSketch. Pink ovals marked with a letter U represent the places in the pipeline where the user can interfere.

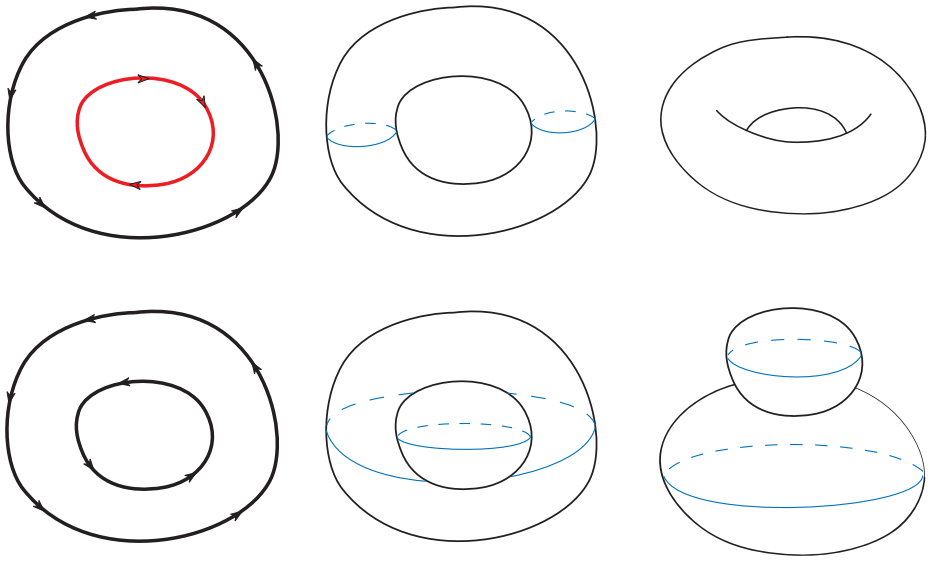


Figure 3.9: SmoothSketch expects oriented contours as input — the user should draw contours so that the surface is to the left of the contour. Thus, to create a donut, the user draws a counter-clockwise outer stroke and a clockwise inner stroke (top row). If the inner stroke was drawn in a counter-clockwise manner, the system would create two blobs one in front of the other (bottom row).

hidden contours do not intersect each other or visible contours except at junctions.

To complete hidden contours, we consider all visible-contour endpoints, and estimate the likelihood of a hidden contour joining each possible pair. Following Nitzberg *et al.* [79], we pair up points using their greedy algorithm, testing multiple configurations for (a) probability, and (b) consistency (*i.e.*, can they be Huffman-labeled?). The probabilistic model presented here is only meant to be suggestive, and not a model of actual probabilities.

3.4.1 Preprocessing an input stroke

User input is gathered from an input device (a mouse in our case). A user indicates when a stroke begins and ends by pressing or releasing a mouse-button. But during the input of each stroke, the 2D points arrive at an arbitrary rate. We re-sample these points so that they are not very close to each other. We use Igarashi's routine for resampling input [47]: when there are too many points close together, we simply ignore some samples[2]. This rather crude re-sampling seems to have no substantial impact on the results.

[2]If the distance between the previous point and the current point is less than 15 pixels, we do not add the current point

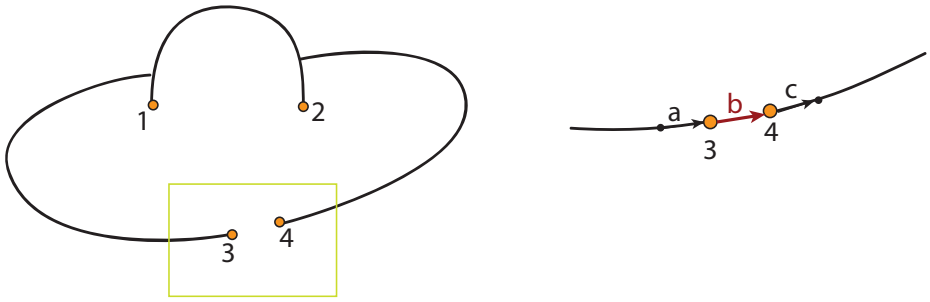


Figure 3.10: (Left) Endpoints 1 and 2 are visible cusps, while points 3 and 4 are "pseudo-cusps": regular points that should be later glued together to lie on a single stroke. (Right) We compute a vector b as a vector connecting endpoints 3 and 4, vector a as connecting the point previous to 3 and 3, and vector c as connecting 4 and the point next to it in the stroke. We classify endpoints as "regular" if the distance between them is small, and the angles between vectors (b, a), and vectors (b, c) are less than epsilon.

3.4.2 Guessing T-points and cusps

Before guessing hidden contours we first identify the visible cusps and tee points. These can be easily found on a drawing:

- We can determine T-points as stroke endpoints that "touch" some other segment ("touch" means "lie within 12 pixels").
- All other endpoints are initially classified as cusps. Some of them are "real" visible cusps and some of them are "regular" endpoints (see Figure 3.10). "Regular endpoints" are points that are close to each other and belong to the same stroke. A user just happened to draw the stroke in several segments. We can classify endpoints as "regular" if the distance between them is small and angles between the segments shown in Figure 3.10 (Right) are below some threshold.

3.4.3 Pairwise completion

First, for each pair of endpoints of the visible contour we compute an initial estimate of the probability that they are connected by a hidden contour. Each endpoint has a location and associated direction for the completion curve. For T-points or "regular" points, the direction is given by the tangent ray of the visible contour; for cusps it is the opposite (see Figure 3.11).

To compute the likelihood of joining two tees or two cusps, we compute an energy function for the pairing, inversely proportional to the likelihood. The energy function of

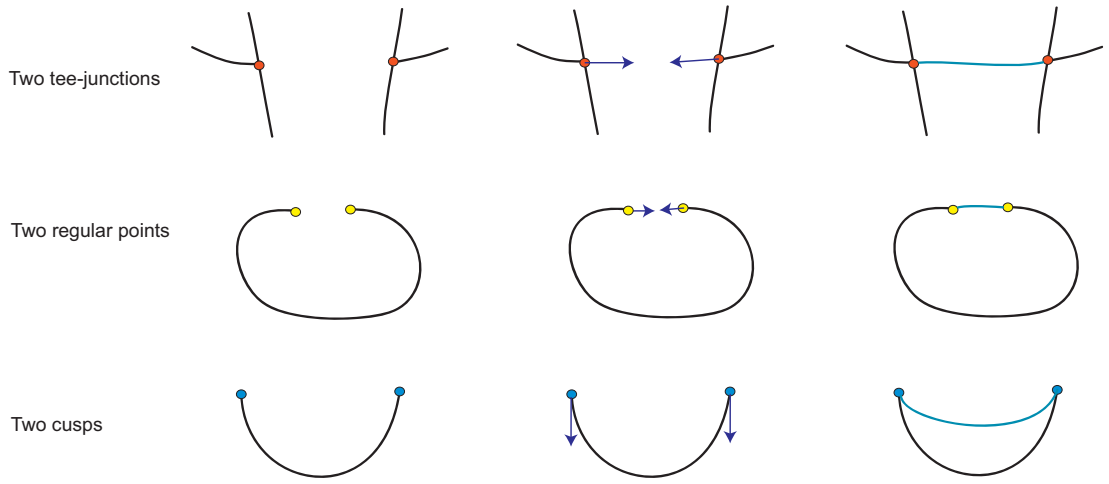


Figure 3.11: The Figure shows how tangent directions and bezier completion curves are computed when two tee-points, two regular endpoints or two cusps are paired up.

the pairing is a sum of two energy functions $E = E_{curve} + E_{endpoints}$, where E_{curve} is the energy of the curve that would connect them were they to be matched and $E_{endpoints}$ is the energy corresponding to the heuristic defined by the endpoint tangent directions. We approximate the elastica curve with a Bézier spline connecting the endpoints given their tangent vectors (see Figure 3.11). The Bézier curve is defined by the two endpoints and the points displaced from the endpoints along the tangent vectors. The distance by which the endpoints are displaced along the tangents is $\frac{1}{3}$ of the distance between the endpoints. The Bézier curve is then uniformly sampled and the energy function of the resulting polyline is computed as follows (see Figure 3.12):

$$E_{curve} = e^{\sum_i l_i} \cdot \sum_i \Delta\theta_i$$

where l_i is the length of the i-th segment of the polyline, and $\Delta\theta_i$ is the absolute value of the angle change between two consecutive segments of the polyline. $E_{endpoints}$ corresponds to another heuristic similar to [79], where we use the tangents at the endpoints to estimate the likelihood of the matches. Intuitively, if the tangent directions at the two endpoints are very similar, it is likely for them to be paired even if the length of the curve connecting them would be long (think of a fat snake whose tail passes behind its body). Similarly, if the tangents at the endpoints are very different, it should be pretty unlikely for them to be paired up. Currently, $E_{endpoints}$ is a constant 1.0 if the angle between the tangents at the endpoints is between $\epsilon_1 = 0.3$ and $\epsilon_2 = 2.5$, 0 if the angle between them is $\leq \epsilon_1$ and proportional to e^{angle} if the angle is $\geq \epsilon_2$.

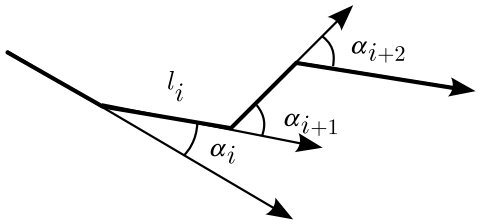


Figure 3.12: The energy of the polyline approximating the Bézier spline is computed as a product of the sum of angle changes between the consecutive segments and the exponent of the sum of the segment lengths.

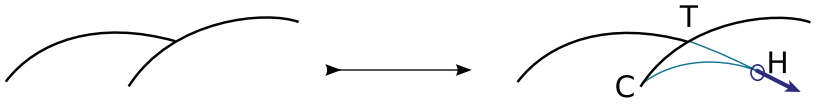


Figure 3.13: When we have a T-point and a cusp to match it to, we seek the location of a hidden cusp such that the two hidden contour parts joining our points to the hidden cusp have the highest probability.

Computing the completion for a tee/cusp pairing

To compute a hidden contour completion for the case of a tee/cusp match (shown in Figure 3.13), we need to estimate the location of the hidden cusp and the tangent direction at the hidden cusp. Formally, given the point T and the tangent vector V_t at T, the cusp point C and tangent vector V_c at C, we need to compute the position H and the tangent vector V_h of the hidden cusp (see Figure 3.14). Following the work of Williams [108] (done for the case of hidden contours connecting tee-junctions), we use the probabilities of directional random walks to estimate H and V_h.

Ideally, we would like to simulate two sets of directional random walks to estimate H and V_h as follows. We would store a table, whose entries are sampled x-coordinates on the plane, sampled y-coordinates on the plane and sampled angles (defining directions); *i.e.*, each entry is a point-direction pair, representing a small box of points and a small range of angles.

To fill in the table, we would start a directional random walk from the point T with the direction V_t. At each step we would vary the angle by $\Delta\theta$ chosen from a $(0, 1)$ Gaussian distribution. After a fixed number of steps (chosen for each random walk from an exponential distribution with mean 60), a walk would end up at a point on the plane with some direction. Then we would increment the table cell corresponding to this point and direction by one. After simulating many random walks, we would divide every value in the table by

the total number of random walks. So, each cell of the table would contain an estimate of the probability that a directional random walk starting from T with V_t ends at the point and with the direction defined by this cell.

The same process would be repeated for C and V_c with a new table of probabilities. In the end, we would have two tables: one for T and V_t, and the other for C and V_c. For each point and direction on the plane these tables would contain probabilities that the corresponding random walk will end up there. To guess a location of a hidden cusp, we could multiply the probabilities in the corresponding cells in two tables and take the point and direction with the highest product probability. This point/direction pair would be the estimate of the location and the direction of a hidden cusp.

To get a reasonable estimate of the hidden cusp location, we would need to run each of two random walks about 10^6 times, which means that our system would not be interactive. To overcome this problem, we precompute a table of probabilities as follows (see Figure 3.14, right):

- Fix the first point at $(0, 0)$ and the direction at angle 0.
- For (α, ϕ) defining the second point (on a unit circle) and its direction, store the resulting point (which is a hidden cusp and calculated as the point where two random walks meet with the highest probability) and the direction at this point (x, y, θ).

Now we can rotate and translate the coordinate system formed by an arbitrary tee/cusp pair so that the tangent vector at the point T lies at the origin and is aligned with an x-axis. Then we scale so that the cusp C lies on a unit circle in the new coordinate system. We can then use the coordinates of these two points and two corresponding directions as an input to our table. After we get the resulting location and direction of "the best" hidden cusp for these two vectors, we rotate, translate and scale it back to the original coordinate system. This simple pre-calculation technique allows us to guess the locations of the hidden cusps interactively.

We connect the hidden cusp to the tee point and the visible cusp with Bezier curves, and compute E_{curve} for their union as described above.

Greedy search for the best configuration

After a likelihood for each pair of endpoints is computed, we need to match up pairs to find the best total *configuration* (a configuration consists of endpoint pairs, where each endpoint appears in only one pair). For instance, if we have four endpoints numbered 1

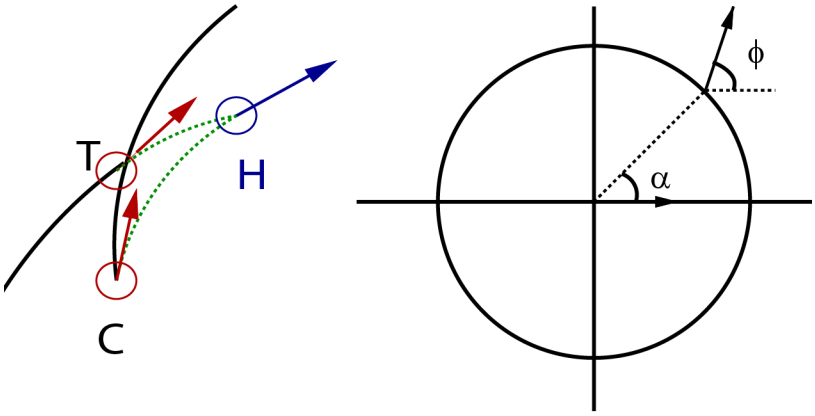


Figure 3.14: (Left) Given a tee point T, a cusp C and tangent vectors at these points, we need to estimate the hidden cusp H and the tangent vector at H. (Right) Precomputing the table of hidden cusp locations for a number of sampled locations and directions.

to 4, the possible configurations are: $\{(1,2),(3,4)\}$, $\{(1,3),(2,4)\}$ and $\{(1,4),(2,3)\}$. The likelihood of a configuration is defined as the product of likelihoods of its pairs. It is not practical to compute the likelihoods of all possible configurations as the number of them grows exponentially in the number of tees and cusps. Instead, we do a greedy search similar to [79].

In particular, we perform a standard 'beam'-search technique [87] to reach an approximate solution. The procedure is not guaranteed to reach the globally optimal solution; in practice we find that it works well for a reasonable number (10-15) of nodes.

Consider the search procedure as being analogous to searching a tree. Each node of the tree contains a particular pair of endpoints and a set of all valid pairs available to the children of this node. Each pair has an associated energy (of the bezier completion curve) computed as described above. We can choose the single pair that has the optimal energy at this point; note that the minimum energy is determined by taking a product of the energy of that pair with the energy of the path leading from the root of the tree to the node under consideration. If we choose to only explore the subtree corresponding to that pair we cannot know if the globally optimal value is indeed present in that subtree. Considering all possible pairs and exploring all possible subtrees at the node will result in exponentially many subtrees; making the procedure computationally intractable. 'Beam'-search presents a compromise between the two extreme choices. At each level of the tree, there is finite

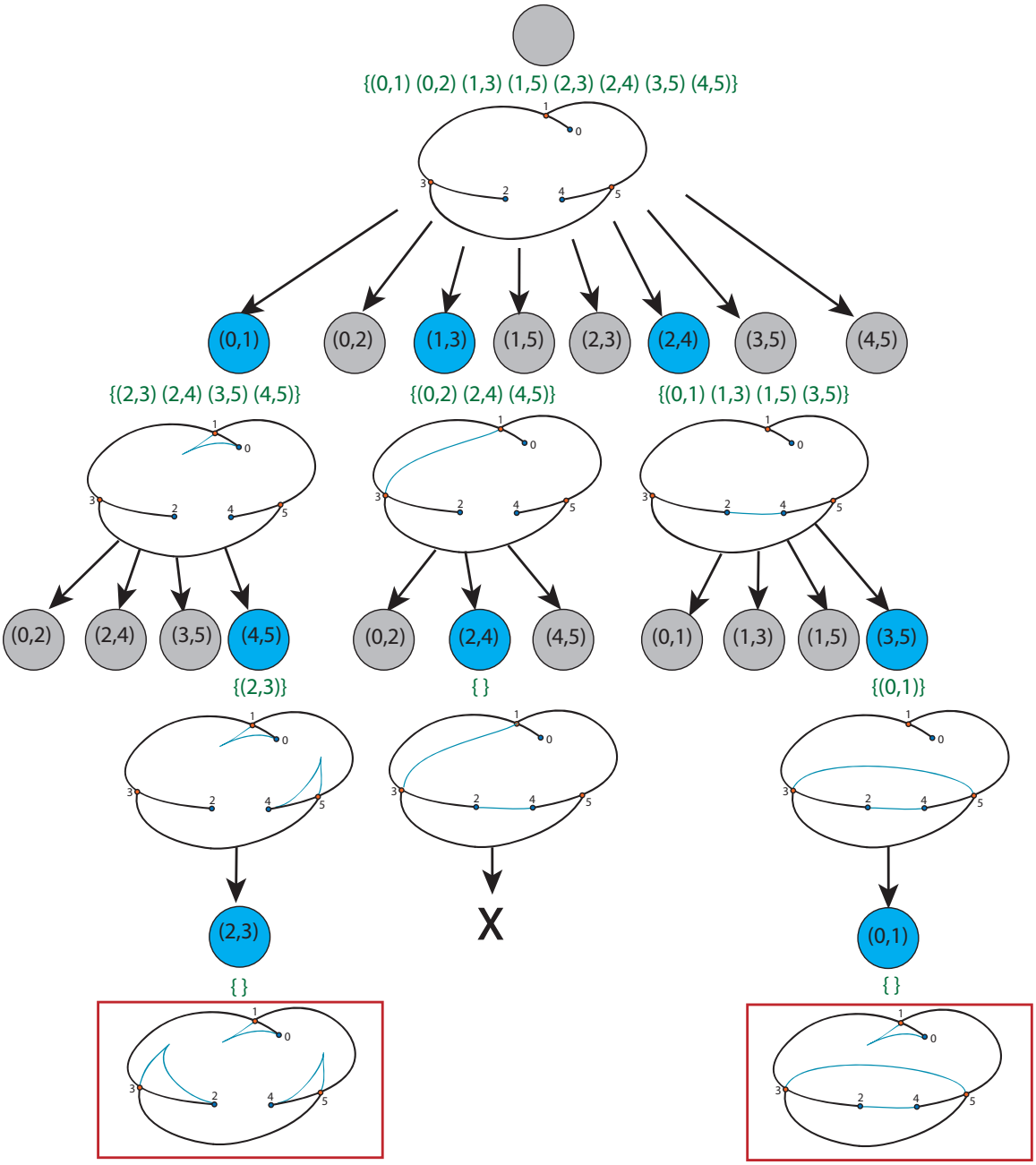


Figure 3.15: The figure demonstrates the 'beam'-search procedure for choosing the best valid configuration on an example sketch that has six junctions. Two best configurations are shown in the red rectangles.

budget of 'm' children (or possible pairs) that we can explore. 'm' is denoted as the 'beam-width' for the search procedure. We consider the set of all possible options at a certain level and choose m children with the least energy. Subsequently, only subtrees rooted at the m children will be searched for the optimal value.

Figure 3.15 demonstrates the 'beam'-search procedure for an example sketch that has six junctions numbered 0 to 5. First, we compute the list of all possible *available valid pairs* $I_0 = \{(0,1),(0,2),(0,3),(0,4),(0,5),(1,2),(1,3),(1,4),(1,5),(2,3),(2,4),(2,5),(3,4),(3,5),(4,5)\}$. For each pair from the list, we check whether the pairing is valid and remove invalid pairs from the set I_0 (we explain validity checks in the next section). Each node in the tree has a list of *available valid pairs* I_k - pairs available to its children (shown in Figure 3.15 in green font) and also every node except the root stores a pair chosen at the node (shown circled). In Figure 3.15, $m = 3$ children are searched and expanded at every level (shown in blue circles). Once a particular optimal pair, say $(1,3)$ is chosen, its node copies an array of available valid pairs from its parent and updates it to remove pairs that contain one of the indices of the pair. For example, for pair $(1,3)$ we would remove all pairs from the array that contain indices 1 or 3. In the end, the algorithm finds two valid configurations outlined in red. The second configuration corresponds to two objects - a bean in front of an oval-like blob.

Checking the consistency of the pairing

We perform several consistency checks for a pair of endpoints before adding it to the configuration. The first three rules we describe come from Huffman's labeling scheme for contours of smooth objects in generic projections (see Figure 3.16, top row) and should hold true for both endpoints in the pairing.

Since we assumed that we only deal with configurations in which hidden contours do not intersect other visible or hidden contours except at the endpoints, we add two other Rules (4 and 5) to eliminate configurations for which it is not true. Please note that rules 4 and 5 will be substituted with different rules (checking the consistency of Huffman's depth labels) if the assumption is removed in the future work.

1. **Rule 1:** Let T be the T-junction location, $\overrightarrow{V_T}$ - the tangent direction at the T, pointing away from the visible stroke, B - the Bezier point lying on the hidden contour closest to T, and $\overrightarrow{V_B} = \overrightarrow{(T,B)}$. Then, Rule 1 says that $dot(\overrightarrow{V_T}, \overrightarrow{V_B}) \geq 0$ (see Figure 3.16a, b).

2. **Rules 2 and 3:** Let C be the visible cusp, B - the point on the hidden contour closest to C, $\overrightarrow{V_C}$ - the tangent direction at the cusp, and $\overrightarrow{V_B} = \overrightarrow{(C,B)}$. Then, Rule

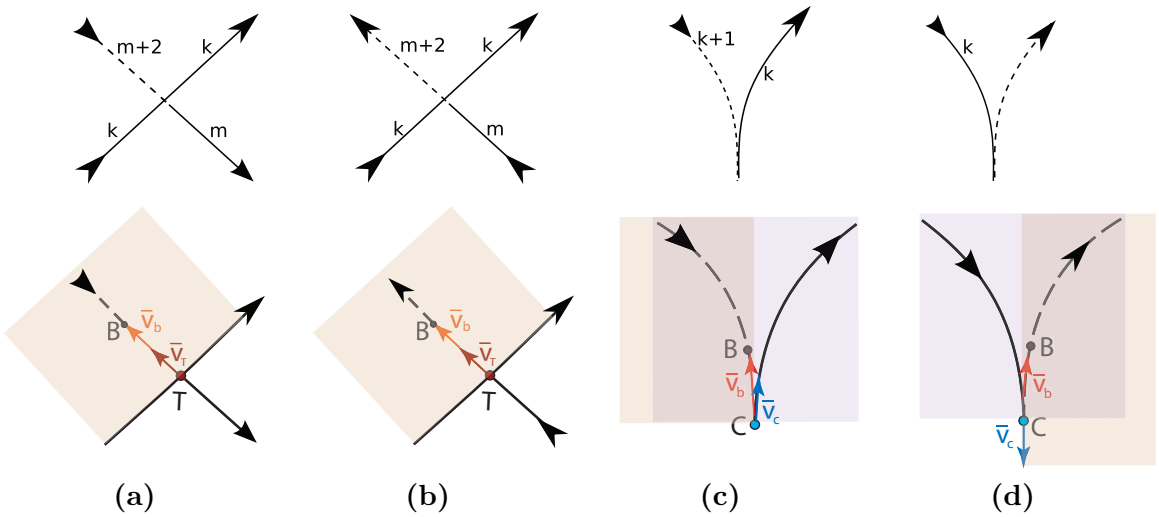


Figure 3.16: Topological consistency checks near the endpoints. Top row shows 4 Huffman's labeling cases, bottom row - the corresponding validity checks. (a)-(b) Consistency check at the T-point. Let T be the T-junction location, V_T - the tangent direction at the T, pointing away from the visible stroke, B - the Bezier point lying on the hidden contour closest to T, and $V_B = (T, B)$. Then, Rule 1 says that $dot(V_T, V_B) \geq 0$. (c)-(d) Let C be the visible cusp, B - the point on the hidden contour closest to C, V_C - the tangent direction at the cusp, and $V_B = (C, B)$. Then, Rule 2 says that the cross-product of V_C and V_B should be equal to $(0, 0, -1)$ (assuming the z-axis is perpendicular to the screen pointing away from the screen). Rule 3 says that $dot(V_B, V_B) \leq 0$.

