Topological Manipulation of Isosurfaces

by

Hamish Carr

B.Sc.(Hons.) 1987 University of Manitoba,

LL.B. 1990 University of Manitoba,

B.C.Sc.(Hons.) 1998 University of Manitoba,

M.Sc. 2000 University of British Columbia

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

we accept this thesis as conforming
to the required standard

The University of British Columbia

April 2004

© Hamish Carr, 2004

Abstract

In this thesis, I show how to use the topological information encoded in an abstraction called the *contour tree* to enable interactive manipulation of individual contour surfaces in an isosurface scene, using an interface called the *flexible isosurface*. Underpinning this interface are several improvements and extensions to existing work on the contour tree. The first, and most critical, extension, is the *path seed*: a new method of generating seeds from the contour tree for isosurface extraction. The second extension is to compute geometric information called *local spatial measures* for contours and store this information in the contour tree. The third extension is to use local spatial measures to simplify both the contour tree and isosurface displays. This simplification can also be used for noise removal. Lastly, this thesis extends work with contour trees from simplicial meshes to arbitrary meshes, interpolants, and tessellation cases.

Contents

Abstract ii

Contents iii

List of Figures xii

List of Tables xvi

Acknowledgements xvii

Dedication xviii

I Introduction 1

1 Overview 2

2 Introduction 4

2.1 Visualization 4

2.2 Contour Lines and Isosurfaces 6

2.3 The Contour Tree 8

2.4 Application of this Thesis 8

3 Roadmap 10

3.1 Contour Trees and Visualization 10

3.2 Contour Trees and Related Ideas 10

3.3 Chapter Interrelationships 13

II Background **15**

4 Mathematical Preliminaries **17**

4.1 Visualization and Data Reduction 17

4.2 Sampling and Reconstruction 18

4.3 Manifolds and Intensity Maps 19

4.4 Level Sets and Contours 19

4.5 Topological Abstractions 20

4.5.1 Watersheds 21

4.5.2 Morse Complexes 21

4.5.3 Reeb Graphs 22

4.5.4 Contour Trees 23

5 Isosurface Extraction **25**

5.1 Early Approaches: Boundary Detection and Implicit Surfaces 25

5.2 Marching Cubes 27

5.2.1 Speed 28

5.3 Continuation 30

5.3.1 Topological Index Structures 30

6 Contour Trees **32**

6.1 Description of the Contour Tree 32

6.2 Previous Work 34

6.3 The Contour Tree 34

6.4 The Augmented Contour Tree 39

III New Applications for Contour Trees **41**

7 Algorithms for Computing Contour Trees **43**

7.1 Parameters for Analysis 43

7.2 Contour Nesting 44

7.3 Skeletonization 44

7.4 Monotone Path Search 44

7.5 Contour Sweep 45

7.6 Sweep and Merge 46

7.6.1 Join and Split Sweeps 47

7.6.2 Merging the Join and Split Trees 47

7.6.3 Computing the Unaugmented Contour Tree Directly 48

7.6.4 Variations on Sweep and Merge 51

7.7 Sample Contour Tree Computation 52

7.7.1 Join And Split Sweeps 52

7.7.2 Augmenting the Join and Split Trees 54

7.7.3 Merging the Join and Split Trees 54

8 Isosurface Seed Generation **60**

8.1 Minimal Seed Sets 60

8.1.1 Properties of Minimal Seed Sets 61

8.1.2 Example of a Minimal Seed Set 63

8.2 Path Seeds 65
8.2.1 Description of Path Seeds 65
8.2.2 Definition of Path Seeds 66
8.2.3 Properties of Path Seeds 66
8.2.4 Path Seed Algorithms 68
8.2.5 Example of Path Seeds 73
8.3 Summary and Comparison 75

9 Flexible Isosurfaces **76**

9.1 Previous Work 76
9.2 Assumptions 79
9.3 A User's View of Flexible Isosurfaces 79
9.3.1 Manipulating Contours Visually 80
9.3.2 Manipulating Contours That Are Not Visible 81
9.3.3 Contour Manipulation Operations 83
9.4 The Flexible Isosurface 83
9.5 The Flexible Isosurface Interface 84
9.6 Evolving A Contour 87
9.6.1 Contour Evolution Policies 89
9.6.2 Continuous and Reversible Contour Evolution 89
9.6.3 Giving Precedence to Evolving Contours 90
9.6.4 Multiple Selection 91
9.7 Data Structures for Flexible Isosurface Manipulation 91
9.8 Algorithms for Flexible Isosurfaces 92
9.8.1 Defining A Level Set 92

9.8.2 Algorithm for Evolving A Contour 93
9.8.3 Choosing Sets of Contours 93
9.9 Layout Problems in the Contour Tree Display 95
9.10 Summary of Contributions 97

10 Local Spatial Measures 99

10.1 Previous Work 99
10.2 Geometric Properties of Regions Bounded By A Contour 100
10.3 Upstart and Downstart Regions 102
10.4 Geometric Properties of Upstart and Downstart Regions 104
10.5 Algorithm For Computing Local Spatial Measures 110
10.6 Example of Local Spatial Measure Computation 113
10.7 Sample Local Spatial Measures 114
10.8 Approximate Local Spatial Measures 115
10.9 Summary 116

11 Contour Tree Simplification 117

11.1 Noise in the Contour Tree 117
11.2 Simplification and Visualization 119
11.3 Applying Graph Simplifications to the Contour Tree 122
11.3.1 Vertex Collapses and Leaf Pruning 123
11.3.2 Collapse Rules 125
11.4 Simplification Algorithms 127
11.4.1 Hierarchical Path Seeds 130
11.5 Examples of Simplification 133
11.5.1 Examples of Simplification with Different Rules 134

11.5.2 Examples of Simplification with Different Local Spatial Measures . . . 134
11.6 Simplified Flexible Isosurfaces . . . 136
11.6.1 Effects of Collapse on Flexible Isosurface . . . 140
11.6.2 Effects of Uncollapse on Flexible Isosurface . . . 142
11.6.3 Using Collapse to Define Layout and Colour . . . 142
11.7 Using Topological Simplification to Remove Noise . . . 143
11.8 Summary of Contributions . . . 143

IV Imperfect Data **145**

12 Contour Trees for Non-Simplicial Meshes **147**
12.1 Introduction . . . 147
12.2 Previous Work . . . 148
12.3 Join and Split Graphs for Arbitrary Meshes . . . 148
12.4 Join and Split Graph Lookup Tables . . . 152
12.5 An Example: the Bilinear Interpolant . . . 159
12.6 Contour Trees for Tessellation Cases . . . 160
12.7 Generating Isosurface Seeds . . . 163
12.8 Piecewise Continuation . . . 163
12.9 Summary . . . 165

13 Contour Trees for Multilinear Interpolants **167**
13.1 Join and Split Graphs for the Bilinear Interpolant . . . 167
13.1.1 Bilinear Spatial Measures . . . 170
13.2 Trilinear Topology Graphs . . . 170
13.2.1 Trilinear Local Spatial Measures . . . 173

13.2.2 Summary . . . 174

13.3 Higher Dimensions . . . 174

13.4 Summary . . . 175

14 Contour Trees for Marching Cubes **176**

14.1 Join and Split Graphs for Marching Cubes . . . 177

14.2 Local Spatial Measures for Marching Cubes . . . 178

14.3 Summary . . . 178

15 Perturbation **181**

15.1 Removing Perturbation . . . 182

15.2 Perturbation and Path Seeds . . . 186

15.3 Perturbation and Non-Simplicial Contour Trees . . . 186

15.4 Summary . . . 186

V Results and Conclusions **188**

16 Results **190**

16.1 Sources of Data . . . 190

16.2 Contour Tree Computation . . . 191

16.3 Results for Marching Cubes . . . 193

16.4 Isosurface Extraction Using Path Seeds . . . 194

16.5 Images Produced Using Simplified Contour Trees . . . 195

17 Conclusions **200**

18 Future Work **202**

18.1 Improvements to Contour Tree Algorithms . . . 202

18.2 Topology of Non-Isovalued Surfaces 203

18.3 Variations in the Flexible Isosurface Interface 203

18.4 Local Spatial Measures 204

18.5 Applications of Topological Simplification and Filtering 205

18.6 Non-Simplicial Meshes and Computational Geometry 205

18.7 Time-Varying Three Dimensional Data 206

Bibliography **207**

Appendix A Code for Generating Piecewise Continuation Tables **214**

A.1 The Code 215

Appendix B Code for Piecewise Continuation **222**

B.1 The Code 222

List of Algorithms

5.1 The Continuation Method of Isosurface Extraction . . . 30
7.1 Computing the Contour Tree For A Simplicial Mesh . . . 46
7.2 Computing the Join Tree For a Simplicial Mesh . . . 47
7.3 Computing Contour Tree By Merging Join and Split Trees . . . 48
7.4 Reducing a Fully-Augmented Contour Tree . . . 49
7.5 Computing the Unaugmented Contour Tree Directly . . . 49
7.6 Computing Join Tree for Unaugmented Contour Tree . . . 50
7.7 Augmenting the Join Tree with Split Supernodes . . . 51
8.1 Computing Reduced Join Tree With Path Seeds . . . 69
8.2 Computing Contour Tree By Merging Join and Split Trees . . . 70
8.3 Computing the unaugmented contour tree With Path Seeds . . . 70
8.4 Algorithm to Extract Seed Edge From Path Seed . . . 71
9.1 Naïve Algorithm to Define a Level Set . . . 92
9.2 Algorithm Using an Interval Tree to Define a Level Set. . . . 93
9.3 Evolving the Contour from the Selection Root and Isovalue . . . 94
9.4 Object-Oriented Visualization from the Contour Tree . . . 94
10.1 Computing Sweep Functions at a Vertex μ . . . 111
10.2 Computing Local Spatial Measures from Leaves Inwards . . . 112
11.1 Collapsing a Vertex in A Contour Tree. . . . 128
11.2 Simplifying a Contour Tree Using Local Spatial Measures. . . . 129
11.3 Algorithm for Hierarchical Extraction of Path Seeds. . . . 131
11.4 Single Interactive Collapse of Contour Tree . . . 141
11.5 Single Interactive UnCollapse of Contour Tree . . . 142
12.1 Using a Lookup Table to Find Join Graph Neighbours . . . 157
12.2 Computing the Join Graph Lookup Table . . . 158
12.3 The Piecewise Continuation Method of Isosurface Extraction . . . 165
15.1 Comparing Vertices with Memory-Address Perturbation . . . 182
15.2 Naïve Removal of Perturbation from Contour Tree . . . 184
15.3 Improved Removal of Perturbation from Contour Tree . . . 184

List of Figures

2.1 Cutaway View of Multiple Contour Surfaces. 6

2.2 Using Flexible Isosurfaces to Explore Contours. 7

2.3 Example of a volcanic crater lake. 8

3.1 Visualizing Scalar Data Sets. 11

3.2 Visualizing Scalar Data Sets, With Contour Trees. 12

3.3 Interrelationships of Concepts. 13

3.4 Interrelationships of Chapters. 14

4.1 A Sample Mesh in Two Dimensions 20

4.2 Watersheds in the Sample Mesh 21

4.3 The Morse Complex of the Sample Mesh 22

4.4 Reeb Graph of a Torus 23

4.5 Contour Tree of the Sample Mesh 24

5.1 Marching Cube Cases, after Montani, Scateni & Scopigno [MSS94a] 28

6.1 Sample Data Set in 3 Dimensions, with Contour Tree. 33

6.2 Unaugmented Contour Tree and Fully Augmented Contour Tree For Figure 4.1 40

7.1 A Sample Mesh in Two Dimensions (Again) 52

7.2 Sweep Through Sample Mesh to Compute Join Tree. 53

7.3 Sweep Through Sample Mesh to Compute Split Tree. 55

7.4 Augmenting the Join Tree and Split Tree 56

7.5 Comparison of Contour Tree, Join Tree and Split Tree. 56

7.6 Merging the Join and Split Trees to Get the Contour Tree. 57

7.7 Merging To Get Fully Augmented Contour Tree. 58

8.1 A Triangulation Requiring a Minimal Seed Set of Size $\Theta(n)$. 61

8.2 A Triangulation Where Some Contours Intersect $\Omega(n)$ Seed Cells. 62

8.3 Sample Mesh, Showing Minimal Seed Set and a Contour. 63

8.4 Annotating The Contour Tree with Monotone Paths to Find Seed Cells. 64

8.5 Detecting Path Seeds During the Join Sweep. 73

8.6 Seed Paths in our Sample Mesh. 74

8.7 Storing Path Seeds From Figure 8.6 in the Contour Tree. 75

9.1 The Contour Spectrum - illustration courtesy C. Bajaj. 78

9.2 Example of Flexible Isosurface Editing to Isolate the Brain. 80

9.3 Heightening Visual Contrast with Colour & Isolation. 82

9.4 Components of the Flexible Isosurface Interface. 85

9.5 Simple Example of an Evolving Contour. 88

9.6 Continuous Contour Evolution: Upwards, then Downwards 89

9.7 Suppressing contours during contour evolution 90

9.8 Object-Oriented Visualization and Largest Contour Segmentation. 95

9.9 A Contour Tree with Intersecting Superarcs 96

9.10 An Unusable Contour Tree 97

10.1 Example of a volcanic crater lake. 100

10.2 Upstart and Downstart Regions in Volcano Example. 103

10.3 Sweep Function For Area in a Single Triangle. 105

10.4 Decomposing A Downward Sweep Function. 106

10.5 Computing the Function For Sweeping Down Superarc $81 - 50$ 113

11.1 Flowchart for Scalar Field Visualization. 119

11.2 Filtering a Topologically Small Detail. 120

11.3 Graph Operations on the Contour Tree 122

11.4 Flat Spot and Modified Slopes Induced by Leaf Pruning Contour Tree. 125

11.5 Why Not All Leaves Are Prunable. 126

11.6 Path Seeds in a Simplified Contour Tree. 131

11.7 Paths Induced by Tree Pruning, And the Collapse Hierarchy. 132

11.8 A Triangulation in Two Dimensions, with Contour Tree 133

11.9 Simplifying by Height with all Leaves Prunable and no Vertex Collapses Permitted. 135

11.10 Simplifying by Height with Leaf Prunes and Vertex Collapses, allowing Y-Prunes. 136

11.11 Simplifying by Height with Leaf Prunes and Vertex Collapses, without Y-Prunes. 137

11.12 Pruning the Contour Tree By Area. 138

11.13 Pruning the Contour Tree By Volume. 139

11.14 Interface to the Simplified Flexible Isosurface. 140

12.1 Finite State Machine To Compute Join Graph for Bilinear Interpolant. 159

12.2 Sample Cases for Tessellation in Two Dimensions. 161

12.3 Nesting of Some Tessellation Cases. 161

12.4 Finite State Machine To Compute Join Graph for Simple 2-D Tessellation Cases. 162

12.5 Areas Swept Through By Contours. 163

12.6 Labels For Piecewise Continuation, With Example 164

13.1 Possible Contour Trees for Bilinear Interpolant. 168

13.2 Subdividing a Bilinear Cell With Vertical and Horizontal Asymptotes. 169

13.3 Computing Area in a Bilinear Cell. 170

13.4 Finite State Machine for Join Graph of the Trilinear Interpolant. 171

13.5 A Trilinear Cell with Two Body Saddles. 172

13.6 Augmenting a Trilinear Cell with Two Body Saddles. 173

14.1 Marching Cubes Case 3 and its Converse Case 3C 177

14.2 Finite State Machine for Marching Cubes cases of Montani, Scateni, & Scopigno [MSS94a]. . 179

14.3 Computing Geometric Measures for Case 2 of Marching Cubes. 180

15.1 Effects of Perturbation on A Small Mesh 183

15.2 Removing Perturbation from a Contour Tree 185

15.3 Effects of Perturbation on Path Seeds. 187

16.1 Topological Simplification of UNC Head Data Set. 196

16.2 Conventional Isosurface of UNC Head Data Set Without Contour Tree Simplification. 196

16.3 Structures Visible in a Noisy MRI Scan of a Pregnant Rat. 197

16.4 Structures in the CT Head data set. 198

16.5 Removing Noise Topologically. 199

List of Tables

11.1 Effects of Size and Data Type on the Contour Tree. 118

11.2 Effects of Smoothing on the Contour Tree of the 3dhead Data Set. After two smoothing operations, the contour tree is reduced by approximately an order of magnitude, but is still much too large for visual display. 121

16.1 Characteristics and Sources of Test Data 191

16.2 Construction Times for the Contour Tree. 191

16.3 Sizes of Contour Trees after Perturbation Removal, with Times Required. 192

16.4 Time Required to Simplify Contour Trees After Removing Perturbation. 193

16.5 Comparison of Construction Times Using Simplicial Subdivision and Marching Cubes. 193

16.6 Sample Isosurface Sizes for Simplicial Subdivision and Marching Cubes. 194

16.7 Some Results for Path Seed Isosurface Extraction. 195

Acknowledgements

Acknowledgements are due to my supervisors, Dr. Jack Snoeyink (UNC-CH), Dr. David Kirkpatrick (UBC), and Dr. Michiel van de Panne (UBC), to the other members of my supervisory committee, Dr. Torsten Möller (SFU), Dr. Jim Little (UBC), and Dr. Will Evans (UBC), and to my departmental, university and external examiners.

Acknowledgements are also due to the Natural Science and Engineering Research Council of Canada (NSERC), the Institute for Robotics and Intelligent Systems (IRIS), Dr. Raymond Ng (UBC) and UBC for financial support.

Thanks are due to the large number of researchers who have shared their time and thoughts with me on various aspects of this research. These researchers include Dr. Janice Glasgow (Queen's), Alan Ableson (Queen's), Dr. Tamara Munzner (UBC), Dr. D.B. Karron (CASI), Dr. Deborah Silver (Rutgers), Dr. Chandrajit Bajaj (UT-Austin), Dr. Kenneth Joy (UC Davis), Dr. Bernd Hamann (UC Davis), Gunther Weber (UC Davis), Dr. Afra Zomorodian (Stanford), and Dr. Valerio Pascucci (LLNL). There are undoubtedly others who deserve thanks but have been omitted: apologies are also due for my poor memory.

Finally, many thanks are due to my parents, for supporting me in innumerable ways in returning to university.

Hamish Carr

The University of British Columbia
April 2004

To the Canadian university system

For giving me many more second chances than I had any right to expect.

Part I

Introduction

Chapter 1

Overview

In this thesis, I describe how to extend scientific visualization tools using a topological structure called the contour tree to generate and manipulate contour surfaces, and as an interface for exploring scalar data.

This thesis is divided into parts. Part I motivates the research by showing some examples of new visualizations in Chapter 2, and provide a roadmap for the rest of the thesis in Chapter 3.

Part II reviews background mathematics and previous work. In particular, Chapter 4 covers some background material, while Chapter 5 describes one of the principal techniques in scientific visualization: *isosurface* extraction. Chapter 5 concludes that *continuation*, the most efficient method of isosurface extraction, requires a systematic source of starting points, or *seeds*. Chapter 6 then describes a topological structure called the *contour tree*, which can be used as a source of seeds.

Thereafter, Part III introduces the principal research contributions of this thesis: new algorithms and interfaces for using isosurfaces for exploratory visualization of data. Chapter 7 describes existing algorithms for computing the contour tree, including the principal contribution of my M.Sc. thesis, the *sweep and merge* algorithm. Chapter 8 shows how to use the contour tree to generate *path seeds*, a new and efficient method for finding seeds for the continuation method of isosurface extraction. Chapter 9 then shows how to use path seeds for the *flexible isosurface* interface that allows, for the first time, efficient independent manipulation of individual isosurfaces. Chapter 11 then addresses the principal weakness of contour trees, topological noise, by showing how to use geometric information to *simplify* the contour tree and the data simultaneously. Chapter 10 follows this up by showing how to compute *local spatial measures*: geometric information computed locally for an isosurface-defined object.

Part III assumes that the data is in the mathematically easiest form possible: a simplicial mesh with distinct isovalues. Because such meshes are rarely used, Part IV covers further contributions of this thesis: how to extend Part III to arbitrary grids. Chapter 12 describes how to modify the algorithms in Part II and Part III for arbitrary grids using *join and split graphs*, then shows how to compute *finite state machines* to generate join and split graphs implicitly. Chapter 13 then applies Chapter 12 to the commonly-used case of trilinear interpolants, generalizing the work of Pascucci & Cole-McLaughlin [PCM02]. Chapter 14 further applies Chapter 12 to the equally commonly-used case of Marching Cubes. Finally, Chapter 15 discusses the use of symbolic perturbation to avoid degeneracies caused by identical isovalues in the input data.

In Part V, I then give some results in Chapter 16, present my conclusions in Chapter 17, and discuss

directions for future research in Chapter 18.

Finally, the appendices give some details of local spatial measures for triangles, tetrahedra, and marching cubes cases.

Chapter 2

Introduction

This thesis introduces a new tool for scientific visualization called the flexible isosurface, based on a topological structure called the contour tree. This tool extends existing isosurface techniques by treating individual contour surfaces as independent objects, permitting the user to select and manipulate individual contour surfaces, and by providing online topological simplification of the data, thus reducing noise and simplifying complex isosurface visualizations.

In this chapter, I attempt to show how such an approach improves upon the existing state of the art. Section 2.1 discusses scientific visualization. Section 2.2 shows how flexible isosurfaces can help resolve one of the principal problems with isosurface visualization: choosing what to look at. Section 2.3 then briefly discusses the contour tree: the topological abstraction of a scalar field that we have chosen to ease this task. Section 2.4 then discusses the conditions under which I expect this thesis to be useful to scientists, engineers and doctors.

2.1 Visualization

Many fields in science and engineering generate large data sets of two- or three-dimensional scalar data. These fields include analytic functions [AAW00, Wes95], cell biology [dLvLV+00, MHS+96], combustion studies [KRS01, RG00, RHC94], computational fluid dynamics [CM97, ECS00, MC00, SSZC94, She98, SCM99, SJ94, SW96, SW98, SH99, SH00], earth sciences [KRS01], electromagnetic fields [DCK+98], geophysics [KRS01], medical imaging [SJ94, SH00, TO91], meteorology [ADM92, LMC01, MCW93, SW98], molecular dynamics [LKS+98, SH99], oceanography [BSRF00] or vulcanology [RG94]. Although this list is incomplete, it gives some sense of the wide applicability of techniques for scientific visualization.

This data generally represents the spatial distribution of a scalar or vector property over some d-dimensional space. There is a practical difference between scalar and vector data in that techniques for visualizing vector data do not usually apply well to scalar data, or vice versa. This thesis does not address the question of visualizing vector data or of visualizing multiple scalar functions simultaneously: it is instead restricted to single scalar-valued functions.

Whatever our source of data, the principal task is to aid the researcher in understanding the data.

The goal is to develop tools and techniques that help an experienced scientist identify the important features of a data set. This takes advantage of two of the greatest strengths of humans: a sophisticated visual system, and expert knowledge of a particular domain. As a discipline, scientific visualization was described by McCormick, DeFanti & Brown [MDB87] as:

> Visualization is a method of computing. It transforms the symbolic into the geometric, enabling researchers to *observe* their simulations and computations. Visualization offers a method for seeing the unseen. It enriches the process of scientific discovery and fosters profound and unexpected insights. In many fields it is already revolutionizing the way scientists do science.
>
> Visualization embraces both image understanding and image synthesis. That is, visualization is a tool both for interpreting image data fed into a computer, and for generating images from complex multi-dimensional data sets. It studies those mechanisms in humans and computers which allow them in concert to perceive, use and communicate visual information. Visualization unifies the largely independent but convergent fields of:
>
> - Computer graphics
> - Image processing
> - Computer vision
> - Computer-aided design
> - Signal processing
> - User interface studies
>
> Richard Hamming observed many years ago that *"The purpose of [scientific] computing is insight, not numbers."* The goal of visualization is to leverage existing scientific methods by providing new scientific insight through visual methods.
>
> An estimated 50 percent of the brain's neurons are associated with vision. Visualization in scientific computing aims to put that neurological machinery to work.

McCormick, DeFanti & Brown then state the three most important themes in scientific visualization: the need to deal with too much data, the need to communicate visually, and the need to steer calculations (i.e. to interact with the data). We will see each of these themes recur throughout this thesis.

Visualization can also be classified into three types of problem, based on whether the user knows what question to ask and what the answer to the question is. If the user knows both question and answer already, then visualization is used to *present* the data to others to explain or illustrate a point. If the user knows the question, but not the answer, then visualization is used to *test* a proposition - e.g. does this dataset contain evidence that the test subject has a tumour? This generally involves applying a domain-specific description of the property to be tested. Finally, if the user knows neither the question nor the answer, at least explicitly, then visualization is used to *explore* the data in an attempt to determine what is unique or characteristic about the particular dataset. These categories are neither static nor necessarily disjoint. Over time, as a visualization technique is better understood, it is likely to move from exploration, to testing, to presentation.

Before determining which category this thesis falls into, the next two sections will give a brief overview of contour lines, isosurfaces and the contour tree.

2.2 Contour Lines and Isosurfaces

We assume that we wish to visualize a continuous scalar function $f : \mathcal{R} \rightarrow \mathbb{R}$ where $\mathcal{R} \subseteq \mathbb{R}^d$. One of the oldest tools for doing so in two dimensions is the contour map, which consists of a set of contour lines, each of which connects a set of points with the same function value. Although no single line contains all the information about the function, we can infer a lot of information from a set of contour lines. For example, the spacing between the lines reveals secondary information about the gradient of the function. By showing multiple contour lines, the contour map therefore gives us a good idea of the function as a whole.

The most familiar type of contour map is the topographic map, where contour lines are used to represent the land elevation. Elevation is a spatial measurement similar to the spatial measurement of the locations for which the function is defined. As a result, experience dealing with topographic maps and landforms can actually lead us astray when interpreting more general functions such as temperature, pressure, or magnetic variation. For example, computing the horizon of a temperature map is much less meaningful than computing the horizon of an elevation map, since the horizon is a visual property that results from the fact that elevation is a spatial measurement.

A more significant problem is that three dimensional contour maps reveal less information in three dimensions than in two. In two dimensions, no two contour lines of a continuous function intersect: thus, we can see more than one value of the function simultaneously. In contrast, a contour in three dimensions is actually a surface, and viewing multiple sets of surfaces is difficult at best since the surfaces occlude each other visually. Figure 2.1 shows an example of this, with the exterior surfaces cut away to enable us to look at the inner surfaces. Without cutting away the outermost surface, neither of the inner surfaces would be visible.

Figure 2.1: Cutaway View of Multiple Contour Surfaces. Outside Surface Occludes Inner Surfaces.

Because of this occlusion problem, a three-dimensional contour map effectively shows only a few contours in their entirety. Thus, we need to find a way to specify which surfaces to show, and more importantly, a way of giving the user information about the surfaces which are not currently being shown.

Existing methods of choosing contour surfaces are restricted to sets of contours that share a single

isovalue. One of the principal contribution of this thesis is to generalize this to *flexible isosurfaces*: arbitrary sets of contours that need not share a single isovalue, and an interface for manipulating these flexible isosurfaces.

As an example, consider Figure 2.2a, which shows a typical isosurface of an CT scan of a head. Note that the exterior surface hides all the details of surfaces inside the head. In this case, the isosurface contains multiple disjoint surfaces. In order to explore this data, it is desirable to be able to remove the exterior surface (b), choose one particular surface and suppress all other surfaces previously shown (c), then manipulate the remaining surface to discover structures inside it (d).

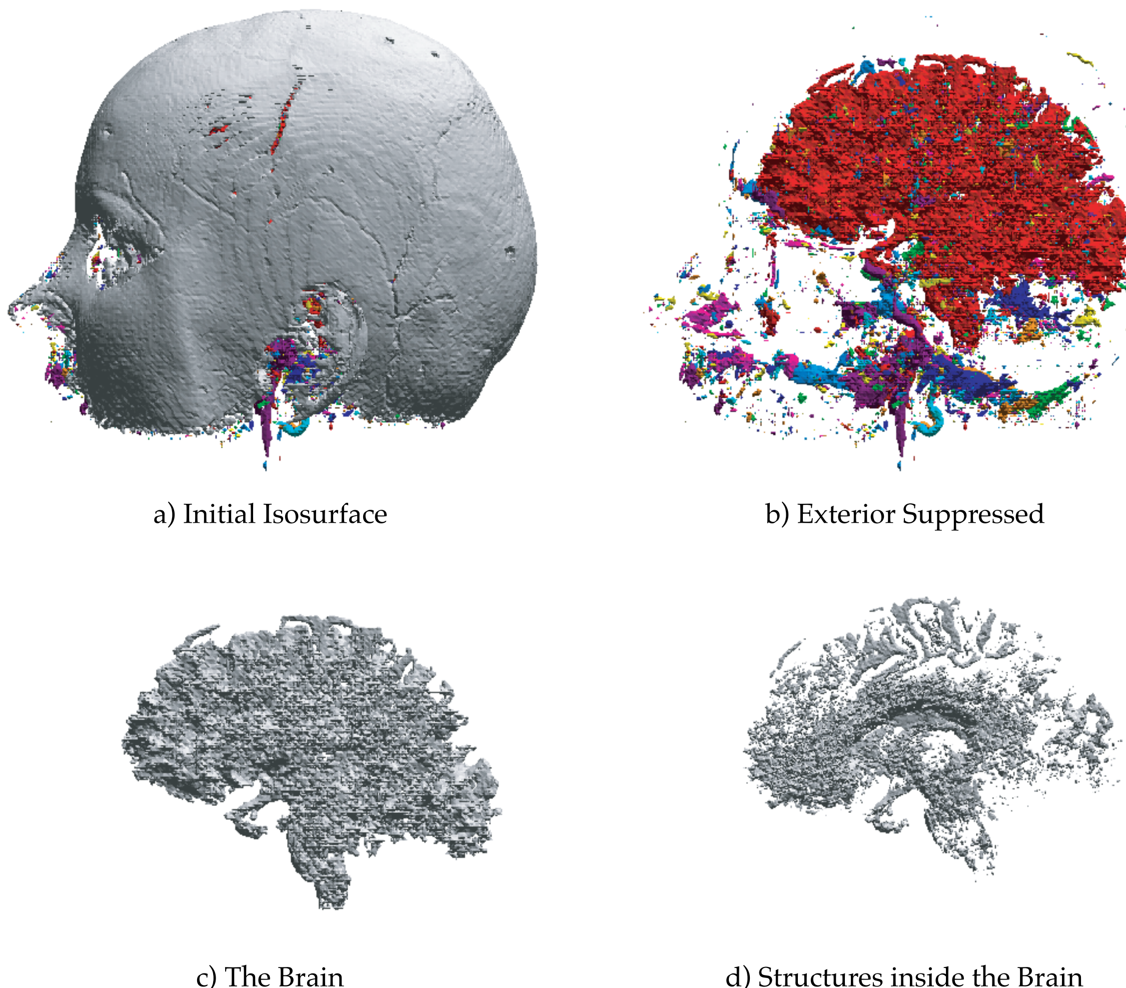
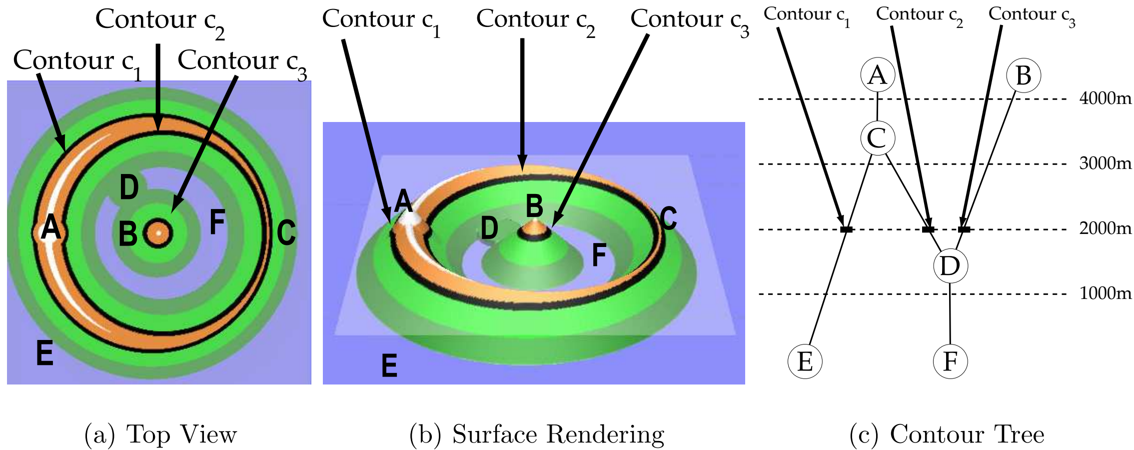


Figure 2.2: Using Flexible Isosurfaces to Explore Contours. The User Starts With a Conventional Isosurface (a), Then Removes an Unwanted Surface (b), Isolates the Principal Surface of Interest (c), and Alters its Isovalue (d)

To work with individual contours, we need a formalism that captures information about individual contours and their relationships. Fortunately, a topological structure called the *contour tree* does exactly this. This thesis will rely heavily on the contour tree, and on geometric and topological information about connectivity encapsulated in this structure.

The next section gives a brief overview of what the contour tree is, and how it relates to contour



(a) Top View (b) Surface Rendering (c) Contour Tree

Figure 2.3: Example of a volcanic crater lake.
The contour tree expresses the adjacency of the coloured regions in the top view. Region E is adjacent to a green region, which is adjacent to contour c_1, which is adjacent to C. C is adjacent to two disjoint regions: A and contour c_2, and so on. The contour tree shows these relationships: here, the isovalue of each regions is used as the vertical dimension in (c).

surfaces and the flexible isosurface.

2.3 The Contour Tree

Chapters 6 through 15 will discuss the contour tree in great detail. For now, it suffices to give a brief description. Figure 2.3 shows a volcanic crater lake as a topographic map and as a surface. In this example, the contour tree expresses which coloured regions are adjacent to which other coloured regions.

If we construct a graph with one node for each contour, we can show these relationships diagrammatically, by connected nodes representing adjacent contours: the result is shown in Figure 2.3(c). This definition of the contour tree, as the dual graph to a set of regions separated by contours, breaks down at the boundaries of the data set. We will return to this question in Chapter 10: the formal definition we provide in Chapter 6 does not, however, fail at the boundaries of the data set.

The principal contributions in this thesis are to compute the contour tree efficiently for simplicial (Chapter 7) and non-simplicial meshes (Chapter 12 to Chapter 15), and to exploit the relationship between contour tree and contours for isosurface extraction (Chapter 5), for visual interaction with isosurfaces (Chapter 9), for topological simplification (Chapter 11), and for computation of geometric properties of isosurfaces (Chapter 10).

2.4 Application of this Thesis

Since this thesis is primarily concerned with extending existing techniques to provide new methods for interacting with data, it falls into the third category of visualization: *exploration*. As such, it is wise to

describe the conditions under which it is most likely to be useful.

Data : I assume that the input data is such that contours (i.e. isosurfaces) are meaningful representations of important features in the data. In the biomedical context, for example, contours would ideally represent anatomical organs or boundaries in the data. As such, these techniques are likely more applicable to anatomical data rather than functional data. In the former, the sampled values of the function depend principally on the nature of the tissue, but in the latter, the sampled values depend on the function or purpose of tissues. An example of this is where a radioactive marker is used to indicate tissues with a high uptake of some chemical. For functional data such as this, and especially for functional data which varies over time, isosurfaces may not give a meaningful view of the data.

Exploration : I also assume that the user is interested in exploring the data, but does not necessarily have an application-specific test available that characterizes whatever is important in the data. In future, this may change, using geometric information stored in the contour tree to support application-specific tests, as indicated in Chapter 18. This, however, fits the general description given above of techniques migrating from exploration to testing as they are better understood.

In practice, different parts of this thesis are likely to be useful to different users. Topologically naïve users may only wish to use the flexible isosurface visually, as described in Section 2.2, without even knowing of the contour tree's existence. More sophisticated users may take advantage of simplified contour trees for direct manipulation, or may choose to use local spatial measures to define new tests of feature importance. All this, however, lies in the future. This thesis will concentrate on techniques that are now possible, and may become useful in the future.

Chapter 3

Roadmap

This thesis includes a variety of concepts, many of which are tightly interlocked, either with each other or with other concepts in scientific visualization. In this chapter, I attempt to lay out some of the relationships between concepts, and between parts and chapters of this thesis.