2 says that the cross-product of $\overrightarrow{V_C}$ and $\overrightarrow{V_B}$ should be equal to $(0, 0, -1)$ (assuming the z-axis is perpendicular to the screen pointing out of the screen). Rule 3 says that $dot(\overrightarrow{V_C}, \overrightarrow{V_B}) \leq 0$. (see Figure 3.16c, d).

3. **Rule 4:** A pairing is invalid if the corresponding hidden contour intersects any visible stroke anywhere except at tee points or cusps it shares with this stroke.

4. **Rule 5:** Before adding any pairing to the beam search tree, check that the corresponding hidden contour does not intersect any hidden strokes of the nodes on the path to this one (except at tee points and cusps that the strokes share).

3.4.4 Gluing segments and assigning Huffman's labels

After the optimal valid configuration is computed, we glue together regular points by filling in the bezier curve between them. Now, these points belong to the same stroke and are removed from the list of endpoints. Then, we split visible strokes at tee junctions and assign orientations and Huffman's depth labels to all strokes. Since the optimal configuration does

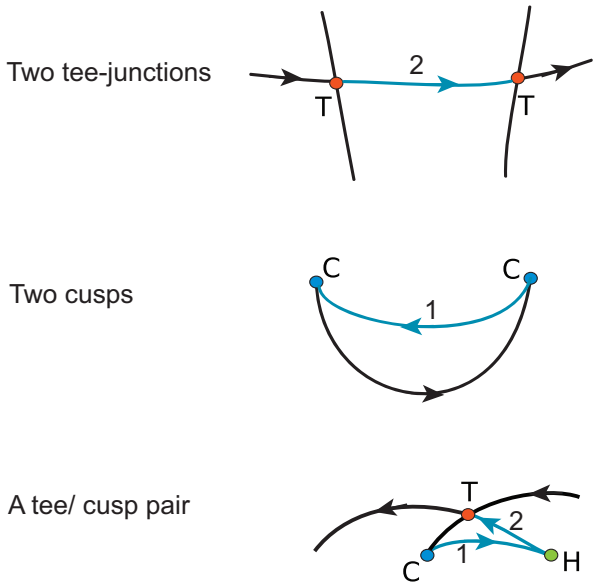


Figure 3.17: Assigning depth labels and orientations to hidden contours is straightforward if hidden curves do not intersect each other or visible contours. In this case, hidden contours connecting a tee point with either another tee point or a hidden cusp are labeled 2, and hidden contours connecting two cusps (visible or hidden) are labeled 1.

not have hidden contours that intersect each other or visible strokes, assigning depth labels is straightforward. We set the labels of all visible strokes to be 0, labels of hidden strokes connecting two tee points or a tee point and a hidden cusp to be 2, connecting cusps (visible or hidden) to be 1 (see Figure 3.17).

3.4.5 Results and Limitations of the figural completion algorithm

Some results of the hidden contour completion performed by SmoothSketch are shown in Figure 3.18. Simple drawings took less than a second to complete, while the more complex one took about 5 seconds.

The figural completion approach that we presented has a number of limitations.

The location of the hidden cusp provided by the above method may be unsatisfactory. Indeed, in the bean-like case shown in Figure 3.19, the hidden cusp is estimated to lie at a point that is not, in fact, hidden. Figure 3.19 (right) shows another example where the locations of hidden cusps are estimated incorrectly because of the simplifying assumption that the precomputed positions of hidden cusps are scale-invariant.

We assumed that hidden contours do not intersect visible and/or hidden contours, which

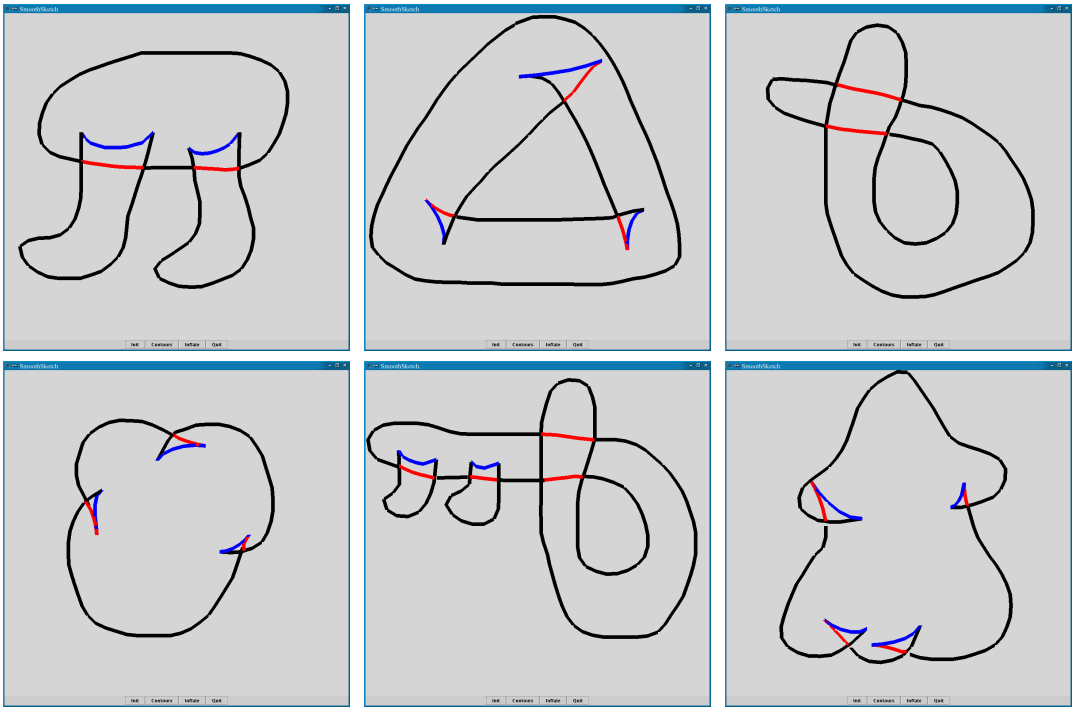


Figure 3.18: Some of the results of the figural completion algorithm. Visible contours are shown in black, hidden contours with Huffman's depth 1 are shown in blue and with depth 2 in red.

is a big simplification. In chapter 6 we discuss how we might change the algorithm to eliminate this assumption.

Consider a dog's body with one leg on the left hand side, seen from the right hand side (see Figure 3.20). This is a case that our contour-completion algorithm cannot handle. The "completion" of the obscured contours consists of two hidden cusps connected by a U-shaped hidden contour, and two "straight" segments connecting the hidden cusps to two t-points. In the two-hidden-cusp completion, the location of the two cusps is ambiguous. The algorithm for finding a hidden cusp for a t-point/visible-cusp pair will not work for this case, because there are no visible cusps.

The figural completion for this case could equally well consist of just an arc joining the two t-points — there's no a priori reason for the system to assume that the shape being drawn has only one connected component. Without the context (knowing that this is a leg), we do not know of any principled algorithm to guess the locations of the hidden cusps.

Our contour-completion algorithm, based on the table-lookup, should probably be improved. We would like to find a good approximation to the data in the table so that a lookup (and the computation and storage of the table) is unnecessary; if such a function

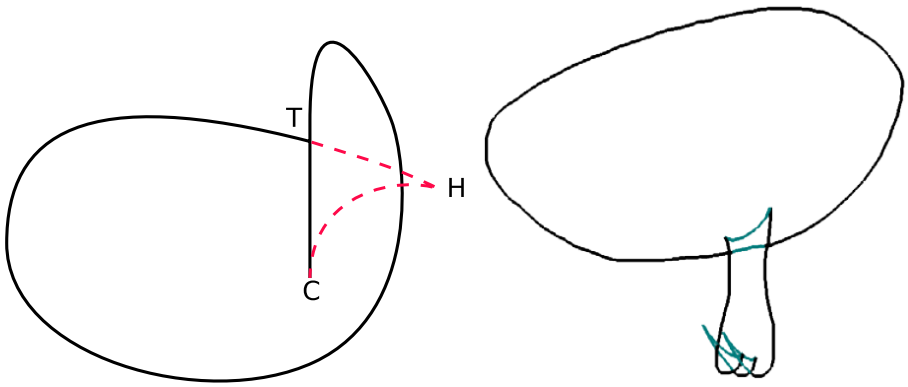


Figure 3.19: Problem cases: our method can produce a contour completion which places the hidden cusps at impossible locations. This happens because our method only considers local probabilities, and not the shape of the remainder of the visible contour.

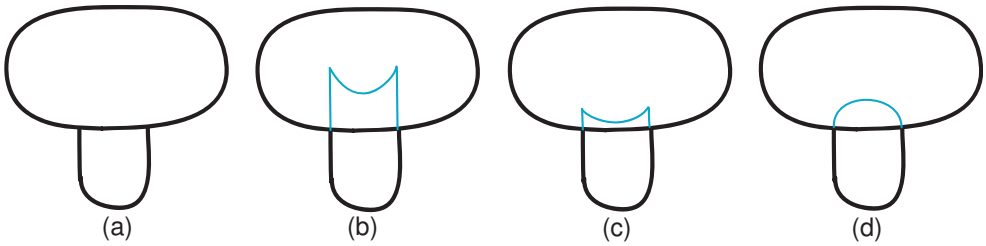


Figure 3.20: (a) The back-leg drawing case; (b),(c), and (d) show possible completions; our system produces completion (d).

were theoretically sound, the unprincipled "scale-invariance" assumption could be eliminated.

Given that figural completion is an expensive search problem, it becomes slower for a large number of tee/cusps (say, more than 15) and is more likely to make incorrect inferences as the drawing gets more and more complicated (we only find an approximate solution to the optimization problem to make it tractable; as a result, the approximate minimum is not always the global minimum).

3.4.6 Completable drawings

Although the class of visible-contour drawings that can be completed has not been characterized, to the best of our knowledge, we can demonstrate that a large class *can* be completed. Once again we consider the 'bean' as our example (see Figure 3.21). Taking a directed visible contour as input, we consider the regions into which it divides the plane;

traversing the boundaries of these regions, and pushing slightly into each region, gives a collection of disjoint embedded curves (which we call *red curves*), each of which may pass by some number of T-points or cusps. We give the key steps in an argument that if (a) the curve for the outermost region passes no such points, and has turning number one (i.e., can be smoothly deformed to a counter-clockwise circle) and (b) all other curves pass an equal number of "starting" and "ending" points, then there is a Huffman-label-able completion of the visible contour. The formal proof involves careful application of the tubular neighborhood theorem, the isotopy extension theorem, and other standard techniques from topology; we present only the essential insights here.

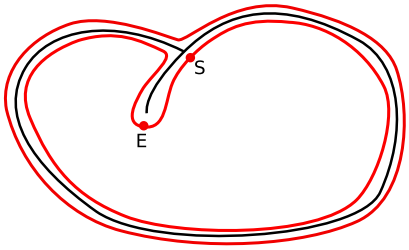


Figure 3.21: The visible contours of the bean divide the plane into two regions; traversing a path slightly displaced from the boundaries of these regions gives the two red curves shown. The inner one passes the cusp and then the T-point (we don't count where it makes a turn at the T-point as "passing" it); the passages are marked with dots. The outer one passes no endpoints at all.

The first step is to assign depth 0 to all visible edges. Now consider one of the red curves that meets at least two endpoints. Starting from any point of the red curve, we traverse it, noting whether the points we encounter are "starting points" (S) or "ending points" (E) of the visible contour arcs. The resulting circular sequence of Ss and Es contains at least one of each, by hypothesis; there must therefore be an adjacent pair of points, one a starting point and one an ending point. We'll show how to remove these from the sequence by completing the two contours; induction then shows that all contours can be completed and we are done.

The adjacent S and E points can be either cusps or Ts. Figure 3.22 shows how these can be joined. In the cusp-tee case, we can add a hidden contour and a hidden cusp, all within the region between the red curve and the contour between the two points being processed; after this addition, the red arc can be redrawn; the points S and E are no longer arc endpoints, and thus the start-end sequence for the arc is now two characters shorter. Similarly, in the T-T case, we can add a short completion arc. The only remaining case is

the cusp-cusp case. Depending on whether the cusps appear in S-E order or E-S order, one of two standard solutions shown provides the necessary completion.

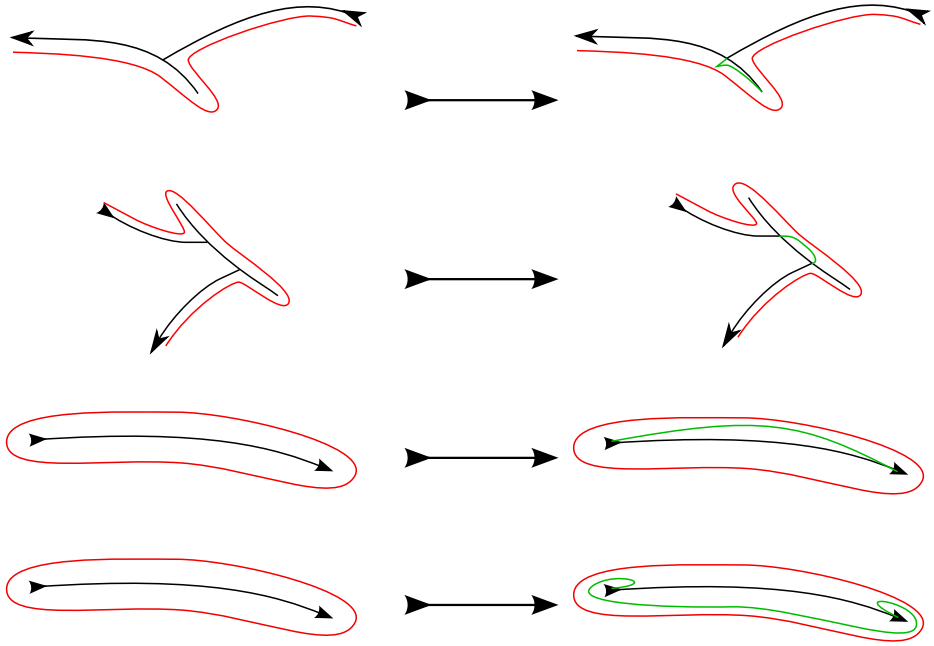


Figure 3.22: Adjacent cusp-T pairs can be joined with two arcs and a hidden cusp; adjacent T-T pairs can be joined with a single arc, as can adjacent cusp-cusp pairs; the side on which the arc lies, in this case, depends on the order in which the two cusps were encountered.

We have shown that a large class of visible contour drawings admit completions; it is known that others do not. But we do not know a complete characterization of which drawings admit completions. We note that there are drawings that do not satisfy the criterion above, but which nonetheless admit completions, so this class, although large, is not exhaustive.

3.5 From Drawing to Topological Embedding

At this point we have a completed contour drawing, with a Huffman-labeling. This drawing consists of directed arcs (which we call *edges*) between *vertices* corresponding to T-points and cusps.

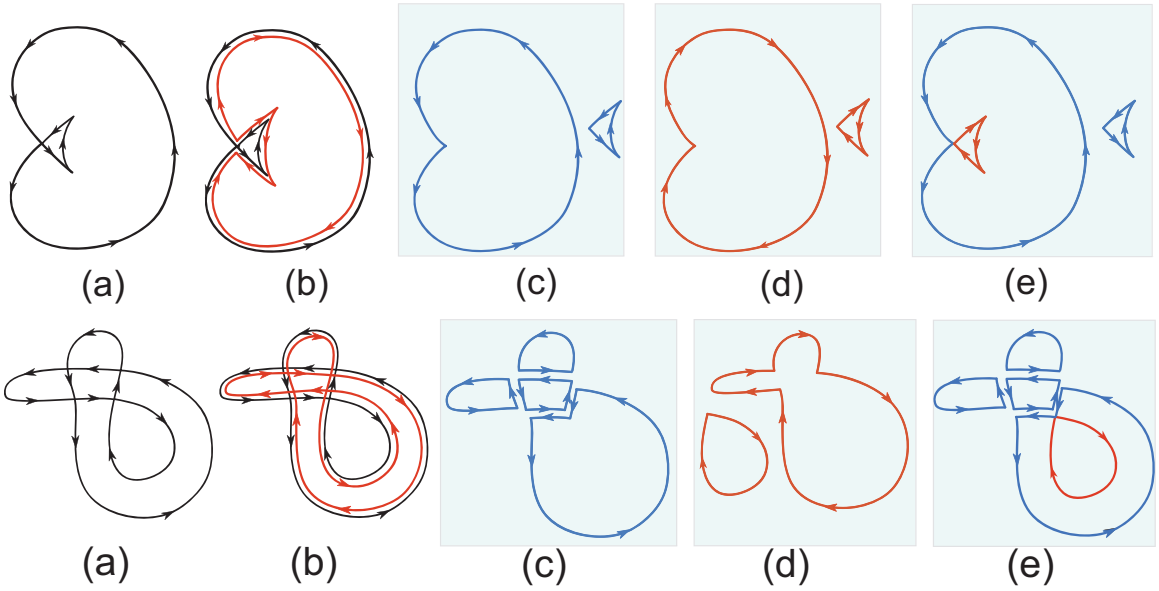


Figure 3.23: (a) shows the labeled drawing (the edges shown in black correspond to the true edges in the graph); (b) shows the graph with extra edges added (shown in red); (c) shows blue counter-clockwise cycles and (d) red clockwise cycles found in the graph. The final set of panels is shown in Figure (e).

3.5.1 Creating 2D panels from a labeled drawing

A set of strokes with orientations assigned to them split the plane into planar regions. The outermost one is ignored, the inner ones (we call them "2D panels") will be determined from the input. As a result of this step, we would like to represent each 2D panel as a closed loop of consecutive strokes that form a boundary of the panel. This is a graph problem. First, we construct a directional graph where the nodes are tees and cusps, and the edges correspond to the strokes connecting them; the orientation of the stroke determines the direction of the corresponding edge (see Figure 3.23a). If a stroke does not contain any tees or cusps, the node is just its first point. Then the algorithm performs the following steps:

1. For every edge of the graph, add the edge going in the opposite direction to the graph. Keep a flag for each edge that either marks it as a "true" edge (it was originally in the graph) or not (in Figure 3.23b the true edges are shown in black and the extra ones in red).

2. Starting from each of the "true" edges, find "valid" counter-clockwise edge cycles in the graph. While traversing the graph searching for the cycles, always choose the "leftmost outgoing edge" (see Figure 3.24). The "leftmost" outgoing edge l for a

given incoming edge e is determined as follows. Find the last segment of the stroke corresponding to the incoming edge e, call it v_e. Find the first segment of each stroke corresponding to the outgoing edges for e. Choose the outgoing edge corresponding to the leftmost segment v_l with respect to v_e (by computing the minimum clockwise angle α between $-v_e$ and v_i). To prevent us from choosing the edge e' going in the direction, directly opposite to the direction of e, we compare α with the threshold (0.03). The strokes corresponding to the counter-clockwise cycles found at this stage are called "preliminary panels" (they are shown in Figure 3.23c in blue).

3. Remove the edges from the graph that are part of the counter-clockwise cycles found above.

4. Find clockwise cycles. Add them to the list of "potential hole cycles" (shown in Figure 3.23d in red).

5. Go along the list of preliminary panels and check if any of the potential hole cycles are entirely contained in some panel. For each panel, we first find the list of holes contained "inside" the panel (by comparing the bounding boxes of the panel stroke and the hole stroke). Then, if a hole lies inside the panel, we check if its stroke can be inserted into the panel stroke (*i. e.* if they have a common node). Figure 3.23e shows examples of hole cycles being inserted into one of the panels[3]. Otherwise, the stroke of the hole is just added to the list of holes maintained by the panel. The holes that do not belong to any panel are ignored.

A more formal description of an algorithm that solves this problem for a case of one connected component is given in Chapter 3.2 of [73].

As a result of this algorithm, we have a list of 2D panels, some containing the array of hole strokes.

3.5.2 Triangulating the panels; the issue of two distinct points having the same 2D location

Each of these 2D panels is now triangulated to form a "3D panel" [93]. From now on, we will use the term "panel" to refer to a 3D panel.

[3]In the current implementation (using JDSL library [2]), the program reports a cycle if it encounters a vertex that was previously visited. As an alternative, one could modify the algorithm to report a cycle only when the next edge is equal to the first edge. Then, at the second step, a set of preliminary panels shown in Figure 3.23e will be computed. That would make a part of the fifth step (merging some of the hole cycles with the panel cycles) unnecessary. Note that one would still need to compute hole cycles since there may be several connected components.

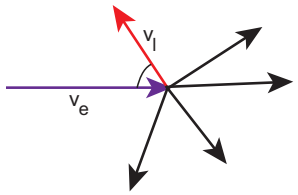


Figure 3.24: The last segment of the incoming edge is shown in purple, the red and the black vectors show the first segments of the outgoing edges. The red one is chosen as the leftmost one.

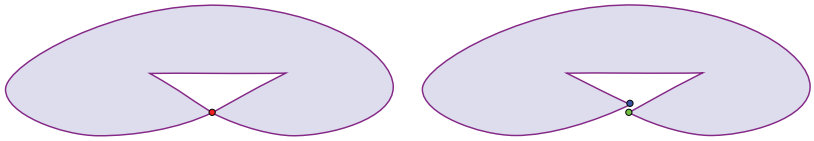


Figure 3.25: (left) A panel for a bean shape. (right) There are actually two distinct points corresponding to the original red 2D vertex.

Consider the stroke of the big panel for the kidney bean drawing - see Figure 3.25. There are two distinct points at the bottom of the drawing and they should remain distinct points in the mesh after the triangulation. We identify such cases at the figural completion stage and add extra information to the panel to be able to keep track of such points on the stroke. Then, before triangulating the panel, we move one of the points slightly towards the average of its neighbors (at this stage all points in the stroke are distinct), triangulate the panel, and then move the corresponding vertex back into its original position.

3.5.3 Paneling construction

We consider (see Figures 3.26, 3.27) the disjoint copies of a region R as being of the form $R_i = R \times \{i\}$, where the index i never appears more than once in all copies of all regions. A typical identification in Williams' scheme is then that the point (r, i) is identified with (r, j), where r is a point on the boundary of region R, and (r, i) and (r, j) lie in R_i and R_j respectively; another might be that (r, i) is identified with (s, j), where R and S are adjacent regions in the plane both containing the point $r = s$ on their boundaries, $(r, i) \in R_i$ and $(s, j) \in S_j$. The disjoint union of all the copies of all the regions will be called U; there's a natural map $\pi : U \to \Re^2 : (r, i) \mapsto r$ in which the multiple copies of any point r are all mapped to r.

For a point P in the plane, the set $\pi^{-1}(P)$ is a set of points of the form (P, i); we call

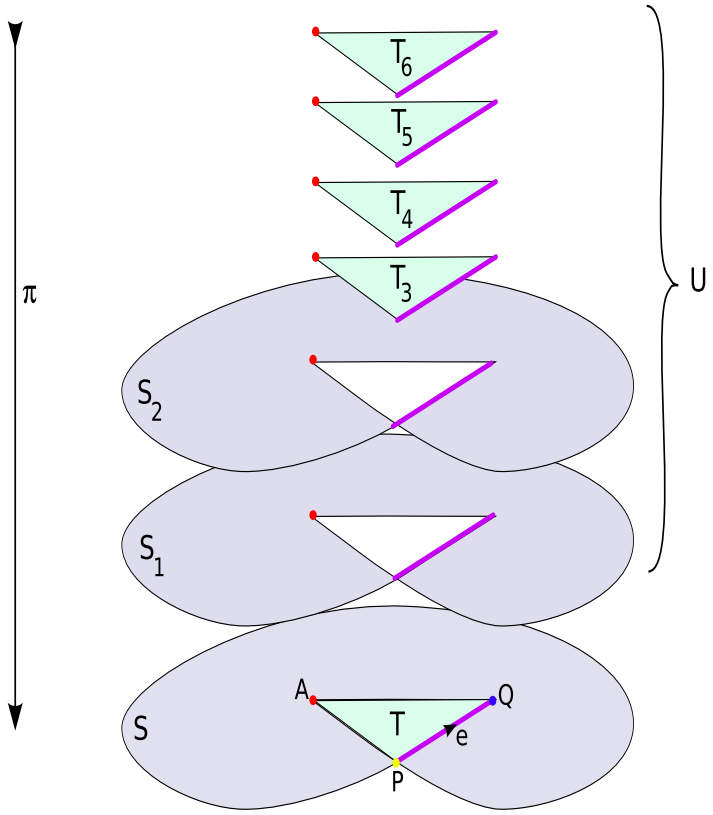


Figure 3.26: Schematic view of the disjoint union of panels that are glued to form the topological manifold homeomorphic to the bean. Each copy of each panel lies in a different layer; the union of all these copies is called U. The map π is "projection back to $\Re^2$ along z." The collection of all points that project to A (the red dots) is called the "stack above A". The magenta edges are the stack above the edge e. Each panel is indexed by its height in z, so all panels have different indices.

this the "stack over P." Similarly, we can consider the stack of edges over an edge in the plane, or the stack of panels over a panel in the plane. If an edge e in the plane goes from P to Q, we write $\partial e = (P, Q)$ to denote that the boundary of edge e consists of the points P and Q, in that order. If e_i is an edge in the stack over e, then $\partial e_i = (P_i, Q_i)$ as well.