3.1 Contour Trees and Visualization

This thesis deals principally with the application to visualization of the topological abstraction known as the contour tree. This structure is principally applicable to visualizing scalar data, so it is germane to ask how scalar data is typically visualized. Figure 3.1 shows that scalar data is often visualized with isosurfaces or volume rendering. Moreover, this diagram also highlights the fact that geometric techniques such as mesh construction are largely independent of signal processing techniques such as reconstruction, although some of the geometric techniques can be expressed in terms of signal processing ideas.

Since the contour tree depends on contours and isosurfaces, which are inherently geometric in nature, it is unsurprising that the contour tree and most other topological techniques are more closely related to geometric techniques than signal processing. If we add this information to the figure, we get Figure 3.2, which clearly reflects the close ties between contour trees and isosurfaces, and the looser ties between the contour tree and volume rendering.

3.2 Contour Trees and Related Ideas

Once we appreciate where the contour tree fits into the broader picture of scientific visualization, we still need to understand the relationship between various aspects of this thesis and related topics. In Figure 3.3, I have attempted to lay out some of the relationships: the coloured patches indicate work that belongs to my M.Sc. thesis, this thesis, other work that I have done, and work that I hope to do in the future. In some cases, the coloured patches do not completely include an idea: this indicates that other researchers have performed the same or parallel work. For example, although this thesis generalizes contour tree algorithms to arbitrary meshes and interpolants, Pascucci & Cole-McLaughlin [PCM02] have already extended contour

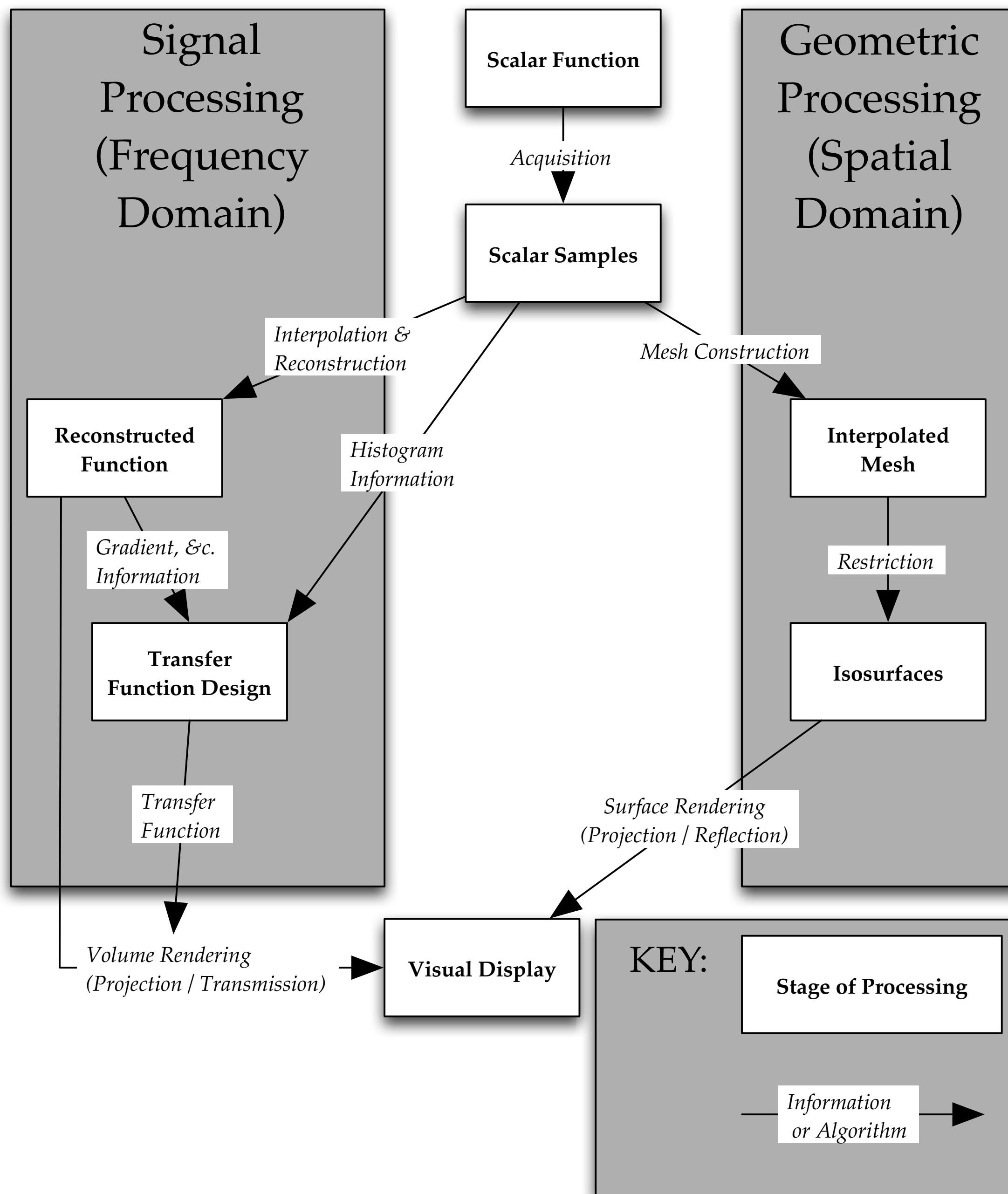


Figure 3.1: Visualizing Scalar Data Sets. Scalar data can be visualized by using signal processing techniques which operate in the frequency domain, or by using geometric techniques which operate in the spatial domain.

tree construction algorithms to cubic meshes using the trilinear interpolant.

As Figure 3.3 indicates, my M.Sc. thesis concentrated on an efficient algorithm to compute the contour tree, dealing with perturbation and path seeds in passing, and discussing simplicial subdivisions as a necessary adjunct to working with simplicial meshes.

The core of this thesis is the use of the contour tree to manipulate isosurfaces using the flexible isosurface interface, based on the ability to isolate individual contours by means of path seeds. To make

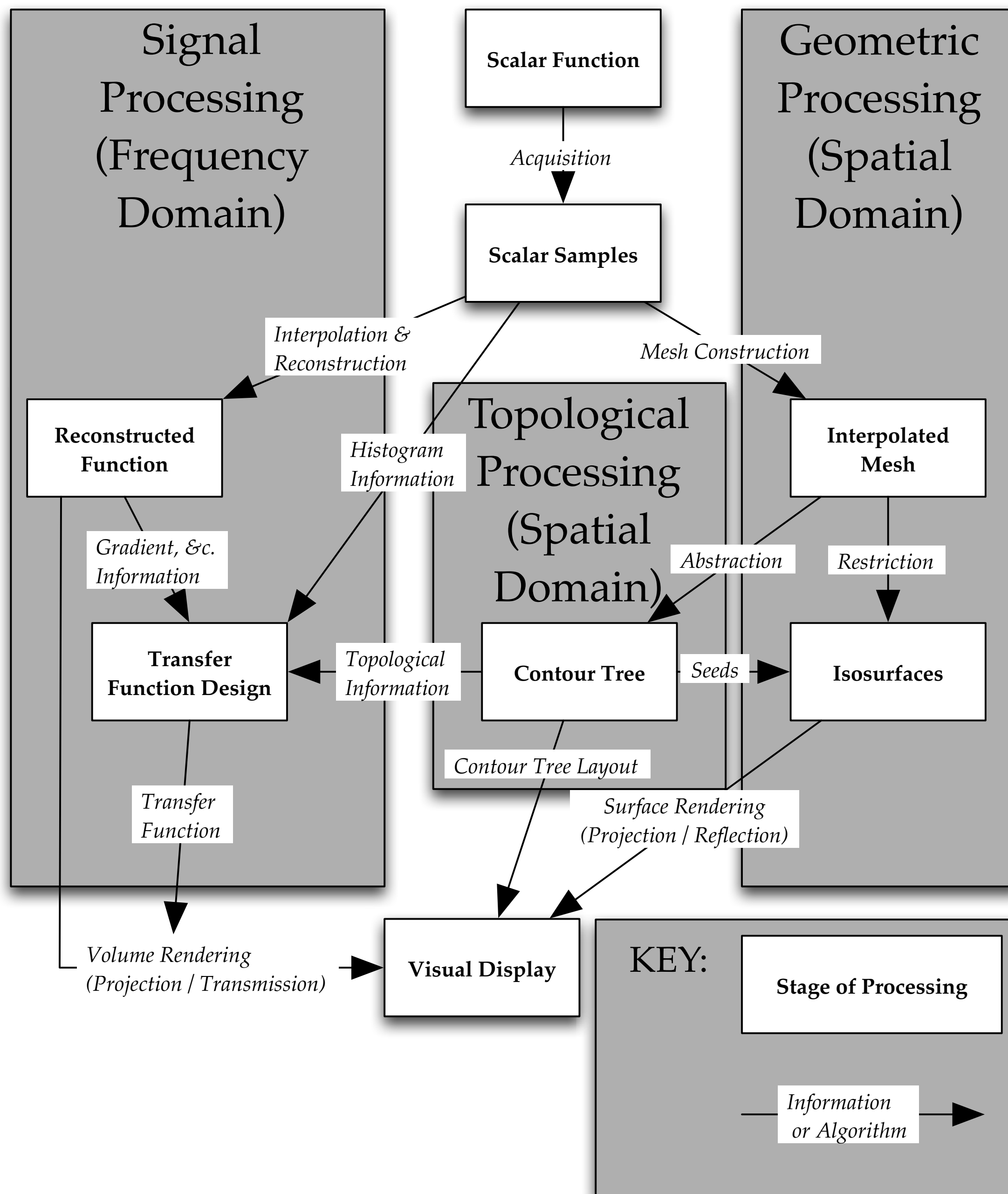


Figure 3.2: Visualizing Scalar Data Sets, With Contour Trees. Topological techniques such as those involving the contour tree depend heavily on geometric processing in the spatial domain, but can be used to inform decisions in transfer function design as well.

this work practical, it was also necessary to deal with contour tree simplification, local spatial measures, generalized contour tree algorithms, and marching-cube based algorithms.

Lastly, areas where I believe there is potential for future research are also marked: these areas are, of course, not exhaustive.

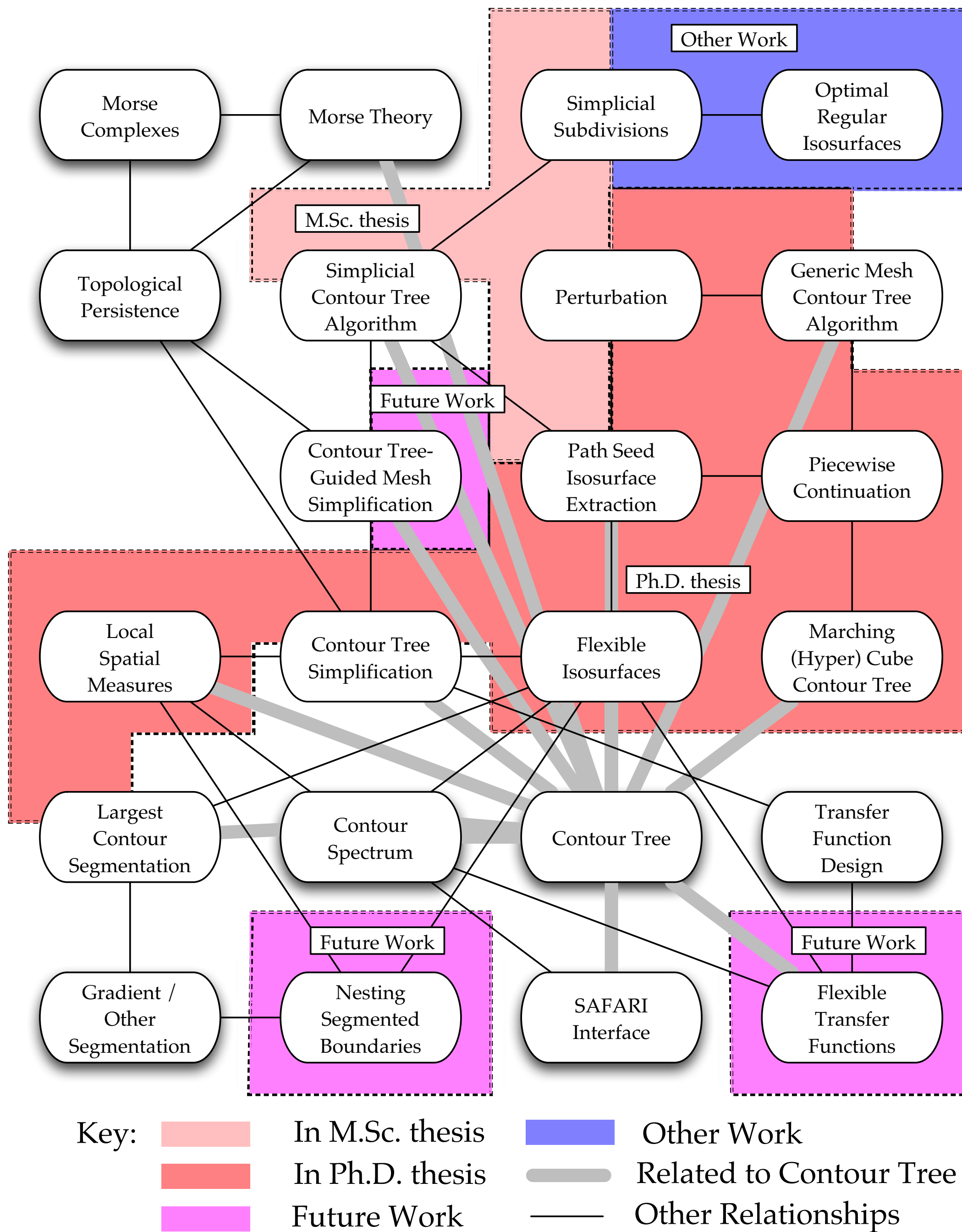


Figure 3.3: Interrelationships of Concepts. This diagram attempts to give some sense of the relationship between various ideas in this thesis, and between these ideas and other researchers' work.

3.3 Chapter Interrelationships

Having illustrated the relationship of the contour tree to scalar visualization in Section 3.1, and to related ideas in Section 3.2, I now turn to the structure of this thesis. In Figure 3.4, I show which ideas belong to which Part and which chapter of this thesis.

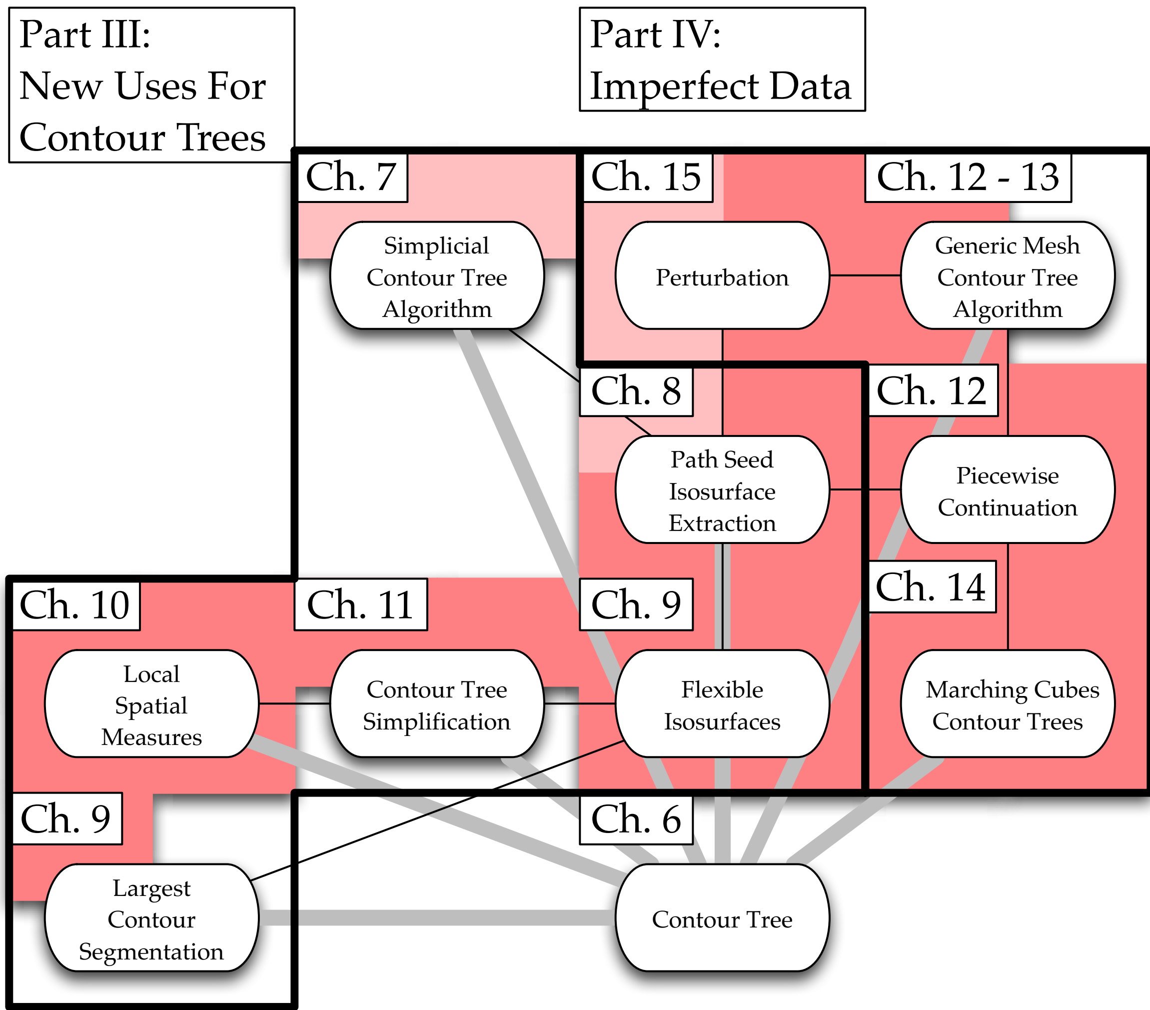


Figure 3.4: Interrelationships of Chapters. This figure shows where some of the topics in Figure 3.3 are to be found in this thesis.

Although these ideas could be discussed in almost any order, it is convenient to work with mathematically ideal data first, then extend to real data later. Thus, Part III (New Uses for Contour Trees) makes several assumptions about the input data which are rarely realized in practice. Part IV (Imperfect Data) then considers how to deal with data that does not live up to these assumptions.

In general, each chapter will depend on most of the previous chapters, and will modify the material in those chapters as needed.

Part II

Background

This Part of the thesis covers the necessary background for discussing the contour tree and isosurface visualization in more detail.

Chapter 4 will review some mathematics which will be crucial to this thesis, ranging from sampling theory, through piecewise interpolation of functions, to topological abstractions of scalar fields.

Because of the fundamental nature of isosurfaces to this work, Chapter 5 will then review in more detail how isosurfaces are constructed. Chapter 6 gives formal and informal definitions of the contour tree, in preparation for Part III, which discusses how to compute the contour tree, and how to use it for exploratory isosurface visualization and topological simplification.

Chapter 4

Mathematical Preliminaries

As with most research, work on the contour tree is heavily dependent on previous research. This chapter aims to cover background ideas that are required to understand the subsequent material, without going into details of specific research papers. The topics covered are:

4.1 Visualization and Data Reduction: a framework for thinking about visualization.

4.2 Sampled Data: generating continuous functions from limited information.

4.3 Manifolds and Intensity Maps: two different ways of thinking about scalar functions.

4.4 Level Sets and Contours: a brief definition

4.5 Topological Abstractions and Visualization

4.1 Visualization and Data Reduction

One of the dominant themes in scientific visualization is coping with huge amounts of data, both algorithmically and perceptually. Algorithmically, we need efficient algorithms so that a computer can process the data at a reasonable rate. Perceptually, we must realize that humans cannot process as much information at a time as we might like: we simply cannot comprehend one million measurements. To visualize these huge amounts of data, we must reduce an unwieldy mass of data to something that a human can deal with.

There are four principal methods used to reduce scalar data: projection, restriction, abstraction, and simplification.

Projection reduces data by summarizing or reducing along straight lines. Projection techniques include perspective projection of surfaces, and volume rendering [DCH88, Gar90, Kaj86, Lev88, Max95, Sab88, UK88], which computes light passing through a volume in straight lines, often producing images similar to the familiar X-ray.

Restriction reduces data by retaining only the data in a lower-dimensional subset of the domain,

discarding all other data. Restriction techniques include cross-sections, time-slices, and contours: we will go into more detail on contours in particular in Section 4.4 and Chapter 8 in particular.

Abstraction usually reduces data by computing a topological description of the function. Abstraction techniques include watersheds, Morse complexes, Reeb graphs and contour trees. These are discussed in more detail in Section 4.5, below.

Simplification reduces data by removing unimportant features from the function, sometimes replacing several unimportant features with one important feature. Simplification techniques include filtering, mesh simplification, surface simplification, topology simplification, and object suppression. These techniques will be dealt with in Chapter 11.

4.2 Sampling and Reconstruction

Scientific visualization generally assumes the existence of a continuous function defined everywhere over some region of interest. Much of the time, scientific visualization also assumes that this function is continuously differentiable.

In reality, the data is generally known only through *point samples*: individual points, usually in a regular pattern, at which the function is known. The pattern in which the point samples are laid out is sometimes referred to as a *grid*. Except at the point samples, the value of the function is unknown. It is assumed, however, that the property being measured, such as heat, temperature, or pressure, is itself a continuous function. Thus, the first step in visualization involves *reconstructing* the underlying function.

In reconstructing the underlying function, the value of the function at each sample is distributed over the immediate neighbourhood by another function called a *kernel*. The field of *sampling theory* [GW02] studies the types of kernels to be used for reconstruction, using kernels such as $sinc(x) = \sin x/x$. These kernels are generally analytically intractable - i.e. there is in general no closed form solution for the roots of the reconstructed function f. For visualization techniques that do not require a closed form solution, kernels with good sampling properties are generally used.

Geometric and topological techniques, however, generally require closed form solutions in order to find critical points and to determine the relationship between them. As a result, scientific visualization techniques that exploit geometry and topology commonly reconstruct the function f using geometric *meshes*.

A *mesh* is a subdivision of the region of interest into geometric primitives called *cells*. In general, the vertices that define the cells are given by the grid - i.e. the mesh is constructed using the point samples as vertices. Common cells include: triangular and rectangular cells in two dimensions, tetrahedral and box cells in three dimensions, and simplicial and hyperbox cells in arbitrary dimensions.

The function f is then reconstructed on the mesh by using an analytically tractable *interpolant*. Triangular, tetrahedral, and other simplicial cells commonly use the *barycentric* interpolant, which has the useful property that the function is guaranteed to be linear along any line through the cell. Rectangular, box and hyperbox cells commonly use multilinear interpolants, which are linear along axis-parallel lines, but not necessarily other lines through the cell. Spline interpolants are occasionally used, but are analytically intractable in higher dimensions. For example, in three dimensions, finding critical points using a trilinear interpolant requires solving a cubic equation. A tricubic interpolant, however, requires solving a ninth-order

polynomial, for which no closed-form solution is known.

This distinction, between visualization techniques that use sampling theory and high-quality kernels, and techniques that use geometric or topological methods and geometric meshes, was reflected in Figure 3.1.

Since this thesis rests on geometric and topological properties, I will assume that a geometric mesh is used for reconstruction. By this, I mean that the region of interest is subdivided into polygonal or polyhedral cells, over each of which an analytically tractable interpolant is defined. More specifically, for Part III, I shall assume that the function is reconstructed using barycentric interpolation over a simplicial mesh.

4.3 Manifolds and Intensity Maps

Once a scalar function $f : \mathcal{R} \rightarrow \mathbb{R}$ (where $\mathcal{R} \subseteq \mathbb{R}^d$) has been reconstructed, it is often useful to think of f as a *manifold*: a mathematical generalization of a surface. Formally, a *topological n-manifold*, or n-manifold for short, is a topological space that is everywhere locally homeomorphic to $\mathbb{R}^n$.

For two dimensional scalar data, f can be written as the set of points of the form $\{(x, y, f(x, y)) : (x, y) \in dom(f)\}$. This set of points forms a special surface in three-dimensional space: one for which no two points share the same x, y coordinates. This surface is also a two-manifold embedded in a three-dimensional space. Similarly, a function f that varies over three spatial dimensions forms a three-manifold embedded in a four-dimensional space.

Where f measures a property such as light intensity or radiated heat, it may be more appropriate to treat the function as an *intensity map*, emphasizing the difference between the spatial dimensions and the function value.

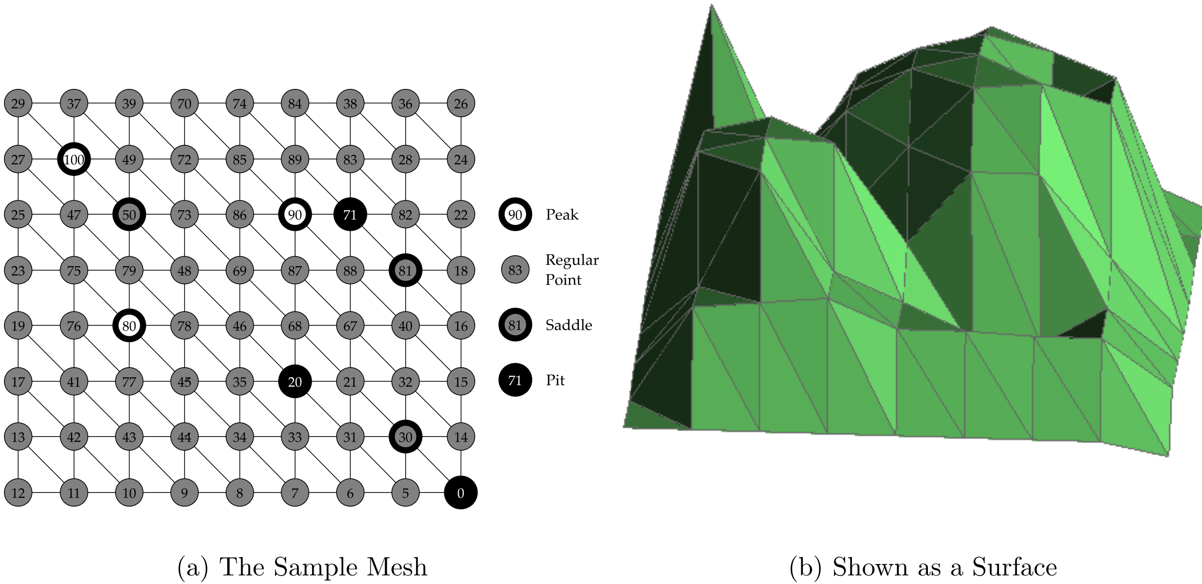
Both approaches can be used simultaneously, as for example in topographic maps, where altitude is frequently encoded with colours as well as contour lines. Here, a spatial dimension is treated as a function value, and converted to intensity.

4.4 Level Sets and Contours

Chapter 2 already introduced the notion of the contour, or more precisely, of the *isocontour line*: a line of points with a common value of f, called the *isovalue*. Formally speaking, a contour derives from the restriction of the function f to an isovalue h. This restriction is called a *level set*: $f^{-1}(h) = \{\vec{x} : f(\vec{x}) = h\}$. This level set need not be connected: each connected component of the level set is called an *isocontour* or *contour*. Moreover, the contours need not be closed curves or surfaces. For some types of data, we can often make the assumption that all contours are in fact closed, but nothing in this thesis requires this assumption.

For two-dimensional data, each isocontour is a linear feature, called an *isoline*. The most familiar use of isolines is on topographic maps, where the function f represents land elevation.

For three-dimensional data, each level set is a set of surfaces in three dimensions: each isocontour is called an *isosurface*. The concept of isosurfaces as an extension of isolines arose in geology (isograds) [Til24] and oceanography (isohalines) as early as the 1920's. Isosurfaces constructed using stacked glass plates



(a) The Sample Mesh (b) Shown as a Surface

Figure 4.1: A Sample Mesh in Two Dimensions

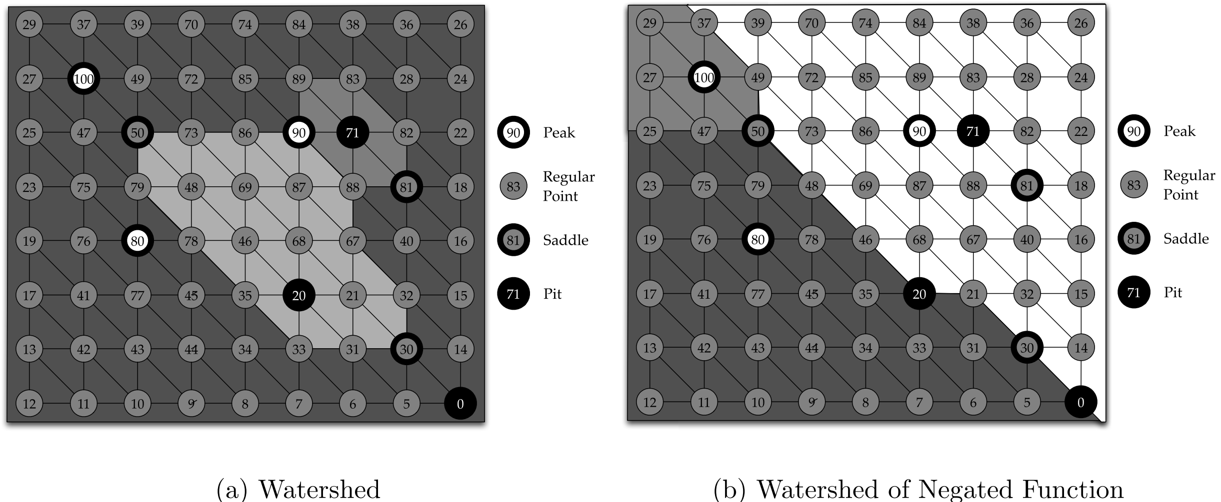
showing contours were introduced for X-ray crystallography in the 1950's. Algorithms for constructing isosurfaces are discussed in Chapter 8.

4.5 Topological Abstractions

Section 4.1 observed that topological abstractions have been used for visualization. There are several principal abstractions that have been used to date:

1. Watersheds (which can also be viewed as geometric abstractions)
2. Morse complexes
3. Reeb graphs
4. Contour trees

To illustrate, let us show each abstraction for the sample mesh in two dimensions shown in Figure 4.1. We will use this as a running example for the balance of this thesis. Figure 4.1(a) shows the triangular mesh itself, while Figure 4.1(b) shows this mesh as a shaded surface in three dimensions. Several things can be seen in this mesh. Firstly, it has three peaks (100, 90, and 80) of varying heights and base areas. Secondly, it has three pits (71, 20, and 0), one of which is a declivity in the peak centred at 90, one of which (20) is a catch-basin near the base of the main peaks, and the third of which is the global minimum. Lastly, it has three saddle points. We will define saddle points formally in Chapter 6: here, we simply characterize them as points where there are multiple distinct directions of ascent or descent. In this case, there is a triple saddle at 50 (also called a monkey saddle) where the three peaks meet, and simple saddles at 81 and 30 which help define the basins with 71 and 20 as local minima.



(a) Watershed

(b) Watershed of Negated Function

Figure 4.2: Watersheds in the Sample Mesh

4.5.1 Watersheds

The simplest topological abstraction applied to scalar data is the *watershed*: a region that drains to a single point. As the name implies, the idea comes from hydrography: mapping the flow of water over a landscape. Figure 4.2(a) shows the watersheds in our sample mesh, using a geometric approach for computing watersheds. Note how each pit (or local minimum) has its own drainage region.

Watersheds are also used for image processing, albeit in a slightly different form. Commonly, the intensity peaks of an image are considered to be important, and the task is to segment the image into regions based on the peaks. This is usually performed by negating the function, then computing the watershed. The effect of this is to define regions which drain *from* a common point. Figure 4.2(b) shows the watershed of the negated function for our sample mesh. Unsurprisingly, the image is divided into regions draining from each of the peaks.

4.5.2 Morse Complexes

The Morse complex comes from the fields of Morse Theory [Mil63] and differential geometry. Assuming that the function f is everywhere differentiable, the gradient at each point is calculated. From each point, a line extends upwards and downwards following the gradient until a local maximum and local minimum are reached. Thus, each point is assigned to a *gradient line*, or path of steepest descent. All gradient lines sharing a common local maximum and local minimum are grouped to form *Morse regions*. The *Morse complex* is simply the set of these regions, usually shown by drawing the boundaries.

Note that, since each Morse region shares a common peak (or source), it must be a subregion of an antiwatershed. Similarly, since each Morse region shares a common pit (or sink), it must be a subregion of a watershed. It is not difficult to see, therefore, that the Morse complex is simply the set of regions defined by intersecting the watersheds and antiwatersheds.

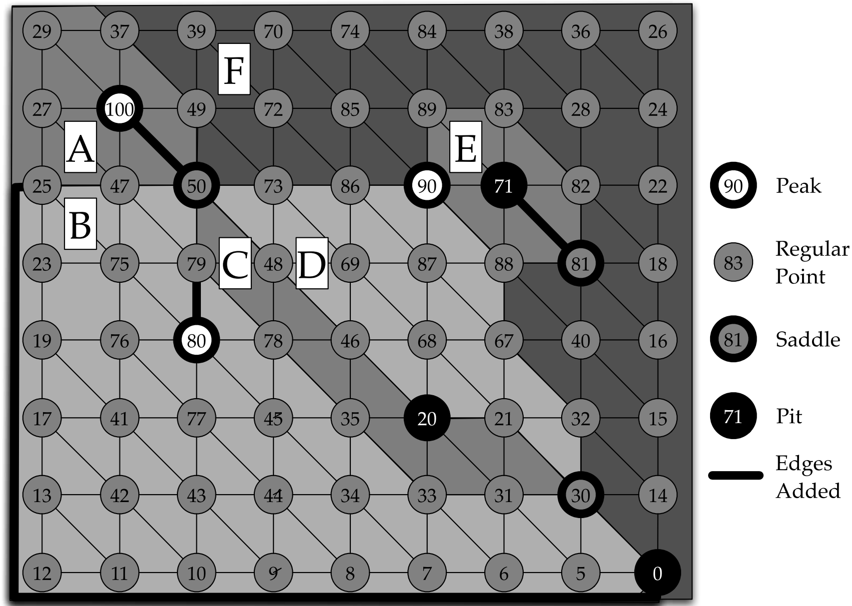


Figure 4.3: The Morse Complex of the Sample Mesh

For piecewise-linear functions such as the one in Figure 4.1(a), the function f is in fact not everywhere differentiable: it is not differentiable along the boundaries between cells of the mesh. This has been dealt with at length by Edelsbrunner, Harer & Zomorodian [EHZ01] and by Bremer et al. [BEHP03], and is not dealt with here. For the purposes of this thesis, it suffices to show an approximation of the Morse complex in Figure 4.3, computed by intersecting the watersheds and antiwatersheds from Figure 4.2. The regions thus obtained were then modified by adding the edges marked to preserve an important property of Morse complexes: that the boundary of each Morse region passes in sequence through a peak, a saddle, a pit, and a saddle which may be the same as the first saddle. Thus, for example, the boundary of region E is $90 - 89 - 83 - 82 - 81 - 71 - 81 - 88 - 90$, and the boundary of region C is $80 - 79 - 50 - 48 - 46 - 20 - 21 - 30 - 31 - 33 - 35 - 78 - 79 - 80$.

4.5.3 Reeb Graphs

In comparison to Morse complexes, both Reeb graphs and contour trees represent, not the line of steepest descent, but the line of least descent (in fact, no descent), i.e. contours or isolines. If you follow an isoline, the value of f never increases or decreases. Both Reeb graphs and contour trees track the changes to a contour as a single parameter is varied. However, the Reeb graph is more general than the contour tree, and is computed for manifolds more complex than a simple surface defined by a function over $\mathbb{R}^d$. We thus illustrate it with a more general manifold - a torus.

The *Reeb graph* [Ree46] expresses the evolution of contours in cross-sections of a manifold. For example, Figure 4.4 shows a set of contours representing horizontal cross-sections through a torus resting on its edge. These contours evolve gradually from a single point to an ellipse, then to a figure-eight, to two ellipses, back to a figure-eight, to a single ellipse, to a single point once more, then disappear. Diagrammatically, the Reeb graph on the right shows how many contours there are, and which ones evolve into which other ones: these contours may break apart, then merge once more, as shown by this example.