Williams identifies certain panel edges in pairs (see Figure 3.27), that is, for certain i and j, he declares that e_i is to be identified with e_j, which means that the point $(x, i) \in e_i$, is identified with the point $(x, j) \in e_j$. This identification induces an identification on the stacks above vertices: if e_i is identified with e_j, and $\partial e = (P, Q)$, we declare $P_i \sim P_j$, and $Q_i \sim Q_j$. The transitive closure of the relation $\sim$ partitions stacks into equivalence classes that we call *clusters*; each cluster in each stack corresponds to a vertex in Williams' surface, which we'll eventually embed.

Ordering the clusters. Williams' construction gives a depth order to the panels in each panel-stack; this order is generally unrelated to the indices above. This order induces an order on the clusters as follows: if P_i and P_j are in two clusters, and R is a region containing $P = \pi(P_i) = \pi(P_j)$ consider all the faces in the stack over R that are adjacent to vertices in the first cluster, and all those adjacent to vertices in the second cluster. By Williams' construction, faces in the first group will either be all in front of or all behind the faces in the second group; we say that the first cluster is in front of or behind the second group accordingly. Again by construction, this order is independent of the adjacent region R that we choose.

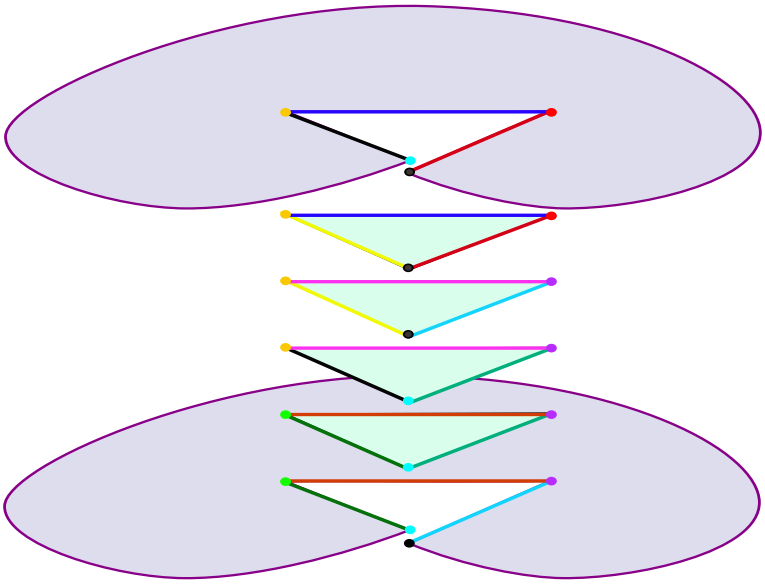


Figure 3.27: The panels, re-ordered for visibility; edges with the same colors are identified. This identifies *clusters* of vertices in each stack; vertices with the same color form a cluster. Note that the near vertex in the two large panels has been split into two copies.

Extra vertices. One important issue remains: if two edges e and e' in the same edge-stack have the same clusters as their endpoints but are *not* identified in the topological manifold, these distinct edges would be assigned the same depth in the constructed surface, which would result in a non-embedding. Figure 3.28 shows two such edges in the lower portion of the leg case. In such cases, we add a new vertex at the midpoint of each of the edges e and e' of the contour (and to any other edges that are identified with these). The stacks and the clusters within these stacks are then created for these newly-inserted points in the same way we described above.

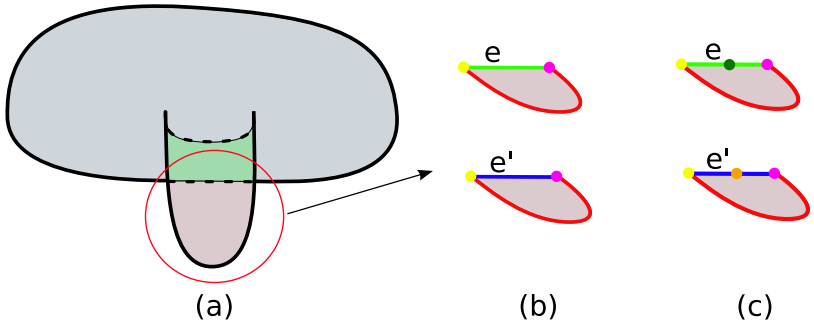


Figure 3.28: (a) A contour-completed drawing of a leg attached to a body, with panels colored. (b) The two panels for the bottom of the leg, colored to show edge identifications and vertex clusters. Note that the top edges e and e' share endpoints but are not identified. (c) We add mid-edge vertices, sort, and cluster them as before.

3.6 Constructing a Topological Embedding

We now present a novel algorithm that constructs a topological embedding from Williams' abstract manifold.

Embedding vertices To each cluster of the vertex stack over a vertex P, we associate a vertex whose xy-coordinates are those of P, and whose z coordinate is yet to be determined (we call these *cluster vertices*). We determine the z-placements using a mass-spring system. Suppose that the vertices corresponding to the clusters of one stack are X_α, where α ranges over the clusters. If cluster α is behind cluster β, we want the z-coordinate, z_α of X_α to be less than that of X_β. For each such order-relation between two of the Xs, we attach a spring whose rest-length z_0 is one, and for which the spring force follows the rule

$$F(d) = \begin{cases} 0 & d \geq 1, \\ Ce^{1-d} & d < 1 \end{cases}$$

which ensures that if the z-order is inverted, there's a substantial force pushing back towards the proper ordering.

This ordering and set of z-values could also be found by simply sorting the vertices; we use the mass-spring system as a way to relate the z-depths for vertices in separate stacks. In particular, if P and Q are distinct vertices joined by an edge e, then each cluster over P is joined to one or more clusters over Q by edges in the stack over e. For each such connection, we add a spring with rest-length zero between the corresponding cluster-vertices; we use a sufficiently small spring constant that the intra-stack ordering is not disturbed. Our goal is to make each edge want to be somewhat parallel to the $z = 0$ plane, rather than having

vertices associated with one stack be far in front of all others, for instance.

The mass-spring system acts on the points, which are constrained to move only in z. Clearly if the spring constant for the inter-stack springs is small enough, each stack will be ordered correctly. In our implementation, we use the constant 1.3, which seems to perform well on examples like the ones shown in this chapter and the associated video. The points of the drawings in our system lie in the bounding box of -1.0 to 1.0 in each direction.

Embedding edges Having embedded the cluster vertices (i.e., the vertices of the manifold that Williams constructs), we can extend the embedding to edges by linearly interpolating depth along each edge. The ordering of edges in Williams' construction is generally sufficient to show that if e_i and e_j are distinct edges of the manifold corresponding to contour edge e, then they do not intersect except, perhaps, at endpoints which they share. In the event that e_i and e_j share *both* their endpoints, linear interpolation would assign them the same depths at all points, and our mapping would not be an embedding. Fortunately, the "extra vertices" step above inserts points exactly when necessary to prevent this; thus we have an embedding of both the vertices and the edges of Williams' manifold.

Embedding faces We extend the embedding over the panel interiors using Poisson's formula to find a harmonic function on the panel whose values on the boundary are the given depth values that we've already assigned to the edges of the panel. Each interior point is assigned a depth that is a weighted average of the depths of points on the boundary edges. To prove that two panel interiors in the same stack never intersect, suppose that P is a point of some panel R, and that X and Y are points in the panel-stack over R, and that $\pi(X) = \pi(Y) = P$. Suppose that the panel to which X belongs, R_i, is in front of the panel to which Y belongs, R_j, so that the z-value for X should be larger than the z-value for Y. Then points on the edges of R_i are in front of (or equal to) the corresponding points on the edges of R_j. The z-coordinates of corresponding points cannot all be equal unless the boundaries of R_i and R_j are identical, in which case the union of R_i and R_j is a spherical connected component of the manifold, and is handled as a special case. In the remaining cases, since the z-values for R_i are greater than or equal to the corresponding values for R_j, and the z value for X is a weighted sum of these values *with all nonzero weights*, and the z-values for Y is the corresponding weighted sum of the other z-values, with the same weights, we find that the z value for X is strictly greater than that for Y. Thus the interiors of faces do not intersect. We have thus constructed a continuous 1-1 map from Williams' abstract manifold into $\Re^3$, i.e., a topological embedding.

3.6.1 Inflation of the objects where some strokes have no tees or cusps

Consider a drawing of a torus that consists of a simple outer stroke and a complex inner stroke corresponding to a hole (see Figure 3.29). In this case, although the internal stroke contains tees and cusps, the algorithm described in the chapter will not work - it is not clear how to assign depths to the 3D edge corresponding to the outer stroke (since a depth of an edge is computed as an interpolation of the depths of vertices that correspond to tees or cusps). We check for this case and add an extra 2D vertex v to the outer stroke - say, the first point of the stroke. One issue remains: we need to make sure the depths assigned to 3D vertices corresponding to v are related to the depths assigned to the vertices of the internal stroke (they should not be assigned independently - the hole of the torus should lie close to its outer boundary). To ensure that, we connect an extra vertex to the internal tees and cusps by extra length springs while creating the mass-spring system. After this stage, the algorithm follows the steps described in the chapter.

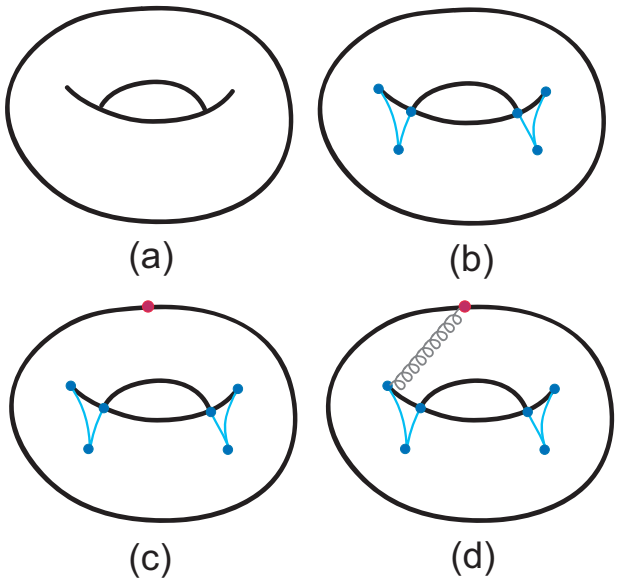


Figure 3.29: (a) A drawing of a torus. (b) 2D vertices corresponding to tees and cusps are shown in blue. (c) An extra vertex shown in red is added to the outer stroke so that the depths could be assigned to all edges. (d) Extra length springs connecting the red vertex to other vertices are added to the mass-spring system that is used to assign depths to vertices. Here only one of them is shown.

3.7 Smoothing the Embedding

3.7.1 Using the mass-spring system

Now that mesh vertices corresponding to each panel have been assigned depths, we "stitch" the meshes of individual panels into a single mesh. We start with the first panel, and stitch panels to it one at a time. If two panels are identified along an edge e, we alter the vertex indices on second to match those of the first. The edge correspondences for the stitched panel (excluding the edge we stitched along) come from the correspondence information of the two component panels. Although the resulting stitched mesh has the proper "contour projection" (for an appropriately modified definition of "contour"), its shape is generally unsatisfactory, as can be seen in the accompanying video. We therefore perform several optimization steps. During these steps, we constrain the vertices lying on the visible silhouette to remain on the silhouette so that the contours will match the drawing. That is, these vertices can only move in z, while others may move in x, y, and z.

First, we remesh the model using the algorithm proposed in [60] and apply ten iterations of Taubin's λ/μ smoothing [94] in order to create more regular triangles, as the behavior of the mass-spring system is sensitive to the quality of the triangulation. These operations are applied only to non-silhouette vertices and edges.

At this point, the mesh is smoother, but rather flat and sharp along the edges (because the silhouette constraints have not been incorporated smoothly). The next goal is to "inflate" the model, making it more rounded. To achieve this, we construct a mass-spring system on the initial mesh, with masses at the vertices and with two types of springs: length springs and what we call "pressure force" springs. The length springs try to keep the length of each edge as close to zero as possible, while the "pressure springs" simply push each triangle outward along its normal with a force proportional to the area of the triangle. We relax this mass-spring system and although the convergence in general is not guaranteed, in practice it converges quite fast. A model like the ones shown in the chapter inflates in several seconds on an AMD Athlon 64 3000+ processor.

Our mass-spring approach has several drawbacks; some of them are common to all mass-spring systems, others are particular to our choice of springs. First, most mass-spring systems approximate the physics of deformable models very crudely. Further, in our case, even the underlying "physical" model is quite *ad hoc*. We intuitively think of the current model as inflating the initial flat shape as a balloon, but with the restriction on the movement of silhouette points and disregard for surface curvature, it is a very weak analogy.

Secondly, our mass-spring system has several tuning constants that have to be chosen so that they work for most of the examples user draws.

Thirdly, there is currently no mechanism in the system to prevent self-penetrations of the surfaces. In fact, although different parts of the initial mesh are in the correct relative order, the mass-spring system could relax it into the configuration that is more or less planar (think of a worm example). We have not observed it in practice, but theoretically, the inflation algorithm does not prevent it from happening. Silhouette constraints prevent this issue to some degree: when we say that something is a cusp, the silhouette has to travel straight into the Z-direction at that point, which tends to mean that "bending back again to touch" is unlikely. This issue can be addressed by doing the relaxation in a way that does not allow silhouette vertices to move, or by inserting springs that preserve the relative order in the inflated mesh. Sometimes, though, we would like to allow self-penetrations: think of a body-with-two-legs example; there, the legs being slightly pushed inside the body is often more desirable than having them stick out far away from the body. Finally, there is a known problem of stiffness [33] with all mass-spring systems that leads to their potential instability.

Having said all this, the mass-spring system we created seems to work reasonably well on most examples. The results of the relaxation of the mass-spring system are satisfactory except at the areas that were completely flat and skinny initially (like the tips of the legs). Finally, we may choose to apply a few iterations of Taubin's anisotropic smoothing [95], which first filters the normals using λ/μ algorithm, and then filters the vertices, integrating the new normal field in the least squares sense. The final results are shown in Figure 3.30.

3.7.2 Evolver: minimizing the discrete squared mean curvature of the mesh

The inflation algorithm based on the mass-spring system relaxation has a number of limitations which we described in the previous subsection. The more principled approach to inflation would be to optimize some functional like squared curvature over the surface, conditioned on the known shapes of the contours.

We experimented with this idea using Brakke's Surface Evolver software [17] [18]. The Evolver lets the user specify constraints on mesh vertices and faces, and several energy functions as surface integrals (such as surface tension and squared mean curvature). Then it 'evolves' the original mesh to minimize the energy while preserving the constraints. Some energies, such as surface tension and several discrete approximations of mean and Gaussian

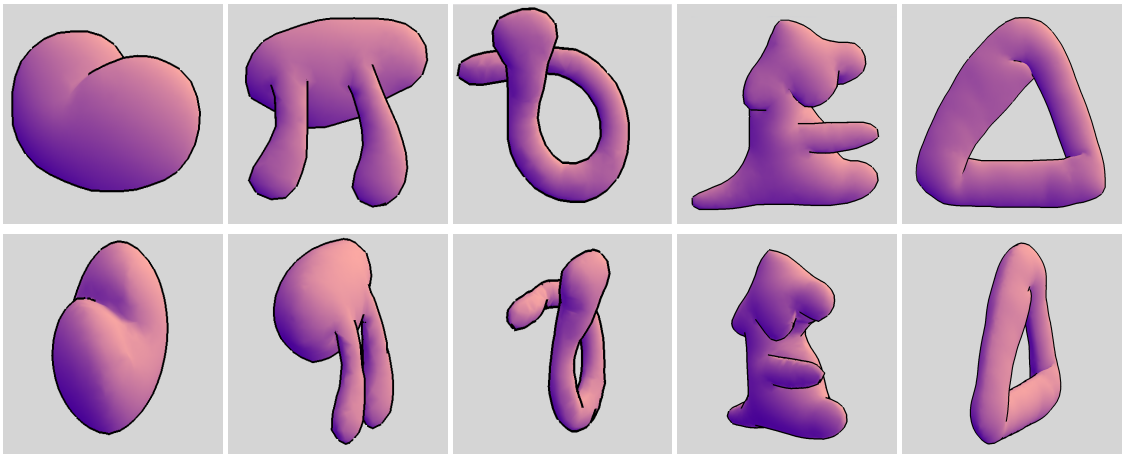


Figure 3.30: The examples of the shapes created with our system from user drawings. The top row shows the shapes from the sketching viewpoint, and the bottom row shows them from a different view.

curvature, are predefined. Additional energy terms can be defined by the user as integrals over the mesh in the input file. The Evolver performs optimization using a gradient descent method. It also provides several remeshing operations such as Laplacian mesh smoothing, face subdivision and face equiangulation.

The Evolver reads the initial mesh data together with the constraints and energy quantities from a text file. We fixed x and y coordinates of vertices on the visible and hidden contours. We marked silhouette edges with a "no-refine" flag to tell the Evolver not to change these edges during the remeshing operations. We specified two energies that we want the Evolver to minimize: E_1 — the "dihedral_hook energy", and E_2 — the "normal energy".

$$E_1 = \sum_e \|e\| \cdot (1 - \cos(\alpha))^2,$$

where e is a mesh edge, α is the angle between the normals of adjacent faces for the edge e, $\|e\|$ is the length of the edge.

$$E_2 = \sum_f (\overrightarrow{nt(f)} - \overrightarrow{n(f)})^2 \cdot flag(f),$$

where $\overrightarrow{n(f)}$ is a normal for the face f, $\overrightarrow{nt(f)}$ is the desired normal at face f, and $flag(f)$ is the flag that is 1 for the faces adjacent to the visible silhouette edges and 0 otherwise.

One evolution step consists of the following commands: compute target normals, do two gradient descent steps, do two Laplacian smoothing steps, compute target normals, and equiangulate non-silhouette edges.

To compute the target normals at the silhouette faces, we go along silhouette edges, compute the adjacent faces and set the target normal at each face to be the cross product of edge direction and line of sight. In the equiangulation step some edges are flipped to make the triangles more equiangular.

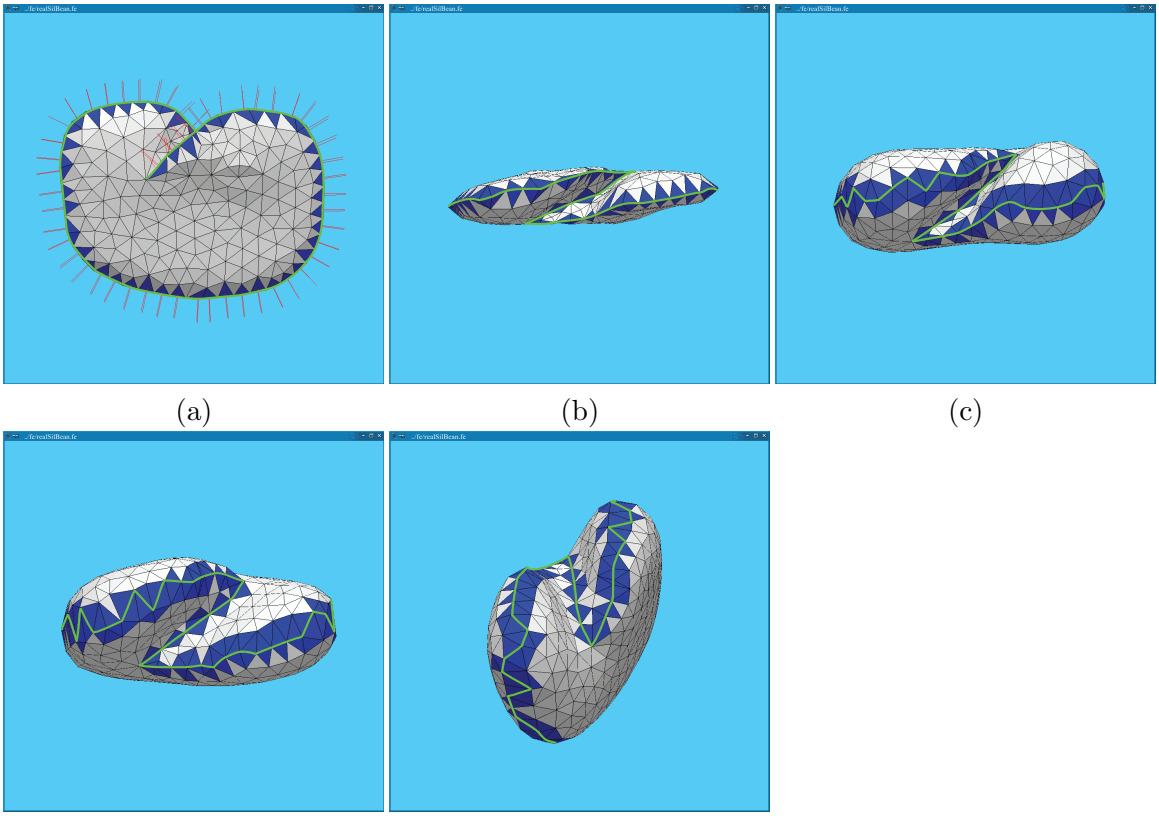
Figure 3.31 shows the result of several evolutions steps on the bean mesh. Although the obtained bean shape is smooth and satisfies the silhouette constraints, it is not the optimal shape we would like to get. One explanation for this could be that the optimization landscape is complex and the local search method simply could not find the global minimum.

We introduced a new inflation technique into SmoothSketch. The initial shape and the corresponding commands are written into a text file and the Surface Evolver is called automatically; the Evolver then performs a fixed number of iteration steps and returns the modified surface to SmoothSketch. Two results of the Evolver inflation are shown in Figure 3.32. We have to specify the target mesh volume for the Evolver. The results shown in Figure 3.32 were obtained by manually trying several different target volume numbers and choosing the best result.

The inflation approach that uses Surface Evolver to minimize the approximation of the squared mean curvature in the current implementation does not perform significantly better than the mass-spring system approach. The weights of the two energy terms: the curvature term and the normal term, the target volume, and the number of evolution steps need to be chosen by trial and error. In the future, we would like to automatically compute these parameters from the user contour drawing.

3.8 Editing Gestures

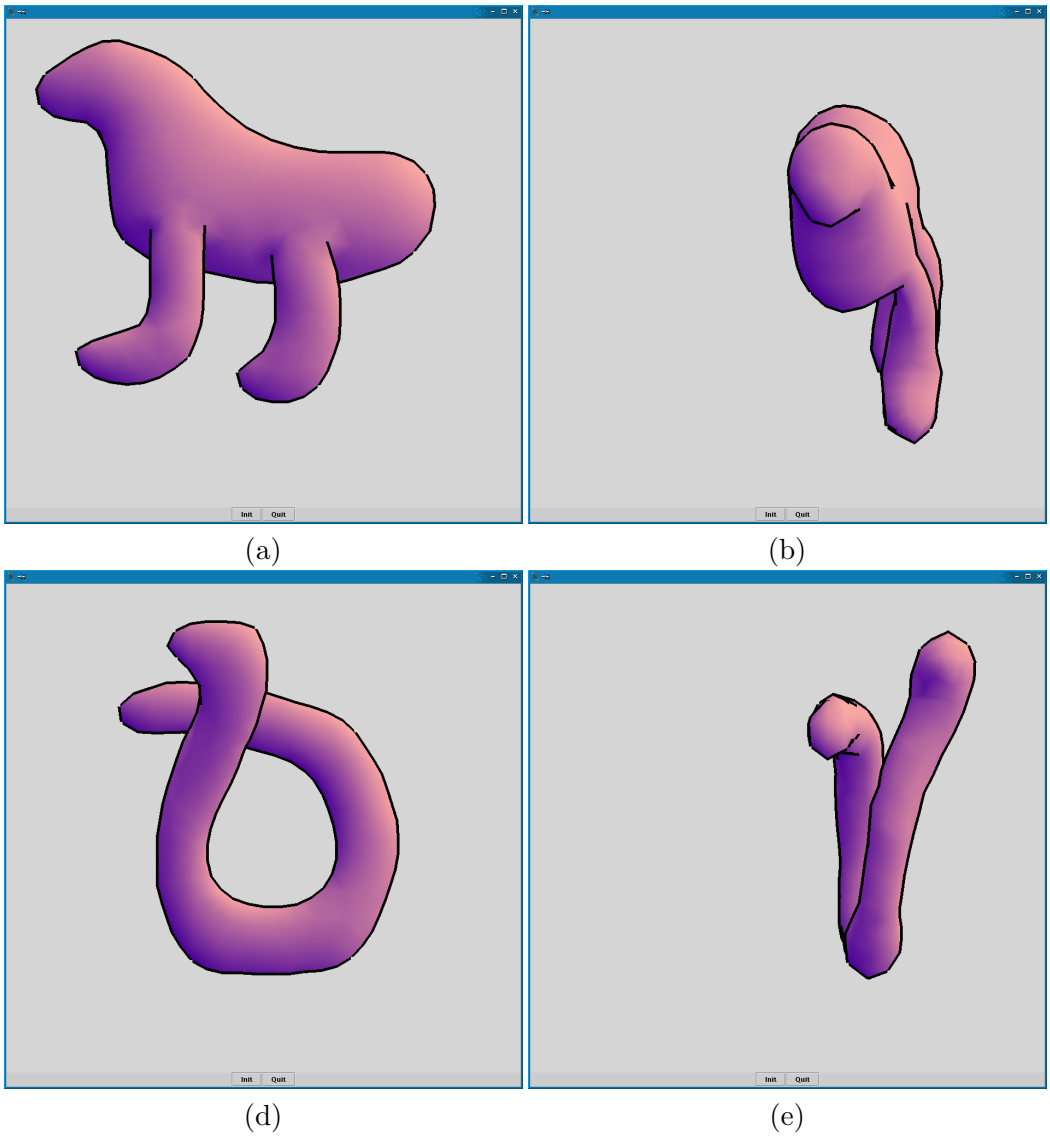
The system may return a different shape from the one that the user had in mind while drawing the sketch. We therefore let the user edit hidden contours or choose a better completion from the list of suggestions offered by the system. The user draws strokes by dragging the right mouse button; when he is finished drawing, he presses the middle mouse button and the sketch is inflated into a 3D model. The user can change the shape by pressing the “Contours” button at the bottom of SmoothSketch window. Then, in the main window, the system displays both visible and hidden contours inferred by the system. The hidden contours are shown in light blue and can be manipulated by dragging default Bezier control points for each hidden curve. In the second, smaller, window that shows up on top of the main one, the system shows up to three alternative hidden contour completions (see Figure 3.33). If the user selects one of the suggestions, the second window closes and the



(a) (b) (c)

(d) (e)

Figure 3.31: (a) The initial mesh of the bean shape. The visible contour is shown in green, the faces adjacent to the visible contour - in dark blue, and the target normals of the "silhouette" faces - in red. (b) The top view of the original mesh. (c) The top view of the bean mesh after 10 Evolver iterations, the weight of the normal energy is 5, the weight of the dihedral hook energy is 1. (d) The top view of the bean mesh after 100 Evolver iterations. (e) A different view of the final bean shape. The pictures were generated using Evolver's graphical interface.