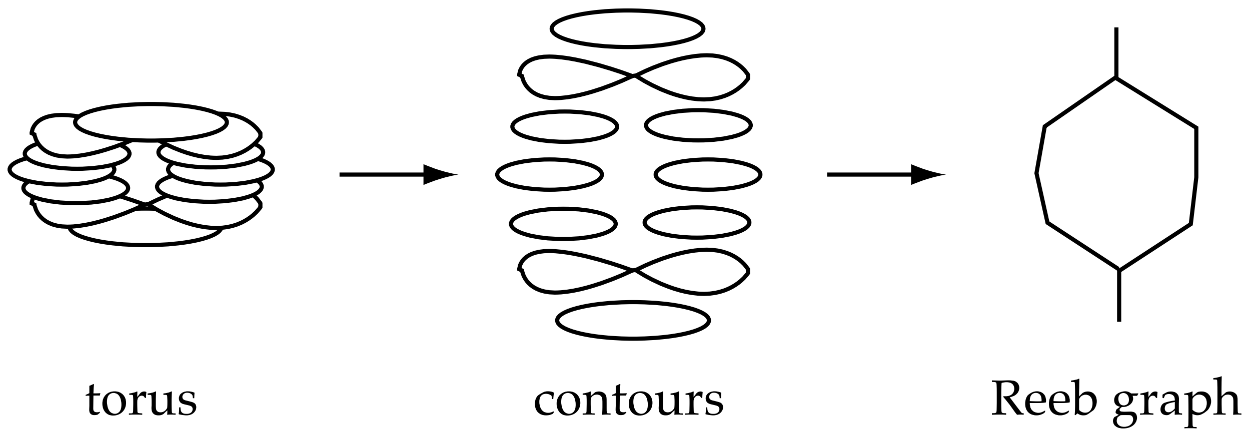
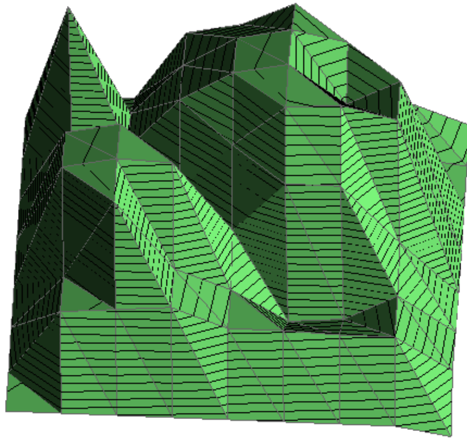


Figure 4.4: Reeb Graph of a Torus

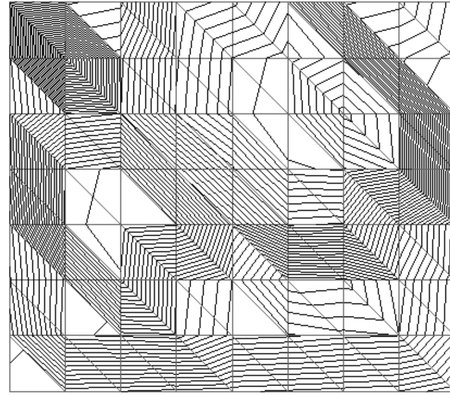
4.5.4 Contour Trees

The Reeb graph is more general than necessary for scalar fields, as it is designed to represent the evolution of any surface with respect to any parameter. In contrast, when working with scalar fields, we are generally only interested in the function's range as the parameter. Furthermore, since we are working with a function $f : \mathbb{R}^d \to \mathbb{R}$, we are interested in a special case of the Reeb graph: the *contour tree*.

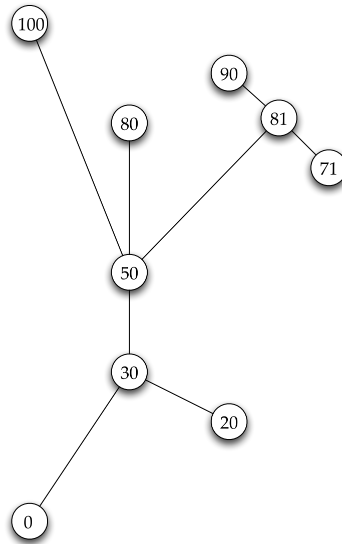
In Figure 4.5, we show the contour isolines for the function represented by the sample mesh, both as lines on the surface, and in an overhead view. As we saw in Chapter 2, the contour tree can be viewed as the dual graph of the regions bounded by the contours and by the boundary of $\mathcal{R}$, the domain of the function f. We show the contour tree for this sample mesh in Figure 4.5. Since this structure is at the core of this thesis, Chapter 6 will give a more formal definition.



(a) Contours on Surface



(b) Contours on Plane



(c) Contour Tree

Figure 4.5: Contour Tree of the Sample Mesh

Chapter 5

Isosurface Extraction

Techniques for extracting and manipulating contours are essentially the same in any number of dimensions. Since the principal focus of this thesis is on isosurfaces, this chapter considers isosurface techniques only, although the techniques reviewed here have their equivalents for two-dimensional data and for higher-dimensional data. We therefore assume for the purposes of this chapter that f is defined over a compact region $\mathcal{R}$ in $\mathbb{R}^3$.

Isosurface extraction grew out of two apparently disparate problems: boundary detection in acquired data and implicit surfaces in mathematically-defined functions. Section 5.1 looks briefly at these two problems, and some early approaches to isosurface extraction. Section 5.2 and Section 5.3 then introduce the two principal modern isosurface extraction algorithms: *marching cubes* and *continuation*.

5.1 Early Approaches: Boundary Detection and Implicit Surfaces

Work on computerized isosurface extraction started in the 1970's, and originated in two research problems - boundary detection, and implicit surfaces. In either case, given the function f, the goal is to define a surface that represents some object or feature of importance in f.

Of these two problems, the boundary detection problem principally arose when data was experimentally acquired, as for example in a medical data set. Then, given the function f, the desired surface was the boundary of some important feature: typically an anatomical organ. This boundary was defined by choosing an isovalue and extracting a geometric surface which partitioned the sample points into two sets: those above the isovalue, and those below the isovalue.

In contrast, the implicit surface problem was driven principally by the desire to study analytically defined functions, by drawing a geometric approximation of the set $\{x : f(x) = h\}$ for some value h (usually 0).

Ultimately, it was realized that these two problems were the same, and that general techniques for extracting isosurfaces could be applied to either.

Boundary Detection Early approaches to boundary detetion took two forms - stacking individual cubes inside the boundary, or connecting stacks of two-dimensional contours.

Herman & Lun [HL79] took the first approach, and rendered a small cube called a *cuberille*, centred on each sample with a value greater than the isovalue. This is similar to building a model using transparent cubes for the samples below the isovalue, and opaque cubes for the samples above the isovalue. Once isosurfaces (see below) were introduced, it was clear that the surface was equivalent to the isosurface generated using the nearest neighbour interpolant function. Artzy [Art79] improved the cuberille approach by rendering only the front half of each cuberille, thus reducing the work by 50%. Cuberilles that were not visible because they were surrounded by others were, however, still rendered, resulting in unnecessary work being performed. This was addressed by Artzy, Frieder & Herman [AFH81], who described an algorithm to follow the surface of the cuberilles, ignoring internal faces.

The other approach to boundary detection connected stacks of two-dimensional contours, as in the early work with stacked glass sheets in X-ray crystallography. This approach was taken by Fuchs, Kedem & Uselton [FKU77], who detected boundaries as contours in 2-D in each slice of the data, then connected the resulting contours. This approach was later extended by Shinagawa & Kunii [SK91] to use Reeb graphs to describe the relationship between contours in adjacent slices. Again, it soon became apparent that this was equivalent to generating isosurfaces.

Implicit Surfaces In contrast to boundary detection, work on implicit surfaces attempted to work directly with the function f. Implicit surfaces are surfaces defined by $f(\vec{x}) = 0$ for some function f, and are rendered using surface modelling techniques: for example, Blinn [Bli82] used a ray-tracing algorithm to render implicit surfaces.

Wyvill, McPheeters, & Wyvill [WMW86a, WMW86b] introduced the term *isosurface* in their work on *soft object* animation. A soft object is an object whose surface is defined to be the zero-set of the sum of several Gaussian distributions. By moving the centres of the distributions, the overall shape of a soft object can be manipulated to produce a smooth animation [WMW86a]. These authors sampled the sum of the distributions on a regular cubic mesh, then constructed a polygonal surface in each cube [WMW86b]. Bloomenthal [Blo88] extended the polygonization method by using an octree to refine the surface adaptively, and introduced the term *implicit surfaces*. Ning & Bloomenthal [NB93] summarized various techniques for implicit surfaces, as did Wyvill [Wyv94], but these techniques are now best viewed as a special case of isosurfaces - i.e. isosurfaces with an isovalue of 0.

Work on boundary detection and implicit surfaces converged with Wyvill, McPheeters & Wyvill's work on soft objects [WMW86b], and Lorenson & Cline's *marching cubes* algorithm [LC87]. Although Wyvill, McPheeters & Wyvill were interested in analytical functions that were well-defined everywhere, and Lorenson & Cline were interested in experimentally sampled functions, they developed similar approaches. Under either of these approaches, the region of interest is subdivided into cubical cells, and the intersection of the isosurface with each cell is approximated with polygons.

5.2 Marching Cubes

Lorenson & Cline's algorithm for isosurface extraction is called *marching cubes* [LC87]. This extracts isosurfaces by extracting surfaces separately in every cell in a cubic mesh. The algorithm iterates through all cells in the volume, hence the term *marching cubes*. In each cell, each vertex is classified as "above" or "below" the surface. For each edge of the cube with one vertex above the isosurface and one below, a point was generated by linear interpolation along the edge. These *edge points* were then used to construct one or more polygonal surface separating the vertices above and below the surface.

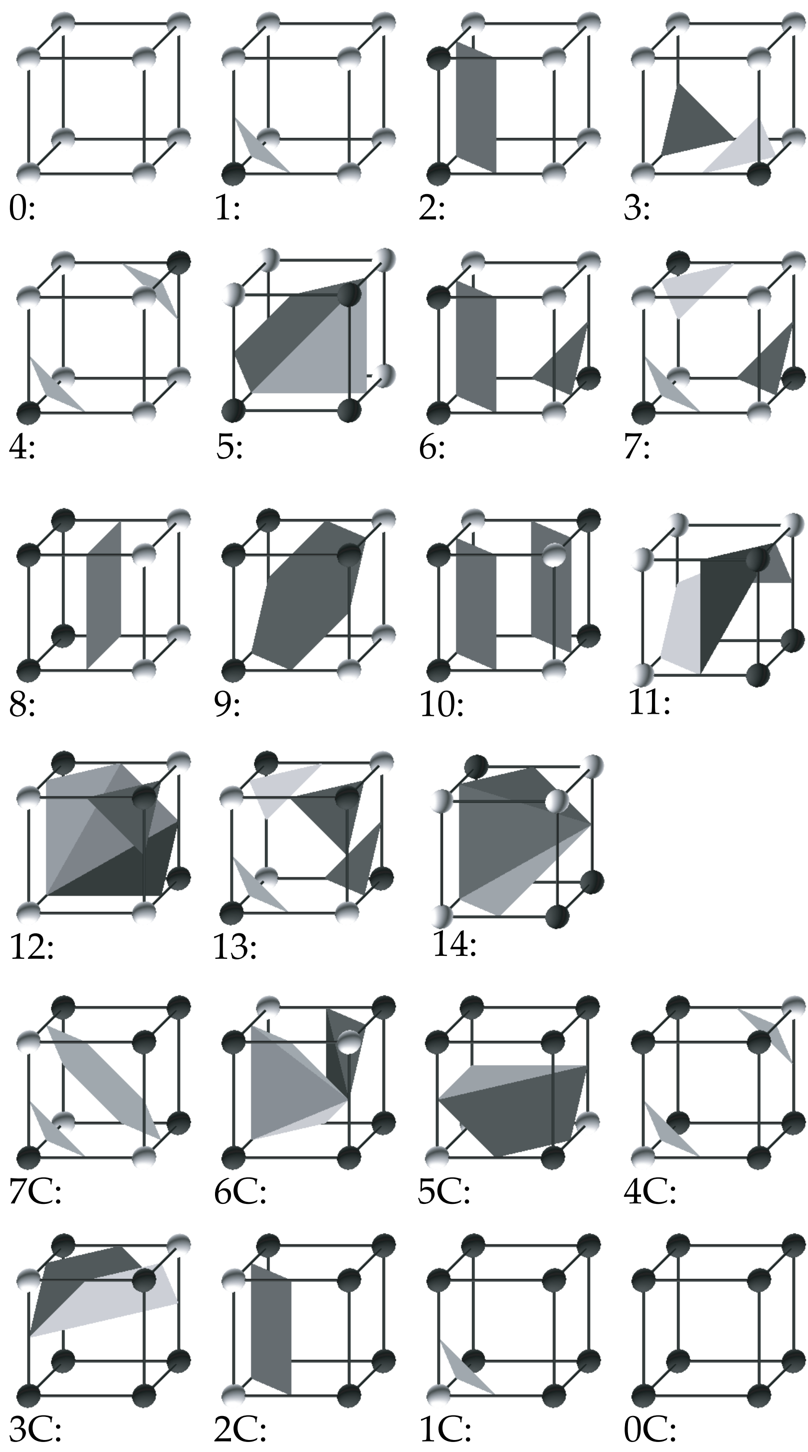
Since a cubic cell has 8 vertices, there are $2^8 = 256$ possible cases. Lorenson & Cline use symmetries between different cases to reduce them to 15 basic cases, and use a look-up table to find the polygonal approximation of the surface in a given cell.

However, the basic cases chosen by Lorenson & Cline were flawed. Given a face of a cube with two diagonally opposite corners above the surface, and the other two below the surface, some of the basic cases assumed that the higher corners fell inside the same connected component of the surface, while others assumed that they did not. Since each face of a cube was shared with an adjacent cube, it was possible to generate surfaces with holes by accident. This was noted by Dürst [Dür88], and solutions proposed by Wilhelms & van Gelder [WvG90], by Nielson & Hamann [NH91], by Matveyev [Mat94], and by Montani, Scateni & Scopigno [MSS94a]. Montani, Scateni & Scopigno's solution is the simplest: they choose different surfaces in the offending cases, resulting in a set of cases which never generates holes: these cases are shown in Figure 5.1. The balance of this thesis will use "marching cubes" to refer to this particular set of cases, unless otherwise specified.

Nielson & Hamann's solution, the *asymptotic decider* [NH91], was based on assuming that f is a bilinear interpolation on each face of the cube. The known topology of contours of bilinear functions was invoked to ensure that the assumption was consistent in both cells. The value of f at the saddle point of the bilinear function was computed and used to distinguish between the two possible solutions. Since this test gave the same result in both cubes, consistent treatment was assured, and the holes disappeared.

Most of these solutions, however, failed to address the underlying problem: that the surfaces extracted did not correspond to contours of any known interpolant. Wilhelms & van Gelder [WvG90] assumed that each cube was subdivided into tetrahedra, and that the barycentric interpolation function was used in each tetrahedron. This technique, usually referred to as *marching tetrahedra*, was also described by Bloomenthal [Blo88], and extended by Ning & Bloomenthal [NB93]. Although attractive for its simplicity, this results in the introduction of topological artifacts to the interpolant function and increases the number of triangles processed [CMS01].

Natarajan [Nat94] defined the interpolant to be the trilinear function used in other contexts, and extended the asymptotic decider by testing body saddle points inside the cube. Cignoni et al. [CGMS00] then implemented this approach, subdividing cubes with high-curvature surfaces to get a more accurate result. However, this solution assumed that only one body saddle could exist in a given cube, although two roots to the equations existed. This problem with body saddle points was identified and corrected by Chernyaev [Che95] and by Lopes & Brodlie [LB03].



black vertices are above the surface (i.e. they have higher isovalues than the surface)

white vertices are below the surface (i.e. they have lower isovalues than the surface)

Figure 5.1: Marching Cube Cases, after Montani, Scateni & Scopigno [MSS94a]

5.2.1 Speed

The principal problem with marching cubes, however, is efficiency. Because marching cubes iterates through every cell in the data, it takes $O(N)$ time to generate an isosurface, where N is the number of cells in the mesh. For typical scientific or medical data, however, it is usually assumed that any isosurface of interest

passes through significantly fewer than N cells. The number of cells intersected by the desired isosurface is often referred to as k - an output sensitive parameter. Itoh & Koyamada [IK95] estimate that $k \approx N^{\frac{2}{3}}$, since a surface is of dimension 2, and the data is of dimension 3.

Two approaches have been used to accelerate isosurface rendering. The first approach is to optimize the work performed for each cell, to minimize overall cost, while the second approach aims to improve the asymptotic performance by avoiding irrelevant cells.

Durkin & Hughes [DH94] note that triangle rendering techniques are inefficient when triangles are smaller than a single pixel. They render a single point for each cell that intersects the isosurface. Montani, Scateni & Scopigno [MSS94b] accelerate the rendering process in a different way. Instead of interpolating points along edges of the cube, they always place the edge point at the centre of the edge, then use the marching cubes cases: thus, their polygonal approximation is faster, but not as precise. Hansen & Hinker [HH92] took a slightly different tack by implementing a parallel version of marching cubes. All three of these techniques, however, merely reduce the constant of proportionality, without altering the asymptotic behaviour.

As noted above, marching cubes tests all cells of the mesh, rather than just the cells through which the isosurface runs. The cost of isosurface generation can be reduced asymptotically in cases where $k < O(N)$ by using an index structure to keep track of which cells intersect which isosurface. Three types of index structures have been tried to date: spatial structures, span space structures, and topological structures. Of these, topological index structures are used in combination with the continuation method: discussion of them is deferred to Section 5.3.1.

Spatial Index Structures: Spatial index structures assume that large regions of the volume have similar values. By taking advantage of this, regions in which few cells intersect the isosurface are discarded at an early stage. Wilhelms & van Gelder [WvG92] use octrees to accelerate marching cubes. Bajaj and Pascucci [BP99] extend this idea to generate progressive isosurfaces, in which coarse approximations of an isosurface are generated at higher levels of the octree, and finer approximations at lower levels where the cells are smaller.

Octree methods divide the region into 8 octants, then repeat this division recursively until each octant is a single cell in size. In the octree, each octant is labelled with the minimum and maximum values of all vertices contained in the octant. Level sets are then generated from the octree by commencing with the entire volume, and propagating downwards through all children spanning the desired isovalue. Construction of the octree takes $O(N \log N)$ time: worst-case time to construct a level set is $O(k + \log N/k)$ [LSJ96]. Shen & Johnson [SJ95] note however that octree-based methods are vulnerable to noise in the data.

Span Space Index Structures: Span space methods index each cell using the minimum and maximum isovalues of the cell, in structures optimized for storing intervals. Livnat, Shen & Johnson [LSJ96] treat the minima as one dimension, the maxima as a second, and call the resulting two dimensional space the *span space*. Gallagher [Gal91] uses buckets and linked lists to store the intervals. Livnat, Shen & Johnson [LSJ96] and Shen et al. [SHLJ96] use Bentley's k-d-trees [Ben75][1]. Construction of the k-d tree takes $O(N \log N)$

[1]It is awkward using k simultaneously as a parameter and as part of the name of the tree structure, but this usage is standard. To distinguish between the two uses in this thesis, the parameter is always printed k, but the name of the tree is always k-d. Since k-d trees are only referred to in this paragraph, this confusion, while regrettable, is not of major concern.

time and $O(N)$ space. Construction of a level set based on the k-d tree then takes $O(\sqrt{N} + k)$ time to retrieve all cells spanning the desired isovalue.

Van Kreveld [vK94] substitutes Edelsbrunner's *interval tree* [Ede80] for the k-d tree, reducing the time to construct a level set to $O(k + \log N)$ for two-dimensional data. Cignoni et al. [CMM+97] apply the same technique to isosurfaces in three dimensions. Construction of the interval tree in either case takes $O(N \log N)$ time and space.

These methods have advantages in addition to speed. Unlike the octree, they apply to rectilinear and simplicial meshes, and to both regular and irregular grids. However, the order of cell retrieval from these structures has little or no correlation with the spatial location of the cells. As a result, triangles representing multiple contours are interleaved, making it difficult to extract single contours, or to distinguish between contours. In this respect, span space is worse than Marching Cubes, since there is no guarantee that adjacent cells are ever processed sequentially.

5.3 Continuation

In comparison to marching cubes, Wyvill, McPheeters & Wyvill's *continuation method* [WMW86a] does not iterate through all cells of the mesh. Instead, the continuation method generates surfaces by following the surface from cell to cell. Wyvill, McPheeters & Wyvill implement this using an explicit queue to store cells yet to be processed, as in Algorithm 5.1. Howie & Blake [HB94] extended continuation to irregular meshes, under the name of *mesh propagation*: another name for continuation is *contour-following* [CSA03].

Input : A seed cell s
Output : Contour surfaces starting from the seed cell s

```
1 Queue the seed cell
2 while the queue is non-empty do
3 |  Remove a cell from the queue
4 |  Compute the Marching Cubes case
5 |  Extract the surface(s), and mark the cell
6 |  Queue all unmarked adjacent cells
  end
```

Algorithm 5.1: The Continuation Method of Isosurface Extraction

Compared to marching cubes, continuation takes $\Theta(k)$ time for isosurface extraction. Moreover, continuation can take advantage of *triangle strips*, which reduce the cost of geometric transformations in rendering by sharing vertices between adjacent triangles [Dee96]. However, continuation has a major drawback: it requires a starting point, or *seed*, from which to start. Since continuation is optimally efficient at the actual extraction, most subsequent research has focussed on generating seeds.

5.3.1 Topological Index Structures

As we saw in Section 5.2, we can reduce the cost for isosurface generation by constructing a spatial or span space index to all the cells in the mesh, indexing them in such a way that we can rapidly identify all cells intersected by any given level set. Thus, these indices are $\Omega(N)$ in size. If we only want to store seeds,

instead of all cells in the mesh, we can reduce this cost to $\Omega(t)$, where t is a measure of the topological complexity of the data. These seeds are then used to initialize Wyvill, McPheeters & Wyvill's continuation method [WMW86a] for isosurface extraction.

Itoh & Koyamada [IK94, IK95] use a heuristic structure called the *extrema graph*, in which all spatially close local extrema are connected with arcs. Each edge of the extrema graph has an associated list of seed cells for different isovalues. No exact analysis is given, but the authors admit that it is not always efficient, and is not guaranteed to succeed. Livnat, Shen & Johnson [LSJ96] show that the worst case time to generate a level set using the extrema graph is $\Omega(N)$ since all edges in the extrema graph are tested for intersection with the desired isovalue.

Bajaj, Pascucci & Schikore [BPS99] use a similar algorithm, in which they choose the entire set of cells as a seed set, then remove redundant cells heuristically, until no more can be removed. The remaining *thinned* set is stored in a segment tree. The same paper also describes a greedy "climbing" algorithm which generates a similar seed set, and a sweep algorithm for constructing seed sets offline. These seeds are then stored in a spatial data structure such as the k-d tree or interval tree. The topology of the data set is not explicitly calculated or stored: the topology of the data set is, however, used to generate contours.

Itoh, Yamaguchi & Koyamada [IYK01] combine the sweep from Bajaj, Pascucci & Schikore's algorithm with the extrema graph of Itoh & Koyamada to produce an *extrema skeleton*, with properties similar to the extrema graph. Again, the topology of the data set is used to accelerate isosurface extraction, but not computed explicitly.

However, there is a topological index structure that is guaranteed to produce correct seeds for every contour: the contour tree. This structure will be introduced in the next chapter, and the question of generating isosurface seeds using it deferred to Chapter 8.

Chapter 6

Contour Trees

In this chapter, we give formal and informal descriptions of the contour tree. Recall that in Chapter 2, we described the contour tree as the structure that described the nesting relationship of a set of two-dimensional contours. Section 6.1 reintroduces this notion for a three-dimensional data set. Section 6.2 then covers some previous work on the contour tree, and Section 6.3 gives a formal definition based on Morse theory. Finally, Section 6.4 describes the *augmented contour tree*.

6.1 Description of the Contour Tree

In Figure 6.1, we show a set of isosurfaces from a small dataset, with large values at 4 corners of a cube, medium values at the other 4 corners and on the faces of the cube, and small values inside and outside the cube. As the height (i.e. the value of the function) decreases, we see contours appear, split, change genus, join, and disappear. In particular, the level set evolves from four sticks (a), to two rings (c), to two cushions (between (c) and (d)), to one surface (d), which gradually turns into two nested surfaces as the "inside" and "outside" separate (between (e) and (f)). Finally (although we cannot see this), the inner surface collapses to a point and disappears, leaving us with a single surface once more.

The *contour tree* is a graph that tracks contours of the level set as they appear, join, split, and disappear. Figure 6.1(g) shows the contour tree for the data set used in Figure 6.1. Starting at the global maximum, four small contours appear in sequence (α, β, γ, δ): these correspond to the four leaves at the top of the contour tree. The surfaces join (ϵ, ζ) in pairs, forming larger contours, which quickly become rings. These rings then flatten out into cushions, which join (η) to form a single contour. This contour gradually wraps around a hollow core, and pinches off, splitting into two contours: one (ι) faces inwards, the other (θ) outwards. The inward contour contracts until it disappears at a local minimum: the outward contour expands until it reaches the global minimum.

Sampled Scalar Field Values:

z = 0:				
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

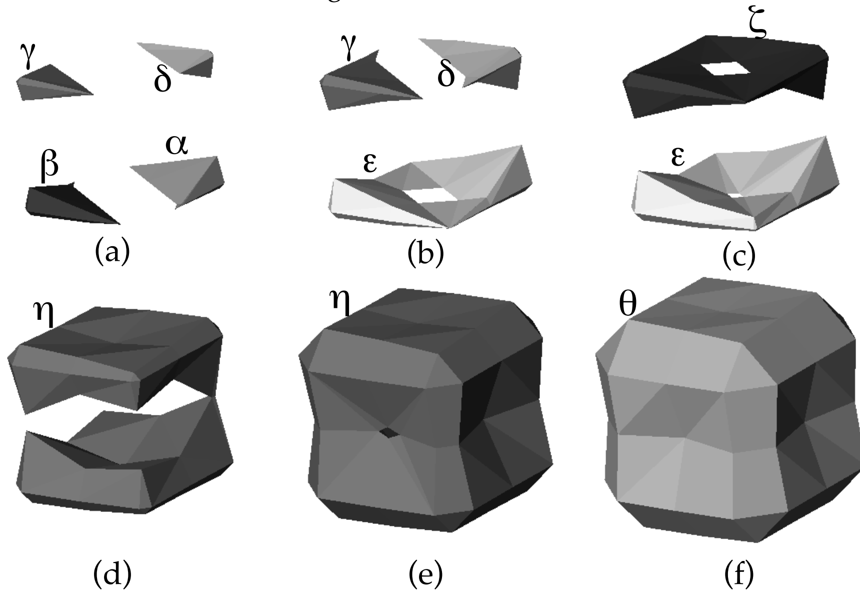
z = 1:				
0	0	0	0	0
0	99	90	85	0
0	95	80	95	0
0	85	90	99	0
0	0	0	0	0

z = 2				
0	0	0	0	0
0	75	50	65	0
0	55	15	45	0
0	60	40	70	0
0	0	0	0	0

z = 3				
0	0	0	0	0
0	97	87	82	0
0	92	77	92	0
0	82	87	97	0
0	0	0	0	0

z = 4				
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

Level Sets (ordered from high isovalue to low isovalue):



Contour Tree:

Figure 6.1: Sample Data Set in 3 Dimensions, with Contour Tree.
Level Sets shown are at 6 distinct points during sweep from high to low isovalue.
The Contour Tree is shown embedded in the plane with the vertices placed vertically according to their isovalue.
Horizontal lines shown with the contour tree correspond to level sets (a) - (f).
Each such line intersects exactly one edge for each contour in the corresponding level set.
(a) Contours α, β, γ and δ appear first, one at each local maximum.
(b) Contours α, β *join* together to form contour ϵ.
(c) Contours γ, δ join together to form contour ζ.
(d) Contours ϵ, ζ join together to form contour η.
(e) Contour η has an internal cavity with a double wall and a single opening.
(f) Contour η has now separated into inner (ι) and outer (θ) contours.

In Section 6.3, we define the contour tree using equivalence classes of contours [CSA03], but it can also be characterized as a continuous contraction of each possible contour to a single point [Pas01, Ree46]. This characterization captures the most useful property of the contour tree: that there is a 1 to 1 correspondence between contours in the data set and points in the tree. These points may be vertices, or *supernodes* of the tree or may be single points along the edges of the tree, treating each edge as a set of points rather than a purely combinatorial object. Another useful property of the contour tree which is almost as useful is that each local extremum of the function is a leaf of the contour tree.

This is not the only possible definition of the contour tree. In Chapter 2, we characterized the contour tree in terms of the nesting relationship of the contours. This, in fact, was the definition of the contour tree introduced by Boyell & Ruston [BR63]. For practical purposes, this is equivalent to the definition in Section 6.3, although basing the contour tree on nesting relationships imposes fewer formal constraints on the input data.

6.2 Previous Work

The contour tree was introduced by Boyell and Ruston [BR63], as a summary of the evolution of contours on a map (i.e. in 2-D), and was used by Freeman and Morse [FM67] to find terrain profiles in a contour map. It has been used for image processing and geographic information systems [GC86, KK94, SC86, TIS+95], as well as for isosurfaces [BPS99, Car00, CS03, CSA00, CSA03, Pas01, PCM02, vKvOB+97, Zyd88].

The contour tree is a special case of the Reeb graph [Ree46], which describes the evolution of arbitrary contours as a parameter changes. As a result, the Reeb graph can be used to trace the evolution of an arbitrary surface in space, such as the surface of a torus. In this case, the parameter used to describe the surface is commonly the z coordinate of the space, but can also be time in a time-varying data set. Thus, Reeb graphs have been used to reconstruct surfaces from medical images [SKK91] and to track individual isosurfaces over time[SW96, SW98]. The Reeb graph belongs to a branch of mathematics known as Morse theory [Ban67, Mil63], which seeks to study topological properties by parameterizing topological manifolds.

Any function can be viewed as a manifold parameterized by the isovalue, and the contour tree can also be defined as the Reeb graph of the manifold with respect to the isovalue parameter. But the contour tree has some additional useful properties compared to the Reeb graph. Because a function can have only one value at a given point, no two isocontours intersect. Moreover, any given contour c divides its complement into disjoint subregions, so that every path from a point (or contour) in one subregion to a point (or contour) in another subregion must pass through the contour c. It follows from this that the contour tree has no cycles: hence the name "contour *tree*".

It is not necessary that a function be defined in order for the contour tree to exist. Following Boyell & Ruston [BR63], the contour tree can instead be defined for any set of isovalued manifolds that nest inside each other. Although nesting properties require that all contours be closed manifolds, it is possible to work around this by taking advantage of the fact that a contour comes from a scalar field, and has a "high" side and a "low" side. We will return to this concept in Chapter 10. For now, we do not rely on this characterization of the contour tree, instead using a Morse-theoretic definition.

6.3 The Contour Tree

Morse theory [Ban67, Mil63, Ree46, SKK91] studies the changes in topology of level sets as some parameter is varied. Points at which the topology of the level sets change are called *critical points*. Morse theory requires that the critical points are isolated – i.e. that they occur at distinct points and values. A continuous function that satisfies this condition is called a *Morse function*. All points other than critical points are called *regular points* and do not affect the number or genus of the contours.

As we will see when considering isosurface extraction in Chapter 8, we are not always interested in

all Morse critical points. In particular, changes of topological genus (e.g. from a disk to a torus) do not affect the number of contours, and therefore do not affect isosurface extraction. However, as Pascucci & Cole-McLaughlin [PCM02] have shown, these Morse critical points can readily be added tothe contour tree. To simplify discussion, however, we defer this to Section 6.4.

In Section 6.1, we described the evolution of level sets as the function value varied. We make this "evolution" more precise by defining an equivalence relation between two contours. Since the critical points are, as the name implies, critical to this process, we deal with them first.

We assume that we are studying a Morse function $f : \mathcal{R} \rightarrow \mathbb{R}$ defined over some compact region $\mathcal{R} \in \mathbb{R}^d$. We do not assume that f is defined outside $\mathcal{R}$. We can also assume that f is continuous because it is a Morse function. We now define level sets and contours as follows:

Definition 6.1 *The* level set *of f at h is the set $L_h(f) = \{x \in \mathcal{R} : f(x) = h\}$.*

Definition 6.2 *A* contour *of f at h is a single connected component of $L_h(f)$.*

We do not assume that contours are closed manifolds: they may intersect the boundary of $\mathcal{R}$. Except for holes at the boundary, however, contours will be closed manifolds because f is continuous.

Once we have these definitions, we can look at the topological changes in the neighbourhood of a point x. Let $B_\epsilon(x)$ refer to the ball of radius ϵ centred at x, and let $f|B_\epsilon(x)$ be the restriction of f to $B_\epsilon(x)$. Then:

Definition 6.3 *The* Morse up-degree *of f at a point x is the number of contours of $f|B_\epsilon(x)$ at isovalue $f(x) + \delta$, as $\delta, \epsilon \rightarrow 0^+$.*

Definition 6.4 *The* Morse down-degree *of f at a point x is the number of contours of $f|B_\epsilon(x)$ at isovalue $f(x) - \delta$, as $\delta, \epsilon \rightarrow 0^+$.*

Definition 6.5 *A* Morse critical point *of f is a point x for which the Morse up- and down- degrees are not both* 1.

However, two or more contours of $f|B_\epsilon(x)$ can belong to the same contour of f. This leads to some more definitions:

Definition 6.6 *The* up-degree *of f at a point x, $\delta^+(x)$, is the number of contours of f that intersect $B_\epsilon(x)$ at isovalue $f(x) + \delta$, as $\delta, \epsilon \rightarrow 0^+$.*

Definition 6.7 *The* down-degree *of f at a point x, $\delta^-(x)$, is the number of contours of f that intersect $B_\epsilon(x)$ at isovalue $f(x) - \delta$, as $\delta, \epsilon \rightarrow 0^+$.*

Definition 6.8 *A* critical point *of f is a point x for which the up- and down- degrees are not both* 1.

It is not hard to see that every critical point is a Morse critical point, but not vice versa. Any Morse critical point that is not a critical point is a point at which some topological property other than connectivity changes. In three dimensions, these are points where the topological genus of the surface (the number of handles) changes.

Once we have these definitions, it is convenient to classify the critical points.

Definition 6.9 *A* join *is a critical point with up-degree of at least* 2*.*

Definition 6.10 *A* split *is a critical point with down-degree of at least* 2*.*

Some papers reverse the meaning of join and split. Since they are dual, this does not affect anything material, although it can be confusing.

Definition 6.11 *A* saddle *is a join or a split.*

Definition 6.12 *A* local maximum *is a critical point with up-degree of* 0*.*

Definition 6.13 *A* local minimum *is a critical point with down-degree of* 0*.*

Definition 6.14 *A* local extremum *is a local minimum or local maximum.*

Once we have classified critical points, we use *regular point* to refer to any other point:

Definition 6.15 *A* regular point *is a point whose up- and down- degrees are both* 1*.*

Occasionally, we will need to classify points under the Morse definitions of up- and down- degree. When we do so, we simplify prefix the class of points with "Morse", as follows:

Definition 6.16 *Morse joins, Morse splits, Morse saddles, and Morse regular points are defined in the same way as joins, splits, saddles, and regular points, using the Morse up- and down- degrees.*

We now define the equivalence relation as follows:

Definition 6.17 *Let $\mathcal{V} = \{v_1, \ldots, v_t\}$ be the set of critical points of f. Then contours c and c' are equivalent ($c \equiv c'$) if there exists some f-monotone path P in $\mathcal{R}$ that connects some point in c with another in c', such that no point $x \in P$ belongs to the same contour as any $v_i \in \mathcal{V}$.*

We refer to the equivalence classes of this relation as *contour classes*. Contours that do not contain critical points belong to contour classes that map 1 to 1 with open intervals $(f(x_i), f(x_j))$, where x_i and x_j

are critical points and $f(x_i) < f(x_j)$. We describe a contour class as being *created* at v_j or at $f(v_j)$, and being *destroyed* at at v_i or at $f(v_i)$, thus preserving the intuitive description of a sweep from high to low values.

Contours that include critical points must be the sole members of their (finite) contour classes. Moreover, since f is a Morse function, only one critical point can belong to a single contour. It follows from this that we can refer interchangeably to a critical point of f and a critical point of the contour tree of f.

This correspondence between critical points and finite contour classes, and between open intervals and infinite contour classes, leads to a graph-theoretic definition of the contour tree for a simplicial mesh.

Definition 6.18 *The* contour tree *for a function $f : \mathcal{R} \rightarrow \mathbb{R}$ (where $\mathcal{R} \subseteq \mathbb{R}^d$) is a graph $(\mathcal{V}, \mathcal{E})$ such that:*

1. *$\mathcal{V}$ is the set of critical points of f, and*
2. *For each infinite contour class created at v_i and destroyed at v_j, the edge $(v_i, v_j) \in E$. For convenience, we will assume that (v_i, v_j) is directed from the higher to the lower end, i.e. that $f(v_i) > f(v_j)$.*

For convenience in later proofs, we also define the following:

Definition 6.19 *Let c be a contour of f that belongs to an infinite contour class created at v_i and deleted at v_j. Then c* belongs *to the edge $e = (v_i, v_j)$ in the contour tree.*

Definition 6.20 *For any edge e of the contour tree of f, define the contour set of e to be $Cont(e) = \{c :$ c is a contour of f and c belongs to $e\}$.*

Definition 6.21 *If e is an edge of the contour tree for the function f, define the* sweep region *belonging to e to be $\mathcal{R}(e) = \bigcup\{c : c \in Cont(e)\}$.*

Any contour c of f is uniquely defined by its isovalue h and the contour class to which it belongs, which may be finite or infinite. Moreover, c can be represented as a point in the contour tree that is either at a vertex or somewhere on an edge. If c belongs to an edge, we can place it precisely by interpolating with respect to isovalue.

Definition 6.22 *Let c be a contour at isovalue h of f that belongs to an edge $e = (v_i, v_j)$ of the contour tree of f. Then c is located in the contour tree at the point found by linear interpolation along e with respect to the isovalues $f(v_i), h, f(v_j)$.*

From this, we see that contours that are spatially close in $\mathcal{R}$ will also be spatially close in the contour tree: contours belonging to the same edge of the contour tree will be interpolated along that edge according to their respective isovalues, and a natural correspondence between monotone paths in the contour tree and f-monotone paths in $\mathcal{R}$ arises.

We start by showing we can convert from monotone paths in the contour tree to f-monotone paths in $\mathcal{R}$ and vice versa. The first is an immediate corollary of our definition of the contour tree:

Corollary 6.1 *Every f-monotone path P in $\mathcal{R}$ maps to the monotone path $Q = \{c_p : p \in P\}$ in the contour tree, where c_p is the contour through a given point p.*

To show the converse, we start with paths in the contour tree that traverse no more than a single edge of the tree:

Lemma 6.2 *Let p_0, p_1 be points on contours c_0 and c_1 respectively that both belong to the same edge of the contour tree. Then there exists a f-monotone path P from p_0 to p_1.*

Proof: Without loss of generality, assume that $f(p_0) < f(p_1)$. By Definition 6.17, there exists a monotone ascending path Q from point q_0 on contour c_0 to point q_1 on contour c_1. Since c_0 and c_1 belong to the same edge of the contour tree, every ascending path from c_0 passes through the same set of contours, because no point on any of these contours may have an up-degree or down-degree > 1.

Note that q_0 need not be the same as p_0, nor need q_1 be the same as p_1. But, since p_0 and q_0 belong to the same contour, we know that they are connected by a path P_0 on c_0. Similarly, p_1 and q_1 are connected by a path P_1 on c_1. If we construct the path P_0QP_1 by concatenating these three paths, we obtain a non-decreasing path from p_0 to q_0, but this path is not f-monotone.

We remedy this by sliding q_0 along c_0 and q_1 along c_1 until they reach p_0 and p_1 respectively. Because f is continuous, if we slide point q_0 along contour c_0 to point p_0, we can keep it connected with path Q by deforming the bottom end of Q that falls into the isovalue range $[f(p_0), f(p_0)+\epsilon]$ for some arbitrarily small ϵ. Similarly, we slide the top end of Q along contour c_1 to point p_1, and we have the desired f-monotone path. $\square$

Corollary 6.3 *Let p_0 be a critical point on contour c_0, and let p_1 be a regular point on some contour c_1 that belongs to an edge e incident to c_0 in the contour tree. Then there exists an f-monotone path in $\mathcal{R}$ from p_0 to p_1.*

Proof: Without loss of generality, assume that $f(p_0) < f(p_1)$. We know from Definition 6.8 that, as $\delta, \epsilon \to 0^+$, the ball $B_\epsilon(p_0)$ intersects one contour at isovalue $f(p_0) + \delta$ for each upwards arc of the contour tree incident to c_0, including e. Let c_δ refer to the contour at isovalue $f(p_0) + \delta$ that belongs to edge e, and, for each δ, ϵ, choose a point $p_{(\delta,\epsilon)}$ in the intersection of $B_\epsilon(p_0)$ and c_δ. By Lemma 6.2, there exists some f-monotone path $P_{(\delta,\epsilon)}$ between $p_{(\delta,\epsilon)}$ and p_1. As $\delta, \epsilon \to 0^+$, the point $p_{(\delta,\epsilon)}$ will approach p_0, and, in the limit, we obtain a f-monotone path P from p_0 to p_1. $\square$

Corollary 6.4 *Let $e = (c_0, c_1)$ be any edge in the contour tree of f, and let p_0 and p_1 be the critical points on contours c_0 and c_1, respectively. Then there exists a f-monotone path in $\mathcal{R}$ from p_0 to p_1.*

Proof: Pick any point p on any contour c belonging to edge e. By Corollary 6.3, there exist f-monotone paths in $\mathcal{R}$ from p_0 to p and from p to p_1. Concatenate these paths to obtain a f-monotone path from p_0 to p_1. $\square$

Lemma 6.5 *For every monotone path Q in the contour tree, there exists at least one f-monotone path P in $\mathcal{R}$ such that P maps to Q in the contour tree.*

Proof: Without loss of generality, Q is a descending path $c_0, c_1, \ldots, c_m, c_{m+1}$, where c_0 and c_{m+1} are contours that belong to edges e_0 and e_{m+1} of the contour tree, and $c_1, \ldots, c_m$ are the contours in the finite contour classes corresponding to some of the vertices of the contour tree. To avoid confusion, we use $p_1, \ldots, p_m$ to refer to the critical points of f corresponding to $c_1, \ldots, c_m$. Let p_0 and p_{m+1} be any points on contours c_0 and c_{m+1} respectively.

By Corollary 6.3, we know that there are f-monotone paths in $\mathcal{R}$ from p_0 to p_1 and from p_m to p_{m+1}. By Corollary 6.4, we also know that there are f-monotone paths in $\mathcal{R}$ from p_i to p_{i+1} for $i = 1 \ldots m-1$. Since the isovalues of p_0 to p_{m+1} are in descending order, we concatenate all these paths to obtain the desired path P. $\square$

Theorem 6.6 *For every path P in $\mathcal{R}$, there exists a path Q in the contour tree to which P maps, and for every path Q in the contour tree, there exists at least one path P in f that maps to Q.*

Proof: ($\Rightarrow$) We decompose the path P into f-monotone subpaths and subpaths that lie on contours. By Corollary 6.1, each of the former maps to a path in the contour tree. And each subpath that lies on a contour maps to that contour in the contour tree. Then Q is simply the concatenation of these subpaths.

($\Leftarrow$) We decompose the path Q into monotone subpaths $Q_1, \ldots, Q_m$ in the contour tree. Unlike paths in $\mathcal{R}$, there are no subpaths that lie on contours, since each contour has collapsed to a single point. By Lemma 6.5, each subpath Q_i maps to some subpath P_i in f. Moreover, if p_i is the first endpoint of P_i and r_{i-1} is the second endpoint of P_{i-1}, both p_i and r_{i-1} must lie on the same contour, else Q_{i-1}, Q_i will not connect in the contour tree. But, if p_i and r_{i-1} lie on the same contour, there is some path R_i along the contour from p_i to r_{i-1}. We now let $P = P_1 + R_1 + P_2 + \ldots + R_{m-1} + P_m$. $\square$

6.4 The Augmented Contour Tree

The contour tree defined in the previous section contains only vertices at which connectivity changes. If other vertices are significant, we simply add them to the contour tree, or *augment* it. Since each such vertex belongs to a contour, which in turn belongs to an equivalence class, this is easy.

Definition 6.23 *The* contour tree augmented by W *for any set W of points in $\mathcal{R}$ is the tree obtained by substituting $\mathcal{V} \cup W$ for $\mathcal{V}$ in Definition 6.17 and Definition 6.18.*

Definition 6.24 *The* unaugmented contour tree *is the contour tree augmented by the empty set.*

Definition 6.25 *The* Morse-augmented contour tree *is the contour tree augmented by the set of Morse critical points which are not critical points.*

Definition 6.26 *The* fully augmented contour tree *or is the contour tree augmented by all vertices in the mesh.*

Van Kreveld et al. [vKvOB+97] call the vertices and edges of the unaugmented contour tree *supernodes* and *superarcs*, and call the vertices and edges of the fully augmented contour tree nodes and arcs. They

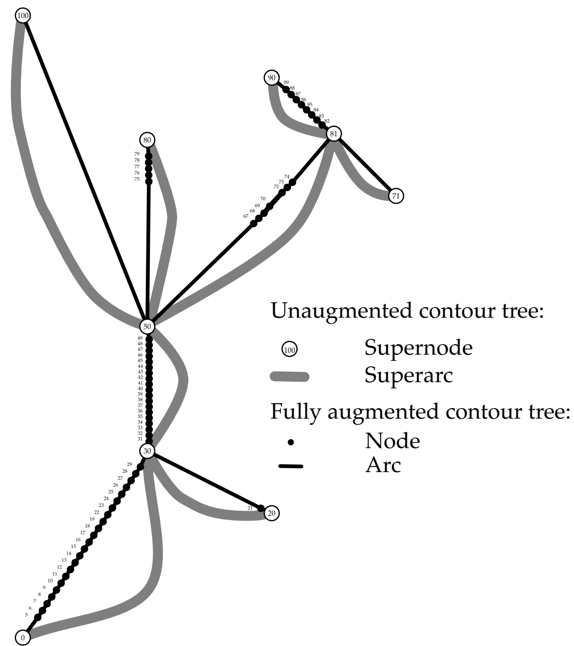


Figure 6.2: Unaugmented Contour Tree and Fully Augmented Contour Tree For Figure 4.1
Following van Kreveld et al. [vKvOB+97], we show both the unaugmented contour tree and the fully augmented contour tree in the same diagram. Note how the superarcs each consist of a string of arcs, and each node except supernodes belongs to some superarc.

also draw both the unaugmented contour tree and the fully augmented contour tree in the same diagram, as shown in Figure 6.2. Where clarity is required, we refer to the unaugmented contour tree or the fully augmented contour tree, in order to make the difference clear. Elsewhere, we use “contour tree” to refer to any contour tree, augmented or not.

Part III

New Applications for Contour Trees

This Part will deal with a several new uses for the contour tree, from extraction of contours using path seeds in Chapter 8 to a description of how to use the contour tree to compute geometric properties in Chapter 10.

For the sake of simplicity, we will make three assumptions in this part. In Part IV, we shall show how to relax these assumptions.

Assumption 1 *The input data consists of a simplicial mesh with isovalues specified at the vertices of the mesh.*

Assumption 2 *The isovalue at any point in $\mathcal{R}$ other than a vertex of the mesh is computed using barycentric interpolation.*

Assumption 3 *No two vertex isovalues are the same.*

Overall, the contribution of this Part of the thesis is to use the contour tree as a abstraction that unifies individual advances in the following areas:

1. *topology*, in the form of efficient algorithms to compute the contour tree (Chapter 7)
2. *isosurface extraction*, in the form of efficient algorithms for extracting single contours using path seeds (Chapter 8)
3. *exploratory visualization*, using the flexible isosurface to explore and manipulate individual contour surfaces (Chapter 9),
4. *geometry*, in the form of local spatial measures (Chapter 10) that measure geometric properties for regions bounded by contours, and
5. *topological simplification*, in which local spatial measures (Chapter 11) are used to guide simplification of the topology in the form of the contour tree, both to identify major features in the data, and to identify and remove noise.

In Chapter 7, I review algorithms used to compute the contour tree. In Chapter 8, I then explore the extraction of single contours using seeds generated from the contour tree. This sets the stage for Chapter 9, in which I discuss the manipulation of individual contour surfaces, and an interface for doing so. In Chapter 10, I show how to compute geometric properties with respect to regions bounded by contours. Finally, in Chapter 11, I use these geometric properties to simplify the contour tree for clearer visual interaction and for noise suppression.

Chapter 7

Algorithms for Computing Contour Trees

A number of different algorithms have been used to compute the contour tree, in two, three, and arbitrary dimensions. These include contour nesting (Section 7.2), skeletonization (Section 7.3), monotone path search (Section 7.4), contour sweep (Section 7.5), and sweep and merge (Section 7.6).

The analysis of these algorithms can be complex, so we start by defining parameters for analysis in Section 7.1. Sections 7.2 through 7.6 then discuss each algorithm in turn. Since the sweep and merge algorithm is fundamental to later contributions in this thesis, an example of its operation is then shown in Section 7.7.

7.1 Parameters for Analysis

In analysing the runtime of contour tree algorithms, a number of parameters have been used. For convenience, these parameters are collected here:

n - the number of vertices in the mesh, which may be regular or irregular.

N - the number of cells in the mesh: for irregular meshes in three or more dimensions, this may be $\Omega(n^2)$. For regular meshes or two-dimensional meshes, it is $\Theta(n)$.

sort - cost of sorting the vertices: notionally $O(n \log n)$, but can be reduced to $O(n)$ for quantized data.

t - the number of vertices in the contour tree (proportional to the number of local extrema).

k - an output-sensitive parameter for the size of extracted isosurfaces. Since isosurfaces are generally rendered as triangles, we will assume that k is the number of triangles generated.

$\alpha(t)$ - the extremely-slowly growing inverse Ackermann function, which is ≤ 4 for practical value of t.

p - an output-sensitive parameter for the cost of using path seeds to generate contours. We discuss this parameter in more detail in Section 8.2.4, but include it here for completeness.

τ - the number of contours present in a level set or flexible isosurface. We discuss of this parameter in Section 9.4, but include it here for completeness.

7.2 Contour Nesting

The earliest contour tree construction algorithm, contour nesting, was introduced by Boyell & Ruston in 1963 [BR63]. In this algorithm, the contour tree represents the nesting relationship of a set of polygonal contours manually converted from a topographic map. An "outside" region is designated that encloses everything, and nodes representing contours are added to the tree one at a time. Because there is a well-defined outside region, each contour has a distinct inside and outside, and the node for each contour is connected to the node for the contour immediately outside it. No details were provided on the mechanics of the construction, which may have been manual.

7.3 Skeletonization

The second contour tree construction algorithm, skeletonization, is heuristic in nature, and was used by Itoh, & Koyamada [IK94, IK95], and by Itoh, Yamaguchi & Koyamada [IYK01]. Starting with every cell in the mesh, cells are removed until no more can be removed without disconnecting critical points of the function: i.e. that for any two critical points, there is a continuous chain of cells connecting the two critical points. This algorithm leaves a tree structure composed of cells to use as seeds for isosurface extraction. Since all critical points are included, and all are connected to each other, this structure intersects all possible contours. And since cycles can be broken by removing a single cell, this structure will be a tree. No proof has been constructed to show that this structure is in fact the contour tree, but it seems to be equivalent.

7.4 Monotone Path Search

Another algorithm was described by Takahashi et al. [TIS+95] for two-dimensional data, referring to the contour tree as a *topological change tree*. This algorithm has two stages: a graph construction stage, and a tree computation stage. In the graph construction stage, edges are added from saddles to local extrema, representing distinct directions of ascent or descent from the saddle. In the second stage, a local extremum is picked, and connected in the tree to the adjacent saddle with isovalue closest to that of the local extremum. The local extremum is then removed from the graph, and any remaining paths redirected to connect to the saddle. This algorithm was later extended to three dimensions by Takahashi, Fujishiro & Takeshima [TFT01]. No formal analysis of the runtime was given, but a bound of $O(n^2)$ is easily proven by observing that at most n vertices are critical points and that no path from a saddle to a local extremum may be longer than n vertices. A tighter bound may be possible. Finally, this method constructs only the unaugmented contour tree: extending it to augmented contour trees is, however, feasible.

7.5 Contour Sweep

The next algorithm was introduced by van Kreveld et al. [vKvOB+97]. This algorithm is based on sweeping through the data set from high to low, as described in Section 6.1, then sweeping in the reverse direction (i.e. from low to high isovalues). In addition to assuming barycentric interpolation over a simplicial mesh with distinct isovalues, van Kreveld et al. assume that all supernodes are simple (i.e. that no more than two contours meet at the supernodes), and guarantee this by pre-processing the mesh to break up multiple saddles.

In each sweep, the level set corresponding to the isovalue is explicitly maintained as a set of contours. For two dimensional data, each contour can be represented as an ordered list of simplices through which the contour passes. However, each list of simplices may correspond to more than one contour in the first sweep, as this sweep does not detect when contours separate.

In the first sweep, as each local maximum is passed, a new contour is constructed in the level set. Contours are represented by tracking the set of simplices currently intersected by the contour. At vertices other than local maxima, the contours are updated locally by adjusting the contours for each simplex incident to the vertex. If the vertex is the lowest vertex in a given simplex, the simplex is removed from the contour. If the vertex is the highest vertex in the simplex, the simplex is added to the contour. And, regardless of the vertex' isovalue, the connectivity between simplices is updated if necessary. Because the contours are separately labelled, it is easy to detect when two contours meet at a vertex: in this case, the smaller of the two contours is assimilated to the larger one, and the vertex is identified as a join.

In the second sweep, in the reverse direction, splits and local minima are detected, and information from the first sweep is used to determine how components break apart at splits, allowing the contour tree to be assembled.

By always joining the smaller of two contours onto the larger, van Kreveld et al. showed that the overall time could be reduced to $O(N \log N) = O(n \log n)$ in two dimensions. In dimensions higher than two, however, they stated a time bound of $O(N^2)$.

However, this algorithm is moderately complex. In addition to preprocessing to break down complex saddles, this algorithm requires special processing at the boundary of the data set. However, this algorithm computes the fully augmented contour tree directly, which can be used to generate minimal sets of contour seeds: see Section 8.1 for details.

Tarasov & Vyalyi [TV98] extended this algorithm to $O(N \log N)$ in three dimensions, by showing that the relabelling process could also be done efficiently in three dimensions, and by extending the pre-processing of complex saddles to at least three dimensions. Their method, however, required subdividing each tetrahedron into 24 new tetrahedra, and potentially 360, with obvious implications for running time.

Pascucci [Pas01] then further extended this algorithm to compute topological indices called Betti numbers for the contours, effectively adding topological genus to the contour tree.

7.6 Sweep and Merge

Carr, Snoeyink & Axen [CSA00, CSA03] improved the algorithm of van Kreveld et al. by dispensing with the explicit maintenance of the contours, the preprocessing for complex saddles, and the specialized boundary processing, while extending it to all dimensions. Two sweeps are performed to obtain partial information about the contour tree by computing the *join tree* and *split tree*, representing the connectivity of $\{x : f(x) \geq h\}$ and $\{x : f(x) \leq h\}$, respectively. A third stage, similar in nature to the second stage of Takahashi et al. [TIS+95], then assembles the contour tree by picking local extrema from the join and split trees and transferring them to the contour tree.

This merge stage can be shown to be equivalent to a sequence of partial sweeps through the data, sweeping one contour at a time through the isovalues represented by a single arc (or superarc) of the contour tree. This sweep passes through all of the contours represented by the arc (or superarc), defining a subregion of the original region. Thus, each vertex can be viewed as belonging to a topologically-defined subregion of the data set. This has powerful consequences, as it allows us to segment the domain of the function into regions with different properties. In particular, one of the principal contributions of this thesis is to use the idea of sweeping up and down through the contour tree to compute geometric properties of isosurfaces, then to use this information to perform topologically-based simplification both of the contour tree and of the underlying data. We will see this in more detail in Chapter 10.

This algorithm generates a t-supernode contour tree from an n-vertex, N-cell mesh in $O(sort + N + n + t\alpha(t))$. Moreover, it operates for simplicial meshes in any number of dimensions, and computes either the unaugmented contour tree or the fully augmented contour tree.

Input : Simplicial Mesh M with value $f(v)$ at each vertex v
Output : T, the contour tree for M

1. Using Algorithm 7.2, compute the join tree $J(M)$
2. Using Algorithm 7.2, compute the split tree $S(M)$
3. Using Algorithm 7.3, merge the join tree $J(M)$ and the split tree $S(M)$ to obtain the fully augmented contour tree T
4. [Optional]: Use Algorithm 7.4 to compute the unaugmented contour tree R from the fully augmented contour tree T

Algorithm 7.1: Computing the Contour Tree For A Simplicial Mesh

As shown in Algorithm 7.1, we invoke Algorithm 7.2 twice to compute the join and split trees for the mesh M, then Algorithm 7.3 to merge these two trees together. Formal proof of these algorithms is omitted, as it can be found in my M.Sc. thesis [Car00] or the corresponding journal article [CSA03].

We will see an example of this algorithm in operation in Section 7.7. In the rest of this section, however, we will give further details of how this algorithm works. We describe the join / split sweeps in Section 7.6.1, and the merge phase in Section 7.6.2. Since this algorithm computes the fully augmented contour tree, we then show how to obtain the unaugmented contour tree in Section 7.6.3, either by reduction from the fully augmented contour tree, or by modifying Algorithm 7.1 to obtain the unaugmented contour tree directly. Finally, in Section 7.6.4, we look at some variations on the sweep and merge algorithm.

7.6.1 Join and Split Sweeps

The sweep to compute the join tree is conceptually quite simple: a join occurs when a vertex v has two higher-isovalued neighbours that belong to different connected components of $\{x : f(x) > f(v)\}$. Efficient implementation of this algorithm depends on the observation that this can be computed using Tarjan's union-find structure [Tar75]. Instead of recomputing this separately at each vertex v, the vertices are processed in order of height, so that the union-find structure progressively computes the desired components. Each join corresponds to a union operation that connects two previously disconnected components. This algorithm is also used by Cox, Karron & Ferdous [CKF03] to compute the *Digital Morse Theory criticality tree*, which is essentially identical to the join tree. These authors also do not assume that unique isovalues exist, adding an additional union-find step to adjudicate at each isovalue.

```
   Input  : Simplicial Mesh M with value f(v) at each vertex v
   Output : J(M), the join tree for M
1  J(M) = ∅
2  for each vertex v in M, in any order do
3  |  Set Comp(v) = v to initialize union-find structure.
   end
4  for each vertex v in M, in descending order do
5  |  for each neighbour n of v, in any order do
6  |  |  if f(n) ≥ f(v) then
7  |  |  |  Find components Comp(n) and Comp(v) of n and v.
8  |  |  |  Let u be the lowest vertex in the component Comp(n).
9  |  |  |  Add (u, v) to the join tree J(M).
10 |  |  |  Add edge (n, v) to union-find.
   |  |  end
   |  end
11 |  Set v as lowest vertex in component Comp(v).
   end
```

Algorithm 7.2: Computing the Join Tree For a Simplicial Mesh

7.6.2 Merging the Join and Split Trees

The third phase merges the join tree $J(M)$ and split tree $S(M)$ to obtain the contour tree T. To do so, we define the join tree and split tree to exist for any graph with isovalues attached to vertices. The success of the overall algorithm relies on showing that the join tree $J(T)$ and split tree $S(T)$ for the contour tree T are identical to the join tree $J(M)$ and split tree $S(M)$ for the simplicial mesh M: we will return to this idea in Chapter 12. Conveniently, the up-degree $\delta^+(v)$ of a vertex v in the join tree is the same as the up-degree of the same vertex in the contour tree. Similarly, the down-degrees of v in the split tree and the contour tree are the same. This allows us to read the degree of a vertex from the join and split trees, even if we have not yet added v to the contour tree.

This definition facilitates a recursive algorithm for computing the contour tree T from the join and split trees $J(T) = J(M)$ and $S(T) = S(M)$. This algorithm, shown in Algorithm 7.3, identifies a single edge e incident to some leaf vertex v in T. Since v is a leaf, removing it from T leaves us with a smaller tree T'. Moreover, removing v from $J(T)$ and $S(T)$ respectively gives us $J(T')$ and $S(T')$ respectively, the join and split trees corresponding to the smaller tree.

Input : Join Tree $J(T)$ and Split Tree $S(T)$ for Contour Tree T
Output : T, the Contour Tree

1 Choose leaf v of T such that $\delta^+(v) = 0$ in $J(T)$ and $\delta^-(v) = 1$ in $S(T)$ or $\delta^+(v) = 1$ in $J(T)$ and $\delta^-(v) = 0$ in $S(T)$
2 Without loss of generality, assume that v is an upper leaf of T (i.e. $\delta^+(v) = 0$ in $J(T)$ and $\delta^-(v) = 1$ in $S(T)$)
3 Let e be the edge of $J(T)$ incident to v
4 Let $J' = J(T) - e$ be the join tree with edge e and vertex v removed
5 **if** v is the global maximum in $S(T)$ **then**
6 | Let $S' = S(T) - v$ be the split tree with vertex v removed
end
7 **else**
8 | Let u, w be the vertices adjacent to v in $S(T)$
9 | Let $S' = S(T) - v + (u, w)$ be obtained by splicing (u, v) and (v, w)
end
10 Let T' be the tree computed by recursive invocation on J' and S'
11 Let $T = T' + e$ be computed by adding e to T'

Algorithm 7.3: Computing Contour Tree By Merging Join and Split Trees

By invoking this procedure recursively, we can assume that we know the smaller tree T', which allows us to construct T.

This algorithm depends on the ability to identify a leaf vertex in T. To do so, we note that the updegree of a given vertex v is always the same in the join tree $J(T)$ and the contour tree T itself. Similarly, the downdegree of v is always the same in the split tree $S(T)$ and the contour tree T. Thus, simply by examining vertex degrees in $J(T)$ and $S(T)$, we can ascertain the up- and down-degrees of T, even if we do not yet know what T is.

Moreover, if a vertex v is an upper leaf of T, it is also an upper leaf of $J(T)$, and the incident edge to v in $J(T)$ is guaranteed to be identical to the incident edge to v in T. Thus, even if we have yet to determine T, we can always determine an edge of T that is incident to some leaf v in T.

Although this algorithm is defined recursively, it is easy to convert to an iterative algorithm based on maintaining a queue of leaf vertices: details can be found in my M.Sc. thesis [Car00] or the corresponding journal paper [CSA03].

7.6.3 Computing the Unaugmented Contour Tree Directly

As described in the foregoing subsections, Algorithm 7.1 computes the fully augmented contour tree, in which most of the vertices are regular points with one edge leading upwards and one downwards. For many purposes, the unaugmented contour tree, with all regular points removed, is more useful, and can be computed in one of two ways: reduction from the fully augmented contour tree, or direct computation.

The first, and conceptually the easiest, is to examine the vertices of the contour tree one at a time. If a vertex v has exactly one upwards arc (u, v) and one downwards arc (v, w), then the vertex v is deleted, and (u, v) and (v, w) are spliced together to get (u, w). This is shown as pseudo-code in Algorithm 7.4

Input : Fully Augmented Contour Tree T
Output : Unaugmented Contour Tree T'

1 Let $T' = T$
2 **for** each vertex v in T' **do**
3 **if** v has one up-arc (u, v) and one down-arc (v, w) **then**
4   Delete v from R
5   Add (u, w) to R
  end
end

Algorithm 7.4: Reducing a Fully-Augmented Contour Tree

Input : Simplicial Mesh M with isovalue $f(v)$ at each vertex v
Output : R, the Unaugmented Contour Tree for M

1 Using Algorithm 7.6, compute the join tree $J(M)$
2 Using Algorithm 7.6, compute the split tree $S(M)$
3 Using Algorithm 7.7, augment $J(M)$ with all supernodes of $S(M)$
4 Using Algorithm 7.7, augment $S(M)$ with all supernodes of $J(M)$
5 Using Algorithm 7.3, merge the join tree $J(M)$ and the split tree $S(M)$ to obtain the unaugmented contour tree R

Algorithm 7.5: Computing the Unaugmented Contour Tree Directly

Alternately, the unaugmented contour tree can be computed directly. Doing so requires modifications to Algorithm 7.1 and Algorithm 7.2. The principal modification is to Algorithm 7.2 so that it ignores regular points. In doing so, however, it also ignores splits and local minima during the join sweep, or joins and local maxima during the split sweep. In order for Algorithm 7.3 to operate properly, both the join tree and split tree must contain all local extrema and forks. We achieve this by augmenting both join and split trees with the missing supernodes. Once we have done this, however, we do not need to modify the existing merge sweep algorithm.

In Algorithm 7.6, we show the modifications necessary to Algorithm 7.2. Instead of adding arcs to the join tree, we add superarcs, consisting of paths between supernodes in the join tree. During the sweep, we maintain the property that each component in the union-find structure corresponds to a unique component of $\{x : f(x) \geq h\}$ for the isovalue h representing the sweep. This correspondence is maintained by storing pointer to edges in the join tree in a structure $joinEdge$ indexed by the components in the union-find structure.

Instead of initializing each vertex in the union-find structure as a distinct component, we start by initializing all vertices to belong to a special *NULL* component in Step 2. This has two advantages: it avoids comparing isovalues at vertices, and it reduces the number of union operations to be performed. During the sweep, we will see later that new components are created at local maxima. For now, we assume that all higher-valued vertices already belong to a non-NULL component.

At each vertex, instead of checking the isovalue of each neighbour, we check the components. Neighbours with NULL components have not been processed yet, and therefore have lower isovalues and are skipped. Neighbours with non-NULL components have higher isovalues, and belong to already existing components.

Vertices are added to the first neighbouring component detected in Step 8. For a regular point, all

```
Input  : Simplicial Mesh M with function value f(v) at each vertex v
Output : J(R), the join tree for the unaugmented contour tree R; JoinSuperarc, the join
         superarcs for each v

 1 for each vertex v in M, in any order do
 2 |  Initialize Comp(v) = NULL to the null component
   end
 3 for each vertex v in M, in sorted order do
 4 |  nNeighbouringComponents = 0
 5 |  for each neighbour n of v, in any order do
 6 |  |  if Comp(n) = NULL then
 7 |  |  |  Do nothing: n has not yet been processed, and is lower than v
   |  |
 8 |  |  else if nNeighbouringComponents = 0 then
 9 |  |  |  Add v to Comp(n)
10 |  |  |  Increment nNeighbouringComponents
   |  |
11 |  |  else if Comp(n) ≠ Comp(v) then
12 |  |  |  Set lower end of joinEdge(Comp(n)) to v
13 |  |  |  Retrieve e_v = joinEdge(Comp(v))
14 |  |  |  Merge Comp(v) and Comp(n)
15 |  |  |  if nNeighbouringComponents = 1 then
16 |  |  |  |  Set lower end of edge e_v to v
17 |  |  |  |  Create joinEdge(Comp(v)) = (v, NULL)
   |  |  |  end
18 |  |  |  Increment nNeighbouringComponents
   |  |
19 |  |  else
20 |  |  |  Do nothing: v and n are already in the same component
   |  |  end
   |  end
21 |  if nNeighbouringComponents = 0 then
22 |  |  Let Comp(v) = v be a new component
23 |  |  Create new edge joinEdge(Comp(v)) = (v, NULL)
   |  end
24 |  Let JoinSuperarc(v) = joinEdge(Comp(v))
   end
25 Terminate the remaining join edge at the global minimum
```

Algorithm 7.6: Computing Join Tree for Unaugmented Contour Tree

other neighbouring vertices belong to the same component. But if a neighbouring vertex belongs to a new non-NULL component, we know that we have found a join at Step 11. When we detect a join at a vertex v, we retrieve the edge of the join tree for the neighbour's component, and terminate it at v (Step 12). If this is the first time we have detected a vertex at v, we must also terminate the edge stored for v's component, start a new edge at v and store the new edge on the component for v, as shown at Step 15.

If we iterate through all adjacent vertices without finding a non-NULL neighbour, we know that we have found a local maximum, and create a new component and corresponding edge at Step 23.

Finally, to aid in the later augmentation phase, we save the superarc to which each vertex belongs to in Step 24.

Input : Join and Split Trees $J(R), S(R)$ for Unaugmented Contour Tree R
$JoinSuperarc$, array of join superarcs to which each v belongs
Output : $J'(R) = J(R)$ augmented by all supernodes from $S(R)$

```
1 Let J'(R) = J(R)
2 for each supernode v in S(R), in sorted order do
3 |  if v is not in J'(R) then
4 |  |  Let (u,w) = JoinSuperarc(v) be the join superarc for v
5 |  |  Without loss of generality, f(u) > f(v) > f(w).
6 |  |  Add join superarc (u,v) to J'
7 |  |  Reset the upper end of (u,w) from u to v
   |  end
  end
```

Algorithm 7.7: Augmenting the Join Tree with Split Supernodes

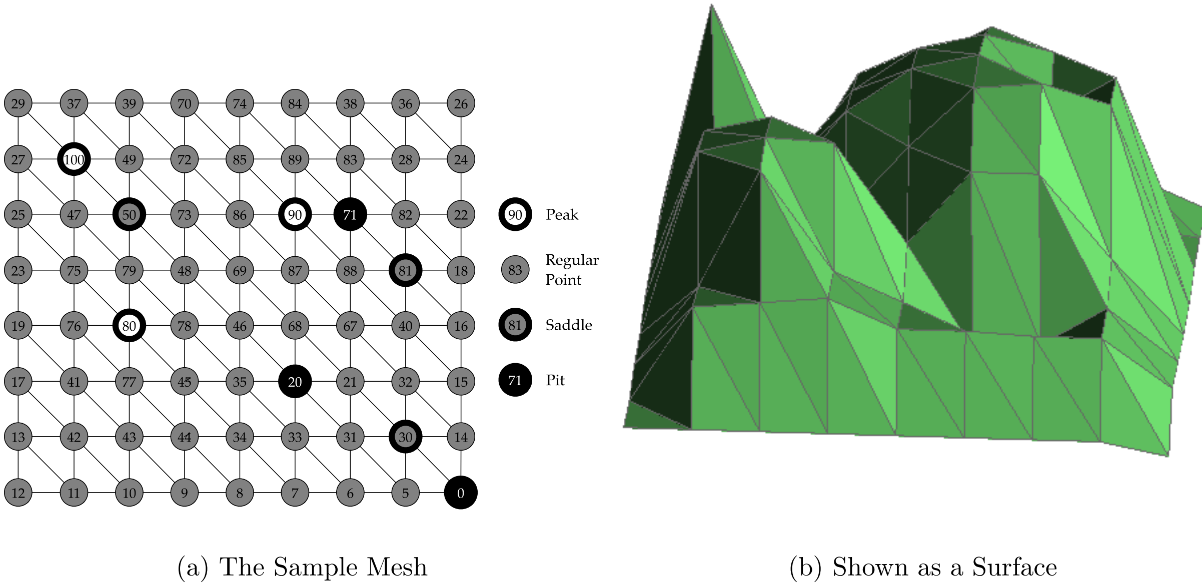
In the augmentation phase, shown as Algorithm 7.7, we check whether each split supernode (i.e. supernode in the split tree) is in the join tree. If not, we insert the split supernode in the join tree. To determine which join superarc the split supernode is inserted into, we refer to the *joinSuperarc* array. This identifies which join superarc the split supernode belonged to during the join sweep. We insert split supernodes in sorted order, as shown, and make sure that the old join superarc is retained as the lower of the two new join superarcs. This ensures $O(1)$ access time to the correct *joinSuperarc* for future split supernodes. If we did not retain the old superarc as the lower of the two, the next split supernode to be processed would retrieve the old join superarc from the *joinSuperarc*, and we would need a further data structure to find the correct new join superarc into which to insert it. By inserting split supernodes in sorted order, and always leaving the old join superarc as the lower of the two, we guarantee that the old join superarc will always be the correct join superarc for future insertions.

7.6.4 Variations on Sweep and Merge

The sweep and merge algorithm was extended by Pascucci & Cole-McLaughlin [PCM02] to trilinearly interpolated cubic meshes. These authors also add the Betti number computation from Pascucci [Pas01], and parallelize the computation of the join and split trees with a divide and conquer algorithm. The trilinear interpolant is modelled by providing an oracle which gives the join and split trees for individual cells. These individual contour trees were used as the input to a block-wise version of Algorithm 7.6. For each block, the reduced join and split trees are computed. To combine these trees into the reduced join and split trees for larger blocks, the union of the block-wise trees and the boundary meshes of the blocks is used as input, once again to Algorithm 7.6. By treating these blocks as individual cells in a larger mesh, we can show that this divide-and-conquer algorithm is itself a special case of the general algorithm for computing contour trees over general meshes.