(a) (b)

(d) (e)

Figure 3.32: Examples of shapes inflated using the Surface Evolver. Visible silhouette points are only allowed to move in z. The minimized energy consists of the "dihedral_hook energy" with weight 1 and the energy term for the normals with weight 5. (a)-(b) The target volume is set to 0.3. (c)-(d) The target volume is set to 0.1.

selection is shown in the main window where the user can edit it.

The list of suggestions lets the user change the topology of the model, while contour editing tool lets him change the shape of the hidden contours. Only valid contour completions are presented to the user, so the user can not use this tool to fix incorrect hidden cusp locations like the ones shown in Figure 3.19.

3.9 Discussion, Limitations, Conclusions

Our system creates 3D shapes for a wide class of contour drawings; but certain limitations prevent it from working universally.

One is that the contour-completion approach is local–the completed contour shape depends on the geometry of the starting and ending points, but ignores the remainder of the input shape; it will require a much deeper understanding of contour completion to address this.

Williams' topological manifold construction, followed by our lifting, creates a mesh embedding with "folds" matching the drawn contour. But mesh contours and smooth contours are different. In particular, the curvature of a smooth contour at a cusp goes to infinity, which a mesh-cusp simply projects to some non-zero angle in the contour-drawing. The problem of exactly fitting the drawing is therefore generally impossible, unless users respect the conditions on curvature at cusps. We need to develop a means to characterize when we have adequately approximated the user's drawing.

The inflation algorithm based on relaxing the mass-spring system, as well as the algorithm based on minimizing the sum of dihedral angles currently require tuning constants; the constants that produce the most satisfactory-looking results actually produce self-intersecting surfaces, especially in locations like "armpits" (i.e., between a limb and a body). We would like to find an algorithm that produces embeddings instead.

In Chapter 6 we discuss these limitations in more detail in the general context of sketch-based interfaces and propose some solutions.

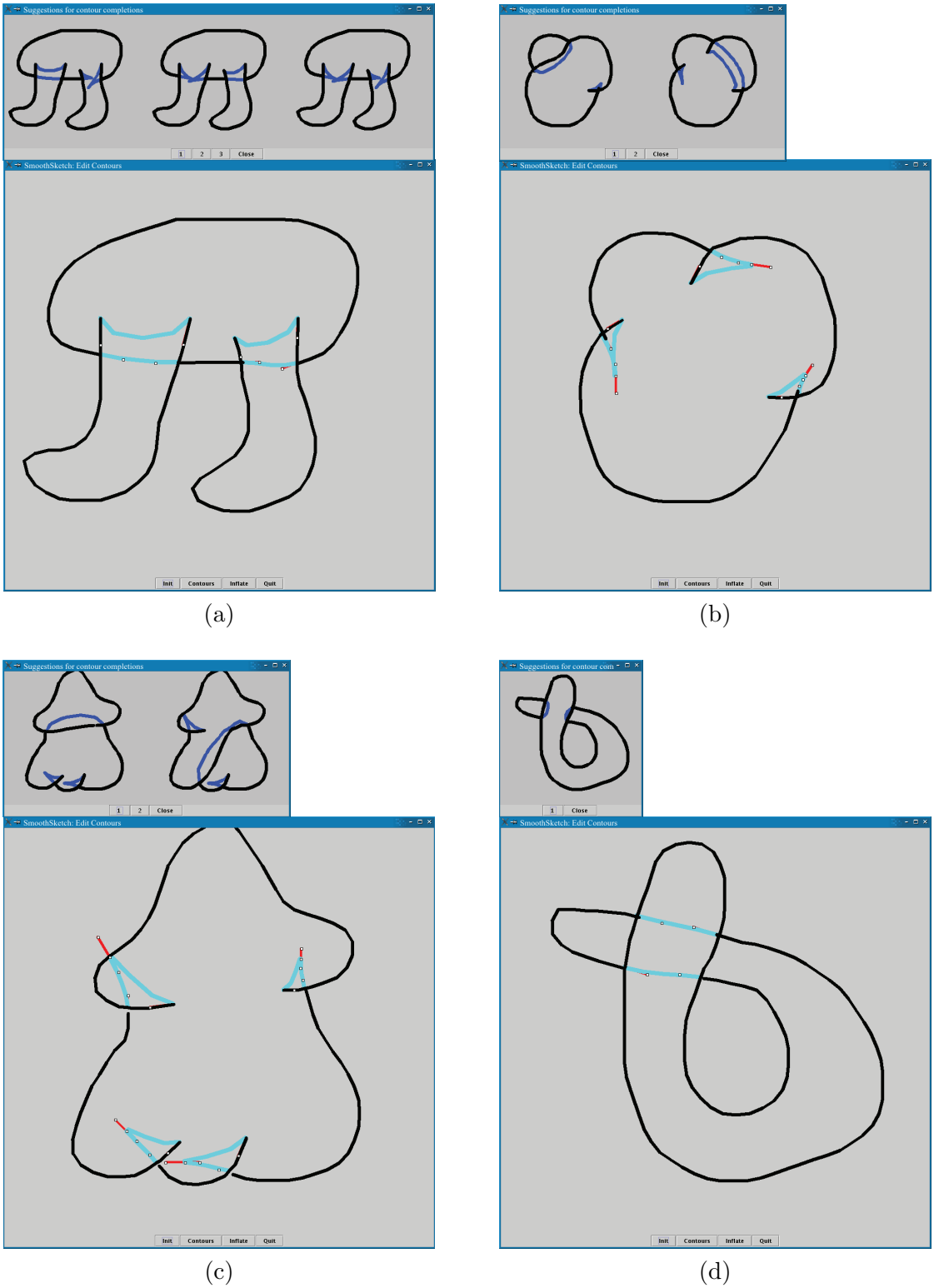


Figure 3.33: For each of the four sketches we show the default hidden contours inferred by the system in the main window and a smaller "suggestions window" with up to three alternative topological interpretations of the sketch.

Chapter 4

Free-form Sketching using Variational Implicit Surfaces

In children's sketching books [6] (see Figure 4.1) one is often taught to first draw the general forms of things starting from simple pieces (cylinders, spheres, cones, blobs, etc.), and then to draw a more careful outline and erase the underlying shapes. The system we describe in this chapter [54] extends this idea to 3D. It provides a user the opportunity to roughly sketch out free-form 3D shapes and then modify them to provide a final form. We call the sketched shapes "blobs" throughout this chapter.

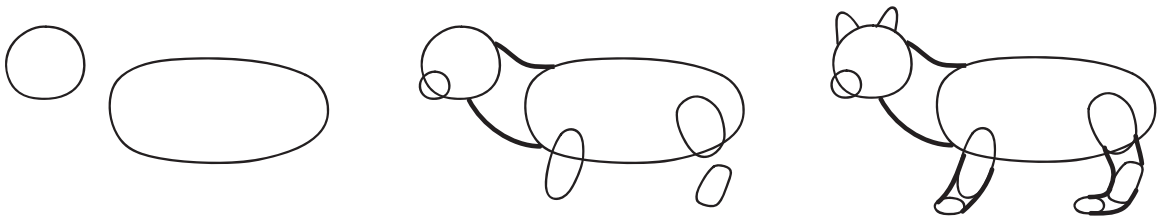


Figure 4.1: A constructive drawing technique is used for teaching children how to draw animals by first sketching ovals and circles, and then connecting them with guidance strokes to refine shapes [6].

In recent years, free-form sketching of 3D shapes has become feasible, as demonstrated by the work of Igarashi *et al.* [47]. But in Igarashi's Teddy system, the underlying surface representation is simply a polygonal mesh, and maintaining smoothness (and performing smooth operations on the surface) is problematic. We present another way of modeling similar free-form shapes - using implicit functions [14]. This allows us to introduce new operations on such models easily. Furthermore, implicit models provide very nice smoothness

characteristics.

In this chapter, we take a particular representation for implicit surfaces — the *variational implicit surfaces* [88] [99] — and show how it can be used to support a collection of free-form modeling operations. The principal contributions of this work are

- operations for easy editing of free-form shapes, and
- a demonstration of how these operations are implemented in the context of variational implicit surfaces.

We describe the user-view of the operations in detail in Section 4.1. The operations are *inflation*, *hierarchy generation*, *merging*, and *local modification*. The first operation is similar to inflation in the Teddy system; the others have some similarities to Teddy and some differences.

The implementation is described in Section 4.3, after an introduction to the surface representation. Except for hierarchy generation, all operations are implemented in a similar way: by the removal of some constraints on an implicit surface and the introduction of others. This uniformity leads to simplicity of structure and coding.

4.1 Overview of Operations

A typical interaction session begins with a view of the world with nothing in it but a horizontal "ground plane" (which corresponds to the x- and z-directions in our coordinates, with the y-direction pointing upwards). The user may draw an outline of a blob depicting the body of an animal, for example, as shown in Figure 4.2(a). This outline is "inflated" into a 3-dimensional shape whose thickness is related to the dimensions of the 2D outline.

The user can then draw further blobs indicating the legs of the animal, as in Figure 4.2(c). These are again drawn at the same depth as the first blob, but the depth may be adjusted as described below. Because these overlap the first blob, the system infers that they are to be attached, and places the new blob in a modeling hierarchy with the first blob as parent; if the parent is later rotated or translated (through a Unicam-like 3rd-button mouse interface [111]) the child is moved as well. As the parent is determined, it briefly flashes pink to tell the user what inference has been made. If a newly-drawn blob overlaps multiple others, one of these is chosen as its parent (based on degree of overlap in the image plane).

If the position of the newly-created blob is not ideal, the user may select the blob (or any other blob) by left-clicking it once; at this point, a transparent bounding sphere appears,

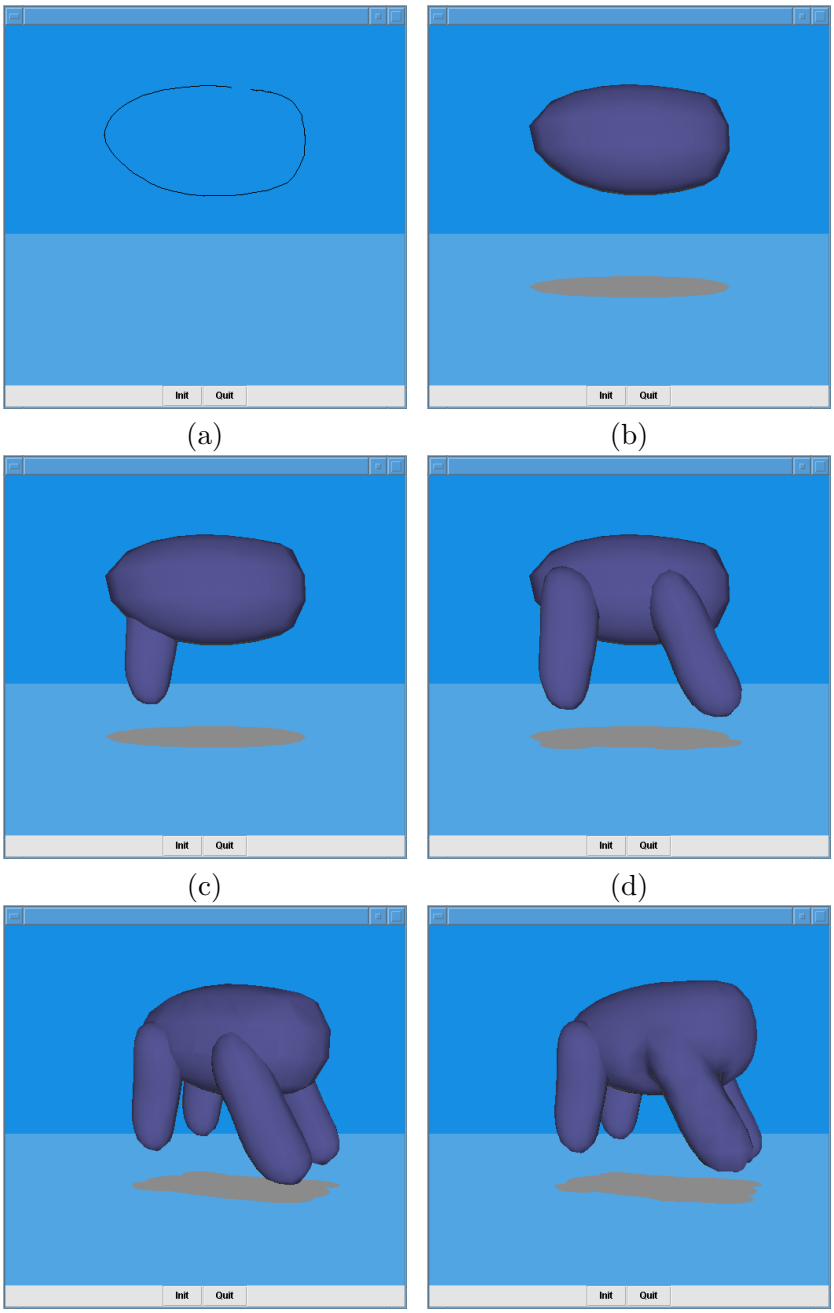


Figure 4.2: (a) A single closed stroke above the ground-plane, which (b) is "inflated" to become the body. (c) Another stroke overlapping the body generates a leg attached to the body. (d) Another leg has been added, and the two near-side legs, their placement having been adjusted slightly, are duplicated with a "symmetrize" operation. (e) A different view. (f) The foreleg is merged with the body.

ready for use as a "virtual sphere rotater," and the color of the shadow of the selected blob is changed to make it easier to select. The user then may

- translate the blob and its children in the xy-plane,
- translate the blob and its children in the xz-plane by dragging the "shadow" of the object, as described by Herndon *et al.* [40], or
- rotate the object around either its center of mass or the center of mass of its intersection with the parent (implemented only for ellipsoids).

The choice of operation is based on where the user clicks when starting to drag the mouse cursor: a click on the bounding sphere indicates trackball rotation; a click on the "shadow" of the blob or near it (which we define as "inside the shadow of the bounding sphere of the blob") moves the object in the xz-plane; otherwise the object moves in the xy-plane. The amount of xy translation is determined by the vector from the first-click to the current mouse-point.

Having drawn the legs, the user may "symmetrize" the model, relative to some parent node, i.e., may create a symmetric copy of all children relative to the plane of symmetry of the parent, by pressing the "S" key, as in Figure 4.2(d, e). The user can add further blobs to the model (a tail, a head, a nose, etc.) and adjust their positions and orientations. When the approximate form is correct, the user may begin to *merge* blobs in one of two ways. In the first, the user selects two blobs and the two are merged together into a single larger blob with a fillet at the junction between them, as in Figure 4.2(f), where foreleg and body have been merged.

In the second form, the user draws a "guidance stroke" starting on the silhouette of one blob and ending on the silhouette of the other, and the fillet is placed so as to match (approximately) this guidance stroke, as shown in Figure 4.3. In this Figure, all blobs are transparent; the original blobs that are merged are gray, the guidance strokes are red, and the resulting surface is pink. Notice that this operation successfully merges even non-intersecting blobs, as long as they are reasonably close to each other. This can be used, for example, to join the head and the body of an animal.

In our current implementation, the two merged blobs are separated from the modeling hierarchy entirely, and the new merged blob is inserted to the hierarchy at the same level as the first blob; all the children of the second blob are also attached to it. If the first blob had no parent, then before a new merged blob is inserted in the hierarchy, the system would try

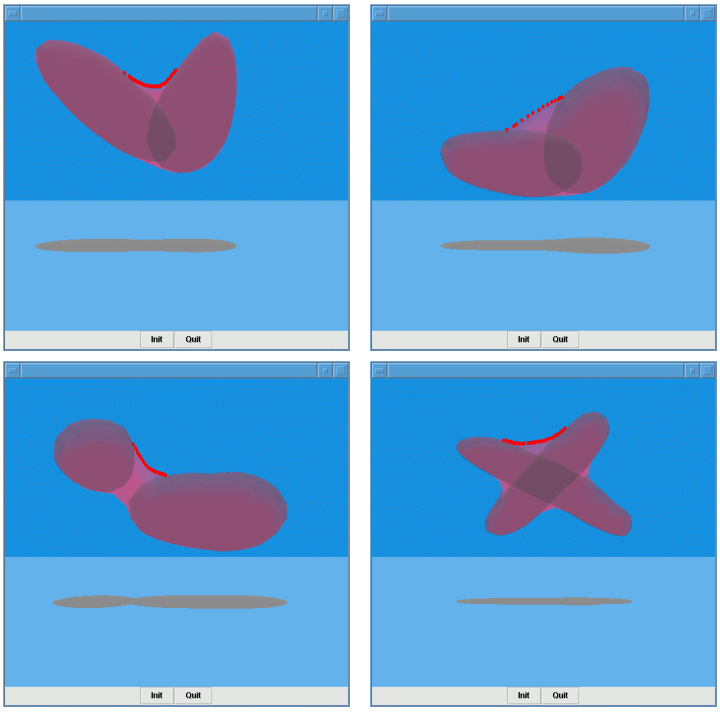


Figure 4.3: Merging with guidance strokes. In each case, the guidance stroke constraint points are shown as red dots, and the new surface is shown in pink.

to find a parent for it. While this is asymmetric, it seems reasonable compared to placing this merged blob into the hierarchy as an independent blob.

Guidance strokes have another use: the user may draw a stroke starting on the silhouette of an object, briefly leaving that silhouette, and then returning to it; points on the surface near the stroke are displaced to lie on the newly-drawn guidance curve. The user can thus make small deformations (giving a camel a hump, for instance, or putting a dent in a cushion) directly by sketching. This operation, called "over-sketching", is demonstrated in Figure 4.4.

Some examples of the animal hierarchies we can build with the program are shown in Figure 4.5.

4.2 Surface Representation

Variational implicit functions are based on thin-plate interpolation [99], which is widely used in solving scattered data interpolation problems where one needs a smooth function, minimizing squared second-derivatives, that passes through a given set of data points. Such

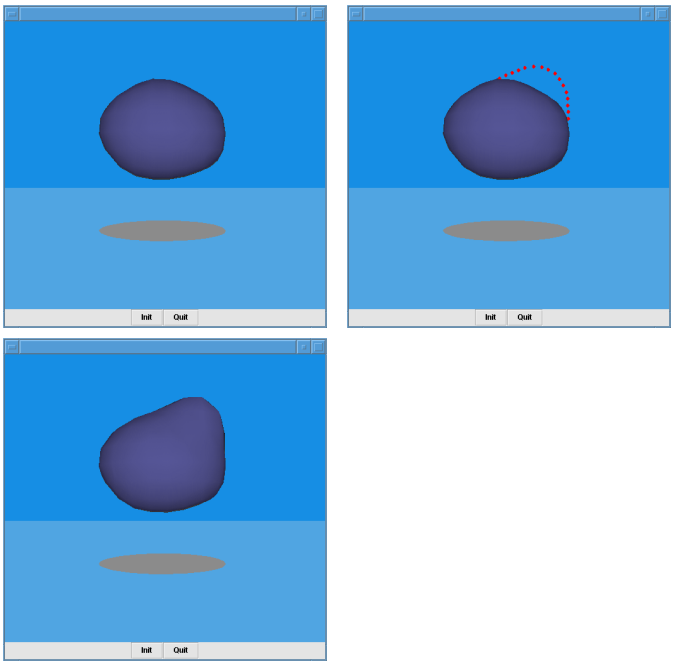


Figure 4.4: Over-sketching: The upper left blob is modified by the user's drawing a stroke near the silhouette (upper right). The surface deforms to match the stroke (lower left).

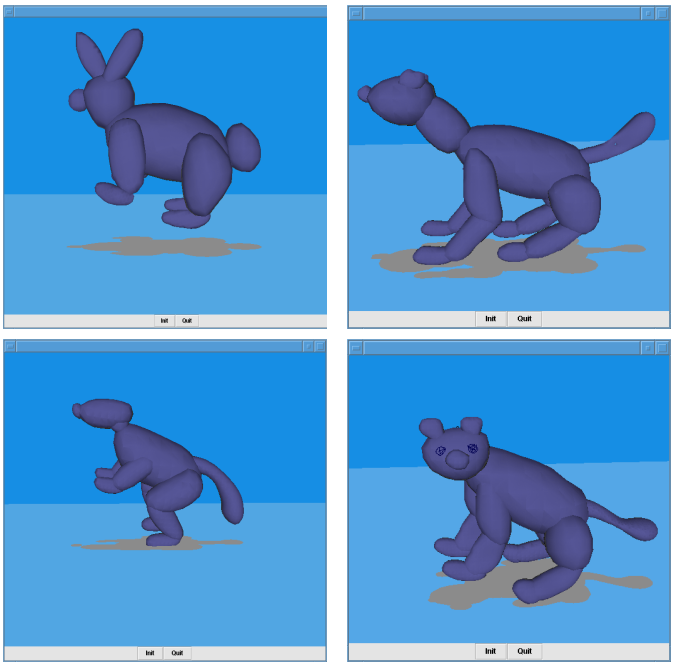


Figure 4.5: Examples of hierarchies created automatically by the system.

a function (a *thin plate* function) is known to be a sum of an affine term and a weighted sum of 'radial basis functions,' i.e., functions of the form

$$f_i(p) = \phi(\|p - q_i\|)$$

which depend solely on the distance from the argument to the ith data point q_i. The exact form of ϕ depends on the dimension; for functions on R^3, the form is $\phi(x) = x^3$.

Thus, any thin plate function can be expressed as

$$f(x) = \sum_{j=1}^{n} d_j \phi(x - c_j) + P(x)$$

where the d_js are real-number coefficients, the c_js are "centers" for the basis functions, and

$$P(x) = a \cdot x + b$$

is an affine function of position x.

Given a collection of n locations at which f is to take on particular values, one can choose the c_js to be the first $n-4$ given locations, and can then solve for the d_js and a and b; this is simply a large linear system.

Thus, the specification of a variational implicit function requires the specification of n values at n points; we refer to such specifications as *constraints*. Turk and O'Brien consider two kinds of constraints: "zero points" and "plus points." A zero point is a point c_j at which $f(c_j) = 0$. This means that the implicit surface $S = \{x | f(x) = 0\}$ passes through such a point.

A "plus point" is one for which $f(x) = 1$ must be satisfied. We will consider points for which $f(x) < 0$ to be "inside," so these "plus points" determine locations which must be outside the implicit surface we are defining.

Turk and O'Brien observed that if one had a set of points through which one wanted a surface in R^3 to pass, one could build a function $f : R^3 \longrightarrow R$ by placing zero-points at the given points, and plus-points at the endpoints of the desired normals. The zero-level set of this function is then a surface interpolating the points and with approximately the given normals.