Similarly, we will see that the Betti number computation is a special case of the algorithm in Chapter 10 for computing geometric properties using the contour tree. The Betti numbers are computed by augmenting all reduced trees with all Morse critical points, then making local revisions to the Betti numbers of each component during the merge phase.

Since this algorithm discards regular points early in the computation, the asymptotic running time is reduced to $O(N + n^{2/3} \log n + t\alpha(t))$. For computations such as the local spatial measures of Chapter 10, or the minimal seed sets discussed in Section 8.1, however, we will still need to compute the fully augmented



(a) The Sample Mesh

(b) Shown as a Surface

Figure 7.1: A Sample Mesh in Two Dimensions (Again)

contour tree, and this divide-and-conquer approach will not improve the asymptotic running time.

In a further variant, Chiang et al. [CLLR02] show that the join and split trees can be computed by using the path-ascent method of Takahashi et al. [TFT01] to compute a graph which is then used as the input graph to the sweep algorithm (Algorithm 7.2). Chiang & Lu [CL03] use the contour tree for simplifying large meshes. In their algorithm, they do not explicitly use the contour tree, relying instead on the join and split trees to determine whether two vertices belong to the same superarc. They test this by determining whether the two vertices belong to the same join superarc and the same split superarc. Since they do not require the contour tree to be extracted explicitly, this allows them to omit the merge phase.

7.7 Sample Contour Tree Computation

Since much of this thesis depends on a detailed understanding of the contour tree and how it is computed, we show a complete example in this section. We will compute the unaugmented contour tree for our two-dimensional sample data set using Algorithm 7.5. Figure 7.1 repeats the illustration of this mesh for ease of reference.

7.7.1 Join And Split Sweeps

In Figures 7.2 and 7.3, we show the join and split sweeps, respectively. Let us start by considering the join sweep in Figure 7.2. Each individual frame corresponds to one additional vertex being processed, with the level set passing through the vertex being shown. Moreover, the shaded area shows the region above this contour, containing all of the vertices and edges processed so far. At each local maximum or join, and at

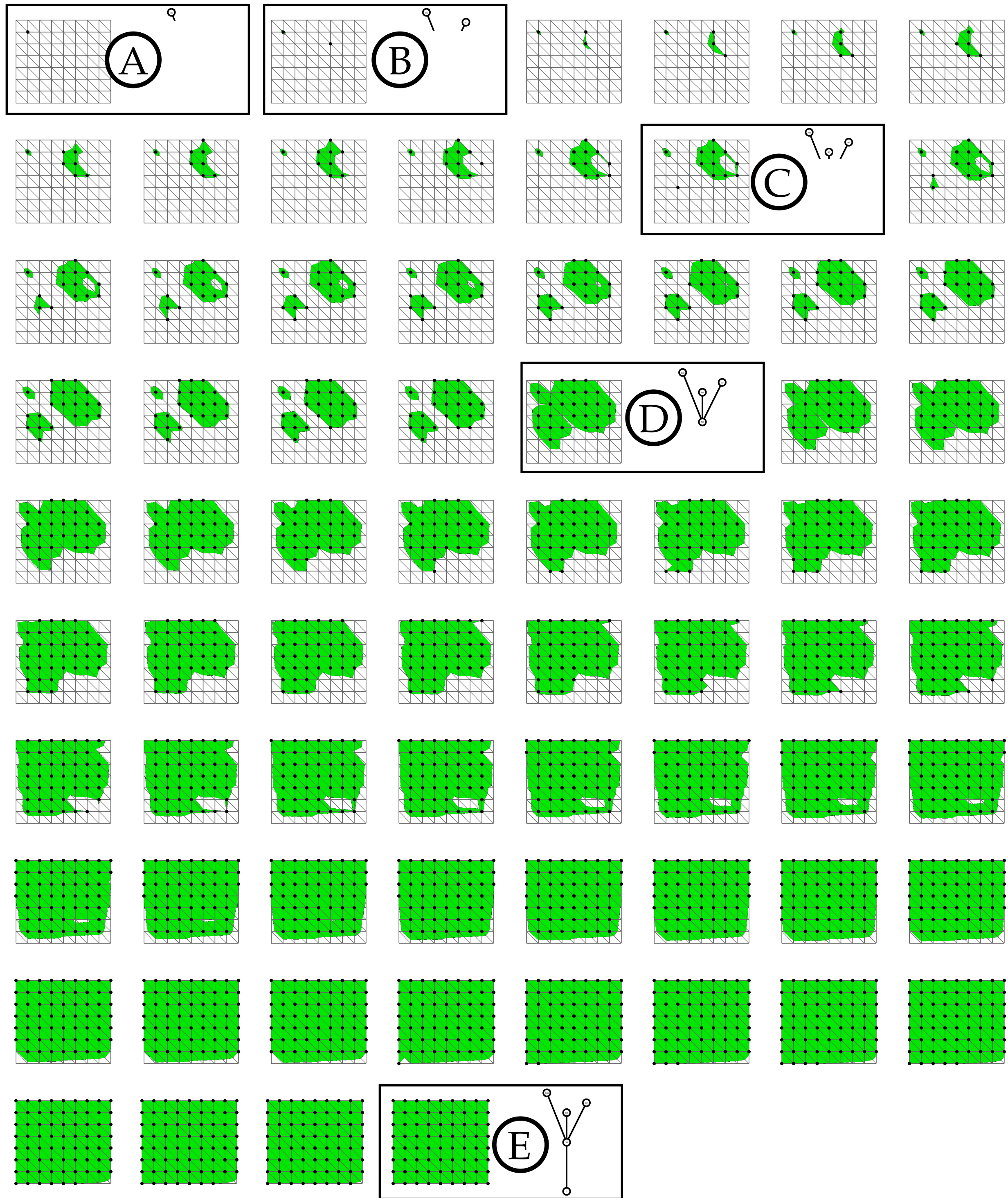


Figure 7.2: Sweep Through Sample Mesh to Compute Join Tree. At each step, one additional vertex is added to the union-find structure.
A: Local maximum detected at 100. Join superarc A created.
B: Local maximum detected at 90. Join superarc B created.
C: Local maximum detected at 80. Join superarc C created.
D: A, B, C meet at 50 and are terminated. Join superarc D created.
E: At global minimum 0, join superarc D terminated.

the global minimum, the reduced join tree is updated, as shown at letters 'A' through 'E'. 'A', 'B' and 'C' mark peaks at isovalue of 100, 90 and 80 respectively, at which new connected components are initialized. As additional vertices are added, the contours expand outwards until they meet, in this case, at 'D', which marks the saddle at 50 at which all three peaks meet. Finally, at 'E', the global minimum of the mesh, the join tree is completed with a single downward edge.

Note that, if we were computing the fully augmented join tree using Algorithm 7.2, the additional vertices would simply be strung out along the edges of the join tree.

Similarly, Figure 7.3 shows the computation for the split tree: in this case, there are three pits at 0 ('A'), 20 ('B'), and 71 ('D'), two splits at 30 ('C') and 81 ('E'), and the global maximum at 100 ('F'). Other than proceeding in reverse order, this sweep is identical to the previous one.

7.7.2 Augmenting the Join and Split Trees

When the join and split sweeps are complete, we have computed the join tree and split tree shown in Figure 7.4. Algorithm 7.7 is then used to compute the augmented join and split trees, also shown in Figure 7.4. At this point, we can verify that these tree do in fact have the correct up- and down-degrees respectively, by comparing them with the contour tree, as shown in Figure 7.5.

7.7.3 Merging the Join and Split Trees

Finally, we show Algorithm 7.3 in Figure 7.6. We transfer leaves from the join and split trees to the contour tree in arbitrary order. In this case, we have placed all upper leaves on the queue first, followed by all lower leaves. At each stage, the dark edges in the join and split trees indicate which edge is being transferred: from the join tree if the vertex is an upper leaf, from the split tree if it is a lower leaf, and which vertex is being reduced in the other tree. The dark edges in the contour tree indicate the known edges, while the grey edges indicate the unknown edges.

If we refer back to our recursive algorithm (Algorithm 7.3), the grey trees are J', S', and T' respectively: the inputs (and output) of the recursive call to compute the next smaller tree.

In Figure 7.6(a), the contour tree is as yet unknown. The edge $100 - 50$ has been selected from the join tree and transferred to the contour tree. Similarly, in Figure 7.6(b), edge $90 - 81$ has now been identified for transfer to the contour tree. Next, in Figure 7.6(c), $80 - 50$ is identified in the join tree for transfer to the contour tree. Since 80 is not a leaf in the split tree, it is reduced and replaced with the edge shown as a dotted line. In Figure 7.6(d), we start processing lower leaves of the tree: 0 is identified in the split tree, and transferred to the contour tree, followed by $20 - 30$, $71 - 81$, and $30 - 50$. Finally, in Figure 7.6(h), we identify and transfer the sole remaining edge: $81 - 50$.

If, instead of the unaugmented contour tree, we are computing the fully augmented contour tree, the merge phase will process every vertex, including regular points. The effect of this is to perform an isovalue sweep along one superarc at a time, sometimes upwards, sometimes downwards. This process is shown in detail in Figure 7.7. Initially, $100 - 50$ is transferred, which is equivalent to starting a contour at 100 and sweeping it downwards to 50, processing all vertices that the contour sweeps through (in this case none): this is shown in the first two images in the figure.

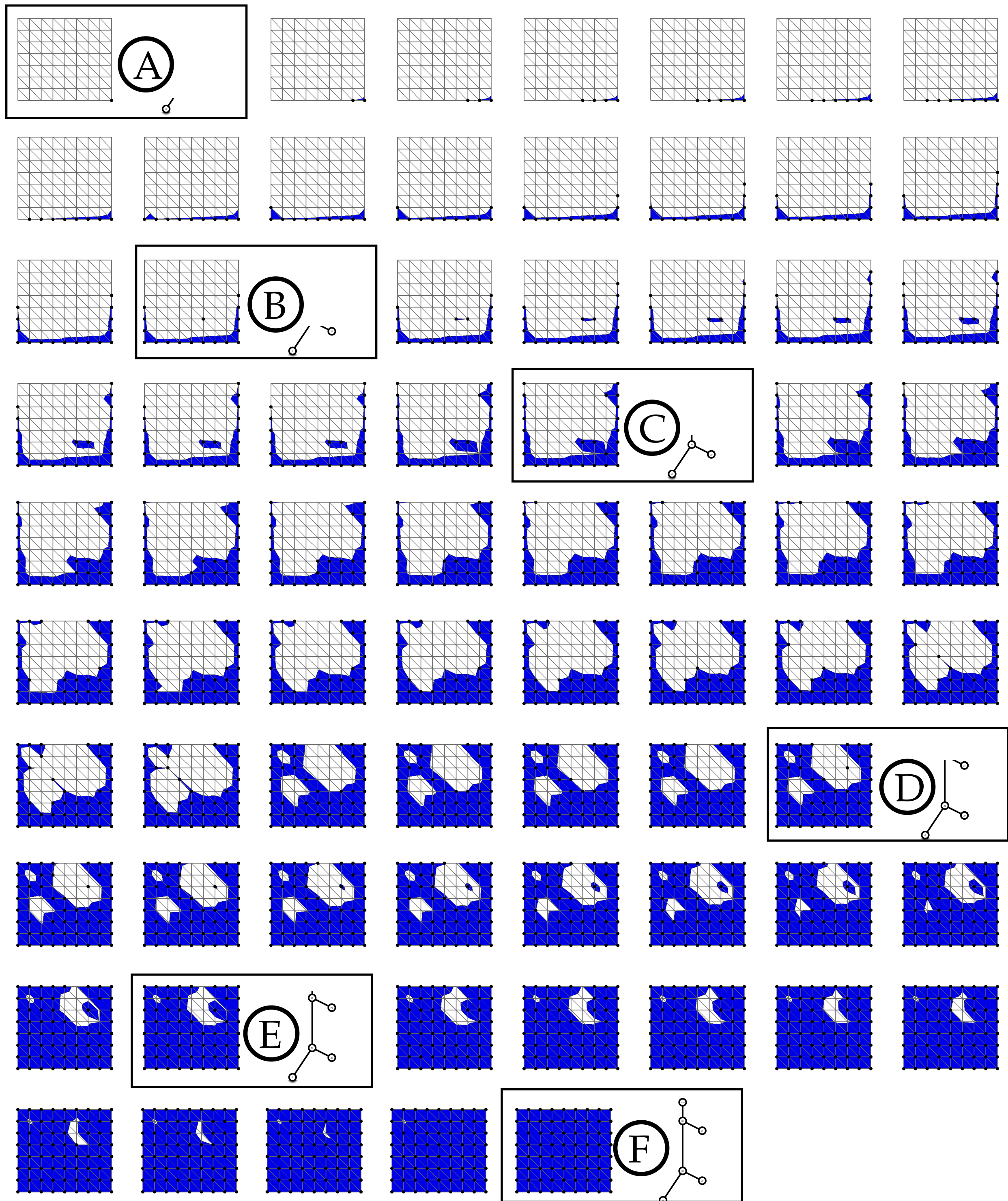


Figure 7.3: Sweep Through Sample Mesh to Compute Split Tree. As with the join tree, each step adds one vertex.
A: Local minimum detected at 0. Split superarc A created.
B: Local minimum detected at 20. Split superarc B created.
C: A, B meet at 30 and are terminated. Split superarc C created.
D: Local minimum detected at 71. Split superarc D created.
E: C, D meet at 50 and are terminated. Split superarc E created.
F: At global maximum 100, split superarc E terminated.

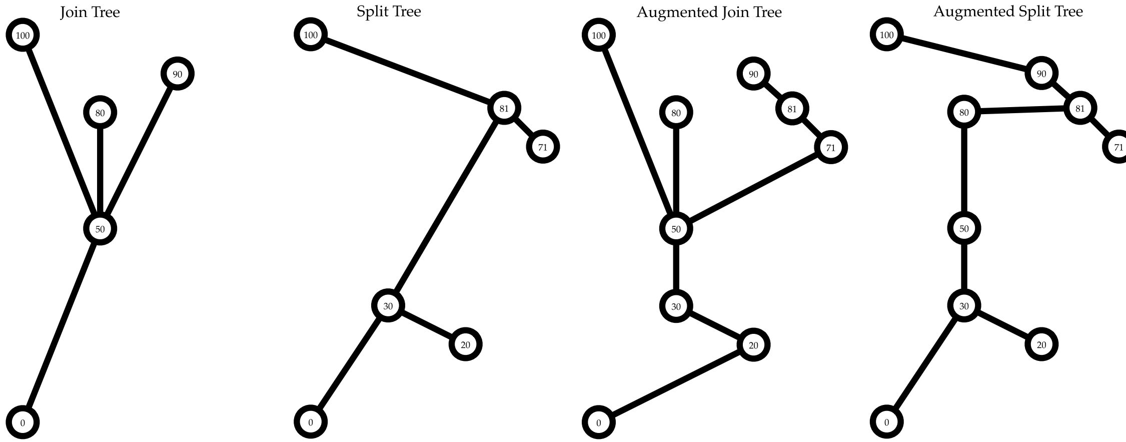


Figure 7.4: Augmenting the Join Tree and Split Tree Augmentation assures that all supernodes are in both the join tree and the split tree. This is achieved by inserting every node from the join tree into the split tree, and vice versa. In each case, the superarc along which the node is inserted can be found in the *joinSuperarc* or *splitSuperarc* arrays, respectively.

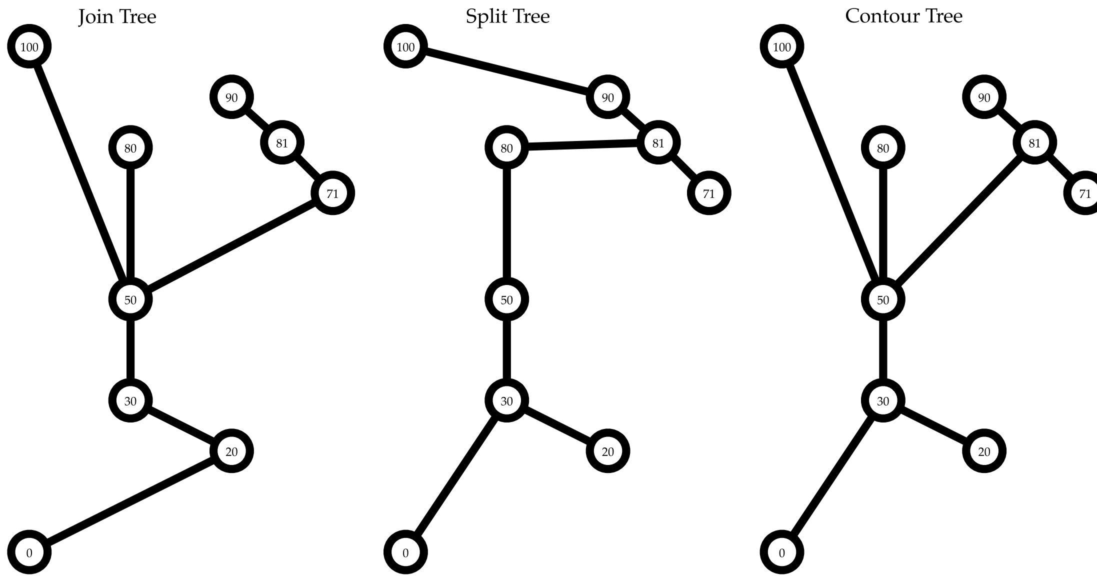
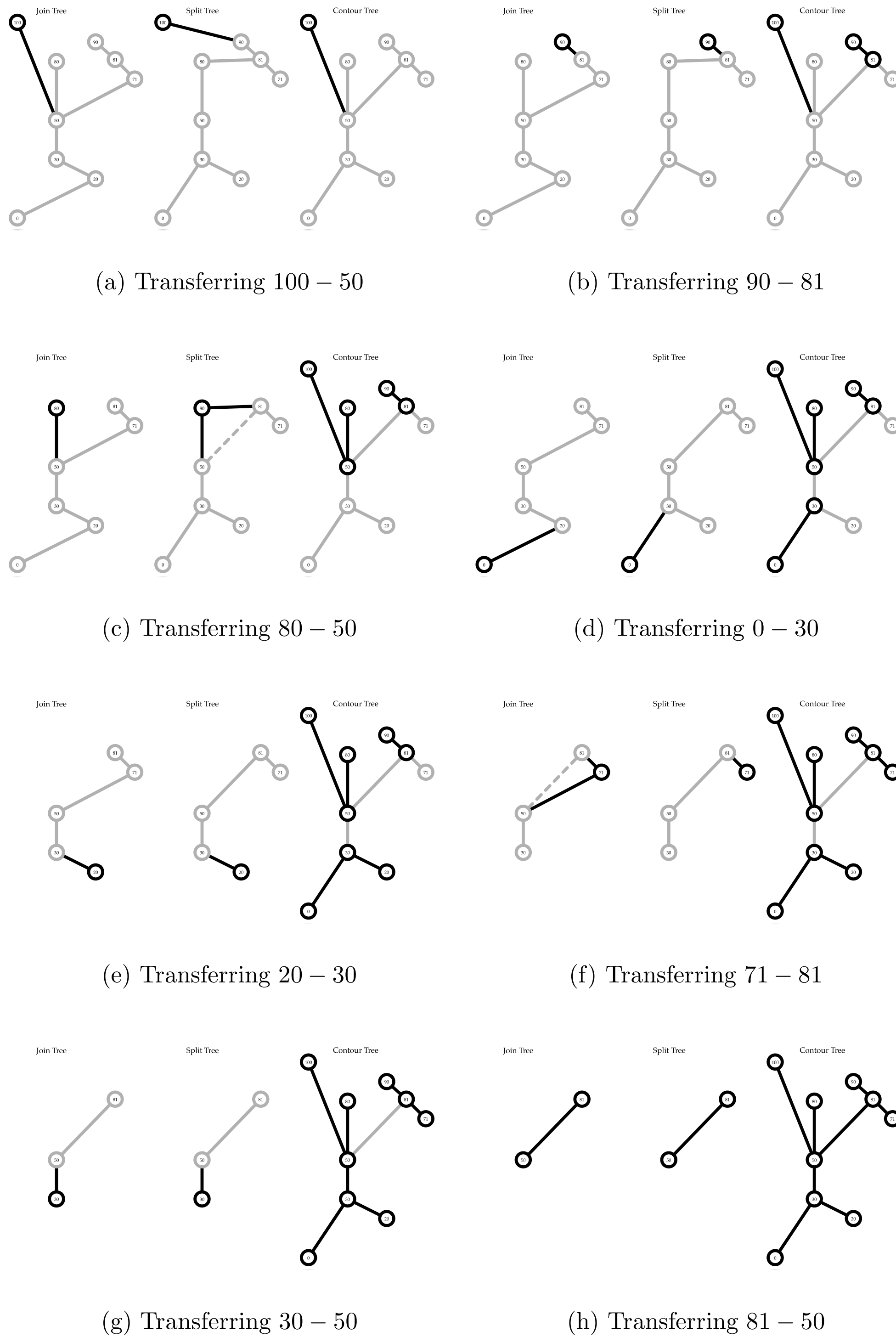


Figure 7.5: Comparison of Contour Tree, Join Tree and Split Tree. Note that the up-degree in the join tree of each vertex is identical to the up-degree in the contour tree. Similarly, the down-degree in the split tree matches the down-degree in the contour tree. We make use of this to determine which vertices are leaves of the contour tree. Also note that upper leaves in the join tree have the same incident arc as they do in the contour tree, and lower leaves in the split tree have the same incident arc as in the contour tree. We make use of this to determine which arc to transfer to the contour tree.

Next, $90-81$ is transferred, sweeping down from 90 to 81 through a total of 10 vertices, then $80-50$, which sweeps down through another 7 vertices. At this stage, we have one contour at 81 and two at 50. In general, the contours sweep until they reach a saddle, where they wait for other contours to arrive at the saddle. When only one superarc at a vertex has not been swept through, we can continue the sweep by combining all of the contours that have reached the saddle, and using the result to sweep down the remaining edge. This can most clearly be seen beginning at the point marked $71-81$, where the edge $71-81$



(a) Transferring 100 − 50 (b) Transferring 90 − 81

(c) Transferring 80 − 50 (d) Transferring 0 − 30

(e) Transferring 20 − 30 (f) Transferring 71 − 81

(g) Transferring 30 − 50 (h) Transferring 81 − 50

Figure 7.6: Merging the Join and Split Trees to Get the Contour Tree.
A black upper leaf in the join tree or a black lower leaf in the split tree indicates a vertex that is about to be transferred to the contour tree. In the other of these two trees, the black edge(s) indicate(s) the vertex to be removed, and the edges to be edited as a result. The grey edges show what remains of the join or split trees after the transfer is complete.
In the contour tree, the black edges indicate edges that have been transferred, and the grey edges indicate edges that are yet unknown. The grey edges in the join and split trees always correspond to the grey edges in the contour tree.

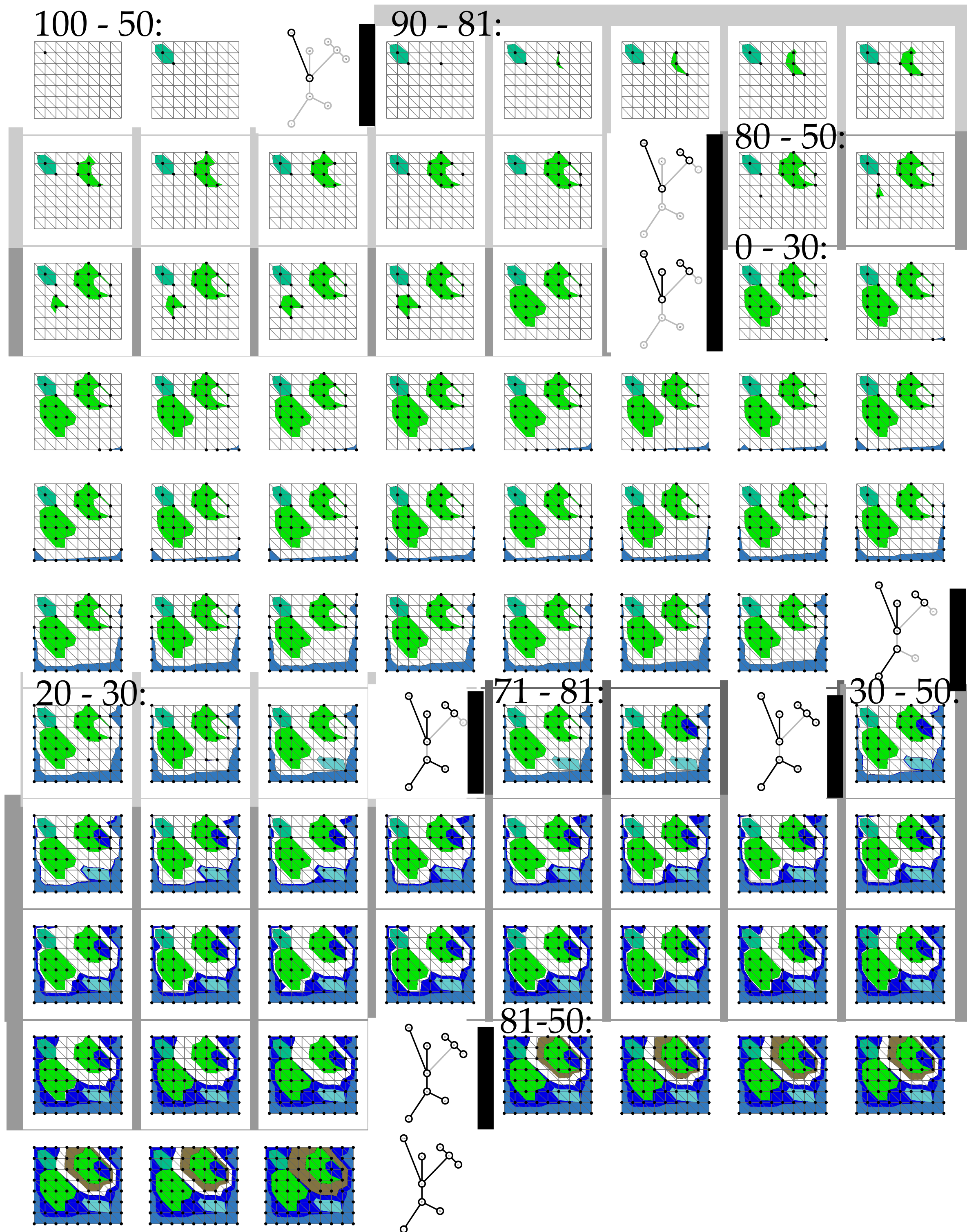


Figure 7.7: Merging To Get Fully Augmented Contour Tree. At each step, one additional vertex is processed, and the contour swept along the corresponding arc of the fully-augmented contour tree. Thus, processing each vertex is equivalent to a series of sweeps along superarcs of the contour tree. The sweeps along individual superarcs are separated by heavy black bars. When complete, each coloured region is the sweep region for a particular superarc, $\mathcal{R}(e)$.

is transferred by sweeping upwards to 81, at which vertex one contour is already waiting:- the contour from edge $90 - 81$. Since we have dealt with all but one of the contours that meet at 81, we can now process the outgoing edge, although in this example, it happens later, at the stage marked $81 - 50$.

Chapter 8

Isosurface Seed Generation

The previous chapter discussed algorithms for constructing the contour tree, and noted that it can be used to generate seeds for use with the continuation method for isosurface extraction. In this chapter, we add the next contribution: *path seeds*, a simple and efficient method of generating isosurface seeds using the contour tree.

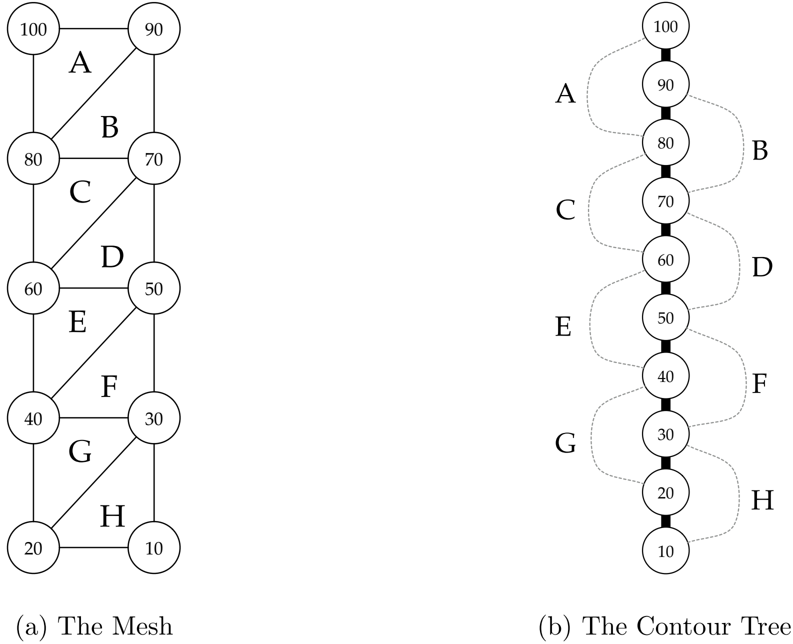
Since van Kreveld et al. [vKvOB+97] also showed how to generate seeds from the contour tree, we start by reviewing their method in Section 8.1, then introduce path seeds in Section 8.2. Finally, in Section 8.3, we summarize the chapter, and compare path seeds and minimal seed sets.

8.1 Minimal Seed Sets

In addition to defining an algorithm for constructing the contour tree, van Kreveld et al. [vKvOB+97] showed how to use it to generate a *minimal seed set*: a set of seed cells that was guaranteed to intersect each possible contour at least once. To determine the minimal set of seed cells, they observed that each cell is a valid seed for every contour that intersects it. Since each contour is represented by a point in the contour tree, each cell in the mesh corresponds to a set of points in the contour tree. More precisely, in a simplicial mesh, each cell intersects exactly one contour for each isovalue between the highest and lowest vertices of the cell. By Corollary 6.1, this edge is equivalent to the corresponding monotone path in the tree. Thus, the contour tree can be annotated with a set of edges, each representing a single cell in the mesh. Finding a minimal seed set now reduces to a graph matching problem: finding the smallest set of these monotone paths that covers all arcs in the contour tree. We will show a complete example of this algorithm in Section 8.1.2.

This algorithm has several undesirable properties. First, although the minimal seed set can be computed in polynomial time, van Kreveld et al.[vKvOB+97] do not specify the polynomial. Instead, they present an approximation algorithm which requires linear storage and $O(n \log^2 n)$ time in two dimensions, linear storage and $O(n^2)$ time in higher dimensions. This approximate algorithm guarantees that no more than twice the minimum number of seed cells are chosen.

Second, the size of this seed set can be significant: at least $\Omega(t)$, and potentially as much as $\Theta(n)$ for a t-supernode contour tree in a n-vertex mesh. Proof is deferred to Section 8.1.1.



(a) The Mesh (b) The Contour Tree

Figure 8.1: A Triangulation Requiring a Minimal Seed Set of Size $\Theta(n)$. A and H must be in the seed set to represent the ranges $90 - 100$ and $10 - 20$. Either C or D must then be present to represent the range $70 - 80$, and so on.

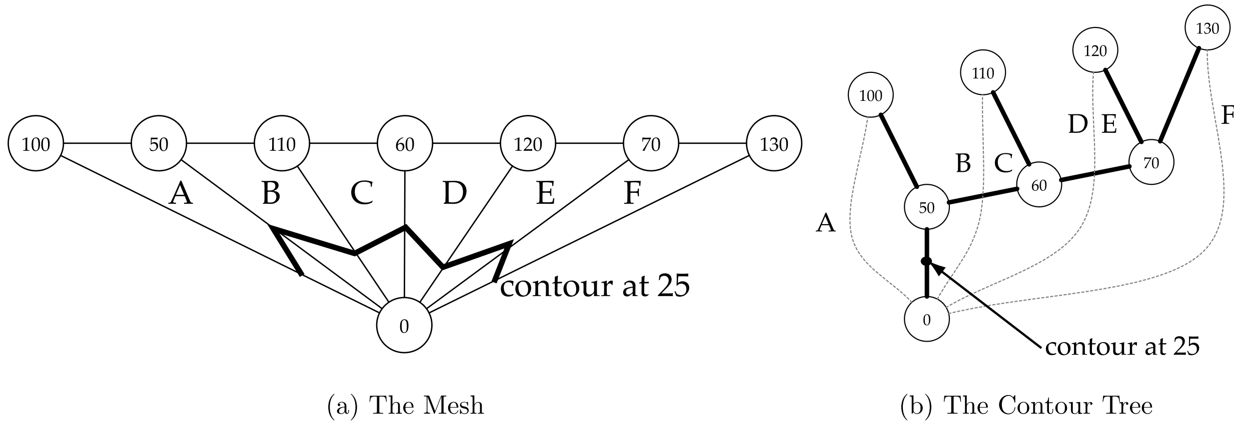
And third, although the contour tree is used to construct the seed set, the correspondence noted between the superarcs of the contour tree and the individual contours in the image is lost. For each given contour, we may in fact have more than one seed cell, which makes it more difficult to relate contours in the image to the superarcs to which they correspond. Moreover, since redundant seed cells for a given contour need not be near each other, we must mark which cells have been visited, in order to guarantee that the same contour is not extracted multiple times. In two-dimensional data sets, where marking cells is not necessary for contour extraction, this adds $O(n)$ additional memory overhead to contour extraction. In higher dimensions, the continuation method needs to mark cells which it has visited anyway, so there is no additional memory required.

8.1.1 Properties of Minimal Seed Sets

We just claimed that minimal seed sets have several undesirable properties. We now prove these properties as a lemma:

Lemma 8.1 *The minimal seed set for a n-vertex mesh with a t-supernode contour tree includes at least $t/4$ seed cells, and in the worst case may contain $\Omega(n)$. In addition, the requirement that every contour intersect at least one seed cell may cause some contours to intersect $\Omega(n)$ seed cells.*

Proof: In a simplicial mesh, each cell can contain at most one local minimum and one local maximum, or at most two local extrema. Since the local extrema are the leaves of a tree of size t, there are at least $t/2$ and at most t local extrema. Therefore, the minimal seed set must include at least $t/4$ seed cells.



(a) The Mesh (b) The Contour Tree

Figure 8.2: A Triangulation Where Some Contours Intersect $\Omega(n)$ Seed Cells. In this mesh, some contours such as the one shown intersect every cell. Since there are $n/2$ distinct peaks, we need $n/2$ seed cells, each of which intersects the contour shown.

Consider the mesh shown in Figure 8.1(a): an inclined strip of triangles labelled A - H. Figure 8.1(b) shows the corresponding contour tree with paths representing seed cells. None of these paths representing seed cells covers more than two arcs in the contour tree. Since the contour tree in this case has 1 superarc consisting of $n-1$ arcs, there must be at least $(n-1)/2$ seed cells in any minimal seed set for this mesh.