It's also worth noting that if the function

$$f(x) = \sum_{j=1}^{n} d_j \phi(x - c_j) + a \cdot x + b$$

interpolates the points c_j with values e_j, then the function

$$\begin{aligned} f(x-k) &= \sum_{j=1}^{n} d_j \phi(x-k-c_j) + a \cdot (x-k) + b \\ &= \sum_{j=1}^{n} d_j \phi(x-(c_j+k)) + a \cdot x + (b - a \cdot k) \end{aligned}$$

interpolates the points $c_j + k$ with the same values e_j, i.e., that translating the "interpolation centers" induces no change on the radial-basis-function coefficients, and a simple change in just the constant of the affine term. Similarly, if one multiplies all the control points by some fixed rotation matrix R, i.e., replaces c_j with $R(c_j)$, then again the coefficients d_j remain the same, and only the coefficient a changes, being replaced by $a' = Ra$.

Finally, note that if we have a real-valued function and build a mesh-approximation of its zero-level surface, then we can take the points of that mesh as zero points for a new variational implicit function, and the endpoints of mesh normals as plus-points, and thus create a new real-valued function whose zero-level surface will closely resemble the mesh. This idea, which comes from Turk and O'Brien is critical in our merging and small-modification algorithms.

4.3 Implementation Details

In this section, we will describe in more detail each of the operations previously mentioned, and then how each is implemented in the framework of a variational implicit surface. We begin with some basic facts about our implementation, and then move on to inflation and the other operations.

Our system's coordinates are such that the initially-visible section of the world, at middle-depth, is about 6 units wide; our view of the world is through a 512 pixel wide window. By "middle-depth," we mean "the depth at which strokes are placed into the world," namely, on the $z = 0$ plane. This choice of size, although arbitrary, is necessary to clarify the other dimensions mentioned below.

4.3.1 Inflation

Inflation is the process of converting a user stroke to a 3D blob, represented as a variational implicit surface, whose silhouette matches the given stroke.

We need to go from a 2D visible-contour drawing to a 3D shape. We follow a sequence of operations, namely:

1. Collect the user-input, i.e. a *stroke*
2. Re-sample the stroke;
3. Assign depths (i.e., distances from the eye) to the points of the stroke; we call the resulting path in 3D the *contour*.
4. Create a surface model consistent with the locations, including depth, of the points of the contour, represented as a variational implicit surface.

Preprocessing an input stroke

User input is gathered from the mouse as a collection of screen-space points. The user begins a stroke by left-clicking, and then drags over the path of the desired stroke, and finishes by releasing the mouse button. During the input, the 2D points arrive at a rate that we cannot control. We re-sample these points so that they are not bunched up too closely. We use Igarashi's [47] algorithm: when there are too many points close together, we simply ignore some samples. If the distance between the previous point and the current point is less than 15 pixels, we do not add the current point. This rather crude re-sampling seems to have no real impact on the results.

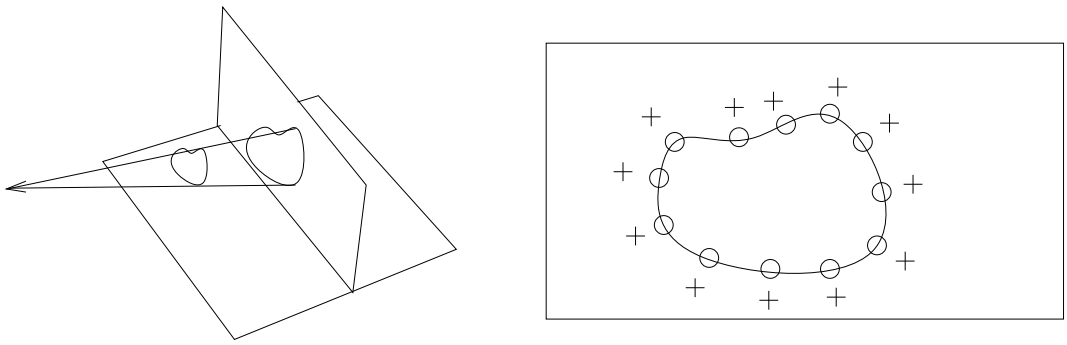


Figure 4.6: Inflating a stroke: (a) the user's stroke is resampled and projected onto the $z = 0$ plane. (b) The points defining the stroke are used as zero-points for inflation (indicated by small circles), and points slightly offset along the normals are used as plus-points (indicated by plus-signs).

To inflate such an input stroke into a blob, we need to explicitly specify the "constraints" ("plus points" and "zero points") that determine a variational implicit function. We first project the 2D points of this stroke onto the $z = 0$ plane; the resulting points are used

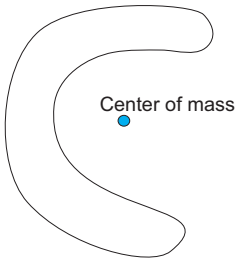


Figure 4.7: Placing extra plus constraints on the line passing through the center of mass of the contour parallel to the z-axis does not work for concave drawings like the one shown here, since the center of mass lies outside the stroke.

as zero-points for the implicit function we want to create; we'll refer to the path defined by these zero-points as the *contour*. (If the view of the world has been rotated, we put the points on a plane through the origin, orthogonal to the current view direction, but it's easier to express the remainder of the construction in an un-rotated frame of reference.) Still within the $z = 0$ plane, we compute points slightly displaced (distance 0.05) from the zero-points along the normals to the contour, and make these all plus-points (see Figure 4.6). To place additional constraints in space (having all constraints in a plane leads to a degenerate situation) we tried three approaches:

Placing the constraints

1. Two plus constraints in 3D are placed at the center of mass of the contour, and then moved off the $z = 0$ plane to depths $z = depthConst$ and $z = -depthConst$, where we used $depthConstant = 1.5$. This makes the "thickness" of the object a constant, which makes a leg look almost as "fat" as a body. This approach does not work well for concave contours whose center of mass lies outside the curve (see Figure 4.7). But since the most common basic shapes used for animal construction are convex blobs, it is a reasonable technique to use for our application.

2. As opposed to having the same thickness for all objects, we make it depend on the shape of the user's stroke. We wanted, for example, that long cigar-shaped contours become blobs with circular cross-section whose radius is the "width" of the cigar, while spherical-looking blobs should be fairly "fat". We therefore use a simple measure for "width" of the stroke shapes:

   Given the 2D shape as a contour consisting of points, we first find the two points that are furthest from each other, and call this the *axis* of the shape. Then we find the

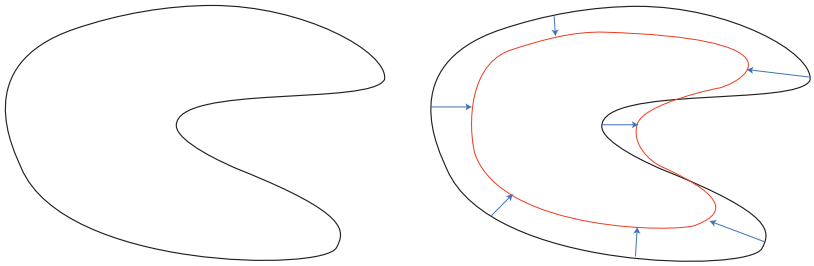


Figure 4.8: First we displace each point of the contour along the inward pointing normal, with the distance proportional to the signed 2D curvature of the contour at this point. Then, two copies of this resulting curve shown here in red could be translated in the positive and negative z directions, and zero constraints placed at these locations.

center of this axis, draw a perpendicular through this point and find the point closest to the center along this perpendicular. This closest distance is our measure for the "width" of the 2D shape drawn by a user.

We now place the two additional "plus points" as before, but instead displace them not by $z = \pm 1.5$, but rather by $z = \pm 1.5 \cdot width$.

3. To ensure that the surface curves "in" (i.e., away from the outward normal at the $z = 0$ slice) as one moves away from the $z = 0$ plane, we take two copies of the stroke points and translate one of them in the positive z direction, the other in the negative z direction, and put plus-points at the resulting locations. The distance we move each of these copies is calculated as in our second method above – but this time using $0.8 \cdot width$.

The third algorithm could be improved in the following way: we could first displace each point of the contour along the inward pointing normal, with the distance proportional to the signed 2D curvature of the contour at this point (see Figure 4.8). Then, one copy of this resulting curve would be translated in the positive z direction, and one copy — in the negative z direction. We could then place extra zero constraints at these locations. We believe this algorithm should produce better results than our third technique and plan to implement it in the future.

In all the examples in this chapter we have used the second inflation technique, although there are some cases where it is not ideal, such as highly convex curves. Finding a more principled approach to inflation is critical for future development.

Having determined the set of all zero-points and plus-points, we use the method described in Section 4.2 to compute a variational implicit surface. We then use Bloomenthal's [13] polygonizer to create a 3D mesh corresponding to this surface, which is what we actually display. We set the size of the grid-cells in the polygonizer to be $0.5 \cdot width$, so that even thin objects get polygonized reasonably.

4.3.2 Hierarchy

Our current implementation is in Java3D, so we organize all the blobs in a hierarchy that's stored in the Java3D scenegraph. Our scenegraph is a tree consisting of branch nodes, each of which contains a blob, with a modeling hierarchy above them. Each branch node has an associated transformation and a shape to render. In our case, the shape for each blob node is the mesh built by polygonizing the implicit surface of the blob. When rendering each blob, Java3D automatically parses the scenegraph transforming each leaf node with the transformation accumulated by multiplying all the transformations from the nodes that are in the path to this shape. All blob nodes have the structure shown in Figure 4.9. Initially, each blob is placed according to the position of the 2D input stroke and at some fixed depth. Next, the system attempts to find a parent of this blob based on the intersection with the other blobs. The algorithm for finding a parent for a blob is given below. If a blob does not have "big enough" overlap with any of the previous blobs, it has no parent and is placed on the high level of the hierarchy as an independent blob. If the system finds the parent, then the transformation of the child is updated as follows:

- Find the "accumulated transformation" of the parent - a multiplication of all the transformations in the nodes above the parent and the transformation of the parent node itself.
- Multiply the blob's transformation from left with the inverse of the parent's "accumulated transformation". Place the result into the transform node of this blob.

Operations such as rotation around the center of mass of the blob and translation either in xy-plane or xz-plane update the transformations of the corresponding blob in the scenegraph. Children are updated automatically because of the way the blobs are placed in the hierarchy.

The algorithm for finding the parent of the blob is the following:

1. Render this blob to an off-screen buffer and save it in an image; find the 2D bounding box of the blob's projection.

2. For each blob previously placed in the hierarchy, render it to the off-screen buffer and find the overlap with the new blob by comparing pixels of the two images. Note that we only need to compare pixels in the sub-area of the image corresponding to the 2D bounding box found in step 1.

3. Calculate the ratio of overlap with this blob by dividing the number of points found in step 2 by the area of the current blob.

4. Among all blobs for which the ratio is more than a certain threshold, choose the one with maximum overlap. Choose this as the parent.

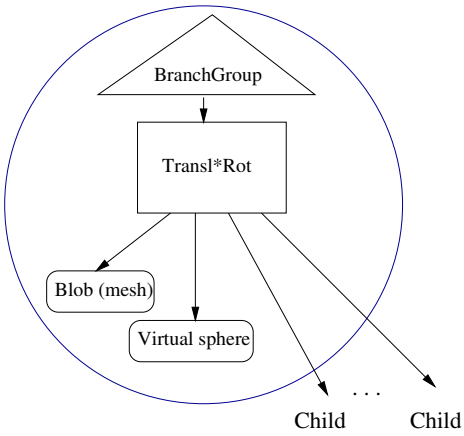


Figure 4.9: The underlying branch graph for a blob

For the "symmetrize" operation, for each child of the parent, we find its "symmetrical" copy as follows:

- First we reflect/flip the vertices and normals of every mesh stored in the branch subgraph of a child around the center of the parent and with respect to the normal of the parent; Then we change the transformations in each transformation node in this subgraph as discussed in the second step.

- Let's say the child has the transformation matrix T relative to the parent, we extract the translational Tr and rotational Rot components from this matrix. We define the operation "flip" applied to the vector v as follows:

$$Flip(v) = v - 2 * (v \cdot n)n$$

where the n is the normal to the plane of symmetry of the parent. Then we create a new transformation whose translation component is $Flip(v)$ and whose rotational

component is determined as follows: if the rotation matrix is the rotation around the axis a with angle ϕ, then the new rotation matrix is the rotation around the axis $Flip(a)$ with the angle $-\phi$.

This set of operations on the hierarchy, although simple, allows us to create a variety of interesting shapes as seen previously.

4.3.3 Merging

Blending operations to allow smooth transition between two implicit surfaces defined by $f : R^3 \to R$ and $g : R^3 \to R$ have been proposed by many researchers; we follow the general idea laid out by Barthe et al. [8]. Here is the central idea:

Define a function H, such that

$$H : R^3 \to R^2 : P \mapsto (f(P), g(P))$$

Clearly H sends all points on the level-zero isosurface of f (which we'll call the "isosurface" from now on, the "level zero" being understood) to points on the y-axis of R^2; similarly it sends those on the isosurface of g to points on the x-axis. Consider a point Q on the positive x-axis. A point in 3-space that maps to Q is evidently in the isosurface for g, but *outside* the isosurface for f. Similarly, anything that maps to a point on the positive y-axis is on the isosurface for f, but outside the isosurface for g. Letting L denote the union of the positive x- and y-axes, we see that the pre-image of L, i.e., $H^{-1}(L)$, is simply the union of parts of the two isosurfaces that are outside each other (see Figure 4.10).

If we have a function, $G : R^2 \to R$ whose level-zero level set is exactly L, the union of the two positive axes, then $G \circ H : R^3 \to R$ is a function on R^3 whose level set is the union of the outside parts of the two level surfaces. If, on the other hand, we have a function G whose level-zero level set is an approximation of L, then the level-surface of $G \circ H$ will be very similar to the union of the two outside parts; in particular, if the level set for G deviates from the axes only near the origin, then the level set of $G \circ H$ will deviate only near the intersection of the level sets of f and g.

Automatic Merging

Let's say we want to merge two blobs with implicit functions f and g. We implement the idea just described in a particularly simple way. First, each blob is sampled at a fixed resolution and polygonized to create vertices and corresponding normals, which are used as zero-points and plus-points for a new implicit representation of the blob.

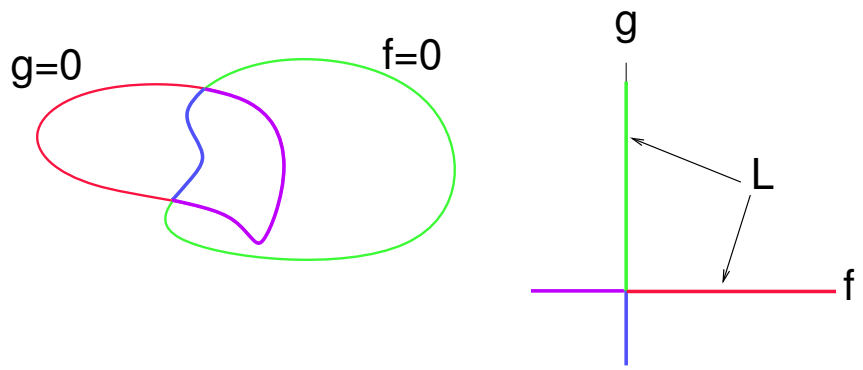


Figure 4.10: Isosurface blending in 2D. The level set for f and the level set for g intersect. When points of these level sets are mapped to R^2 by H, they land at corresponding points in the right-hand figure. One can see that the pre-image of the positive axes consists of a merge of the "outside" parts of the two level sets.

We now create a new implicit function from a subset of these vertices and normals. The vertices (and corresponding normals) to be eliminated are determined by the restriction $(G \circ H)(P) < 0$; in other words, we eliminate all vertices v (and their corresponding normal points) that lie inside the intersection of the two blobs, i.e.,

$\forall v \in$ first blob, eliminate v if $g(v) < 0$.

$\forall v \in$ second blob, we eliminate v if $f(v) < 0$.

The implicit surface reconstructed from the remaining vertices (and corresponding normals) is a smooth merge between the two surfaces (see Figure 4.11). The same idea can be used for merging three or more blobs.

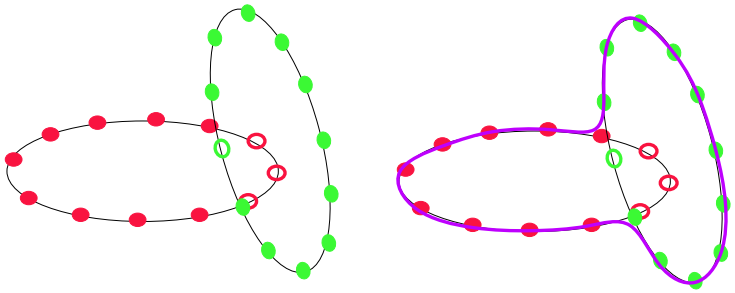


Figure 4.11: Automatic merging: the hollow constraint points on the left are eliminated, leading to the new iso-set in purple on the right.

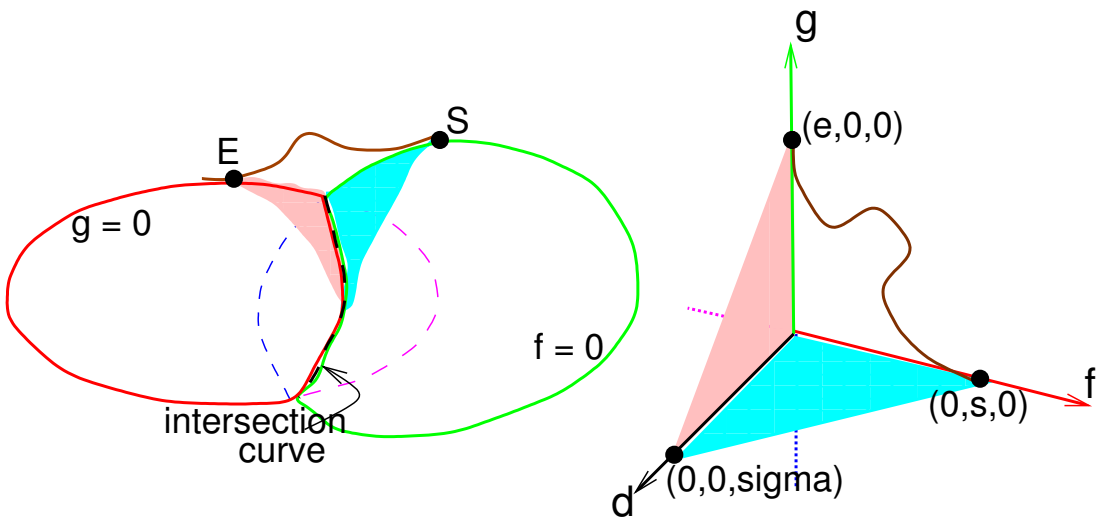


Figure 4.12: Merging with a guidance stroke: The region of influence is shown in pink and blue colors.

Merging with a guidance stroke (with local influence)

The user can specify the shape of the merged blob in one of the cross-sectional planes along the intersection. In this case, the input consists of two blobs and a 2D guidance stroke (Figure 4.12). In addition, the user may specify a limit on the region of influence of the guidance stroke specified by the 3D distance σ (set to the constant value 0.8 in our implementation). The influence of the guidance stroke decreases with distance from the stroke, until at distance σ there is no influence at all. Beyond the region of influence on the two implicit surfaces, the resultant merged surface is same as the one produced by automatic merging. Below we describe the process of computing the blending function G for this guided version of merging.

We first find the 3D location of the start and end point of the guidance stroke, S and E, by finding the nearest silhouette point on the appropriate blob. We then project the 2D guidance stroke onto the plane passing through S and E and parallel to the average normal at those two points. Without loss of generality, let's say S lies on the first blob, defined by the function f and E lies on the second, defined by the function g. The "extrusion space coordinates" for the point S are then $(f(S), g(S)) = (0, s)$; for E they are $(f(E), g(E)) = (e, 0)$. We sample the guidance stroke and plot the corresponding points in the extrusion space by computing implicit function values (f, g). To achieve a smooth transition with limited region of influence, we add a third dimension to the extrusion space. The third

coordinate $d(x)$ for a 3D point x is the minimum of the Euclidean distance to the points S or E. In the extrusion space, the surface generated by our guidance stroke in the f-g plane and the point $(0, 0, \sigma)$ defines the blending surface on which $G(f, g, d) = 0$. The shape of this surface naturally determines the shape of the new surface in the blended region. (Note, however, we do not explicitly construct this surface in the extrusion space.)

For a practical implementation, we eliminate all vertices in the region of influence (and the plus-points corresponding to their normals) and add new zero points corresponding to the sampled locations on the 3D guidance stroke. For example, for vertices on the first blob, we eliminate all v for which $(g(v) < \delta_2)$ or $(f(v), g(v), d(v))$ is inside the triangle with vertices $(0, s, 0), (0, 0, \sigma)$ and $(0, 0, 0)$. Similarly for the second blob: we eliminate all vertices v for which $(f(v) < \delta_1)$ or $(f(v), g(v), d(v))$ is inside the triangle with vertices $(e, 0, 0), (0, 0, \sigma)$ and $(0, 0, 0)$.

For each new zero-point added from the projected and sampled guidance stroke, we add a corresponding plus-point by using the normal to the guidance curve in the plane of projection as the surface normal, and putting a plus-point a small distance (0.05) out along this normal. Once the appropriate constraints on the original blobs have been eliminated, and the new ones corresponding to the guidance stroke have been added, a new merged surface is generated.

4.3.4 Small modifications on the blobs

The user can make local modification of the profile of a blob simply by drawing a new target profile. The shape near to the target profile stroke is modified with the influence limited by 3D distance from the stroke. There are many 3D techniques to make local distortions, but most of them work on polygonal meshes. They involve cumbersome process of finding displacement vectors in the region of influence that vary smoothly across the region. Some 2.5D techniques deform the polygonal meshes so that they conform to the target profile stroke from the given viewpoint, but may have artifacts when seen from a different viewpoint [24].

We instead use an idea similar to the one used for merging with guidance stroke above. The input is a blob and a target profile stroke is drawn near its silhouette. In addition, there is a limit on the region of influence specified by 3D distance τ (again, set to 0.8 in our implementation). The main idea is to eliminate zero points (and corresponding plus points) in the region of influence and add new zero (and plus) points from sampled and projected 3D target profile stroke (see Figure 4.13). Note, that the user draws only a target stroke

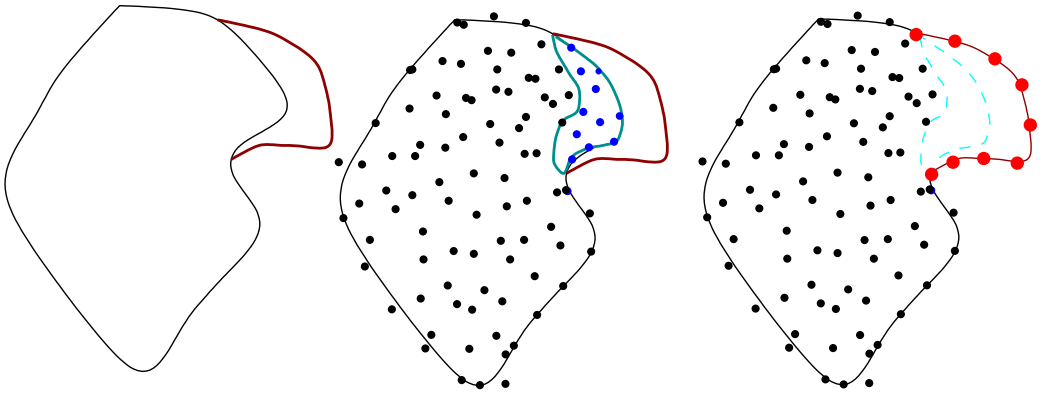


Figure 4.13: (Left) Blob and modification stroke. (Middle) Blue dots indicate zero points in the green region of influence. (Right) The blue points are eliminated and the new red zero points from the target stroke are added.

and does not draw a source stroke, as required earlier work [47], [24]. The source silhouette is automatically computed for the viewpoint in which the target profile was specified. This is an important improvement. It provides simplicity and eliminates the often seen error in clicking on or specifying the source curve.

By providing only new constraints (and relying on the mechanism of variational implicit surfaces), we also avoid the problem of enforcing smooth variation in the displacement vector along the modified surface.

To create the new implicit surface constraints, we first find the nearest silhouette point for start and end of the target (profile) stroke, and assign 3D coordinates, S and E as before. The target stroke is assumed to be snapped to these points if the stroke is within a few pixels. Next, we project the target stroke on the plane passing through S and E and parallel to the average normal at those two points, creating a planar 3D curve. For each point on the target curve, we find the nearest silhouette point on the blob. This creates the corresponding source stroke.

We find the region of influence, by comparing 3D Euclidean distance of each vertex on the blob with respect to the source stroke.

We assign a normal to each point on the sampled target stroke in the direction normal to the stroke curve and parallel to image plane. This provides new zero points and corresponding plus point constraints. The vertices in the region of influence are eliminated. The reconstructed implicit surface has the matching target profile because the surface interpolates the zero points on the target stroke and respects the specified normals parallel to the image plane.