To see that some contours may intersect $\Omega(n)$ seed cells, consider Figure 8.2, which shows a small mesh and the corresponding contour tree. Any minimal seed set must contain A, one of B or C, one of D or E, and F. Moreover, we can build similar meshes of arbitrary size, where any minimal seed set must contain $1+n/2$ seed cells. Observe that the contour at 25 intersects every single seed cell, so, in a similar mesh of n vertices, we can have $\Omega(n)$ seed cells intersect a single contour. □

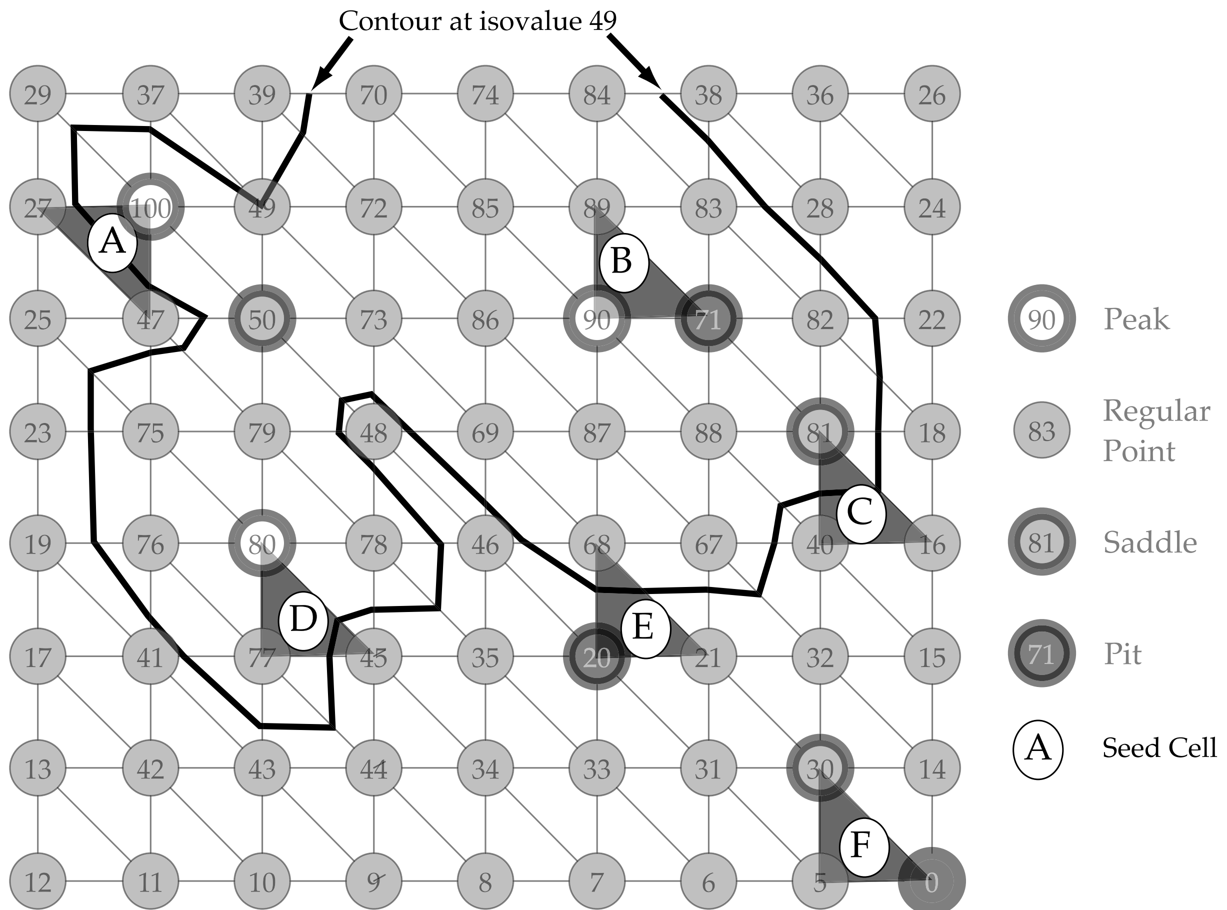


Figure 8.3: Sample Mesh, Showing Minimal Seed Set and a Contour. Here, 6 seed cells are sufficient to intersect every contour, but some contours intersect as many as 4 of the seed cells.

8.1.2 Example of a Minimal Seed Set

To illustrate the computation of a minimal seed set, we return to our sample two-dimensional mesh, shown once more in Figure 8.3. Figure 8.4 shows the fully augmented contour tree, annotated with edges representing the potential seed cells.

For example, consider the cell marked A in Figure 8.3: it intersects a set of contours in the range $(27, 100)$, and corresponds to the heavy grey edge marked A in Figure 8.4. This edge covers all of superarc $(100, 50)$, all of superarc $(50, 30)$, and part of superarc $(30, 0)$. Similarly, each and every cell in Figure 8.3 corresponds to an edge in Figure 8.4.

In this example, one possible minimal seed set is shown in Figure 8.4: the edges marked A to F. The corresponding cells are also marked A to F in Figure 8.3. This particular seed set also illustrates the problem noted above of redundant seed cells. If we wish to extract the contour at isovalue 49, no fewer than 4 of our seed cells intersect it. As noted above, this makes it difficult to take advantage of the one-to-one correspondence between contours and points in the contour tree.

In the next section, we describe an improvement to the minimal seed set approach: path seeds,

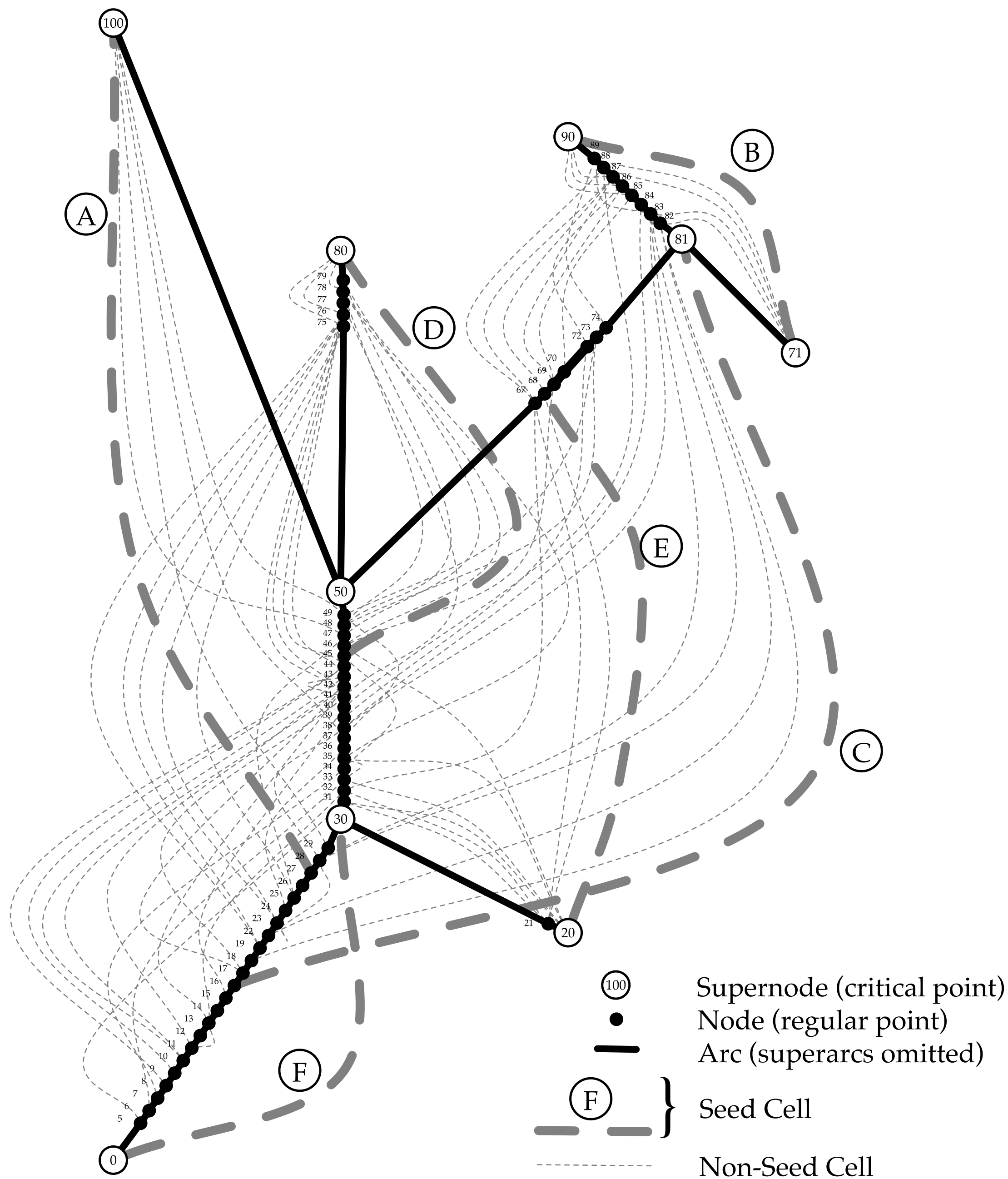


Figure 8.4: Annotating The Contour Tree with Monotone Paths to Find Seed Cells. In this figure, the fully-augmented contour tree for Figure 8.3 is shown with a dotted path added for each cell in the mesh. The paths marked with letters were chosen as a minimal seed set, and the corresponding cells marked in Figure 8.3.

which we generate directly from the contour tree.

8.2 Path Seeds

The distance is nothing: it is only the first step that is difficult: (the Marquise du Deffand (1697-1780), remarking on the legend that St. Denis walked two leagues carrying his head after it was cut off)

In the previous section, we reviewed the construction of minimal seed sets from the contour tree, and noted some undesireable properties. In this section, we introduce *path seeds* to generate exactly one seed for each possible contour directly from the contour tree.

We start with an intuitive description of a path seed in Section 8.2.1, followed by a formal definition in Section 8.2.2, some useful properties in Section 8.2.3, algorithms for working with path seeds in Section 8.2.4, and a detailed example in Section 8.2.5.

8.2.1 Description of Path Seeds

In Section 8.1, we described the minimal seed set, which consists of a set of seed cells that intersect every possible contour at least once. We noted, however, that minimal seed sets are complex and expensive to compute, and generate redundant seed cells which make it difficult to take advantage of the one-to-one correspondence between contours and points in the contour tree.

Instead of seed cells, we will generate f-monotone paths through the mesh, each of which intersects only the set of contours represented by a given superarc. We call these paths *seed paths*. Each seed path starts at a critical point, and ascends (or descends) through the set of contours represented by the given superarc until the isovalue of the other end of the superarc is reached.

Although any f-monotone path through these contours will suffice, we prefer to choose a path consisting of edges of the mesh. These seed paths are similar to the f-monotone paths used by Takeshima et al. [TIS$^+$95] and Chiang et al. [CLLR02] to compute the contour tree. Unlike the paths used by these authors, however, seed paths need not extend to a local extremum, need not take the steepest descent (although that will often be the most efficient), and need not terminate at a critical point. Finally, note that these paths are computed only as needed, and are not used to compute the contour tree itself.

For each superarc, we will show that, provided that the first edge is chosen correctly, any f-monotone sequence of edges through the mesh will generate a suitable seed path. All that then remains is to determine the first edge for each superarc. We call such edges *seed path seeds*, or *path seeds* for short.

To extract path seeds, we take advantage of the join and split sweeps described in Section 7.6.1. In the join sweep, a join is detected when two components merge in the union-find structure. This happens when a vertex has higher-valued neighbours that belong to two distinct components. This in turn implies that there are two distinct directions of ascent from the vertex identifiable with the two higher-valued neighbours. Computing path seeds merely requires remembering the distinct directions of ascent, and carrying them through the merge phase of the algorithm.

To extract a specific contour using path seeds, we identify the superarc to which the contour belongs, take the path seed for that superarc, and generate a seed path through the mesh until we reach the desired isovalue. When an edge of the path intersects the desired isovalue, we use that edge as a seed edge for the

continuation method. Thus, instead of storing a minimal seed set, which may be $O(n)$ in size, we store a single piece of information for each superarc, reducing the space required to $O(t)$.

8.2.2 Definition of Path Seeds

We will start by defining a seed path as follows:

Definition 8.1 *A* seed path *for a superarc s is a f-monotone path $P : (0,1) \to \mathcal{R}(s)$ such that for every contour c that belongs to s, there is some $x \in (0,1)$ so that $P(x)$ belongs to c.*

From this, we can define a (seed) path seed:

Definition 8.2 *A* seed path seed *or* path seed *for a superarc s is a pair $(\mu, \vec{v})$ consisting of a supernode μ to which s is incident and a non-zero vector $\vec{v}$ such that the path $Q = \{\mu + k\vec{v} : k \in (0,1]\}$ is a f-monotone path contained in $\mathcal{R}(s)$.*

This definition specifies the direction of departure from the supernode as a vector, which need not itself be a seed path. This vector merely specifies the first step to take before ascending or descending through $\mathcal{R}(s)$. Note that not all vectors $\vec{v}$ at μ will form valid path seeds with μ, either because they lie on contours, or because f is not monotone with respect to $\vec{v}$. Under our assumptions that f is given by barycentric interpolation over a simplicial mesh with unique vertex isovalues, however, this does not pose a problem. Even for fields where no vector satisfies the definition, we can substitute a path that follows the gradient of the field. For simplicity of treatment, we will simply assume that a vector suffices.

8.2.3 Properties of Path Seeds

There are two important properties of path seeds that we will need: that we can generate seed paths from path seeds, and that we can generate isosurface seeds from seed paths for use with the continuation algorithm for isosurface extraction. Also, under Assumptions 1, 2, and 3 on page 42, any edge (λ, μ) incident to μ in the mesh generates a valid path seed, either upwards or downwards.

Lemma 8.2 *Given a path seed $(\mu, \vec{v})$ for the superarc $s = (\lambda, \mu)$, there exists a seed path P for s which leaves μ in direction $\vec{v}$.*

Proof: We will prove this by constructing a suitable seed path P from the path seed $(\mu, \vec{v})$. Without loss of generality, assume that μ is the lower supernode of the superarc s. From Definition 8.2, we take the monotone path Q from μ to $\mu + \vec{v}$, excluding μ. Since μ is the lower supernode of s, we know that Q must be an ascending path. We let Q be the beginning of path P. From $\mu + \vec{v}$, we let P follow any ascending path R until the isovalue $f(\lambda)$ is reached, excluding the isovalue $f(\lambda)$ itself. Since both Q and R are ascending paths, P must also be an ascending path for the isovalues $(f(\mu), f(\lambda))$.

From Corollary 6.1, we know that P maps to a monotone path P' in the contour tree. Since each point on Q belongs to s, then P' must start by ascending s. Therefore P' must continue ascending s at least

until the isovalue $f(\lambda)$. But we know that P stops short of $f(\lambda)$. So every point $x \in P$ must map to some contour c_x which in turn maps to s, and Definition 8.1 is satisfied. □

We next need to show that, given a seed path, we can initialize the continuation algorithm with a suitable seed cell.

Corollary 8.3 *Let $c = (s, h)$ be a contour belonging to a superarc $s = (\lambda, \mu)$ and let P be any seed path for s. Then every cell K that contains the intersection of P and c is a valid seed cell for c.*

Proof: By Definition 8.1, there is some point x on P that belongs to c. Moreover, since P is monotone, there is only one such point x, so x *is* the intersection of P and c. Let K be any cell that contains x. Since x is on c, the intersection of K and c is non-empty, and it follows that K is a valid seed cell for c. □

Finally, we show that edges in the mesh make satisfactory path seeds.

Lemma 8.4 *Given a supernode μ of a contour tree, and an edge $e = (\xi, \mu)$ incident to μ in the mesh, there exists some $k_0 > 0$ and a superarc s incident to μ such that $(\mu, k(\xi - \mu))$ is a valid path seed s.*

Proof: From Assumption 3, we know that $f(\xi) \neq f(\mu)$. Without loss of generality, assume that $f(\xi) > f(\mu)$, and let $\vec{v} = \xi - \mu$. We need to show that $Q = \mu + k \times \vec{v} : k \in (0, k_0)$ is contained in $\mathcal{R}(e)$.

Let Q_0 be the path defined by edge e. From Assumption 2, we know that any line segment in any cell of the mesh is linear and therefore f-monotone in $\mathcal{R}$. And since e is an edge in the mesh, it is wholly contained in some simplex K. Since f is linear along edge e and $f(\xi) > f(\mu)$, it follows that Q_0 is an ascending path. From Corollary 6.1, we know that a corresponding ascending path R exists in the contour tree between μ and ξ.

Let $s = (\lambda, \mu)$ be the first superarc in this path R. If $f(\lambda) > f(\xi)$, let $k_0 = 1$, and by Definition 8.2, the result follows, since $Q = Q_0$ must be contained in $\mathcal{R}(s)$. If $f(\lambda) < f(\xi)$, we use only part of the edge e. Choose $k_0 < \frac{f(\lambda) - f(\mu)}{f(\xi) - f(\mu)}$, and the result also follows. □

With this lemma in hand, we can now treat edges incident to supernodes as path seeds. For the purposes of the algorithms in Section 8.2.4, this formulation is more natural, and we will simply assume that all of our path seeds are edges of the mesh.

These path seeds are efficient to store, requiring $\Theta(t)$ space:

Lemma 8.5 *Path seeds for contour extraction require $\Theta(t)$ storage for a t-supernode contour tree in a n-node mesh.*

Proof: For each superarc s in the contour tree, one path seed is stored. This path seed is sufficient to extract all contours belonging to s, by Corollary 8.3 and Lemma 8.2. Since we have a t-supernode contour tree, we have $t - 1$ superarcs, and the result follows. □

We will see some time bounds in the next subsection, once we have formally defined algorithms to compute and use path seeds.

8.2.4 Path Seed Algorithms

Now that we have defined path seeds, and demonstrated that they can be used to generate isosurfaces seeds for the continuation method, we define algorithms to extract path seeds for each superarc of the contour tree, and to generate seed paths and isosurface seeds.

To extract the path seeds in the first place, we note that, except at saddle points, any ascending (or descending) direction is satisfactory, because there is at most one distinct set of contours through which we can ascend or descend. For saddle points, where more than one direction of ascent (or descent) is possible, consider a join. At a join, there are at least two distinct directions to ascend. However, we detected these joins during the join sweep of the contour tree algorithm precisely because it had higher neighbours belonging to two distinct connected components. And each of these directions corresponds to a distinct direction of ascent.

Thus, to extract path seeds, all we have to do is to keep track of the edges by which we detected the join in the first place. These will suffice as path seeds. As a result, we modify Algorithm 7.6 to the algorithm stated in Algorithm 8.1: note that all we have done is to add Step 9 and Step 17.

No modifications to the augmentation algorithm (Algorithm 7.7) are required. In the merge algorithm, we add an explicit check for path seeds when we transfer an edge to the contour tree in Step 12: otherwise, the algorithm is unchanged.

Theorem 8.6 *Algorithm 8.3, correctly computes the unaugmented contour tree with valid path seeds for every superarc s.*

Proof: We start by observing that Algorithm 8.3 is essentially the same as Algorithm 7.5, which computed the unaugmented contour tree directly. The only changes are to substitute Algorithm 8.1 for Algorithm 7.6 and to substitute Algorithm 8.2 for Algorithm 7.3. In each of the substituted algorithms, we have merely added code to store and transfer the path seeds, without affecting the computation of the unaugmented contour tree itself.

From Step 9 and Step 17 of Algorithm 8.1, we know that every edge of the join tree incident to a join has a path seed attached to it at the lower end. During the merge phase, these path seeds are transferred to the contour tree.

Without loss of generality, assume that we are transferring a superarc $s = (\lambda, \mu)$ to the contour tree from the join tree. We are therefore transferring an edge $e = (\lambda, \mu)$ where μ is the supernode at the lower end of the superarc s. From Lemma 8.4, we know that these edges are equivalent to valid path seeds.

To see that the path seeds transferred are valid path seeds for the superarc to which they are attached, augment the join tree by the vertex ξ. Without loss of generality, we assume that $f(\lambda) > f(\xi) > f(\mu)$, using the same construction as in Lemma 8.4.

The transfer of s to the contour tree can then be decomposed into two transfers: the transfer of (λ, ξ) and the transfer of (ξ, μ). If we then reduce the contour tree at ξ, we will end up with ξ belonging to the superarc s, as required.

If no path seed is attached to (λ, μ), Step 12 generates a path seed by choosing an arbitrary edge in

Input : Triangulation T with function value $f(v)$ at each vertex v

Output : $J(U)$, the join tree for the unaugmented contour tree U; $JoinSuperarc$, the join superarcs for each v

```
1  for each vertex v in T, in any order do
2  |  Initialize Comp(v) = NULL to the null component
   end
3  for each vertex v in T, in sorted order do
4  |  nNeighbouringComponents = 0
5  |  for each neighbour n of v, in any order do
6  |  |  if Comp(n) = NULL then
7  |  |  |  Do nothing: n is lower than v
8  |  |  else if nNeighbouringComponents = 0 then
9  |  |  |  Store path seed v → n on joinEdge(Comp(n))
10 |  |  |  Add v to Comp(n)
11 |  |  |  Increment nNeighbouringComponents
12 |  |  else if Comp(n) ≠ Comp(v) then
13 |  |  |  Set lower end of joinEdge(Comp(n)) to v
14 |  |  |  Retrieve e_v = joinEdge(Comp(v))
15 |  |  |  Merge Comp(v) and Comp(n)
16 |  |  |  if nNeighbouringComponents = 1 then
17 |  |  |  |  Store path seed v → n on joinEdge(Comp(n))
18 |  |  |  |  Set lower end of edge e_v to v
19 |  |  |  |  Create joinEdge(Comp(v)) = (v, NULL)
   |  |  |  end
20 |  |  |  Increment nNeighbouringComponents
21 |  |  else
22 |  |  |  Do nothing: v and n are already in the same component
   |  |  end
   |  end
23 |  if nNeighbouringComponents = 0 then
24 |  |  Let Comp(v) = v be a new component
25 |  |  Create new edge joinEdge(Comp(v)) = (v, NULL)
   |  end
26 |  Let JoinSuperarc(v) = joinEdge(Comp(v))
   end
27 Terminate the remaining join edge at the global minimum
```

Algorithm 8.1: Computing Reduced Join Tree With Path Seeds

the mesh ascending from μ. But, since no path seed is attached to (λ, μ), μ cannot be a join, so all directions of ascent must lead through the same set of contours: those represented by s. It follows that the arbitrary edge we chose is a satisfactory path seed. □

Once we have extracted the path seeds, Algorithm 8.4 shows the algorithm for generating isosurface seeds. This algorithm simply implements Lemma 8.4, Lemma 8.2, and Corollary 8.3 by choosing a new ascending vertex at each edge in the mesh.

Theorem 8.7 *For any superarc $s = (\lambda, \mu)$ and isovalue h strictly between $f(\lambda)$ and $f(\mu)$, Algorithm 8.4 generates a valid seed for the h-isovalued contour c that belongs to s.*

Input : Join Tree $J(C)$ and Split Tree $S(C)$ for Contour Tree C
Output : C, the Contour Tree

1 Choose leaf v of C such that $\delta^+(v) = 0$ in $J(C)$ and $\delta^-(v) = 1$ in $S(C)$ or vice versa
2 Without loss of generality, assume that v is an upper leaf of C (i.e. $\delta^+(v) = 0$ in $J(C)$ and $\delta^-(v) = 1$ in $S(C)$)
3 Let e be the edge of $J(C)$ incident to v
4 Let $J' = J(C) - e$ be the join tree with edge e and vertex v removed
5 **if** v is the global maximum in $S(C)$ **then**
6 | Let $S' = S(C) - v$ be the split tree with vertex v removed
7 **else**
8 | Let u, w be the vertices adjacent to v in $S(C)$
9 | Let $S' = S(C) - v + (u, w)$ be obtained by splicing (u, v) and (v, w)
end
10 Let C' be the tree computed by recursive invocation on J' and S'
11 Let $C = C' + e$ be computed by adding e to C'
12 **if** e does not have a path seed attached to it **then**
13 | Choose any edge (u, v) incident to v in the mesh such that $f(u) > f(v)$
14 | Add (u, v) to e as a path seed
end

Algorithm 8.2: Computing Contour Tree By Merging Join and Split Trees

Input : Simplicial Mesh M with isovalue $f(v)$ at each vertex v
Output : U, the unaugmented contour tree for M

1 Using Algorithm 8.1, compute the join tree $J(M)$ with path seeds
2 Using Algorithm 8.1, compute the split tree $S(M)$ with path seeds
3 Using Algorithm 7.7, augment $J(M)$ with all supernodes of $S(M)$
4 Using Algorithm 7.7, augment $S(M)$ with all supernodes of $J(M)$
5 Using Algorithm 8.2, merge the join tree $J(M)$ and the split tree $S(M)$ to obtain the unaugmented contour tree U

Algorithm 8.3: Computing the unaugmented contour tree With Path Seeds

Proof: From Lemma 8.4, we know that the input edge e is a valid path seed for s, ascending from μ into $\mathcal{R}(s)$.

We claim that each time through the loop at Step 7, we always locate an ascending edge in Step 9. Assume not. Then, for the current value of *here*, there are no ascending edges incident. This implies that *here* is a local maximum. As such, *here* must be a supernode, so *here* $= \lambda$, the upper end of the superarc s. Before we started Step 7, we executed Step 3, at which point *there* referred to the vertex now referred to as *here*. In other words, when we executed Step 7, *there* referred to λ, at which time $f(\textit{there}) = f(\lambda)$. But we know that $h < f(\lambda)$, so the comparison in Step 7 must have returned false, and Step 9 was never reached.

Since each edge in the mesh is a f-monotone path, it follows that the sequence of edges generated by Algorithm 8.4 is an ascending path. Since Algorithm 8.4 follows the construction in Corollary 8.3, it follows that the edge s is a valid seed edge for the contour c. □

Now that we have algorithms for working with path seeds, we can look at the asymptotic cost of using these algorithms.

Input : A superarc $s = (\lambda, \mu)$ of a contour tree C
An edge $e = (\xi, \mu)$ that is equivalent to a *path seed* by Lemma 8.4
An isovalue h between $f(\lambda)$ and $f(\mu)$
Without loss of generality, $f(\lambda) > h > f(\mu)$
Output : A seed edge s for the contour c at h that belongs to s.

```
1  here = μ
2  there = ξ
3  if f(there) < h then
4  |  here = there
5  |  foundAscendingEdge = false
6  |  Let (here, there) be the first edge incident to here
7  |  while not foundAscendingEdge do
8  |  |  if f(there) > f(here) then
9  |  |  |  foundAscendingEdge = true
   |  |  end
10 |  |  else
11 |  |  |  let (here, there) be the next edge incident to here
   |  |  end
   |  end
   end
12 return s = (here, there)
```

Algorithm 8.4: Algorithm to Extract Seed Edge From Path Seed

Theorem 8.8 *Algorithm 8.3 has the same asymptotic running time as Algorithm 7.5.*

Proof: We noted that Algorithm 8.1 differs from Algorithm 7.6 only in the addition of Step 9 and Step 17. At each vertex v, Step 9 is invoked at most once, when a neighbour is detected that already belongs to a distinct component. This step takes $O(1)$ time, and is invoked at most $O(n)$ times.

At each join j with degree $degree(j)$, Step 17 is invoked once for each union operation. There are at most $degree(j) - 1$ such union operations at j, and at most $O(N)$ such union operations in the entire mesh. Since each invocation of Step 17 takes $O(1)$ time, we can bound total cost of adding this step by $O(N)$.

Algorithm 8.2 differs from Algorithm 7.3 only in Step 12. In the worst case, however, this step may cost $O(degree(v))$ at supernode v to find a suitable path seed. But since it is called at most once for each supernode, the overall cost is bounded by the sum of the degrees of all vertices in the mesh - $O(N)$.

Since $n = O(N)$, and Algorithm 7.5 requires $O(sort + N + t\alpha(t))$ time, the result follows. □

We now introduce a new output-sensitive parameter, p for the cost of extracting an isosurface seed from a seed path. In practice, this p is usually fairly small, as we will see in Chapter 16. In the worst case, however, $p = \Omega(n)$.

Definition 8.3 *The parameter* $p = \sum_{(u,v)\in P} degree(u)$ *measures the cost of extracting an isosurface seed from a path seed using a seed path* P *generated by Algorithm 8.4.*

For irregular meshes, we define p in this way because each vertex we pass may have $\Omega(n)$ edges in the worst case, and we may need to consider all but one of them before proceeding. For regular meshes,

however, the regularity allows us to give a slightly better description of p.

Lemma 8.9 *For regular meshes, $p = O(\delta \times l)$, where δ is the degree of each vertex in the mesh and l is the length of the path seed generated.*

Proof: Follows immediately from the fact that each vertex in a regular mesh has degree δ.□

Although this parameter p is convenient, in the worst case it can be proportional to the size of the mesh:

Lemma 8.10 *There exist meshes for which path seeds require $p = \Omega(N)$ time to extract contour seeds.*

Proof: We use Figure 8.1 as our counterexample, which consists of four squares divided into triangles. Since the contour tree has one superarc, we have one path seed. Without loss of generality, this path seed starts at 10. To extract the contour at an isovalue of 99, we must ascend through at least one edge for each square in the mesh. Since there are $\Theta(n)$ such squares, our seed path P must contain at least $\Omega(n)$ vertices. Since this mesh is regular, Lemma 8.9 implies that $p = \Omega(\delta \times n) = \Omega(N)$. □

In practice, however, p is significantly smaller than N. Absent a theoretical model of contour complexity, it is hard to prove a more useful bound. Only one estimate of contour complexity exists. Itoh & Koyamada [IK95] estimate that contours in a d-dimensional field involve $O(N^{(d-1)/d})$ cells of the mesh, based on the observation that each contour is a $(d-1)$-manifold and the entire data set is a d-manifold. For typical data sets in scientific visualization, no other estimate has been advanced. For lack of a better estimate, we shall accept it.

By analogy, we can estimate that a seed path, which is a 1-dimensional manifold, should involve $O(N^{1/d})$ edges of the mesh, and will be dominated by the $O(N^{(d-1)/d})$ cost of the actual contour extraction.

Thus we get the following assertion:

Assertion 1 *We estimate that $k = O(N^{(d-1)/d})$, and that $p = O(k)$, where k is the output-sensitive parameter that measures isosurface size.*

In Chapter 16, we will see that the length of seed paths is typically quite short, lending some credence to this estimate.

Now that we have defined p, we can analyse the cost of using Algorithm 8.4

Lemma 8.11 *Algorithm 8.4 takes $O(p)$ time to extract an isosurface using an ascending seed path P.*

Proof: In Algorithm 8.4, we initialize $P = (\mu, \xi)$. Each time through the loop that starts at Step 3, we choose another edge $(here, there)$ and add it to P. At each vertex $here$, we execute Step 7 at most $degree(here)$ times at constant cost. The result then follows from Definition 8.3. □

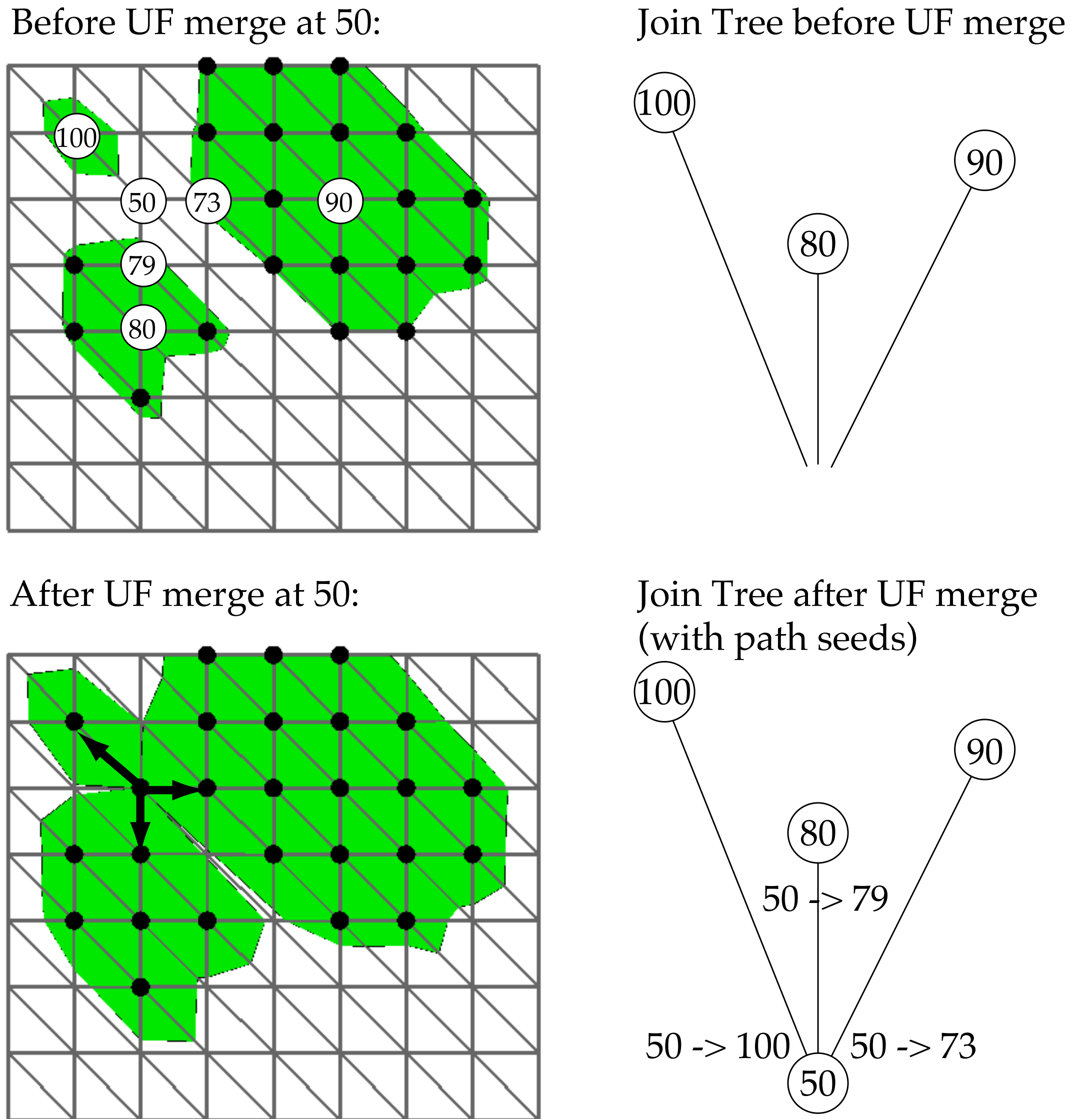


Figure 8.5: Detecting Path Seeds During the Join Sweep.
During the join sweep, we know that 50 is a join because 100, 79 and 73 belong to distinct components in the union-find structure. When we terminate the join superarcs that correspond to these three components, we add the corresponding edges to the join tree as path seeds. When transferring an edge from the join tree to the contour tree, we also transfer the path seed.

8.2.5 Example of Path Seeds

Now that we have defined path seeds and described an algorithm for working with them, we show an example using the same sample mesh as before. Figure 8.5 shows the step during the join sweep of Figure 7.2 at which vertex 50 is added to the union-find structure. Immediately before 50 is added, there are three distinct components, with join superarcs initialized at 100, 90, and 80. Initially, edge $(100, 50)$ is processed, and is

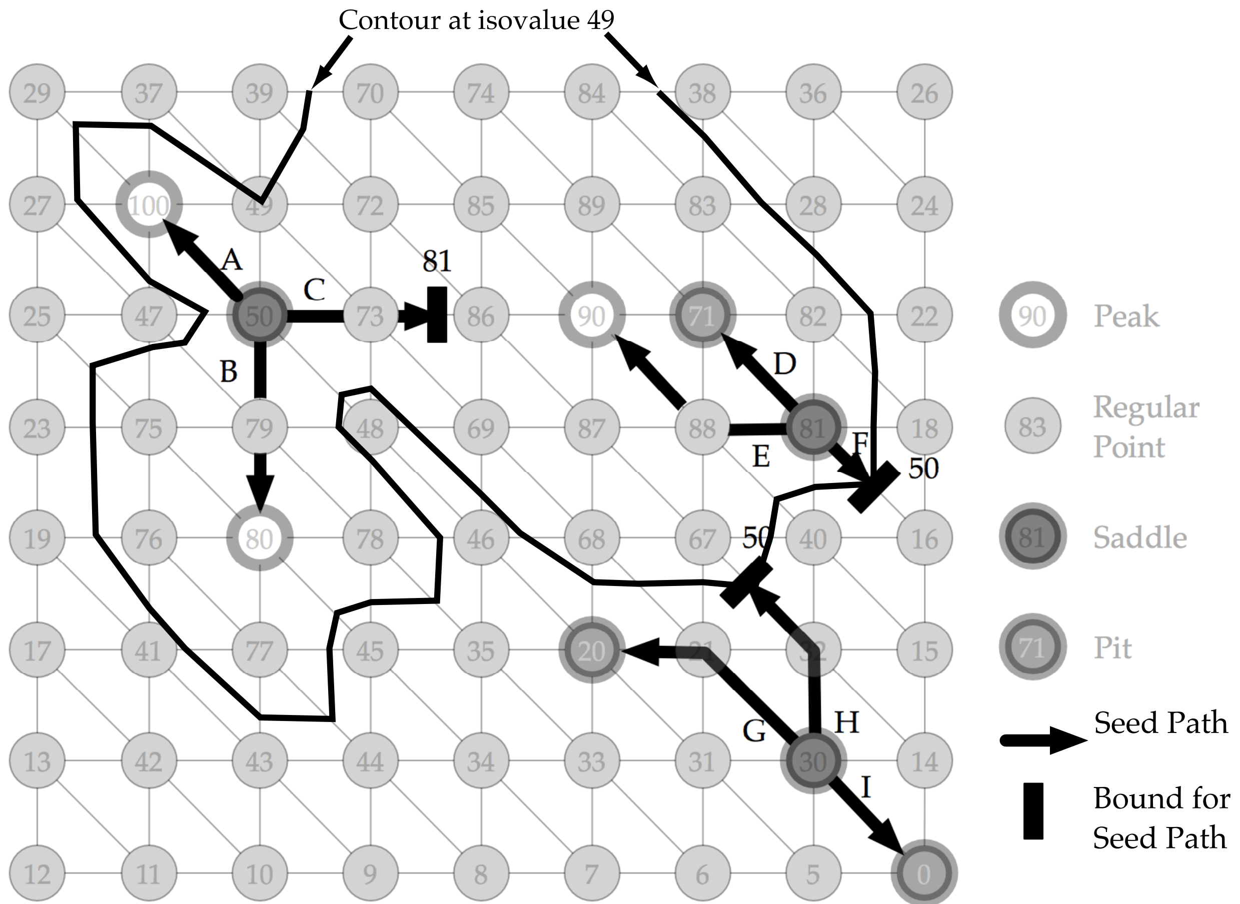


Figure 8.6: Seed Paths in our Sample Mesh.
Here, we show a set of path seeds (A - I) generated for the sample data set, some seed paths generated from these path seeds, and a sample contour. Note that seed paths can terminate in the middle of a mesh edge: this prevents path F from intersecting the contour at 49, guaranteeing exactly one seed per contour.

added to the join superarc $(100, NULL)$ in case 50 turns out to be a join. If the next edge processed is $(80, 50)$, three things happen. First, $(100, NULL)$ is terminated at 50 to obtain $(100, 50)$, with the path seed left as $(100, 50)$. Second, $(80, NULL)$ is terminated at 50 to obtain $(80, 50)$, with path seed $(79, 50)$ set. And third, edge $(50, NULL)$ is initialized to continue downwards from 50.

After the contour tree has been computed, the path seeds shown inFigure 8.7 are attached to the superarcs. Figure 8.6 then shows example seed paths that can be extracted using these path seeds. We also show the contour at 49, as we did in Figure 8.3. In comparison to minimal seed sets, however, exactly one of our seed paths $(30 \rightarrow 32 \rightarrow 67)$ intersects this contour. Although the path from 81 to 50 appears to, this is due to crowding in the image. Since the path descends from 81 and stops at isovalue 50, it cannot actually do so.

Note that the seed paths may take any route: e.g. $30 \rightarrow 31 \rightarrow 33 \rightarrow 34 \rightarrow 35 \rightarrow 46 \rightarrow 78$ would also be a valid seed path for the superarc $(50, 30)$. It is simple, however, to optimize by always taking the steepest ascent, which tends to result in shorter seed paths.

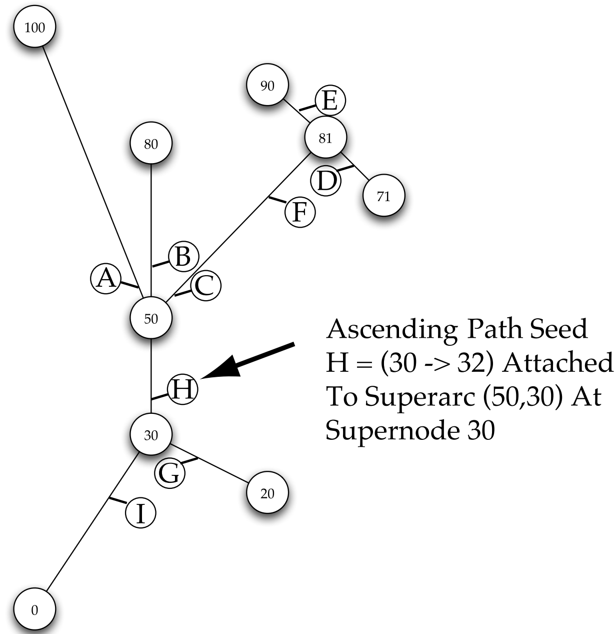


Figure 8.7: Storing Path Seeds From Figure 8.6 in the Contour Tree.
Here, we show the path seeds used in Figure 8.6 attached to the corresponding superarcs of the contour tree.

8.3 Summary and Comparison

In this chapter, we have introduced one of the principal contributions of this thesis: the *path seed* for generating isosurface seeds. As with the previous minimal seed set of van Kreveld et al. [vKvOB+97], it is guaranteed to provide seeds for all possible contours in the set. However, in comparison with the minimal seed set, the path seed enjoys a number of advantages:

1. Path seeds are simpler to calculate than minimal seed sets. Instead of a complex post-processing step, all that is required are some simple additions to the main contour tree algorithm.
2. Path seeds are cheaper to compute, requiring $O(t)$ computation time in any dimension, instead of $O(n \log^2 n)$ time in two dimensions and $O(n^2)$ in higher dimensions
3. Path seeds are cheaper to store, requiring $\Theta(t)$ storage even in the worst case, instead of $O(n)$ storage.
4. Path seeds guarantee a one-to-one correspondence between seeds and contours, allowing us to track individual contours by mapping them to the contour tree.

These are offset by slightly slower time for extracting isosurfaces from path seeds compared with minimal seed sets, a disadvantage that is negligible in practice, due to the $O(k) \approx O(n^{\frac{d-1}{d}})$ cost of extracting the isosurface itself.

Of these advantages, the most powerful is the last: a tight one-to-one correspondence between path seeds and contours. In the next chapter, we show how this can be used to generalize the concept of an isosurface, and to use the contour tree as a direct control for manipulating or annotating individual contours.

Chapter 9

Flexible Isosurfaces

We saw in Chapter 2 that isosurfaces, unlike isolines, occlude each other. This makes it crucial that we choose carefully which isosurfaces we display. Prior to the introduction of path seeds, users were typically limited to *level sets*: sets of isosurfaces at a single fixed isovalue. In many cases, however, interesting isosurfaces are contained inside and occluded by uninteresting isosurfaces. And, in some data sets, different interesting objects have different isovalues. Thus, a restriction to level sets hampers exploratory visualization. Moreover, the isosurfaces are often generated as an interlaced set of polygons belonging to multiple surfaces. With path seeds, however, we can lift this restriction, and develop new ways of exploring scalar data.

In this chapter, we introduce the second principal contribution of this thesis: the *flexible isosurface*: a metaphor and interface for interactive exploration of sets of contours at different isovalues. This interface uses the contour tree as a topological index to the potential contours, and as a source of seeds for extracting and rendering the contours, taking advantage of the tight one-to-one correspondence between contour tree and contours that was enabled by the introduction of path seeds in the previous chapter.

Section 9.1 sketches previous work relevant to this chapter. Section 9.2 then states some assumptions, and Section 9.3 discusses the tasks we wish a user to be able to perform, and states a set of operations for manipulating contours. Section 9.4 then gives a formal definition of a *flexible isosurface*, and Section 9.5 presents an interface for manipulating contours in a flexible isosurface. Section 9.6 then examines the evolution of individual contours rather than entire level sets. Section 9.7 then introduces the data structures that we use for implementation, and Section 9.8 discusses how individual operations are implemented. Finally, Section 9.9 discusses some problems that lead into the discussion of simplification in Chapter 11, and Section 9.10 summarizes the contributions of this chapter.

9.1 Previous Work

Research relating to individual contour manipulation can be grouped under three headings: local surface choice, isosurface choice, and transfer function design. The first group, local surface choice, is the smallest, but closest in spirit to this work, and includes work on displaying several contour surfaces with different isovalues, based on a global understanding of what constitute important objects. The second group, isosurface choice, is the largest, and includes work on choosing a single isovalue to apply globally. The third group,

transfer function design, uses isovalue as the principal parameter for mapping a function to colour and opacity for use in volume rendering.

Local Surface Choice: There has been relatively little work on isolating and manipulating contour surfaces, except as entire level sets. Silver [Sil95] introduced *object-oriented visualization*, in which objects were defined to be individual contours determined by expanding surfaces from the local maxima until a connectivity criterion was satisfied. Manders et al. [MHS+96] took a similar approach, called *Largest Contour Segmentation*, in which objects were defined as the largest surface containing only one local maximum. In neither case was it possible to explore the contour surfaces interactively.

Shinagawa, Kunii & Kergosien[SKK91] described a visual code for contour changes, based on the Reeb graph, but do not seem to have used their visual coding as an interface for isosurface exploration. Moreover, this visual code was not accompanied by an algorithm for generating codings automatically.

Isosurface Choice: More work has been published on choosing "important" isovalues, either manually or automatically. Manually chosen isovalues are often used to explore otherwise ill-understood data. The simplest exploration method displays the isosurface interactively. The user interacts with the software, trying different isovalues until a suitable isosurface is found. The user will often make a rough guess at the relevant isovalue, then adjust the isovalue in small increments, and watch how the isosurfaces evolve. This exploratory process is significantly faster if cues are available to guide the user to interesting isovalues. We will return to this idea in Section 9.6.

Some work aims to provide context, or cues, to the user, in order to guide isosurface choice. Bajaj, Pascucci & Schikore[BPS97] described such an interface, called the *Contour Spectrum*, shown in Figure 9.1. This interface consists of two parts: a main window showing the data, in this case two-dimensional data with isovalue mapped to colour, and a separate window, the *Contour Spectrum*, which graphs summary characteristics of isosurfaces as a function of the isovalue. These summary characteristics include the surface area and enclosed volume of the isosurface. In addition, the contour spectrum displays the contour tree as a cue to topological change in the data. In particular, note that the vertical line representing the isovalue chosen intersects the contour tree in the spectrum exactly once for each individual contour in the image. This interface, however, does not contain sufficient information to identify which contour is which in the contour tree, except by deducing it from the pattern of peaks in the image. Although the Contour Spectrum uses topological information to guide isovalue selection, the user is still restricted to choosing a single isovalue of interest. Moreover, the geometric information such as area, contour length, &c., is not integrated with the topological information represented by the contour tree. Instead, the two sets of information are displayed in parallel. We will see in Chapter 10 how to integrate geometric and topological information: this could then be added back in to the Contour Spectrum, if desired.

In a closely related approach, Kettner, Rossignac & Snoeyink [KRS01] modified the Contour Spectrum in an interface called SAFARI. This interface was designed to display time sequences of three-dimensional scalar fields. In a panel similar to the Contour Spectrum window, a colour map is used to represent the connectivity of isosurfaces over two independent variables: time and isovalue. Moreover, the panel is also used directly to specify the time and isovalue to display in the main panel. The connectivity of the isosurfaces at different isovalues and times is computed from the contour trees for the individual time steps, without explicitly extracting the isosurfaces.

Both of these interfaces use topology in the form of the contour tree or information derived from the contour tree. But both of them treat isosurface choice as a global problem with one parameter: the isovalue.

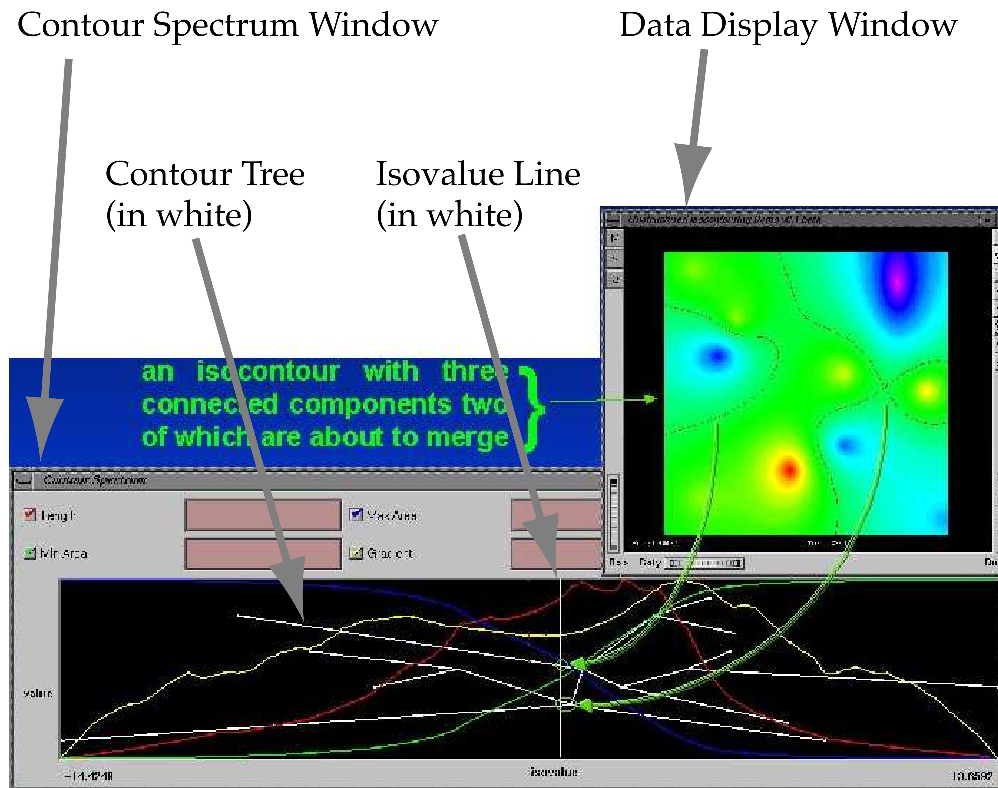


Figure 9.1: The Contour Spectrum - illustration courtesy C. Bajaj. In this illustration, a 2-D field is displayed by assigning colours to different isovalues: boundaries between colours are contours. In the Contour Spectrum window, the isovalue is plotted horizontally, while properties such as area above the isovalue are plotted as coloured lines, and the contour tree is drawn as a set of white line segments. The vertical bar indicates a particular isovalue, in this case a medium green colour in the data display.

And neither the Contour Spectrum nor SAFARI allows manipulation of individual contours.

Transfer Function Design : Choosing an appropriate isovalue is closely related to designing transfer functions for volume rendering. Transfer functions specify the opacity and optical properties of different types of material, commonly based on isovalues in the data. Like isosurfaces, transfer functions assign meaning to particular isovalues, or more commonly, ranges of isovalues.

As with the Contour Spectrum and SAFARI, this assumes that one global choice, or set of choices, will be applied everywhere. Kniss, Kindlmann & Hansen KKH01 noted that this assumes that a given isovalue has uniform meaning throughout the data set. They observed that this assumption causes problems, and designed an interface to construct multi-dimensional transfer functions interactively. This interface added gradient information as a parameter to the transfer function, but continued to apply the same transfer function everywhere in the data. Kindlmann et al. [KWTM04] added the curvature of the function as a parameter, but still without any spatial locality.

Some authors use statistical methods instead of user interaction to determine important isovalues. Pekar, Wiemker & Hempel[PWH01] use a gradient histogram to find isovalues for which the gradient is steepest, assuming that steep gradients mark significant boundaries. Although these authors generated isosurfaces, their work also applies to transfer function design. Similarly, Tenginakai, Lee & Machiraju[TLM01]

use statistical signatures of the local distribution of voxel values to define a transfer function. And in a technical report published in 2001, Takahashi, Fujishiro & Takeshima[TFT01] described how to automate transfer function design by using the contour tree to detect isovalues at which major changes in isosurface topology occurred, then emphasizing those isovalues.

With the exception of the work by Silver [Sil95] and Manders et al. [MHS$^+$96], the emphasis has been on using global properties to detect isovalues that are significant. Implicitly, this decides that the isovalue is significant everywhere in the data. If we wish to determine significance on a local basis, we must extend these approaches and use different properties or isovalues in different spatial locations.

9.2 Assumptions

Before discussing the details of the flexible isosurface, we must state some important assumptions:

Assumption 4 *Interesting objects in the data are not connected to each other, and each can be represented by a single contour.*

This assumption is based on the general assumption that isosurfaces are useful representations of important features, modified to recognize that distinct objects are usually topologically disconnected, or separable.

Assumption 5 *Not all contours in a level set are interesting, or important.*

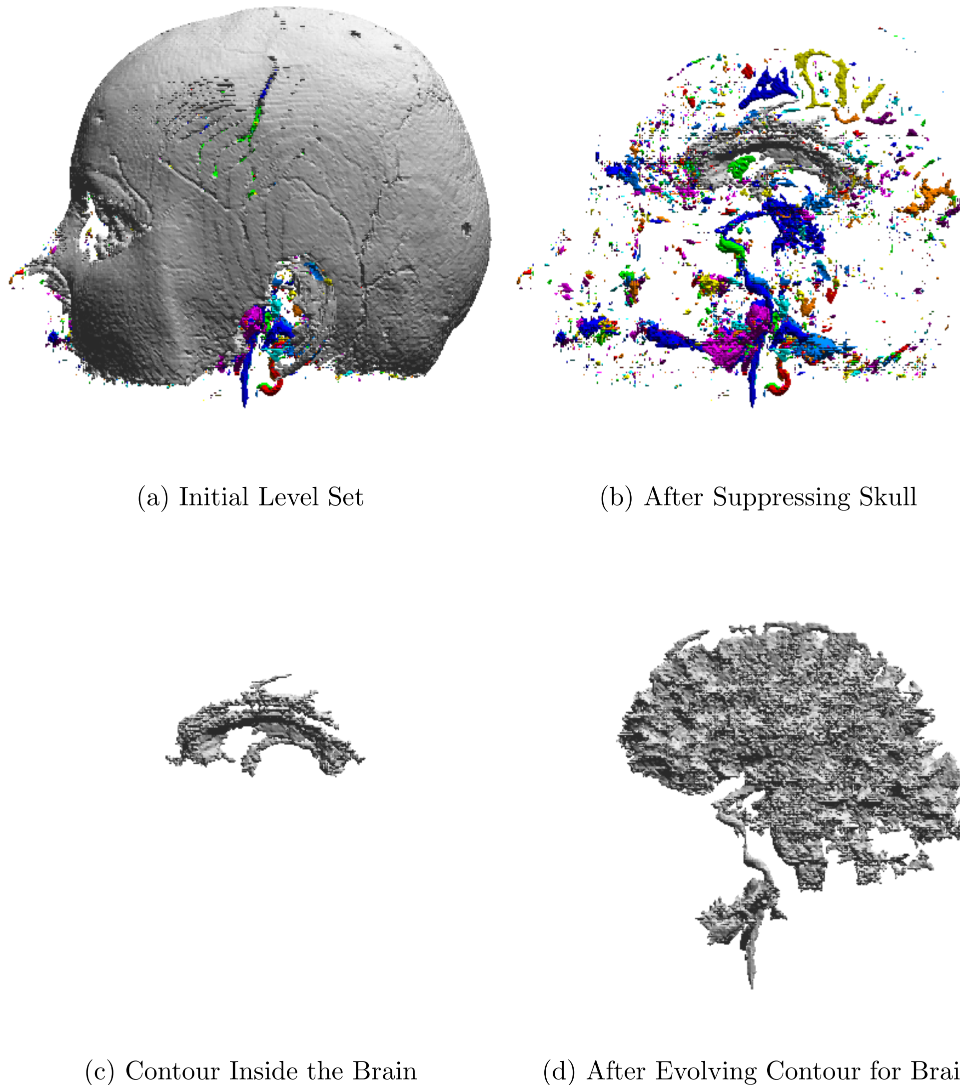
Since 3-D surfaces generally occlude each other, if we wish to look at an object inside or behind an occluding object, we must find a way of suppressing the less interesting occluder. In large data sets, there may also be so many contours that the one of most interest is occluded, or fails to catch the user's eye due to the visual complexity of the display.

Several consequences flow from this assumption. First, we must provide an automatic or manual method to identify which contours are interesting, and which are not. Secondly, we must provide the user with some mechanism for deciding which contours to show, especially initially. And thirdly, we must give the user cues as to which contours are interesting. Since visual complexity and occlusion are problems, we wish to find an abstract representation of the data that can meaningfully be used to guide the user. We will use the contour tree for this purpose. Unlike the Contour Spectrum [BPS97], however, we use the contour tree as an active means of manipulating contours, rather than a passive display of the topology.

9.3 A User's View of Flexible Isosurfaces

We wish to relax the restrictions imposed by previous isosurfacing algorithms, and allow user manipulation of individual distinct contours. We start by introducing some examples of user interactions that we wish to make feasible, starting with visual manipulation of visible contours in Section 9.3.1.

In Section 9.3.2 we discuss manipulation of contours that are not currently visible, using the contour



(a) Initial Level Set

(b) After Suppressing Skull

(c) Contour Inside the Brain

(d) After Evolving Contour for Brain

Figure 9.2: Example of Flexible Isosurface Editing to Isolate the Brain.
(a) Initial selection of an isosurface by specifying the isovalue.
(b) After deleting (suppressing) an occluding surface (the skull).
(c) After isolating a single contour of interest.
(d) After changing the isovalue (evolving) of the contour.

tree as an index to all possible contours. Finally, in Section 9.3.3, we state a set of operations that will support the manipulations described.

9.3.1 Manipulating Contours Visually

We will start by considering manipulation of currently visible contours by looking at two possible scenarios of a user's interaction with the system. In Chapter 2, we described a possible sequence of user interactions and showed results of individual choices in Figure 2.2. We describe a similar scenario in Figure 9.2, and consider it in more detail.

Suppose that a radiologist has performed an MRI scan of a patient's head, and wishes to segment the surface of the brain as a geometric object for further processing. If the radiologist is using conventional isosurfacing, and chooses a suitable isovalue, he or she might get the image in Figure 9.2(a). Clearly, this is not satisfactory, as the isosurface for the skull occludes the brain isosurface.

To deal with this occlusion, the radiologist might wish to suppress the display of this skull isosurface, so that objects inside the skull would become visible, as shown in Figure 9.2(b). At this stage, the radiologist would probably realise that the original choice of isovalue was unsatisfactory, as the outer boundary of the desired object (the brain) is not well-defined. However, one of the contours (shown in grey) is in the right location, and can be guessed to be some central structure in the brain of relatively high intensity. It is hard to be certain, however, due to occlusion by other isosurfaces and due to the visual complexity resulting from a large number of contours.

To deal with these problems, the radiologist might then ask to suppress all of the isosurfaces except the chosen one, resulting in the image shown in Figure 9.2(c). Even now, though, the surface is unsatisfactory, as it does not show the surface of the brain, but rather some interior structure, because of a poor initial choice of isovalue.

To deal with this final problem, the radiologist might adjust the isovalue of this particular isosurface, without having to remove the skull and other isosurfaces at each new isovalue. Adjusting the isovalue, the radiologist would watch this contour evolve, growing larger as the isovalue decreased, until a reasonable final surface was arrived at, as shown in Figure 9.2(d).

In a second scenario, a molecular biologist is using isosurfaces to visualize an X-ray crystallographic data set. An initial isosurface might look like the one shown in Figure 9.3(a). In this case, even when the scene is rotated, it can be difficult to distinguish visually between individual isosurfaces. In order to increase the visual contrast between components, the biologist might ask for the surfaces to be assigned colours, with the result shown in Figure 9.3(b). Here, although it is easier to distinguish individual components, many of the components are partially or wholly occluded. As with the previous example, the user may wish to isolate a single component for individual study: in Figure 9.3(c), a single alpha-helix in the molecule has been isolated. And, again as in the previous example, the user may then choose to adjust the isovalue of this contour. In Figure 9.3(d), the isovalue has been increased in order to see the individual peaks contained in this particular contour.

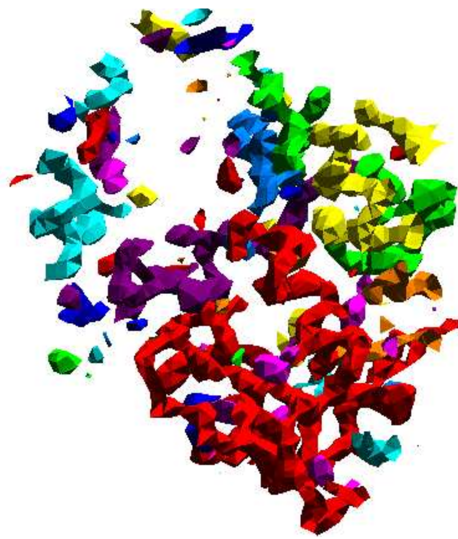
In both of these scenarios, the tasks can be broken down into smaller subtasks, generally involving a single contour at a time: identification or *selection* of an individual contour, *deletion* of a single contour or of all contours except one (*isolation*), *evolution* of a single contour with respect to the isovalue, and *initialization* of the set of contours currently in view to a level set.

9.3.2 Manipulating Contours That Are Not Visible

In the previous section, we looked at manipulating contours visually. Contours that are not already visible can be brought into view in several ways: evolving a contour until the desired contour is found, by specifying a set of contours, for example a level set, or by choosing a contour from an index of possible contours.

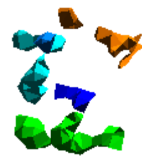
To specify a set of contours, we choose some abstract criterion which is easy to apply. Level sets are chosen by applying the criterion that the contours must have a specified isovalue. Another abstract criterion chooses maximal contours that contain only one local maximum: this criterion was used by Silver [Sil95] for

(a) All One Colour



(b) With Arbitrary Colours Assigned

(c) Alpha Helix Isolated



(d) Helix Evolved

Figure 9.3: Heightening Visual Contrast with Colour & Isolation. Note how adding colours to individual contours makes it easier to tell them apart. Moreover, isolating a single helix to study it separately removes distracting or occluding surfaces.

Object-Oriented Visualization and by Manders et al. [MHS+96] for *Largest Contour Segmentation*. We will generalize this later, by allowing non-maximal contours instead of maximal contours.

While manipulating individual contours, we need a way of indicating what contours could be added to the currently displayed set, and a way of specifying which to add. In other words, we need an index to all possible contours. Every possible contour should be in the index, and the index should be as compact as possible. Fortunately, such an index is available: the contour tree, using path seeds for contour extraction.

To use the contour tree as an index, we will draw the contour tree so that one dimension represents the isovalue. We will then allow the user to define a contour by choosing the point on a superarc at the desired isovalue. We will treat this as equivalent to choosing a contour visually from the data display. Once we have chosen a contour in this way, we can also allow contour evolution to be applied directly in the contour tree.

9.3.3 Contour Manipulation Operations

Based on the descriptions of desired user interactions given above, we decompose the tasks into the following operations:

1. Defining a level set
2. Increasing visual contrast with colour
3. Choosing a contour (selection)
4. Choosing no contour (unselection)
5. Evolving a contour by adjusting its isovalue
6. Deleting a contour
7. Isolating a contour by deleting all other contours
8. Undoing a deletion or isolation
9. Adding a contour
10. Defining a Largest Contour Segmentation

We will see how to implement these operations in Section 9.8. First, however, we will define a flexible isosurface formally, then describe an interface that supports these operations.

9.4 The Flexible Isosurface

In the previous section, we saw some examples of tasks that might require displaying a set including one or more contours with distinct isovalues. Before considering an interface for visually manipulating sets of contours, let us give some slightly more rigorous definitions.

We start with the following convention:

Convention 9.1 *A contour c is denoted by a pair $c = (s, h)$ consisting of a superarc s in the contour tree and an isovalue h.*

For convenience, we do not allow contours that pass exactly through critical points. Although this may seem like a limitation, it is in keeping with most isosurface extraction techniques, which classify vertices as "above" or "below" the isosurface, never "at" the isosurface. This has the effect of perturbing any isovalue by an arbitrarily small epsilon, while simplifying the analysis of possible cases. Since critical points in a simplicial mesh are at vertices of the mesh, we lose nothing by this choice.

We can now use this to generalize the concept of an isosurface from the single-parameter control discussed in Section 9.1, to the flexible isosurface, in which we can specify and manipulate individual surfaces:

Definition 9.2 *A* flexible isosurface *is a set* $\{c_1, \ldots, c_\tau\}$*, where each* $c_i = (s_i, h_i)$ *is a contour of* f*.*

It is easy to show that we can efficiently display any arbitrary flexible isosurface using path seeds, according to Lemma 8.11.

Lemma 9.1 *A flexible isosurface containing* τ *contours can be generated in* $O(\tau p + k) \approx O(k)$ *time, where a path seed takes* $O(p)$ *time to extract, and the contours generated are of size* k *in total.*

Proof: Recall that we represent each contour γ_i as a pair (s_i, h_i) where s_i is a superarc in the contour tree and h_i is an isovalue. From Lemma 8.11, we know that we can extract each isosurface seed in $O(p)$ time. From Section 7.1, we know that k measures the cost of extracting surfaces with the continuation method. Since we extract τ contours, the total cost is therefore $O(\tau p + k)$. □

As in Section 8.2.4, we expect τp, the cost of path seed extraction, to be dominated by k, the cost of isosurface extraction from seed edges.

9.5 The Flexible Isosurface Interface

For flexible isosurfaces to be useful, we must provide an interface through which a user can define and manipulate them. In Figure 9.4, we show the interface we use to manipulate the flexible isosurface. Broadly speaking, the interface is split into three parts: the data display area, the contour tree, with attached isovalue slider, and miscellaneous controls. Although superficially similar to the contour spectrum shown in Figure 9.1, there are some important differences.

The Data Display The data display shows the current state of the flexible isosurfac, using path seeds to extract each contour individually. If coloured contours are not enabled, all contours are shown in a uniform colour. If coloured contours are enabled, each contour is assigned a colour. Since the surfaces are rendered using shading, we are restricted to only a few colours, and are forced to reuse them. Even a few colours, however, enhance visual contrast, and identify which contour in the data display corresponds to which tag in the contour tree display.

The Contour Tree Display The contour tree display shows the contour tree, with each contour in the flexible isosurface indicated by a small rectangle or *tag*. Each tag is positioned along the corresponding

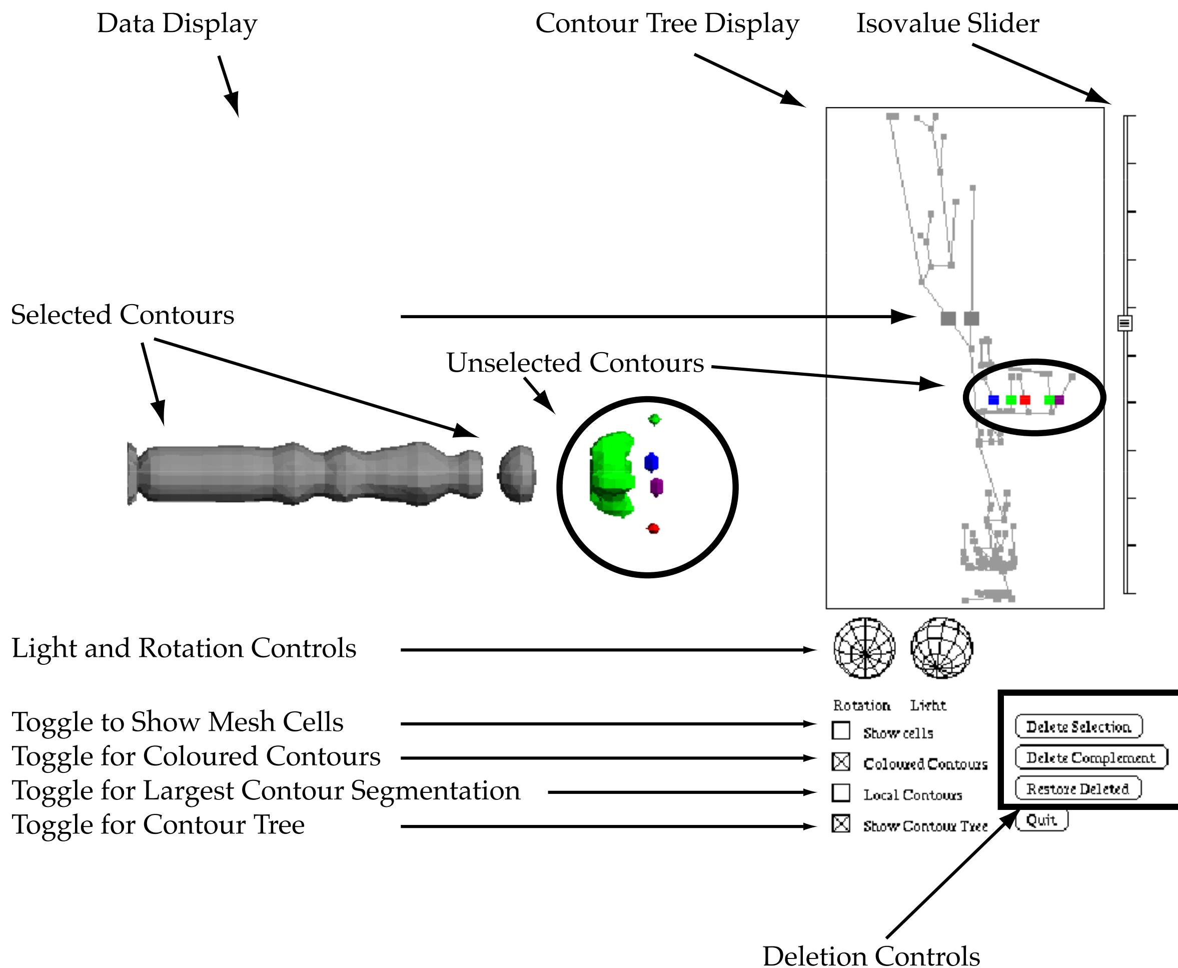


Figure 9.4: Components of the Flexible Isosurface Interface.
The interface as shown here has three regions. On the left is the data display where contours are rendered. On the right is the contour tree display where the contour tree is displayed with colour-coded tags representing where each contour displayed is in the contour tree. Next to the contour tree display is a slider which controls the isovalue either of a single selected contour or of a level set if no contour is selected. Underneath the contour tree display are a set of miscellaneous controls, ranging from arcballs to control rotation in the data display to buttons controlling component deletion. A more sophisticated version of this interface can also be seen in Figure 11.14, which incorporates additional controls for contour tree simplification.

superarc, with the vertical position set to the isovalue of the contour. Moreover, these tags are colour-coded to match the colour assigned to the contour in the data display. And finally, these tags do not just provide passive information: they can actively be manipulated, in which case the corresponding contours are also manipulated in the data display.

Choosing (Selecting) a Contour The metaphor used for the user interface is the standard graphical metaphor of “select and operate”. The user chooses a contour visually, either in the main image, or in the contour tree display, then performs an operation on the contour, such as deletion, isolation, or contour evolution.

In the data display, we can select only contours that are currently visible. We do so using standard graphical picking techniques: when the user clicks the mouse in the data display, we either trace a ray through the scene from the viewplane, or simply query the frame-buffer to determine the closest contour to the viewplane at that point. As with many graphical interfaces, if no contour exists at the location chosen, the user is considered to have *unselected* or released the current selection.

In the contour tree display, the user can select any possible contour by clicking the mouse over any of the superarcs displayed. If there is already a tag on that superarc, it is moved to the isovalue corresponding to the vertical position of the mouse, and the contour in the data display updated accordingly. If there is no tag on the superarc, a new tag is created, and the corresponding contour rendered in the data display. At present, we do not allow multiple contours on a single superarc to be selected for two reasons. First, for most data sets, one of the contours will be inside the other and therefore occluded. And second, by allowing only one contour per superarc, we keep the interface and processing relatively simple. It would, however, be feasible to permit the user to choose multiple contours on the same superarc.

Following standard user-interface conventions, the contour selected is highlighted to distinguish it from contours that are not selected. Highlighting is done by changing the colour of the contour and the contour tag, and by enlarging the tag slightly. When coloured contours are disabled, a distinctive colour such as red is used to separate the highlighted contour or contours from the others, which are rendered in grey. When coloured contours are enabled, however, we reverse this convention, and render the highlighted contour in grey, to distinguish it from the unselected contours in colour.

When we discuss contour evolution in Section 9.6, we will see that the selected contour can evolve into multiple contours as the isovalue varies: we will treat all such evolved contours as selected, so that they are easily distinguished from other contours in the data display. Except for this case, we do not currently support selecting more than one contour at once: this avoids complications when evolving contours.

Arcballs, Check Boxes and Buttons Two arcballs are provided: one to control the viewing direction, and one to control the direction of incoming light. If needed, controls for zooming in and out and for moving the image in the data display laterally can also be provided. Figure 11.14 shows an example of a version of the flexible isosurface interface with controls for zooming and scaling.

Check-boxes are provided to toggle the cells of the mesh, to enable coloured contours, to enable "local contours", and to turn the contour tree display off for faster rendering of the data display. Three of these have obvious functions: the fourth, the check-box labelled "local contours" , principally affects the behaviour of the isovalue slider, which we will discuss shortly.

Four buttons are shown in the lower right hand corner. Three of these implement operations from Section 9.3.3. The first deletes the currently selected contour(s), the second isolates the currently selected contour by deleting all other contours in the flexible isosurface, and the third one reverses the most recent deletion or isolation operation. A fourth button, labelled *Quit*, is provided to exit from the program.