4.4 Related Approaches

There is a long history of implicit-surface modeling, nicely summarized in a book by Bloomenthal [14]. Here we describe a few areas of research that are particularly relevant to this work.

The "Skin" system [69], developed by Markosian *et al.*, supports a form of constructive drawing, in which the user creates a set of basic forms over which he places a skin, whose characteristics are then modified by small adjustments to offset distances at various scales of subdivision of the skin mesh. This skin is basically a polygonization of an implicit surface, but one that's defined by a combination of signed-distance representations of the underlying forms. No provision is made for directly creating the underlying blobs, aside from the operations described in Zeleznik's "Sketch" work [112], or for any blob hierarchy.

By contrast, Wyvill *et al.* [110] describe the blob-tree, a CSG-like hierarchy of implicit models in which shapes are combined by a rich collection of operators, including various deformation operators. Their emphasis is on the representation of complex implicit surfaces through a tree-like structure. We, too, build a tree-like structure with implicit surfaces at the leaves, but ours is a more conventional modeling-transformation hierarchy, in which the internal nodes represent grouping or linear transformations; when we merge surfaces or make small modifications to them, these edits are applied directly to the underlying implicit representation.

Our editing operations use a key idea described by Barthe *et al.* [7], a method for blending a pair of implicit surfaces defined as zero-sets of functions f and g by considering each point of R^3 as having f and g "coordinates," i.e., by treating the map

$$(f, g) : R^3 \to R^2 : p \mapsto (f(p), g(p))$$

as a kind of "coordinatization" of 3-space. The surfaces are then the pre-images (i.e., points that map to) of the $f = 0$ axis and $g = 0$ axis respectively: by looking at the pre-images of other curves in (f, g)-space, especially ones containing large portions of these two axes, they create blends between the two surfaces.

Finally, our surface representation is based on *variational implicit surfaces*, as described by Turk and O'Brien [99] (see section 4.2). Such variational implicit surfaces can be used to represent some very complex objects, as shown by Carr *et al.* [19]; their methods show how one can simplify the representation somewhat by judicious deletion of constraint points.

Another approach is to use the polygonal representation and direct mesh algorithms, but improve the smoothness of the meshes by using techniques like mesh subdivision and

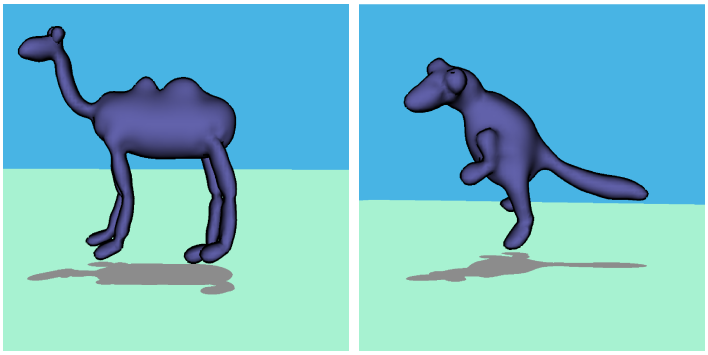


Figure 4.14: Two animal models created using our system. Each took about 5 minutes to create by the user familiar with the system.

mesh fairing [67] [90].

Our interaction techniques are derived from those in Sketch, Teddy, and the curve-over-sketching idea presented by Baudel [9] for the case of 2D piecewise parametric curves.

4.5 Results, Limitations, and Future Work

We have described our system for creating free-form models from gestural input, using variational implicit surfaces in a modeling hierarchy as a surface representation. A couple of models created using the system are shown in Figure 4.14. Our choice of modeling operations allows a user to construct shapes in a way that closely parallels the way that they draw, starting from basic forms and then refining.

The advantage of the implicit surface representation is that the natural modeling operations — inflation, merging, stroke-based merging — are all easy to implement in this context. Nonetheless, there are some limitations in our approach. As the number of constraints increases, the time it takes to compute the coefficients for the variational implicit surfaces grows as well. Presumably it's possible to reduce the number of constraints substantially, using methods like that of Carr *et al.* [19], and we hope to do so in the future.

Because the ratio of an overall size of an object to the cube size in the polygonizer is fixed, it is impossible to represent objects smaller than a certain size. In particular, when a small object is merged with a large one the details of the small one may disappear.

Our representation cannot support sharp edges in a surface; doing so would require many more constraints. Presumably this can be remedied by some sort of hybrid polygonal/implicit representation or by using anisotropic basis functions [27].

Because our operations depend on implicit function values, and because these values, far from our surfaces, may have no intuitive meaning, our operations may produce unexpected results unless the guidance strokes (for example) stay close to the underlying surface. Furthermore, the region of influence of an operation depends on the gradient(s) of the implicit function(s) involved, and hence may be unpredictable.

There are also some non-critical limitations in our current implementation: our shadow-selection algorithm is based on the shadow of the bounding sphere of the blob rather than the blob itself. We would also like to improve our overlap-detection algorithm to make it faster.

4.5.1 Future work

We would like to extend our merging algorithm to take into account the intersection curve between the two blobs: it would be nice to use a single guidance stroke to make a "reasonable" blend or fillet at all points of this curve.

We would also like to extend the hierarchy-detection algorithm to include better placement of child blobs (sticking legs to the sides of bodies, for instance, rather than placing them along the body's medial plane and requiring that the user translate them), and a better use of the inferred hierarchy in determining inflation-depths for child blobs.

Finally, we want to add a "painting" component to allow the user to decorate the resulting shapes, drawing feathers on birds' wings, for example, or spots on a leopard.

Chapter 5

Multi-View Sketching

Since multiple shapes may have identical projections, no single solution that a sketching interface would suggest will be right every time. Because a second/third/etc. views of an object very often resolve the ambiguities introduced by projection, it seems as though the obvious extension to the one-view sketching would be to consider an interface where a user could draw contours of the same object from different points of view and the system would infer a shape that matches these contours as well as possible. Engineers have been using 3-view (top/front/side) drawings of CAD-objects for a long time, so it is a natural question to ask: would similar multi-view approach work for free-form shapes?

Despite the obviousness of the idea, the actual preliminary results (that we describe for the case of 3D curve sketching from multiple views) are somewhat disappointing. The main reason for this is the fact that people are not good at drawing projections of shapes from different points of view. The system that we are interested in, is intended for a general user, not for an artist, and as a result, we can not expect the sketches of the projections to be either correct, or consistent with one other.

5.1 Epipolar Methods for Multi-view Sketching of Generalized Cylinders

A simple proof-of-concept system that we built is designed to support the construction of 3D curves ("backbones") along which generalized cylinders are extruded. The user draws the backbone from one point of view, redraws it from another, and the constraints of epipolar geometry are used to find the curve matching both sketches as closely as possible.

Describing generalized cylinders by a shape that varies along a backbone curve is by now

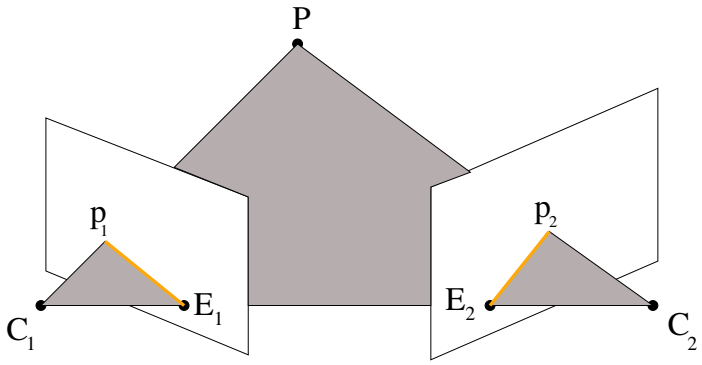


Figure 5.1: Given an object point P, two pinhole cameras with the optical centers C_1 and C_2, and two image planes, we can define an epipolar plane (C_1, C_2, P), two epipoles E_1 and E_2 (the points of intersection of the line (C_1, C_2) with the image planes) and two epipolar lines (shown in orange).

a standard topic in computer-aided design, well-covered in any textbook; the only (slight) novelty in our formulation is the use of Bishop's framing of a curve [12].

5.1.1 Epipolar geometry

Consider two images of the same object taken by two different pinhole cameras with centers at C_1 and C_2 . For each point P on the object, there is an *epipolar plane* passing through the centers C_1, C_2 and P (see Figure 5.1).

The lines formed by the intersection of this plane with two image planes are called *epipolar lines*. Each epipolar line is a projection of the ray connecting the object point with the other camera position onto the current image plane. Intuitively, one can think of an epipolar line as a line along which we can move the projection point in the second image while preserving its projection on the first image. We can determine the $3D$ position of a point P by sketching it from two points of view. Only by making the restriction that the second "sketch" of the point P lies on the epipolar line defined by the first camera position can we guarantee that the resulting $3D$ position of P will be consistent with both views. This idea forms the basis for our multi-view sketching system.

5.1.2 Multi-view sketching: the user perspective

These simple observations about epipolar geometry above lead us to an interface for $3D$ curve sketching, where a user is given epipolar lines as a guide for modifying the curve from a different point of view (see Figure 5.2). We added this interface to our previous system

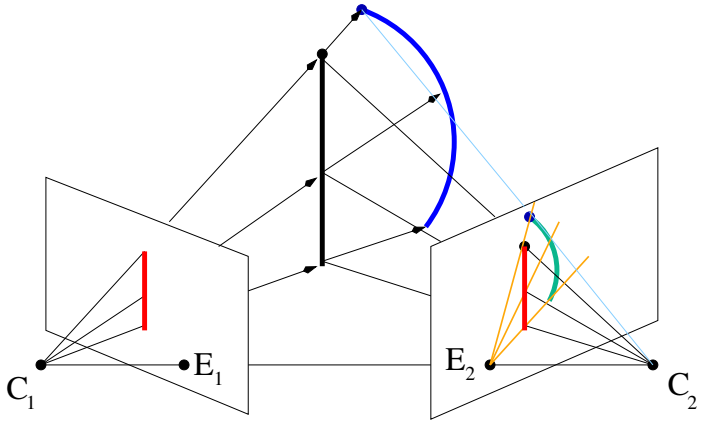


Figure 5.2: Sketching a 3D curve using epipolar constraints: a user draws a red line on the left image plane to create a black 3D curve; then she can modify the shape of the black curve by drawing its green projection from a different point of view (on the right image plane). The green curve is projected onto epipolar lines (that are shown in orange and intersect at the epipole E_2). As a result, a blue 3D curve is created. The projection of this final blue 3D curve onto the left image plane remained unchanged because of the epipolar constraints.

[54] for drawing free-form animals, and found it to be particularly useful for creating a variety of animal tails and other objects whose shape can be represented by generalized cylinders.

A user typically carries out the following steps (see Figure 5.3):

1. The user rotates the virtual camera with the right mouse button and draws a curve (by dragging the mouse) from any point of view. The user stroke is projected on the plane passing through the world origin and perpendicular to the look vector of the virtual camera.

2. When the user starts rotating the virtual camera to find another view from which to redraw the curve, the system displays the segments of epipolar lines — yellow segments passing through each point of the curve — that serve as a sketching guide.

3. As the user redraws the curve from a new point of view, the new stroke gets projected on the epipolar lines.

4. Once the redraw operation is complete and the user rotates the virtual camera again, the epipolar lines are updated — they become projections of the rays connecting the previous eye point with the stroke points onto the current image plane. We only keep

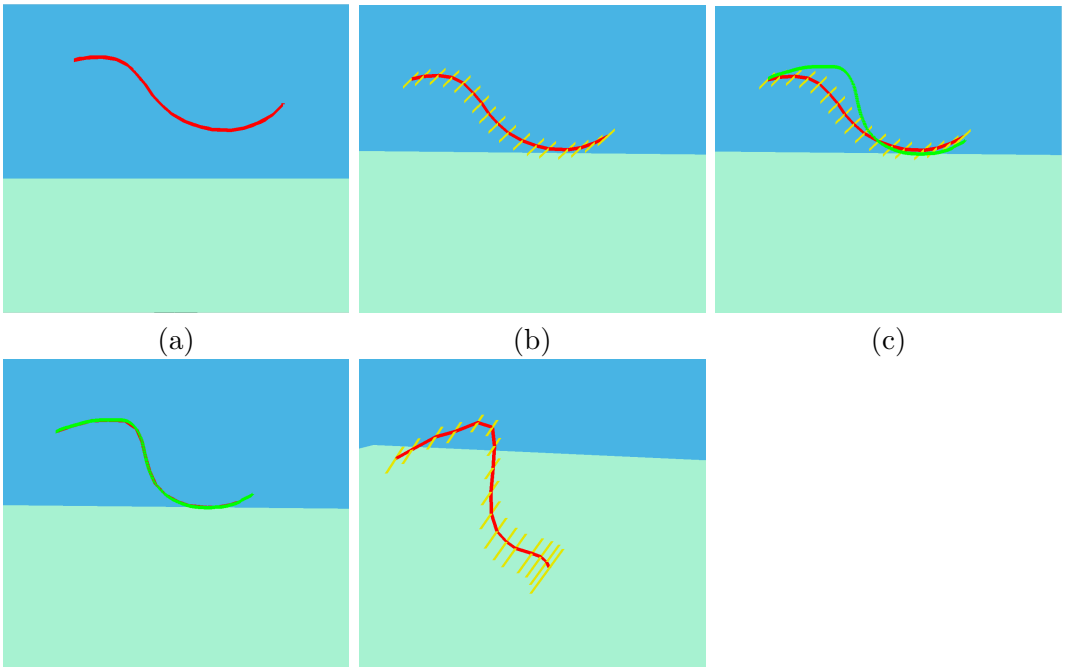


Figure 5.3: (a) A user draws a curve on the image plane. (b) When the virtual camera is rotated, epipolar lines appear as yellow segments at each point of the stroke. (c) The user redraws the selected curve from a new point of view by inputing a new green stroke. (d) A new stroke is projected on the epipolar lines and the coordinates of the points of the original curve are updated to satisfy new constraints. (e) The resulting curve from a different angle - notice that new epipolar lines are calculated for each point of the new curve.

track of the most recent previous camera position, so the modification of the curve in the current view will only be consistent with the curve projection in the view last used for sketching. The user repeats steps 3 and 4 till a satisfactory result is achieved.

In addition to redrawing, the user can perform the following operations:

- Select and move individual points or segments of the curve along the epipolar lines.
- Select a stroke and translate it along the epipolar lines.
- Place the curves in a hierarchy by attaching them to one another (useful if the user wants to create a wireframe chair, for example).
- "Shear" the stroke by dragging just one of the points. A stroke with the end points A and B can be modified by dragging any intermediate point C along the corresponding epipolar line. The end points A and B will remain fixed and positions of all the points in between will be calculated as the intersections of the plane defined by (A, B, C) and the corresponding epipolar lines.

In practice, we found the "redraw" operation to be the most useful and intuitive.

For each segment of the newly redrawn stroke we need to decide which epipolar line to project it to, because normally it would intersect more than one epipolar line. The algorithm we are currently using is the following:

1. Consider the first segment of the stroke and check whether it intersects the epipolar line corresponding to the first point of the old stroke. If it does not, check intersection with the epipolar line corresponding to the second point of the old stroke, then third, etc.
2. Assume that the segment with the index $i-1$ was projected onto epipolar line with the index j. Then, for a segment with the index i, search for the first epipolar line it intersects starting with epipolar line indexed $j+1$.
3. Skip the point if the corresponding segment does not intersect any epipolar lines satisfying the criteria above.

This simple algorithm seems to work moderately well for curves with relatively low curvature; its drawbacks are discussed below.

After the desired backbone curve is obtained, a user can select two planes perpendicular to the curve and draw cross-section strokes on them; these are then interpolated along the

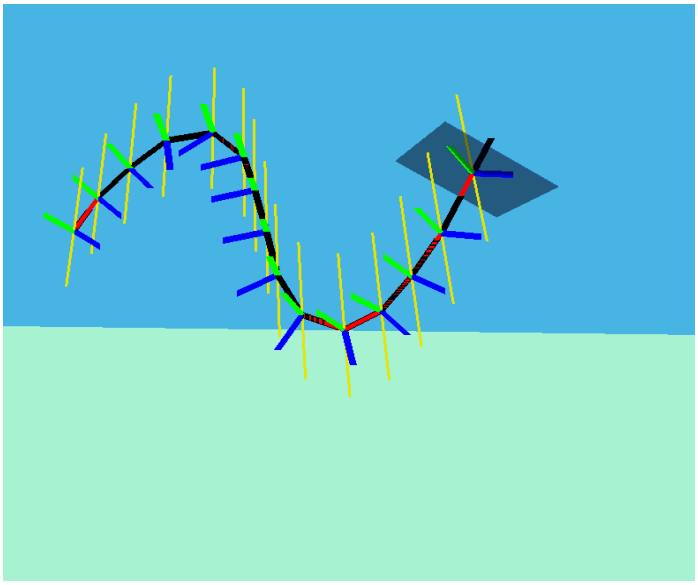


Figure 5.4: The Bishop framing along the curve. The blue and green lines are the two vectors of the Bishop frame; the red lines are in the direction of the tangent vectors. Notice that the Bishop frame is continuous even at the inflection between the two bends.

axis to construct the generalized cylinder. To do that, the user clicks near some point on the backbone curve, and a small gray semi-transparent square shows up at this point perpendicular to the curve segment (Figure 5.4). This is the "plane" on which the user can draw a cross-section stroke. The virtual camera (or the curve) can be rotated to adjust the view in such a way that the "plane" is approximately parallel to the view plane. It is the most convenient view to draw a cross-section stroke as there are no distortions as the stroke gets projected onto the "plane" (as the user draws it).

The second cross-section stroke is drawn in the same manner. The system also allows the user to draw just one cross-section; in this case its scaled or non-scaled copy is propagated along the curve. In any case, once two cross-sections are specified, the system needs to interpolate the intermediate cross-section strokes between them.

5.1.3 Interpolation between the Cross-sections

To interpolate between cross-sections, we need a continuous framing of the curve, i.e., a basis for the plane normal to the curve at each point, but one which changes continuously. The normal and binormal vectors of the Frenet frame [72] [28] almost work, but at points where the curvature is zero, the normal is undefined, and in practice often "flips" to its

negative. The Bishop framing of a curve [12], however, works fine (see Figure 5.4). It is defined by first picking a single unit vector $B_1(0)$ which is orthogonal to the unit tangent vector $T(0)$ there, and then extending B_1 along the curve by the rule that $B_1'(t)$ is a linear combination of $B_1(t)$ and $T(t)$. The second vector in the frame of the tangent plane is then simply defined by $B_2(t) = T(t) \times B_1(t)$. For a polyhedral curve, the vector $B_1(t)$ changes only at vertices; at vertices it is changed by applying the rotation M that rotates the previous edge direction into the next edge direction by a rotation about an axis determined by their cross-product. When the angle between the two direction vectors is zero, the axis is undefined, but the rotation amount is zero, so one uses the identity matrix for M. The sole constraint on this process is that no two sequential edge-vectors of the polyhedral curve be in exactly opposite directions.

With this Bishop framing of the curve, we can interpolate cross-sections easily: we take the starting and ending cross-sections and represent each in the coordinate system of the Bishop frame at the corresponding point; we then interpolate between the cross-section drawings, and place the interpolated cross-sections at the corresponding points of the curve, using the Bishop frame as a coordinate system. In our testbed system, the interiors of cross-section curves are restricted to be star-shaped with respect to the origin: a ray from the origin in direction θ must intersect the curve at a single point whose distance from the origin is called $r(\theta)$. We then interpolate the curves by interpolating between the associated r-functions. More sophisticated approaches, based on 2D curve morphing ([4], [91]), are possible, but were not germane to the testing of the epipolar-manipulation method, so we did not implement them.

5.1.4 Typical Results

Figure 5.5 shows how our system can be used to model a shape like that of the wire that trails behind the Luxo lamp in "Luxo, Jr.". The curve has a wavy shape in the xy-plane and a bump in the xz-plane; constructing such a curve in Teddy or similar systems would be cumbersome. Figure 5.6 demonstrates the various tails (with different backbones and cross-sections) that can be created.

The mouse and the monkey were generated using our previously presented free-form sketching system [54]. The new interface for sketching with epipolar constraints fits well in the existing framework. By default, a "blob" (a node in the scenegraph containing a blob-shaped object) is created whenever a user draws a stroke, but the user can press a key to activate "curve sketching with epipolar constraints" mode. After the editing of the

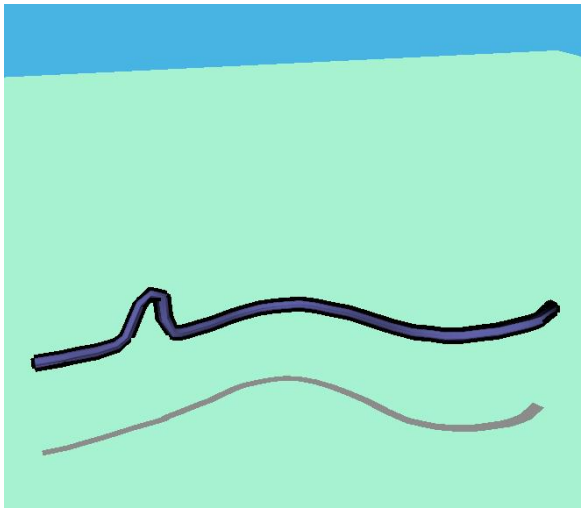


Figure 5.5: A generalized cylinder in a shape like that of Luxo, Jr.'s power cord; the small vertical bump is in a different plane from the main curvature of the cord.

curve is complete and cross-sections are inputted, the resulting generalized cylinder becomes a "blob" and can be selected, transformed, inserted into a hierarchy and merged with the other blobs as described in the paper [54]. The system is interactive and the interface allows simple shapes to be created in a couple of minutes.

5.1.5 Problems and Limitations

Figure 5.7 demonstrates the case on which "redraw" operation fails. Assume that a flat spiral curve was drawn in the xz-plane and a user wants to add torsion to this flat curve in y, to create a three-dimensional spiral curve. When a virtual camera is rotated to any intuitive view for the "redraw" operation, the epipolar lines obstruct the view of each other and confuse the user rather than help. When the user attempts the "redraw" operation, the system does not project a newly drawn stroke on epipolar lines correctly, because it is not clear which point matches which epipolar line. A better algorithm for matching two sketches drawn from different points of view might make the modeling of curves with high curvature easier.

5.1.6 Simple Geometry Editing

Suppose that a user wants to draw a simple shape, like the back and seat of a chair; both the back and seat are rectangular, and they are at 90 degrees to one another. Figure 5.8 shows

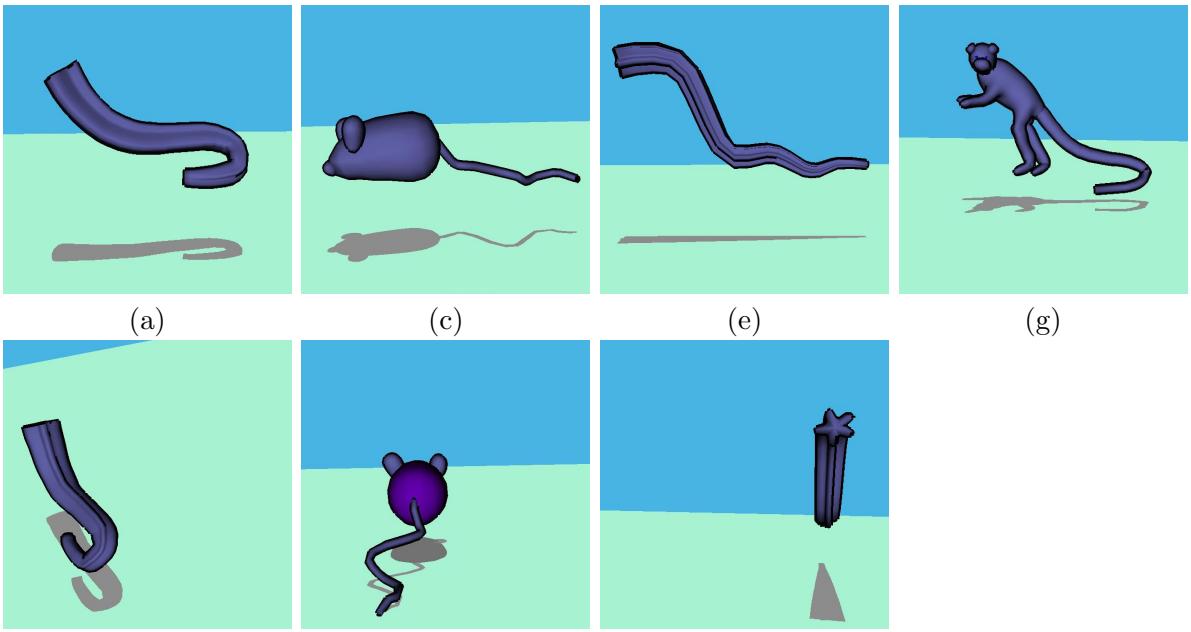
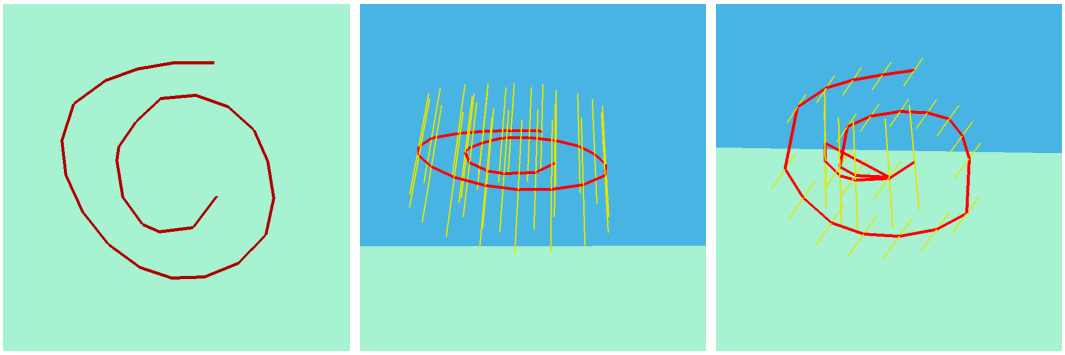


Figure 5.6: Examples of the generated generalized cylinders. (a)-(b) The lizard's tail is generated from two cross-sections that are a morph between a circle and a triangle. (c)-(d) The mouse tail is created from two circular cross-sections. (e)-(f) This generalized cylinder shows off the interpolation between flower-shaped and circular cross-sections. (g) The monkey's tail has a simple circular cross-section.



(a) (b) (c)

Figure 5.7: (a) Assume that the following flat spiral curve was drawn in the bottom plane (xz-plane) and a user wants to add torsion to this flat curve in y. (b) When the virtual camera is rotated to the most intuitive view for "redraw" operation, the system cannot project a newly drawn stroke on the epipolar lines because it is unclear which point matches which epipolar line (c).

the process the user employs in the epipolar editing system. First the user draws the back and seat, but as she rotates the view, it becomes clear that the seat, rather than sticking out towards the user, is actually a parallelogram lying in the first view-plane. It proves to be difficult to get the seat to be perpendicular to the back by re-sketching the sides of the seat. The only view in which this would be easy is one that looks along the edge joining the seat and back . . . but in that view, one cannot properly sketch the new position of the seat! The user tries to do it by using the "shear" operation: the stroke corresponding to the seat of the chair is selected and one of the corners is dragged along the epipolar line in a new view to shear the seat and make it perpendicular to the back of the chair. Unfortunately, shearing is not very intuitive and it is hard to adjust the seat and the back of the chair from just one attempt.

From situations like this, it becomes clear that although a pair of views may completely determine the shape of an object, that determination depends critically on the relative view directions and the orientation of the object itself; a user with a casual camera-positioning tool may not be able to determine how these are related and what the eventual consequences of her actions will be.

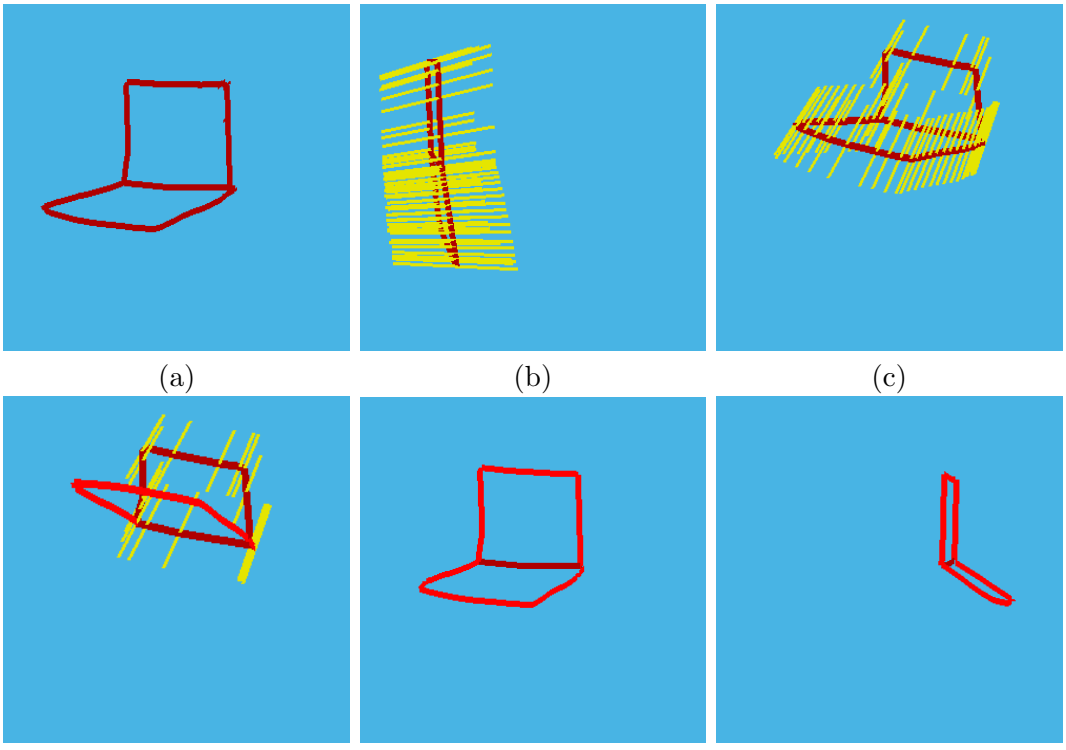


Figure 5.8: (a) The user has drawn the back of a chair (as a rectangle) and its seat (as a parallelogram). (b) When the user views it from a new direction, it is clear that the back and seat are actually coplanar. (c)-(d) The user tries to modify the seat in this new view by using a shear operation to make them actually perpendicular, but (e)-(f) the results, seen from another view, show that, although the seat and the back of the chair are not coplanar anymore, they are still not at 90 degrees to one another.

5.2 Discussion and Future Work

Epipolar-based multi-view sketching might prove, in its current form, to be a useful tool in a larger collection of methods, but one would probably not want to build a system based on this idea alone. On the other hand, there may be ways to improve the method by making cleverer guesses of the meaning of the sketch in the first view - there's no particular reason to infer that this sketch is planar, for instance. In this section, we discuss the limitations and possibilities for future work on this topic.

Despite the "obviousness" of multi-view sketching, it is surprisingly hard to use in practice. Why is this? The problem, in cases like the spiral above, is partly due to the simplicity of our matching-and-reconstruction algorithm. If it were easier to indicate which points in the new sketch matched which ones in the old sketch, some of these problems would be resolved. But it is part of the nature of *sketching* that we want to make informal drawings, not spend time precisely indicating correspondences. Furthermore, for views that are nearly parallel, the sensitivity of the reconstruction to small changes in the input can be enormous — multi-view sketching works best for nearly orthogonal pairs of views.

As the "chair" example demonstrates, there's a further problem: as the user sketches the back and seat of the chair, she has a clear 3-dimensional idea of what she has drawn. But as the view is rotated, this notion becomes untenable, and the world must be readjusted to make the notion valid again. This cognitive dissonance — between what the user "knew" that she'd drawn and what she sees as the result — makes the user's task one of "fixing the world" rather than "saying what she knows." We suspect that it is this aspect of the experience that most makes our multi-view sketching awkward. Of course, the problem mentioned in describing the chair construction is relevant as well: the particulars of the relationship between two views may have a large effect on the result of a multi-view drawing, but these particulars are difficult for a user to determine, and their effects may not be at all obvious.

The depth-inference used in our system is the simplest possible: all strokes are assumed to be in the view plane until they're redrawn from a new view. If we used some more sophisticated inference like Lipson's geometric constraints [65], the rotated view might be less jarring, as it would be only *slightly* unexpected rather than totally unexpected. Of course, if the user intended to draw a square and diamond in the same plane, the rotated view would be *more* jarring in the latter case. This naturally leads one to consider the question "Which is more *likely*?" It may be that a probabilistic formulation of the problem could form the basis for a fruitful approach. For 3D curve sketching, for instance, it is

conceivable that one could create a reasonable prior on curves and then ask for a maximum likelihood estimate of the curve drawn, given that its projection must match the drawing.

The sensitivity of the curve reconstruction when the views are near-parallel hints at a more general problem: since it is difficult to draw the same object from two different views and have the two drawings be consistent, is there a way to treat the two drawings as each providing *evidence* about the shape rather than absolute constraints?

Chapter 6

Discussion and Conclusions

While great progress has been made in the recent years in creating powerful graphics tools, they have not become widespread among non-experts. The main reason for this is that existing graphics tools are too sophisticated and take months or even years to learn. We believe that better, more intuitive and easy-to-use tools will make computer graphics more accessible to the general public and will significantly increase its use and effect on everyday life.

With the increasing popularity of pen-based devices such as tablet PCs, the impact would be even larger. Although a mouse and a keyboard are quite common input devices, they are not as intuitive to people as a pen. Interaction techniques based on pen-based input are promising since they are based on user's natural ability to draw. People are reasonably good at drawing on paper, and paper sketches are great for communicating ideas and concepts. In fact, many ideas start on a cocktail napkin. Can we turn it into a "smart" napkin by combining the power of pen-based hardware with accessible graphics software? Imagine sketching on a palm device, which immediately interprets the sketch, helping one create a 3D model, or a simple animation.

Sketch-based user interfaces have been developed for a number of different domains such as 3D modeling, animation, mathematical sketching and electronic circuit diagrams. The applications range from entertainment (for instance, the tools could let a user quickly create his own game character and/or control the character using the pressure and tilt of a pen [81]), to education (teachers explaining the physics of a pendulum by sketching the setup and corresponding formulas and animating the sketch; or a soccer coach sketching and simulating the play [64]), to early-stage design support (for experts).

This dissertation has explored ways to create more intuitive 3D modeling tools than the

ones available today. In this Chapter we summarize the contributions of the dissertation and talk about possible future directions.

6.1 Contributions

6.1.1 Intuitive 3D modeling interfaces

In Chapter 3 we introduced SmoothSketch—a system for inferring plausible 3D free-form shapes from visible-contour sketches. SmoothSketch accepts a complex user input stroke that may contain so-called cusp and T-junctions that are endpoints of hidden parts of the surface contour. Based on Williams' work [106], we created a practical system that goes from a contour drawing to a fairly smooth surface with that drawing as its visible contour, one which works for a wide variety of useful cases.

In Chapter 4 we presented a novel 3D shape modeling interface based on the constructive drawing idea. It lets a user model a complex 3D shape as a hierarchy of basic blobby shapes. The hierarchy is automatically created by the system as the user draws strokes.

Chapter 5 presented our preliminary efforts on multi-view sketching. In particular, we proposed using epipolar constraints as guides for the user during the 3D curve modeling interaction. That led to a multi-view sketching interface for modeling generalized cylinders, where first a user draws a medial axis of the cylinder from multiple points of view and then draws several cross-section strokes. Chapter 5 summarized some of the challenges of multi-view sketching and possible improvements over the current system.

6.1.2 Contour completion as a step on the way to object recognition

Part of our contribution in SmoothSketch was in the area of contour completion. Partial occlusion of objects in the scene or in the drawing is one of the reasons that the problem of object recognition is hard. Thus, contour completion is an important step in the systems that perform object recognition. We proposed an algorithm for completing contours of certain smooth surfaces in generic views where contours are given as a collection of strokes (where each stroke is a sequence of 2D points). As opposed to previous work, we can handle contours that contain cusps in addition to T-junctions. We restricted ourselves to contours where hidden parts don't intersect each other or visible contours and where the gaps to be filled are relatively short.

6.2 Limitations and Directions for Future Work

As we indicated in Chapter 3, both the hidden contour completion algorithm and the inflation algorithm for SmoothSketch have a number of limitations. In Table 6.1 we categorized the limitations in terms of the step of the framework they are encountered at, and whether they can be addressed by minor modifications of the existing algorithms, or by user interaction, or whether it is an open problem that requires more careful research. The table serves as a roadmap to the rest of the discussion about future work in this section.

Limitation	*Framework step*	*Minor extensions to the framework*	*UI solution*	*Open problem*
Hidden cusp scale invariance	I	✓	Interactive adjustment of hidden cusp position	✓
User-specified contours need to be oriented	I	✓	✓	
Hidden contours don't intersect	I	✓	Regional contour editing	
Only relatively small gaps can be filled in Step I	I	Specialized techniques	Oversketching hidden contours	✓
Beam search may not find a solution or best solution	I	More efficient implementation	Regional contour editing	Better heuristics based on perception
Back leg case	I		Editing gesture	
Smooth embedding can result in incorrect relative order	III	Incorporating springs for controlling relative order	Multi-view sketching	
Tuning constants are required for inflation	III	Estimate from contours	Sliders	✓
Surfaces with creases are not handled	I, II, III	✓	Inflate from contours, then draw creases	Extend paneling construction
Surfaces with boundary are not handled	I, II, III			Extend all three steps to handle boundaries
User can not control thickness of the generated shape	III		Multi-view sketching	✓

continued on next page

continued from previous page				
Limitation	*Framework step*	*Minor extensions to the framework*	*UI solution*	*Open problem*
Which visible contours accept completions?	I			✓
Contour completion is local	I		Oversketching hidden contours	✓
The system may not return the optimal shape	III	Global optimization solver	Multi-view sketching	✓
No principled prior on 3D shapes	III			✓
Redrawing contours locally from the same view is not allowed	I, II, III	✓	✓	✓
Redrawing contours locally from other views is not allowed	I, II, III		✓	✓

Table 6.1: We categorize different limitations of SmoothSketch. Steps I, II and III refer to Figural Completion, Paneling Construction and Smooth Embedding respectively. We also specify how the limitations can be addressed: whether minor extensions are required; if a user interface solution will suffice, or if the problem requires further research.

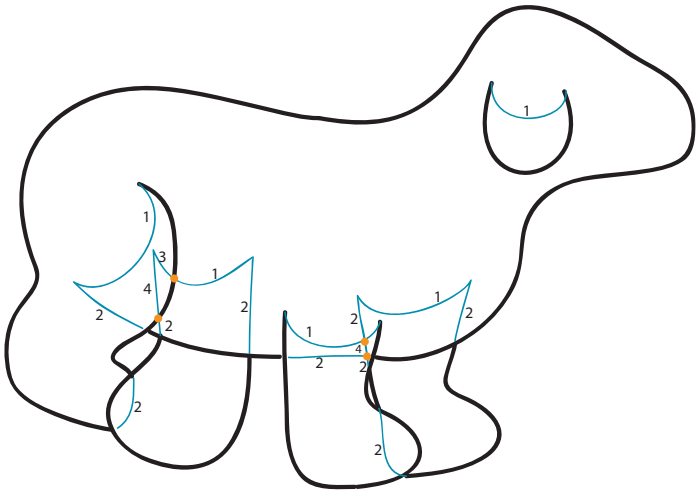


Figure 6.1: A drawing that currently can not be handled by SmoothSketch since hidden contours intersect with each other and with visible contours. Currently, the system would reject such a completion. Generalizing our figural completion algorithm to handle such cases would let a user create much more interesting shapes like the one shown here. Here, we mark new tee-junctions in orange and show valid Huffman labels for hidden contour parts. All visible contours parts have label 0.

6.2.1 Contour completion

Scale invariant algorithm for guessing hidden cusp locations

The algorithm for guessing hidden cusp locations assumes scale invariance. Thus the algorithm may produce incorrect or impossible location of a hidden cusp (see Figure 3.19). One way to avoid this would be to compute several tables of hidden cusp locations and tangent directions for several different scales. Then, we could automatically select the appropriate scale, and if the solution at the current scale leads to invalid completion, choose a different scale. A better solution would be to find a way to compactly represent scale variance in this problem.

Figural completion for the case of hidden contours intersecting other visible or hidden contours

The figural completion algorithm described in Section 3.4 does not work for drawings where hidden contours intersect visible and/or hidden contours (see Figure 6.1). We would like to generalize our algorithm so that for each pairing, it first identifies new junctions in the completed drawing (only new tee junctions can appear) and then computes the validity by

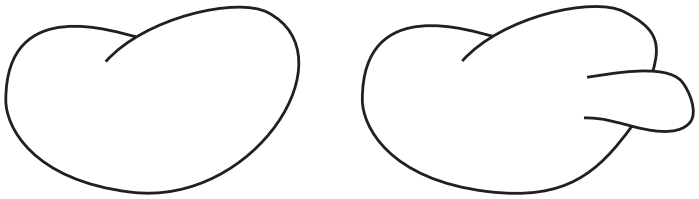


Figure 6.2: How can we modify the algorithm so that the 3D shape is modified whenever a user adds details to the original drawing?

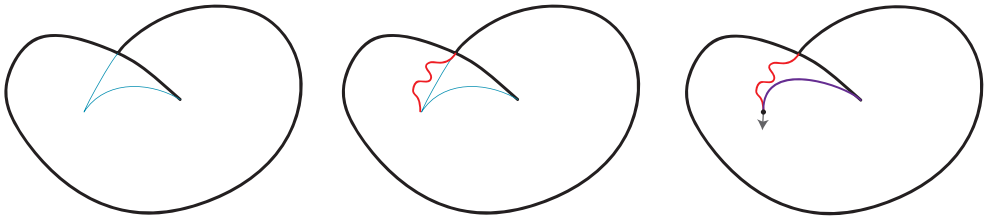


Figure 6.3: (a) A bean drawing with hidden contours computed by SmoothSketch. (b) The user oversketches one of the hidden contours, the new contour is shown in red. (c) The second hidden contour is automatically modified by the system so that the tangents of two curves meeting at the hidden cusp are equal.

trying to find a valid Huffman labeling. For each new junction there will be two choices regarding which contour is in front, and which is in the back. We have to consider all these choices and try to find a valid labeling or reject the pairing. Since the search for the best pairing grows exponentially in the number of junctions, it would be reasonable to restrict the number of new intersections to 4 or 5, and reject the pairing if it has more junctions.

More editing gestures

What gestures can we add to SmoothSketch to let a user add new parts to the drawing? Ideally, the algorithm would not start from scratch with a new input drawing, but would rather modify only the necessary portions of the shape created from the original contour drawing (see Figure 6.2).

Currently, SmoothSketch only provides a user with the ability to edit hidden contours by either manipulating the corresponding Bezier curves, or choosing a different topological completion from the list of suggestions.

More editing gestures would make SmoothSketch even more versatile and would let users

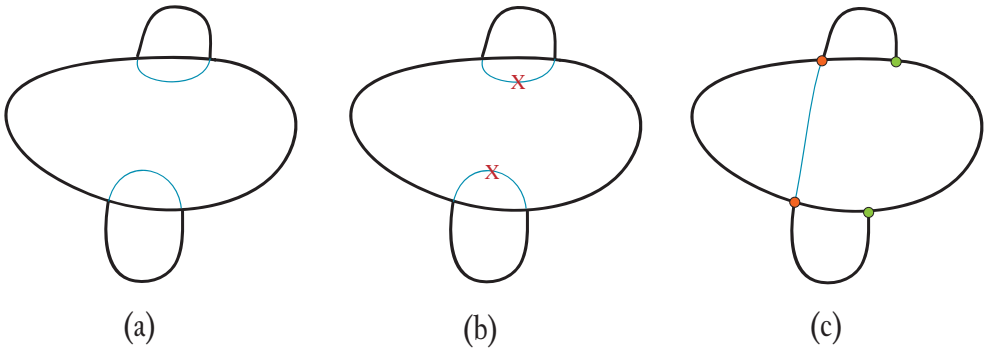


Figure 6.4: (a) A drawing with hidden contours computed by SmoothSketch. (b) The user removes hidden contours by either selecting them and pressing delete, or drawing a "cross" stroke. (c) Then the user pairs up two tee points shown in red by clicking on one of them and dragging the mouse till the other junction. The other two points can be automatically paired up by the system.

create more complex shapes.

We would like to allow a user to oversketch hidden contours to either change the shape of the contour (see Figure 6.3) or change the pairing of tees and cusps (see Figure 6.4).

Currently, we formulate figural completion as an optimization problem; for more complex contours, the search for an approximate solution can become too slow to be interactive and the resulting approximate solution is less likely to be the best one. Also, for complex drawings, the list of suggestions returned by the system is small or empty, since finding several valid completions given a limited beam width is hard. To overcome it, we could let the user select a portion of the drawing and ask the system to recompute figural completion for that portion alone, keeping the completion for the remainder of the drawing fixed (see Figure 6.5). It is a simpler problem (since there are fewer junctions that need to be paired up), thus it can be solved more efficiently.

Describing the set of visible contour drawings that admit completions

The question of what visible contour drawings admit completions remains open. In Chapter 3 we proved a certain class of contours is completable, but this class does not include all completable drawings as we demonstrated by examples.

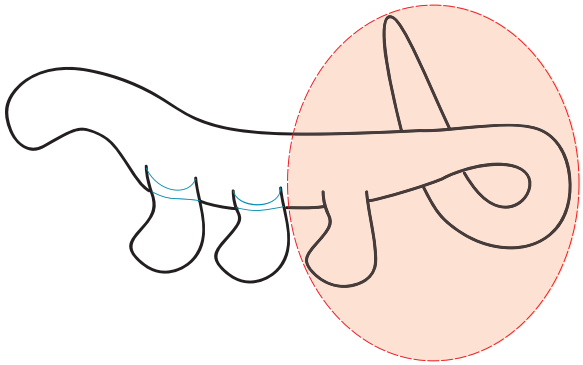


Figure 6.5: Regional contour editing gesture: a user draws a red stroke to specify a region of the drawing for which the contour completion should be recomputed.

6.2.2 Inflation

The inflation algorithm in SmoothSketch based on the mass-spring system relaxation requires tuning constants and produces self-intersecting surfaces, which may be undesirable for some applications. We would like to produce mesh embedding with "folds" matching the drawn contour.

Prior on free-form shapes

If we knew the right prior on free-form shapes, we could develop a more principled inflation algorithm that would optimize something about expected shapes of the inflated surface, conditioned on the known shapes of the visible contours.

'Push-off' curve idea

Here we describe an idea of how to obtain a smoother bean surface from its contour drawing.

Figure 6.6(a) shows a 3D curve γ that is a contour generator for the bean surface. Note that γ is smooth at the cusps, unlike the embedded silhouette edges in the inflation algorithm described in Chapter 3. We would like to compute this 3D curve from the given 2D contour drawing in the xy-plane. Assume for a moment that the z-coordinates Z_C and Z_H of the visible cusp C and the hidden cusp H are known. We also know the tangent vectors to γ at the cusps: $(0, 0, 1)$ and $(0, 0, -1)$. We can compute the z-coordinates of γ by fitting a Bezier curve to the known positions and tangent vectors.

Then, for each point on the curve we could compute the normals N to the **surface** at this point (note that the surface normal at a point and the normal to the curve lying on the

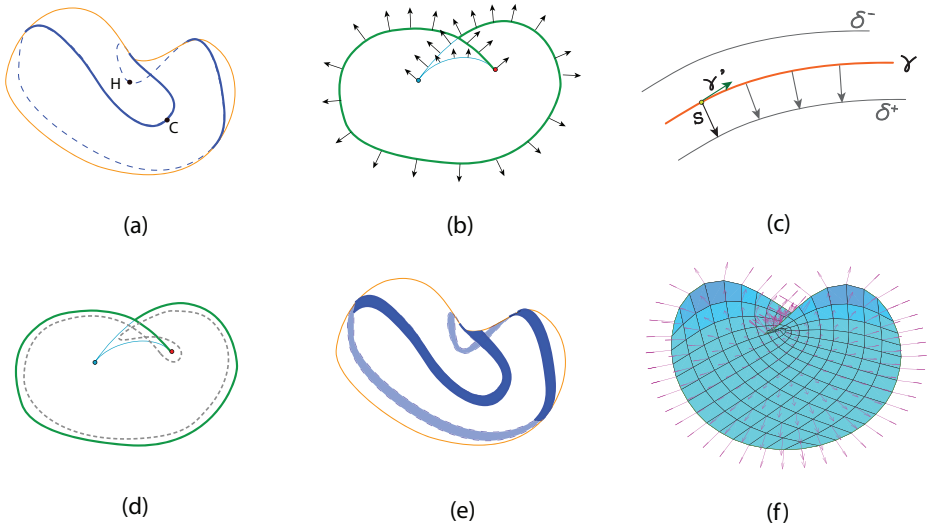


Figure 6.6: (a) A contour generator γ of the kidney bean surface is shown in blue. The visible cusp is marked with C and the hidden cusp with H. (b) The 2D contour of the bean is shown together with the normals to the surface at each point. (c) The curves $\delta^{\pm}$ are obtained by displacing γ along the vector $S = \frac{\gamma'}{||\gamma'||} \times N$. (d) One of such curves, δ^{+}, is shown here dotted in gray. (e) γ is extruded slightly in the tangent plane to create a thin ribbon. (f) The depths and normals are interpolated to the interior of the bean.

surface and passing through the point are not the same). Since this is a silhouette curve, the z-component is 0 for all normals. Normals to the surface of the bean along the contour are shown in Figure 6.6(b). We can also find a vector $S = \frac{\gamma'}{||\gamma'||} \times N$ at each point, the normal to γ **in the surface**.

The curve γ could then be slightly displaced in the tangent plane to get $\delta^{\pm} = \gamma \pm \epsilon S$ (see Figures 6.6(c, d)). Taking ϵ in some small interval yields a thin piece of ribbon as shown in Figure 6.6(e); the edges of the ribbon have no cusps. The normals along δ^{+} (or δ^{-}) are interpolated to the interior of the bean. We can then fit a smooth surface to this normal field, subject to the positional constraints of the contour generator curve (see Figure 6.6(f)).

So far, we assumed that we fixed z coordinates at the cusps. Instead, we could make them parameters to the optimization problem, and find the smooth surface that has the least energy (probably in the least squared mean curvature sense) over values of Z_C and Z_H.

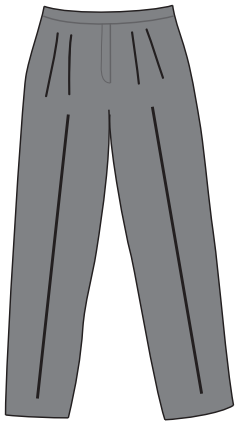


Figure 6.7: Extending SmoothSketch to handle surfaces with creases (like the creases on the pants shown here) would allow a user to model a larger variety of shapes.

Tracing contours in the image and automatically texturing the model

Some details are easier to add to the model as texture. One can think of loading an image in SmoothSketch (for example, an image of the dog in Figure 1.3), tracing the contours in the image and then automatically texturing the model.

Surfaces with sharp features

We would like to extend SmoothSketch to include minor surface discontinuities—things like ridges or creases on a surface (see Figure 6.7), which often are perceptually significant, and indeed, in cases like armpits, creased shapes are what a user might want to create.

One way to handle creases would be to create a smooth surface from contours as we did in SmoothSketch, and then let a user draw crease lines on the surface. Then, we could use our second inflation technique based on energy minimization, where for all edges except the crease edges, the energy minimized is the sum of dihedral angles. For crease edges, we could define a different energy term that favors sharp angles.

Alternatively, we could start with a user drawing that consists of contours and crease lines, and extend the paneling construction algorithm to deal with such drawings. The distinction between creases and contours would have to be noted by the user. Figure 6.8 demonstrates how different shapes can be obtained depending on whether a stroke is handled as a crease or a contour.

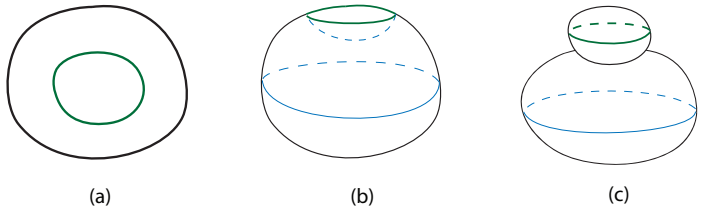


Figure 6.8: (a) A contour drawing, where the inner green stroke can be interpreted as either a contour or a crease. (b) The shape inferred from the sketch assuming that the inner green stroke is a crease (side view). (c) The shape inferred assuming that the inner green stroke is a counter clockwise contour (side view).

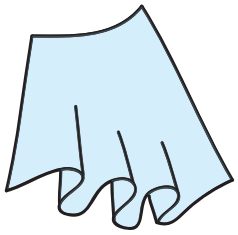


Figure 6.9: A piece of cloth is one example of surface with boundary.

Smooth surfaces with boundaries

We would like to extend figural completion, the paneling construction, and the inflation algorithms to handle contour drawings of smooth surfaces with boundary. This would be useful for modeling garments with folds and wrinkles, for example (see Figure 6.9).

Figure 3.1 shows four cases of Huffman's labeling for the case of smooth closed surfaces. In a similar fashion, there are several cases of Huffman's labeling for surfaces with boundary. The paneling construction algorithm developed by Williams does not apply to the cases of general surfaces with boundary. So before we can approach the problem of inflating such drawings, we need to extend paneling construction to such cases.

Multi-view sketching

Inferring a 3D shape from a single-view sketch is an inherently underconstrained problem. While the system could return one of the reasonable interpretations of the drawing, this interpretation will not always match the user's intention. Drawing strokes from other points of view may resolve some of these ambiguities and provide the user with more control on the shape.

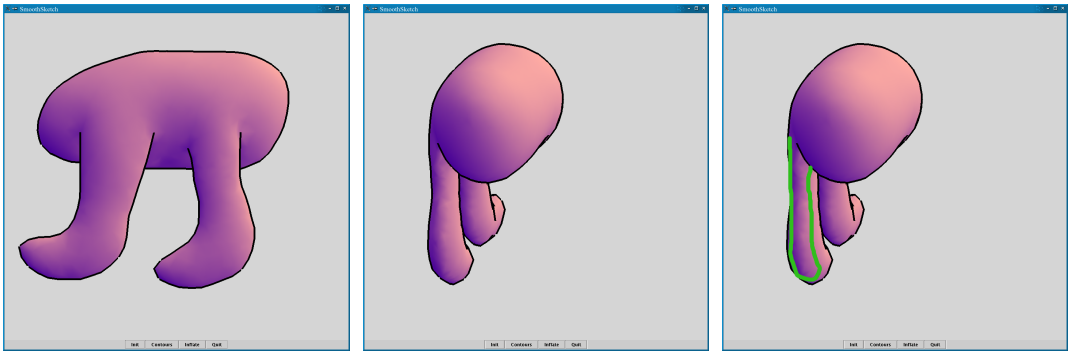


Figure 6.10: From left to right: an inflated drawing of a dog's body with two legs; the user changes the camera view to look at the dog from the back; the user oversketches the silhouette of one of the legs to make it thinner.

The problem can be posed probabilistically: given a set of 3D contours drawn from different points of view, find a 3D shape that matches the curves well and belongs to the class of smooth free-form shapes. We should take into account the fact that the user may draw contours that are inconsistent with each other. Thus, rather than satisfying the contour constraints precisely, a probabilistic approach is a better idea. We could assume that we know the prior on shapes, and that the user-drawn contours are true contours plus some noise. We could then find a shape such that the probability of the contours given the shape multiplied by the prior is maximized.

From an interface point of view, it would be useful to let the user specify or change thickness of different parts of the shape in SmoothSketch. One could imagine a system where a user would draw the dog from one point of view, then inflate the drawing, change the camera view so that he is looking at the dog from the back, and then oversketch the silhouette strokes of the legs to make them thinner (see Figure 6.10).

A hybrid system: matching a user's sketch against a database of 3D models

We have developed SmoothSketch to be agnostic about shape, treating it purely geometrically, yet users *are* familiar with many shapes. We imagine the possibility of a hybrid system, in which the user's sketch is both inflated *and* matched against a large database of known forms, for possible suggestions ("You seem to be drawing a dog; would you like us to add the hidden legs for you?"). The problems of searching such a database and forming reliable hypotheses, however, seems daunting.

Bibliography

[1] Google SketchUp – a 3D sketching software for the conceptual phases of design. http://www.sketchup.com.

[2] JDSL: The data structures library in Java. Brown University software. http://www.cs.brown.edu/cgc/jdsl/.

[3] Autodesk maya, 2007. http://www.autodesk.com/maya.

[4] Alberto S. Aguado, Eugenia Montiel, and Ed Zaluska. Modeling generalized cylinders via fourier morphing. In *ACM Transactions on Graphics*, volume 14, pages 293–315, 1999.

[5] Anca Alexe, Loic Barthe, Marie-Paule Cani, and Véronique Gaildrat. Shape modeling by sketching using convolution surfaces. In *Pacific Graphics*, Short paper, 2005.

[6] Lee J. Ames, editor. *Draw 50 Animals*. Main Street Books, 1985.

[7] Loic Barthe, Vronique Gaildrat, and Ren Caubet. Implicit extrusion fields. In *The 2000 International Conference on Imaging Science, Systems, and Technology (CISST'2000)*, pages 75–81. CSREA press, June 2000.

[8] Loic Barthe, Vronique Gaildrat, and Ren Caubet. Extrusion of 1D implicit profiles: Theory and first application. In *International Journal of Shape Modeling*, volume 7/2, December 2001.

[9] Thomas Baudel. A mark-based interaction paradigm for free-hand drawing. In *ACM Symposium on User Interface Software and Technology*, pages 185–192, 1994.

[10] G. Bellettini, V. Beorchia, and M. Paolini. Topological and variational properties of a model for the reconstruction of three-dimensional transparent images with self-occlusions.

[11] Irving Biederman and Ginny Ju. Surface versus edge-based determinants of visual recognition. *Cognitive Psyhcology*, 20:38–64, 1988.

[12] R.L. Bishop. There is more than one way to frame a curve. *American Mathematical Monthly*, 82:246–251, 1975.

[13] Jules Bloomenthal. An implicit surface polygonizer. In P. Heckbert, editor, *Graphics Gems IV*. Academic Press, New York, 1994.

[14] Jules Bloomenthal, editor. *Introduction to Implicit Surfaces*. Morgan-Kaufmann, 1997.

[15] Mario Botsch, Mark Pauly, Christian Rossl, Stephan Bischoff, and Leif Kobbelt. Geometric modeling based on triangle meshes. SIGGRAPH 2006 Course Notes, 2006.

[16] David Bourguignon, Marie-Paule Cani, and George Drettakis. Drawing for illustration and annotation in 3D. *Computer Graphics Forum*, 20(3):114–122, 2001.

[17] Kenneth A. Brakke. The Surface Evolver software. http://www.susqu.edu/facstaff/b/brakke/evolver/.

[18] Kenneth A. Brakke. The Surface Evolver. *Experimental Mathematics*, 1(2):141–165, March 1992.

[19] Jonathan C. Carr, Richard K. Beatson, Jon B. Cherrie, Tim J. Mitchell, W. Richard Fright, Bruce C. McCallum, and Tim R. Evans. Reconstruction and representation of 3D objects with radial basis functions. In *Proceedings of SIGGRAPH 2001*, pages 67–76, August 2001.

[20] Joseph J. Cherlin, Faramarz Samavati, Mario C. Sousa, and Joaquim A. Jorge. Sketch-based modeling with few strokes. In *SCCG '05: Proceedings of the 21st spring conference on Computer graphics*, pages 137–145, 2005.

[21] Roberto Cipolla and Peter Giblin. *Visual motion of curves and surfaces*. Cambridge University Press, 1999.

[22] Jonathan M. Cohen, Lee Markosian, Robert C. Zeleznik, and John F. Hughes. An interface for sketching 3d curves. In *Proceedings of the ACM Symposium on Interactive 3D Graphics*, pages 17–21, 1999.

[23] Frederic Cordier and Hyewon Seo. Free-form sketching of self-occluding objects. *IEEE Computer Graphics and Applications*, 27(1):50–59, 2007.

[24] Wagner Toledo Corrja, Robert J. Jensen, Craig E. Thayer, and Adam Finkelstein. Texture mapping for cel animation. In *SIGGRAPH'98*, pages 435–446, August 1998.

[25] Thomas Crulli. A hierarchical constructive drawing approach to freeform sketching. Master's thesis, Brown University, 1994.

[26] James Davis, Maneesh Agrawala, Erika Chuang, Zoran Popovic, and David Salesin. A sketching interface for articulated figure animation. In *ACM Symposium on Computer Animation*, pages 320–328, 2003.

[27] H. Q. Dinh, G. Turk, and G. Slabaugh. Reconstructing surfaces using anisotropic basis functions. In *Proc. ICCV*, volume 2, pages 606–613, Vancouver, Canada, July 2001.

[28] M. P. Do Carmo. *Differential geometry of curves and surfaces*. Prentice-Hall, Englewood Cliffs, NJ, 1976.

[29] L. Euler. Methodus inveniendi lineas curvas maximi minimive proprietate gaudentes. 1744.

[30] Ramakant Nevatia Fatih Ulupinar. Shape from contour: Straight homogeneous generalized cylinders and constant section generalized cylinders. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 17/2, February 1995.

[31] David Forsyth and Jean Ponce. *Computer Vision: A Modern Approach*. Prentice Hall, 2003.

[32] Thomas Funkhouser and Michael Kazhdan. Shape-based retrieval and analysis of 3d models. SIGGRAPH 2004 Course Notes, 2004.

[33] S. Gibson and B. Mirtich. A survey of deformable modeling in computer graphics. Technical Report TR-97-19, Mitsubishi Electric Research Lab., Cambridge, MA, 1997.

[34] Richard L. Gregory. Knowledge in perception and illusion. In *Phil. Trans. R. Soc. Lond. B*, volume 352, pages 1121–1128, 1997.

[35] Ulf Grenander. *Lectures in Pattern Theory*, volume 1-3. Springer-Verlag, 1981.

[36] H. B. Griffiths. *Surfaces*. Cambridge University Press, 1981.

[37] V. Guillemin and A. Pollack. *Differential Topology*. Prentice Hall, 1974.

[38] G. Guy and G. Medioni. Inferring global perceptual contours from local features. In *Image Understanding Workshop*, pages 881–892, 1993.

[39] H. Helmholtz. *Physiological Optics, Vol. III: The perceptions of vision*. Optical Society of America, Rochester, NY, 1925.

[40] Kenneth P. Herndon, Robert C. Zeleznik, Daniel C. Robbins, D. Brookshire Conner, Scott S. Snibbe, and Andries van Dam. Interactive shadows. In *ACM Symposium on User Interface Software and Technology*, pages 1–6, 1992.

[41] Eric Hill. *Spot Goes to the Circus*. Puffin, August 1994.

[42] D. D. Hoffman and Whitman Richards. Parts of recognition. Technical Report AIM-732, 1983.

[43] Donald D. Hoffman. *Visual Intelligence: How We Create What We See*. W. W. Norton, 2000.

[44] D. A. Huffman. Impossible objects as nonsense sentences. In B. Meltzer and D. Michie, editors, *Machine Intelligence 6*. American Elsevier Publishing Co., New York, 1971.

[45] Takeo Igarashi and John F. Hughes. A suggestive interface for 3d drawing. In *ACM Symposium on User Interface Software and Technology*, 2001.

[46] Takeo Igarashi and John F. Hughes. Smooth meshes for sketch-based freeform modeling. In *Symposium on Interactive 3D Graphics*, 2003.

[47] Takeo Igarashi, Satoshi Matsuoka, and Hidehiko Tanaka. Teddy: A sketching interface for 3D freeform design. In *Proceedings of SIGGRAPH 99*, pages 409–416, August 1999.

[48] Takashi Ijiri, Shigeru Owada, Makoto Okabe, and Takeo Igarashi. Floral diagrams and inflorescences: interactive flower modeling using botanical structural constraints. *ACM Trans. Graph.*, 24(3):720–726, 2005.

[49] Katsushi Ikeuchi and Berthold K. P. Horn. Numerical shape from shading and occluding boundaries. *Artificial Intelligence*, 17:141–184, 1981.

[50] Scott F. Johnston. Lumo: illumination for cel animation. In *Proceedings of the Symposium on Non-photorealistic animation and rendering*, pages 45–52, 2002.

[51] Michael Kallay. Constrained optimization in surface design. In *In Modeling in Computer Graphics*. Springer Verlag, 1993.

[52] G. Kanizsa. *Organization in vision*. Praeger, New York, 1979.

[53] Olga Karpenko and John F. Hughes. Inferring 3D free-form shapes from contour drawings. In *SIGGRAPH 2005 Sketches Program*, 2005.

[54] Olga Karpenko, John F. Hughes, and Ramesh Raskar. Free-form sketching with variational implicit surfaces. In *Eurographics Computer Graphics Forum*, volume 21/3, pages 585–594, 2002.

[55] Olga Karpenko, John F. Hughes, and Ramesh Raskar. Epipolar methods for multi-view sketching. In *Eurographics Workshop on Sketch-Based Interfaces*, 2004.

[56] Olga A. Karpenko and John F. Hughes. SmoothSketch: 3D free-form shapes from complex sketches. *ACM Transactions on Graphics*, 25(3):589–598, 2006.

[57] M. Kass, A. Witkin, and D. Terzopolous. Snakes: Active contour models. In *International Conference on Computer Vision*, pages 259–268, 1987.

[58] Youngihn Kho and Michael Garland. Sketching mesh deformations. In *Symposium on Interactive 3D Graphics and Games 2005*, 2005.

[59] David C. Knill and Whitman Richards. *Perception as Bayesian inference*. Cambridge University Press, 1996.

[60] Leif P. Kobbelt, Thilo Bareuther, and Hans-Peter Seidel. Multiresolution shape deformations for meshes with dynamic vertex connectivity. *Computer Graphics Forum*, 19(3), 2000.

[61] J. J. Koenderink. What does the occluding contour tell us about solid shape. In *Perception*, volume 13, pages 321–330, 1984.

[62] J. J. Koenderink. *Solid Shape*. MIT Press, 1990.

[63] Joseph LaViola, Randall Davis, and Takeo Igarashi. An introduction to sketch-based interfaces. SIGGRAPH 2006 Course Notes, 2006.

[64] Joseph J. LaViola and Robert C. Zeleznik. Mathpad2: a system for the creation and exploration of mathematical sketches. In *ACM Transactions on Graphics*, pages 432–440, 2004.

[65] H. Lipson and M. Shpitalni. Conceptual design and analysis by sketching. In *AIDAM-97*, 1997.

[66] Angeline M. Loh. *The recovery of 3D structure using visual texture patterns.* PhD thesis, The University of Western Australia, 2006.

[67] Jerome Maillot and Jos Stam. A unified subdivision scheme for polygonal modeling. In *Proc. Eurographics*, volume 20/3, 2001.

[68] Shahzad Malik. A sketching interface for modeling and editing hairstyles. In *Eurographics Workshop on Sketch-Based Interfaces and Modeling*, pages 185–194, 2005.

[69] Lee Markosian, Jonathan M. Cohen, Thomas Crulli, and John F. Hughes. Skin: A constructive approach to modeling free-form shapes. In *Proceedings of SIGGRAPH 99*, pages 393–400, August 1999.

[70] David Marr. *Vision: a computational investigation into the human representation and processing of visual information.* W.H. Freeman, San Francisco, CA, 1982.

[71] S. Masnou and J. Morel. Level lines based disocclusion. In *5th International conference on Image Processing*, pages 259–263, October 1998.

[72] Richard S. Millman and George D. Parker. *Elements of Differential Geometry.* Prentice-Hall, Inc., 1977.

[73] Bojan Mohar and Carsten Thomassen. *Graphs on Surfaces.* The Johns Hopkins University Press, 2001.

[74] Henry Packard Moreton. *Minimum Curvature Variation Curves, Networks, and Surfaces for Fair Free-Form Shape Design.* PhD thesis, EECS Department, University of California, Berkeley, 1993.

[75] T. Moscovich and J. F. Hughes. Animation Sketching: An approach to accessible animation. Technical Report CS-04-03, Brown University Computer Science Department.

[76] David Mumford. Elastica and computer vision. In Chandrajit L. Bajaj, editor, *Algebraic Geometry and Its Applications.* Springer-Verlag New York Inc., 1994.

[77] Andrew Nealen, Takeo Igarashi, Olga Sorkine, and Marc Alexa. FiberMesh: Designing freeform surfaces with 3D curves. In *ACM Transactions on Computer Graphics*, 2007.

[78] Andrew Nealen, Olga Sorkine, Marc Alexa, and Daniel Cohen-Or. A sketch-based interface for detail-preserving mesh editing. *ACM SIGGRAPH Transactions on graphics*, 24(3), 2005.

[79] M. Nitzberg, D. Mumford, and T. Shiota. *Filtering, Segmentation, and Depth*. Springer-Verlag, 1993.

[80] Makoto Okabe, Shigeru Owada, and Takeo Igarashi. Interactive design of botanical trees using freehand sketches and example-based editing. *Computer Graphics Forum*, 24(3):487–496, 2005.

[81] Masaki Oshita. Pen-to-mime: A pen-based interface for interactive control of a human figure. In *Eurographics workshop on sketch-based interfaces and modeling*, pages 43–52, 2004.

[82] Stephen E. Palmer. *Vision Science: Photons to Phenomenology*. The MIT Press, May 1999.

[83] Alex Pentland and Jeff Kuo. The artist at the interface. Technical Report 114, MIT Media Lab, 1989.

[84] Joao P. Pereira, Vasco A. Branco, Joaquim A. Jorge, Nelson F. Silva, Tiago D. Cardoso, and F. Nunes Ferreira. Cascading recognizers for ambiguous calligraphic interaction. In *Eurographics Workshop on Sketch-Based Interfaces and Modeling*, 2004.

[85] Gabriele Peters. Theories of Three-Dimensional Object Perception - A Survey. In *Recent Research Developments in Pattern Recognition*. Transworld Research Network, 2000.

[86] Mukta Prasad and Andrew Fitzgibbon. Single view reconstruction of curved surfaces. In *CVPR '06: Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 1345–1354, 2006.

[87] Stuart Russel and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 1995.

[88] Vladimir V. Savchenko, Alexander A. Pasko, Oleg G. Okunev, and Tosiyasu L. Kunii. Function representation of solids reconstructed from scattered surface points and contours. *Computer Graphics Forum*, 14(4):181–188, 1995.

[89] Ryan Schmidt, B. Wyvill, M. C. Sousa, and J. A. Jorge. Shapeshop: Sketch-based solid modeling with blobtrees. In *Eurographics Workshop on Sketch-Based Interfaces and Modeling*, pages 53–62, 2005.

[90] Robert Schneider and Leif Kobbelt. Geometric fairing of irregular meshes for free-form surface design. *Computer Aided Geometric Design*, 18(4):359–379, May 2001.

[91] T. W. Sederberg and E. Greenwood. A physically based approach to 2d shape blending. In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, volume 26, pages 25–34, July 1992.

[92] Amit Shesh and Baoquan Chen. Smartpaper–an interactive and user-friendly sketching system. In *Eurographics 2004*, 2004.

[93] Jonathan R. Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, May 1996.

[94] Gabriel Taubin. A signal processing approach to fair surface design. *Computer Graphics*, 29:351–358, 1995.

[95] Gabriel Taubin. Linear anisotropic mesh filtering. Technical Report RC-22213, IBM Research, 2001.

[96] Bill Tilton, editor. *Drawing and Painting Animals*. North Light Books, 1996.

[97] Osama Tolba, Julie Dorsey, and Leonard McMillan. A projective drawing system. In *2001 ACM Symposium on Interactive 3D Graphics*, pages 25–34, March 2001.

[98] Greg Turk. Re-tiling polygonal surfaces. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 55–64, 1992.

[99] Greg Turk and James O'Brien. Shape transformation using variational implicit functions. In *Proceedings of SIGGRAPH 99*, pages 335–342, August 1999.

[100] Emmanuel Turquin, Marie-Paule Cani, and John Hughes. Sketching garments for virtual characters. In John F. Hughes and Joaquim A. Jorge, editors, *Eurographics Workshop on Sketch-Based Interfaces and Modeling (SBM)*, August 2004.

[101] P. Varley. *Automatic Creation of Boundary-Representation Models from Single Line Drawings*. PhD thesis, Cardiff University, 2002.

[102] William Welch and Andrew Witkin. Variational surface modeling. *Computer Graphics*, 26(2):157–166, 1992.

[103] William Welch and Andrew Witkin. Free-form shape design using triangulated surfaces. *ACM SIGGRAPH Transactions on graphics*, 28:247–256, 1994.

[104] H. Whitney. On singularities of mappings of euclidean spaces. In *Annals of Mathematics*, pages 374–410, 1955.

[105] Lance Williams. Shading in two dimensions. In *Graphics Interface*, pages 143–151, 1991.

[106] Lance R. Williams. *Perceptual Completion of Occluded Surfaces*. PhD thesis, University of Massachusetts, 1994.

[107] Lance R. Williams. Topological reconstruction of a smooth manifold-solid from its occluding contour. *Intl. Journal of Computer Vision*, 23(1):93–108, 1997.

[108] Lance R. Williams and David W. Jacobs. Stochastic Completion Fields: A Neural Model of Illusory Shape and Salience. In *Neural Computation*, volume 9(4), pages 837–858, 1997.

[109] Andrew P. Witkin. *Shape from contour*. PhD thesis, MIT, 1980.

[110] Brian Wyvill, Andrew Guy, and Eric Galin. Extending the CSG tree - warping, blending and boolean operations in an implicit surface modeling system. *Computer Graphics Forum*, 18(2):149–158, June 1999.

[111] Robert C. Zeleznik and Andrew Forsberg. Unicam — 2D gestural camera controls for 3D environments. In *1999 ACM Symposium on Interactive 3D Graphics*, pages 169–174. ACM SIGGRAPH, April 1999.

[112] Robert C. Zeleznik, K. Herndon, and J. Hughes. Sketch: An Interface for Sketching 3D Scenes. In *Proceedings of SIGGRAPH 96*, pages 163–170, August 1996.

[113] Ruo Zhang, Ping-Sing Tsai, James Edwin Cryer, and Mubarak Shah. Shape from shading: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(8):690–706, 1999.