Isovalue Slider The only remaining interface element is the isovalue slider, whose behaviour is somewhat complex. If we compare the effects of the interface elements described so far with the list of operations in Section 9.3.3, we see that the only remaining operations are: defining a level set, defining a largest contour segmentation, and evolving a contour. Each of these operations uses the isovalue slider. Depending on whether a contour is selected, and depending on whether the "local contours" check-box has been selected, the slider does one of the following:

Contour Selected	Local Contours	Operation
No	No	Defines a level set
No	Yes	Parameterized Largest Contour Segmentation
Yes	No	Evolves the selected contour
Yes	Yes	Constrained evolution of the selected contour

Thus, when no contour is selected, and local contours are disabled, the isovalue slider is used to specify the isovalue of a level set, as with previous interfaces. When no contour is selected, and local contours are enabled, we perform a parameterized version of Largest Contour Segmentation: for details, see Section 9.8.3, below.

If a contour has been selected, and local contours are disabled, the isovalue slider controls the isovalue of that contour only, allowing contour evolution as described in Section 9.3. We will discuss contour evolution in some detail in Section 9.6, below. For superarcs whose height is small, this does not give fine enough control over the isovalue in many cases, so we use the "local contour" check-box to constrain the evolution to a single superarc of the contour tree. As with Largest Contour Segmentation, the isovalue slider is now re-parameterized to the range of isovalues corresponding to the superarc.

For convenience, when a contour is being evolved, we allow the user to use the contour tag in the contour tree display in the same way as the isovalue slider. We note that the vertical location of the box on the isovalue slider corresponds to the isovalue, as does the vertical location of each contour tag in the contour tree. Thus, instead of selecting the contour tag, then using the slider to specify the isovalue, we permit the user to drag the contour tag up and down in the contour tree, evolving the contour in the same way as if we had used the slider.

9.6 Evolving A Contour

In the previous section, we saw an interface that enables all of the operations discussed in Section 9.3.3. Most of these operations are straightforward, except for one: *contour evolution.* This operation involves adjusting the isovalue of one specific contour, showing the results in both the data display and the contour tree display. We will start with a formal definition of what it means to evolve a contour:

Definition 9.3 *A contour c at isovalue h evolves into the set of contours $c_1, \ldots, c_m$ at isovalue h' such that for each c_i there exists an f-monotone path P_i in $\mathcal{R}$ from c to c_i.*

This definition emphasizes the expectation of the user that they are seeing the results of increasing (or decreasing) the isovalue *of a specified contour.* Since we use the contour tree as our index, we must convert this definition into one in terms of the contour tree, an easy task given the correspondence between contours and the contour tree:

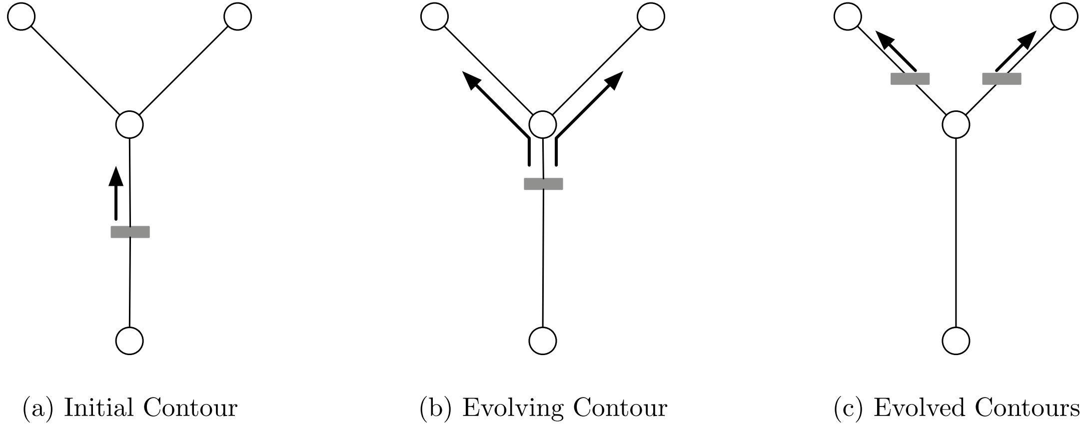


Figure 9.5: Simple Example of an Evolving Contour.
(a) As the contour is evolved upwards, we generate contours on the same superarc at progressively higher isovalues.
(b) As the contour evolves upwards past a saddle, it splits into one contour for each of the upwards superarcs at the saddle.
(c) Once past the saddle, each contour continues to evolve upwards on its own superarc.

Lemma 9.2 *Definition 9.3 is equivalent to the following formulation: A contour c at isovalue h evolves into the set of contours $c_1, \ldots, c_m$ at isovalue h' such that for each c_i there exists a monotone path Q_i from c to c_i in the contour tree.*

Proof:

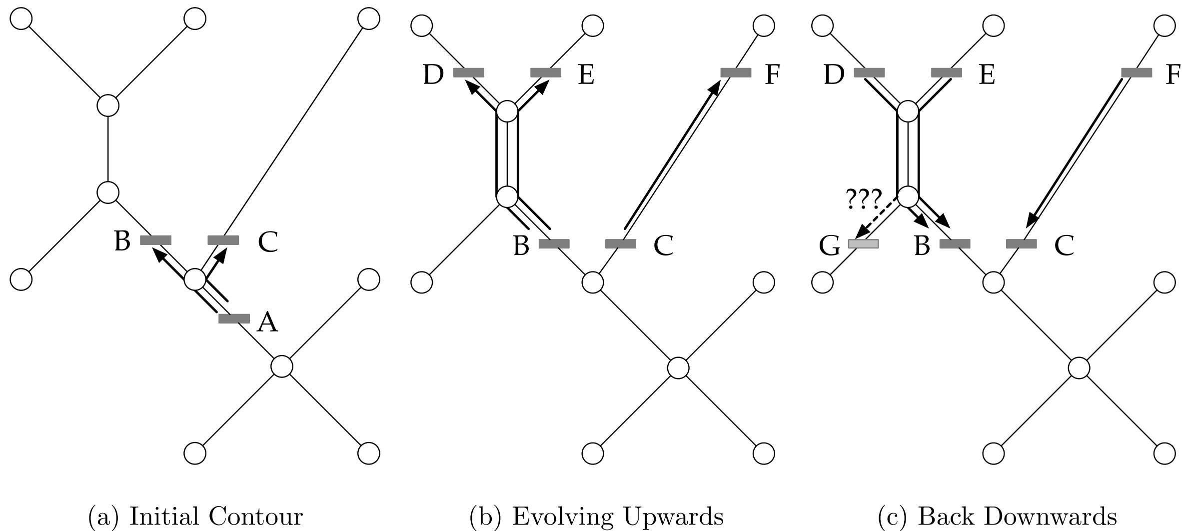
($\Rightarrow$) Let P_i be an f-monotone path in $\mathcal{R}$ from c to c_i at isovalue h'. By Corollary 6.1, we know that there is a corresponding monotone path Q_i in the contour tree from c to c_i.

($\Leftarrow$) Let Q_i be a monotone path in the contour tree from c to some contour c_i at isovalue h'. By Lemma 6.5, we know that there is a corresponding f-monotone path P_i in $\mathcal{R}$ from c to c_i. □

This result allows us to discuss contour evolution solely in terms of the contour tree, while still remaining true to the user's expectations.

We show a simple example of contour evolution in Figure 9.5. In this case, a single contour is chosen in a small contour tree. As the isovalue is increased using the isovalue slider, the contour tag slides up the superarc of the contour tree from Figure 9.5(a) to Figure 9.5(b). As the contour evolves past the isovalue of the join, the contour separates into two distinct contours, one for each branch of the contour tree at the join.

Semantically, the basic meaning of this evolution is clear: the user expects to see a smooth evolution of contours as the parameter is changed. As the parameter is changed repeatedly, however, several difficulties arise, and we must decide how to handle them.



(a) Initial Contour (b) Evolving Upwards (c) Back Downwards

Figure 9.6: Continuous Contour Evolution: Upwards, then Downwards
(a) As in Figure 9.5, a contour A evolves upwards into contours B and C.
(b) Contours B and C then develop into contours D, E and F.
(c) When the parameter is returned to the isovalue of B and C, should G be shown?

9.6.1 Contour Evolution Policies

Although it is easy to define contour evolution, we still have some decisions to make about the interface. These decisions result in the following rules, which we state briefly, then explain in the following sections:

(9.6.2) Contour evolution can be viewed as a continuous process from an initial to a final state.

(9.6.3) An evolving contour is more important than a contour that is not evolving.

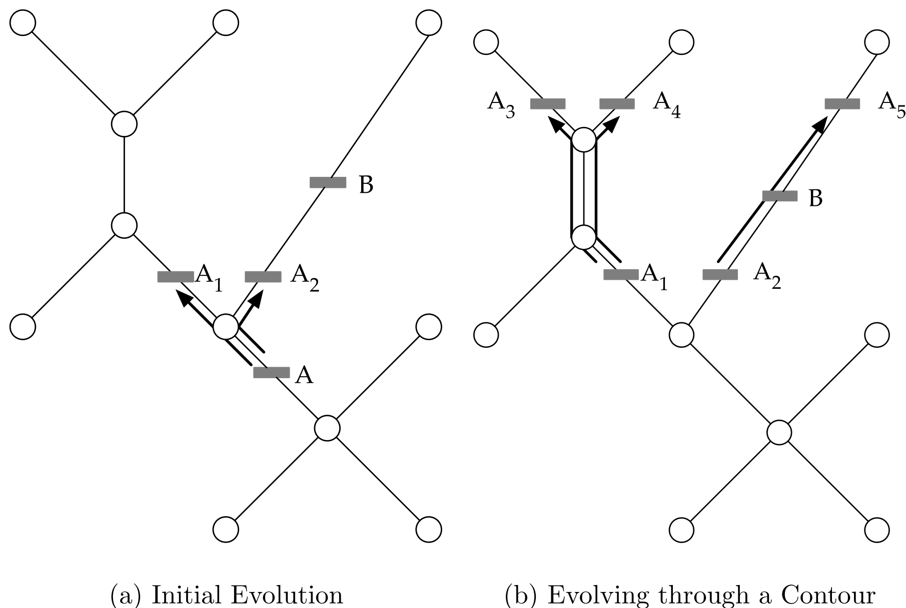
(9.6.4) The user is not permitted to select more than one contour at one time.

9.6.2 Continuous and Reversible Contour Evolution

From the standpoint of the user, evolving a contour should be a continuous and reversible process. It should be *continuous* by providing visual feedback as the isovalue is changed. It should be *reversible*, in that reversing the direction of evolution should return to a previous stage in the evolution. This emphasizes the assumption that the contour represents an object which we wish to study at different isovalues.

To achieve this, we generate a sequence of evolutions, starting at an initial contour. We must decide whether to treat this sequence of evolutions as independent evolutions, or to treat them all as evolutions from the initial contour. We choose to treat them all as evolutions from the initial contour.

Consider Figure 9.6, in which the user starts with contour A in Figure 9.6(a), then evolves it upwards to get first B and C in Figure 9.6(a), then D, E, and F in Figure 9.6(b). When the user reduces the isovalue with the isoslider once more, one of two things can happen.



(a) Initial Evolution (b) Evolving through a Contour

Figure 9.7: Suppressing contours during contour evolution
As A evolves past B, B will normally occlude either A_2 or A_5 as A sweeps past B. We assume that the user is more interested in the evolution of A through A_2 and B to A_5 than in B itself, and suppress B so that the evolution of A is clear.

If we treat the changes to the isovalue slider as independent evolutions, then, as the user reduces the isovalue, the set of contours D, E and F should be evolved downwards. Evolving each independently, we get contours B, C, and G, as shown in Figure 9.6(c). In a more complex contour tree, each reversal of direction potentially adds more contours to the current selection.

In comparison, if we treat the changes to the isovalue slider as interim evolutions, and evolve each new set directly from the original contour A, then, as the user reduces the isovalue, G will not be generated. Instead, the effect will be to reverse the evolution shown in Figure 9.6(b), returning to contours B and C only. This has the merit that, if the user returns multiple times to the same isovalue, they see the same set of contours each time.

Thus, when evolving a contour, we call the initial contour, in this case A, as the *selection root*. As the isovalue is changed, we extract the selection by searching monotone paths through the contour tree from the selection root to the new isovalue. Since the selected set of contours will change over time, we will keep track of them separately from the contents of the flexible isosurface. When a new contour, or no contour, is selected, we will *commit* the selection, transferring each of them from the selection to the flexible isosurface proper.

9.6.3 Giving Precedence to Evolving Contours

Another issue arises when we adjust a contour so that it moves to or past a contour that is already present, as shown in Figure 9.7. Assume that contour is already present, and that the user has just selected contour A in the contour tree for evolution. For many data sets, B will be entirely inside A or vice versa. As the isovalue is increased in Figure 9.7(a), A evolves into A_1 and A_2. When the contour is evolved further, in

Figure 9.7(b), we must decide what to do with B.

If B was inside A, then, as the isovalue increases, A evolves past B. If we leave B visible, then the user sees A_2 disappear into B, and does not see any further evolution to A_5, even though A_3 and A_4 are visibly evolving. If A was inside B, we see only B until A passes B, at which point we see A evolving but not B, due to occlusion.

We assume that the user is less interested in B than in the evolution of A. Thus, any time the set of contours evolves past a contour such as B, we treat B as belonging to the evolving contour thereafter. If the user then reverses the evolution so that A_1, A_2 are generated from A, we must make B visible once more. As with repeated evolutions in Section 9.6.2, we do this so that the evolution in *reversible*: i.e. the user can return to an earlier isovalue and obtain exactly the same set of contours.

We implement this by marking B as *suppressed* while A evolves past it. Suppressed contours are not rendered in the data display for the duration of the evolution. When the evolution reverses back past B, we remove the suppression mark from B. When the selection is committed, or transferred, to the flexible isosurface proper, any currently suppressed surfaces are deleted, so that unselection does not have the effect of adding contours to the data display.

9.6.4 Multiple Selection

The final issue that we have to tackle is *multiple selection*: whether the user should be able to select several contours simultaneously, as for example A and B in Figure 9.7. We have chosen to disallow this in order to keep the semantic meaning of the user interface elements as clear as possible.

We have chosen to have the isovalue slider show the isovalue of the contour selected: i.e. the vertical position of the slider corresponds to the vertical position of the tag in the contour tree. If we allow multiple selection, we must assume that the contours selected all have different isovalues: which isovalue should we show in the slider? And, as we move the slider, should we move all the contours to the same isovalue, or should we make the isovalue change relative to the original isovalue? If we evolve all of the contours to the same isovalue, then we will see large perceptual jumps, as contours farthest away in isovalue expand very rapidly, while contours close to the new isovalue change little. If we evolve the contours to different isovalues, we are faced with the problem of defining which isovalue each contour will jump to. Moreover, any such definition should be one which is meaningful to the user.

It would be feasible, for instance, to have all of the contours in the selection evolve at the same rate. We have elected not to implement this at present, but may do so in the future. In practice, which solution is most useful for a particular purpose is likely to be domain-dependent. Since this thesis is principally concerned with the principles of manipulating contours via the contour tree, we have chosen a simple solution: restricting evolution to a single contour at a time. This also has the merit that the data structures to keep track of one evolving contour are simpler than those needed to keep track of multiple evolving contours.

9.7 Data Structures for Flexible Isosurface Manipulation

Now that we have established the operations we wish to perform, we can define suitable data structures and algorithms for carrying them out. The relevant data structures are arrays, taking $O(t)$ storage space.

active: This array stores the contours in the current flexible isosurface. Each contour is stored as a superarc ID and an isovalue. For each superarc, we also store the position in the active array, if any, for efficient deletions from the active array.

suppressed: This array stores a flag marking whether the contour on each superarc is currently suppressed due to a contour evolution. For convenience, we store this in the superarc data structure.

selected: An array storing the contours that are currently selected. As described in Section 9.6, this array is used while a contour is being evolved. Upon unselection the contours in this array are transferred to the active array, while suppressed contours are permanently removed from the active array. We refer to this transfer as *committing* the selection.

restorable: This array stores the set of contours most recently deleted as a result of a deletion or isolation operation. Ancillary variables keep track of whether the operation was deletion or isolation and, if it was deletion, what the corresponding values of selectionRoot and currentSelectionValue were.

selectionRoot: We saw in Section 9.6 that we need to store the selection root - the contour from which the evolution commenced. We also need to store the current isovalue of the evolving selection, in a variable called *currentSelectionValue*.

9.8 Algorithms for Flexible Isosurfaces

In Section 9.3.3, we gave a list of the operations our interface supports: in Section 9.5 and Section 9.6, we described how these operations are invoked, and described how these operations relate to the contour tree. In this section, we define algorithms to implement these operations.

Input : A contour tree $C = (V, E)$
An isovalue h
Output : A flexible isosurface F that consists of the level set at h.

```
1 F = ∅
2 for each superarc s in E do
3     if s spans h then
4         Add (s, h) to F
      end
  end
```

Algorithm 9.1: Naïve Algorithm to Define a Level Set

9.8.1 Defining A Level Set

To define a level set, we use the isovalue slider when no contour is selected, and the "local contour" toggle is off. A naïve implementation of this is shown as Algorithm 9.1. It is worth noting that this algorithm as stated is $O(t)$, since it searches every superarc of the tree.

Input : A contour tree $C = (V, E)$
An interval tree I storing each superarc $s = (u, v)$ as the interval $(f(u), f(v))$
An isovalue h
Output : A flexible isosurface F that consists of the level set at h.

1 $F = \emptyset$
2 StartSearchingForSuperarcs(I, h)
3 **while** not DoneSearchingForSuperarcs(I) **do**

end
4 Let s = GetNextSuperarc(I);
5 Add s to F

Algorithm 9.2: Algorithm Using an Interval Tree to Define a Level Set. This algorithm assumes that the interval tree supports the operations *StartSearchingForSuperarcs()*, *DoneSearchingForSuperarcs()* and *GetNextSuperarc()* to search the interval tree for all superarcs spanning a desired isovalue.

This can be reduced to $O(\log t + \tau)$ by storing all the superarcs of the contour tree in an interval tree [Ede80]: we show the improved algorithm using the interval tree as Algorithm 9.2. This, however, adds an additional $O(t \log t)$ cost to the contour tree computation.

In practice, we have chosen to accept the linear cost, since we will be reducing t to a few hundred with the techniques in Chapter 11. Moreover, with online simplification of the contour tree, we would need to modify the interval tree (or k-d tree) to keep track of which superarcs were currently valid. Although feasible, this adds further complexity to the program.

9.8.2 Algorithm for Evolving A Contour

We described how to evolve a contour to a desired isovalue in Section 9.6, and defined some interface policies in Section 9.6.1. Algorithm 9.3 shows how evolution is implemented with a simple search through the contour tree, suppressing active edges as needed. Since we have implemented this as a breadth-first search, it is not difficult to see that it executes in $O(t)$ time in the worst case.

9.8.3 Choosing Sets of Contours

We have also used the flexible isosurface to reproduce the functionality of Silver's Object-Oriented Visualization [Sil95] and the Largest Contour Segmentation of Manders et al. [MHS$^+$96]. Fortunately, each of these is a special case of a flexible isosurface, and each is easily computed.

Silver [Sil95] defined contours which were the largest topologically unique objects near local maxima, as shown in Figure 9.8(a). As noted in the caption of the figure, this implicitly assumes that "large" refers to the volume of the upstart region (see Chapter 10 of these contours. In this case, extracting the relevant contours from the contour tree is trivial: all that is required is to descend from each local maximum until a supernode is detected, then take a contour at a short distance above the supernode.

Manders et al. [MHS$^+$96] defined contours that contained only one local maximum, using a region-growing algorithm similar in nature to the join sweep algorithm in Section 7.6.1. Figure 9.8(b) shows the

Input : A contour $R = (r, h_0)$ as the selection root.
The current flexible isosurface F
An isovalue h
Wlog, $h < h_0$, the isovalue of the selection root.

Output : The set S of contours at h into which R evolves
Suppress active contours between $f(R)$ and h.

```
1  S = ∅
2  searchQueue = {r}
3  for each active arc a in F do
4  |  Clear suppressed flag on a
   end
5  while searchQueue is not empty do
6  |  remove a from searchQueue
7  |  if a is active then
8  |  |  if isovalue of contour on active arc a is less than h then
   |  |  |
   |  |  end
9  |  |  Suppress a.
   |  end
10 |  if a spans h then
11 |  |  Add a to S
   |  else
12 |  |  for each arc b whose upper end is the lower end of a do
13 |  |  |  add b to searchQueue
   |  |  end
   |  end
   end
```

Algorithm 9.3: Evolving the Contour from the Selection Root and Isovalue

results in terms of the contour tree. Note that the region that grew from the local maximum labelled $LM1$ stops at the isovalue of B: thus, A, which has yet to reach the local minimum, will probably enclose a hollow in the middle of the object defined with this segmentation. The authors do not indicate whether this was a problem for their application.

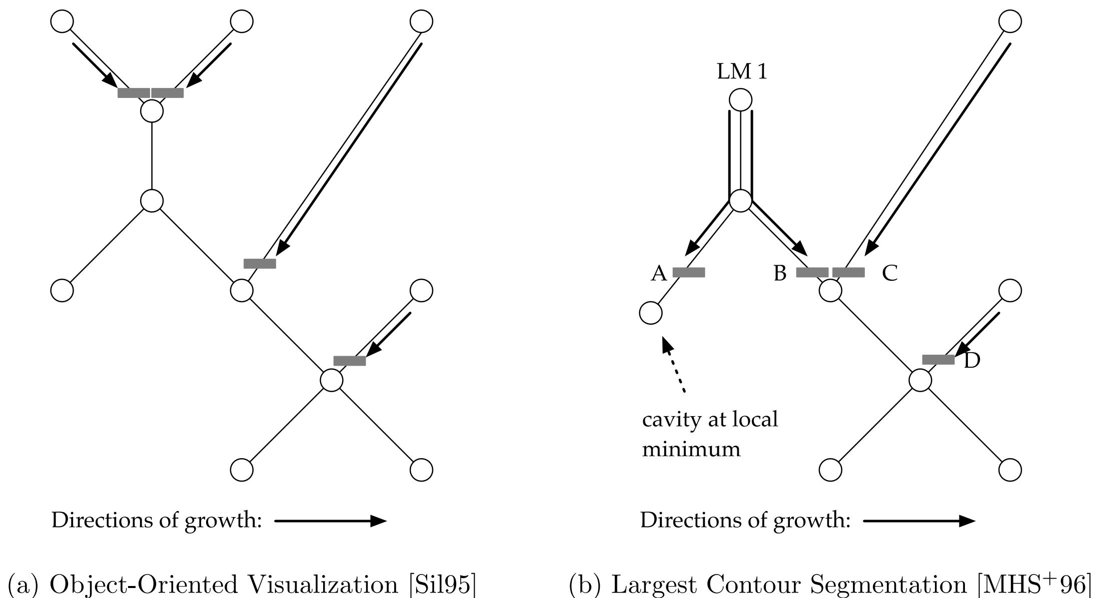
Input : A contour tree $C = (V, E)$
A proportion r

Output : A flexible isosurface F that shows object-oriented visualization, parameterized along the upper edges of the tree with a parameter r.

```
1  F = ∅
2  for each e = (u, l) in E, where f(u) > f(l) do
3  |  if u is an upper leaf of C then
4  |  |  Let h_e = f(l) + r × (f(u) − f(l))
5  |  |  Add (e, h) to F
   |  end
   end
```

Algorithm 9.4: Object-Oriented Visualization from the Contour Tree

Since Object-Oriented Visualization and Largest Contour Segmentation both propagate outwards from local maxima (i.e. a downwards isovalue sweep), they can both be found by starting a contour at each local maximum, and evolving it until a suitable cutoff is reached. For simplicity, we have implemented



(a) Object-Oriented Visualization [Sil95] (b) Largest Contour Segmentation [MHS+96]

Figure 9.8: Object-Oriented Visualization and Largest Contour Segmentation.
In (a), sweeping down from each peak to the nearest saddle defines the largest unique object surrounding that peak. Implicitly, the size of the object refers to the volume of the upstart region, which we discuss in Chapter 10.
In (b), growing regions from peaks achieves essentially the same result, although there is some ambiguity in how to treat cavities.

Silver's Object-Oriented Visualization in Algorithm 9.4. Moreover, we have parameterized the isovalue so that the contours need not appear immediately above the supernode bounding the superarc, but can be adjusted simultaneously to view their evolution.

9.9 Layout Problems in the Contour Tree Display

In this chapter, we have seen how the contour tree can be used to manipulate individual contours, and as a visual index to possible contours. However, as a visual index, it has some drawbacks. In all the examples so far, we have carefully chosen contour trees that are easy to draw. Some contour trees, however, are difficult to draw for one of two reasons. Some contour trees cannot be drawn without edge crossings, and some simply have too many superarcs to be practical to draw or to use as a visual index.

Although any tree can be laid out in two dimensions without superarcs crossing, this is no longer true if one coordinate is constrained. Since we wish to define the y coordinate according to the isovalue at the supernode, trees such as Figure 9.9 cannot be drawn without crossing superarcs.

In general, graph layout is a difficult problem, with entire texts being devoted to the question [dBETT99]. Since the contour tree does not have a root and since we wish to define the y-coordinate, our layout problem can be classified as drawing a layered free tree while minimizing edge crossings. Although algorithms for drawing layered graphs and free trees exist, this particular problem does not seem to have been addressed, perhaps due to a lack of a driving application.

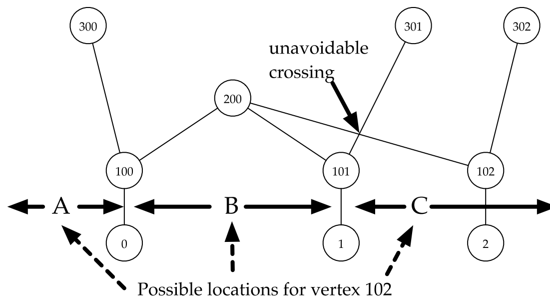


Figure 9.9: A Contour Tree with Intersecting Superarcs

Moreover, minimizing the number of crossings in a layered graph is known to be a hard problem [dBETT99]. Whether the fact that we have a tree instead of graph simplifies the task also does not seem to have been addressed. For graphs in general, tools such as *dot*[1], assign layers to the vertices in a graph, then employ heuristics based on energy-minimization to lay out graphs with few crossings. These methods, however, are relatively slow, often taking a minute or more to lay out a graph with a few hundred nodes. For contour trees with upwards of a million nodes, this is clearly impractical.

Because the topic of suitable layout is clearly a major undertaking in its own right, we do the following:

1. We show in Lemma 9.3 that there exist layered free trees in which crossings are unavoidable.
2. We initially assign arbitrary x-positions to supernodes.
3. We allow the user to adjust supernode positions by hand, and to save and load supernode positions for a given contour tree.
4. We have implemented a routine that invokes *dot* to lay out contour trees of less than 256 superarcs.

In practice, the difficulty of laying out large contour trees in a meaningful fashion is one of the major motivations for undertaking simplification of the contour tree in Chapter 11.

Lemma 9.3 *It is not always possible to draw a contour tree C such that the y-position of each supernode is proportional to the isovalue at that supernode, and such that no two superarcs intersect in the drawing.*

Proof: A counter-example is shown in Figure 9.9. Supernode 300 must be placed either to the left or to the right of supernode 200. Without loss of generality, assume that it is to the left of 200. Supernode 100 can be on either side of 200: for clarity, it is also shown to the left. If supernode 101 is placed to the left of 100, then superarc $200 \rightarrow 101$ must intersect superarc $300 \rightarrow 100$. Therefore, supernode 101 must be placed to the right of supernode 100. And if superarc $301 \rightarrow 101$ is to avoid intersecting edge $200 \rightarrow 100$, supernode 301 must be to the right of supernode 200.

There are then three possible locations for supernode 102, shown in Figure 9.9 as A, B, and C. If A is chosen, then superarc $200 \rightarrow 102$ will intersect superarc $300 \rightarrow 100$. Similarly, if C is chosen, then

[1] Available at `http://www.research.att.com/sw/tools/graphviz/`.

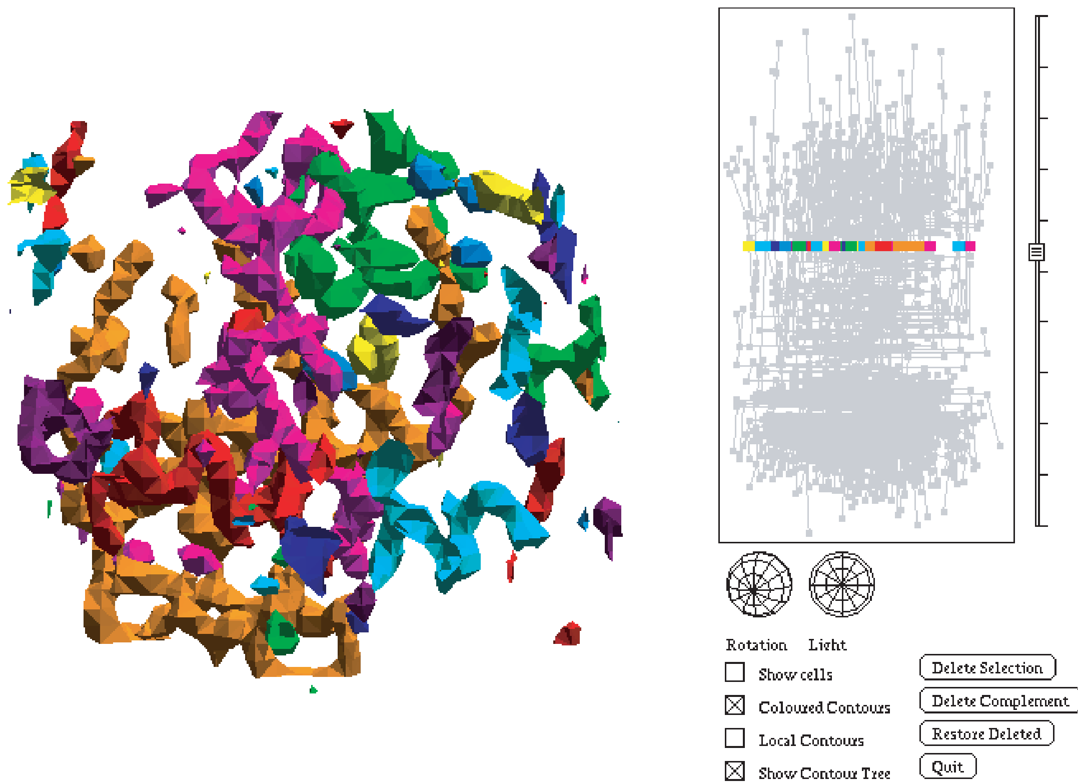


Figure 9.10: An Unusable Contour Tree

superarc 200 → 102 will intersect superarc 301 → 101, as shown in Figure 9.9. And if B is chosen, then superarc 302 – 102 will have to intersect either 200 – 100 or superarc 200 – 101. □

The other major difficulty in drawing contour trees is that they may consist of thousands or millions of superarcs. Such trees are next to useless for an interface, as shown in Figure 9.10. Trees with such a large number of superarcs can be caused by inherently complex data, or by noisy data. In either case, however, if we wish to use the contour tree as a visual index to contours, we need to reduce the tree to a manageable size. This, however, is a separate topic, which we deal with in Chapter 11.

9.10 Summary of Contributions

In this chapter, I have introduced the following contributions:

1. I have shown how to use the tight correspondence between the contour tree and individual contours to treat individual contours as distinct objects.
2. I have shown how to use this to heighten visual contrast of complex surfaces by assigning independent colours to different surfaces.
3. I have shown how to isolate individual contours by suppressing all other contours in an image, or to delete specific contours from view.

4. I have shown how to use the contour tree to track the evolution of individual contour surfaces as the isovalue of that surface is varied.

5. I have introduced the *flexible isosurface*, which enables user-driven exploration of arbitrary scalar data, without any *a priori* information about the data being studied.

6. I have shown that Silver's Object-Oriented Visualization [Sil95] and the Largest Contour Segmentation of Manders et al. [MHS+96] are special cases of the flexible isosurface.

In the next chapter, we start addressing the principal difficulty when using the contour tree as a visual representation of data: complexity and vulnerability to noise.

Chapter 10

Local Spatial Measures

In the previous chapter, we showed how to use the contour tree as a systematic index to individual contours of the data set. The principal contribution of this chapter is to show how to add geometric functions such as area, perimeter, and volume to the contour tree, combining topological and geometric information. In the next chapter, we will then use the geometric information to guide simplification of the contour tree.

We will show how to use regions bounded by contours to localize these measurements to particular regions of the domain of the input function. Since these measurements are spatial in nature, we will refer to them as *local spatial measure*.

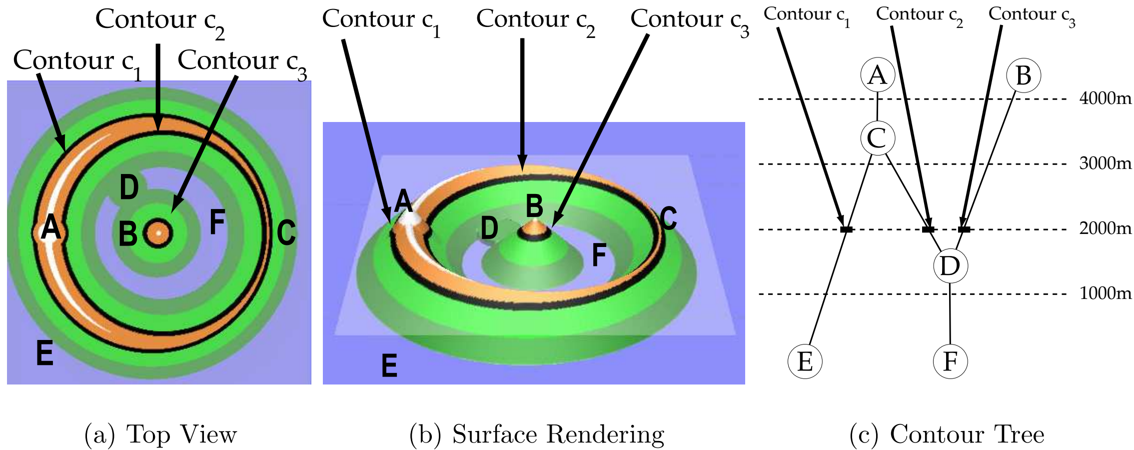
In this chapter, Section 10.1 discusses previous work, while Section 10.2 describes the regions for which we will compute geometric functions. In Section 10.3, we define the regions we will work with. In Section 10.4, we define functions with respect to any region bounded by a single contour, and consider some properties of these functions. Then, in Section 10.5, we give a formal algorithm for computing these functions, while in Section 10.6, we give an example of how this algorithm works. In Section 10.7, we discuss some spatial measures that we can compute in this fashion, including geometric error between the original and simplified function. In Section 10.8, we show how to compute approximate local spatial measures using node and cell counts. Finally, in Section 10.9, we review the contributions in this chapter.

10.1 Previous Work

One way to compute geometric properties of a contour is to extract the contour explicitly and measure it geometrically. This, however, can cost as much as $O(N)$ time in the size of the mesh for each contour considered, a prohibitive cost if we wish to examine many contours during simplification.

In their work on the Contour Spectrum [BPS97], Bajaj, Pascucci & Schikore show that for simplicial meshes, certain geometric functions of the region "above" a level set are described by a piecewise polynomial function with breakpoints at the isovalues of the vertices.

This result follows principally from the observation that functions such as contour length, area, &c. are decomposable [Ben79] into the sum of the function over each individual cell of the mesh. In a simplicial



(a) Top View (b) Surface Rendering (c) Contour Tree

Figure 10.1: Example of a volcanic crater lake.
A: peak on crater rim. B: peak of island in lake. C and D: saddle points. F: lake surface.

mesh, this reduces the problem to computing the function in each simplex independently. In each simplex, contour size (length in 2-D, area in 3-D) is given by a piecewise polynomial function of the isovalue parameter h, with knots at the isovalues of the vertices of the cell. Since piecewise polynomial functions are closed under addition, it follows that the total size of the contours in the level set is also a piecewise polynomial function. Similarly, since the cross-sectional area (in 2-D) or volume (in 3-D) is just the integral of the contour size with respect to the isovalue, and is therefore also a piecewise polynomial function.

However, Bajaj, Pascucci & Schikore [BPS97] computed piecewise polynomial properties only for the region (and contours) defined by a level set. They did not compute properties for individual contours, merely for the aggregate of all contours at a given isovalue. Moreover, although the contour tree was also computed for the same data, the geometric properties were not tied to individual contours in the contour tree.

Pascucci [Pas01] tied a topological property to the contour tree: the topological indices called the Betti numbers, building first on the contour sweep algorithm of van Kreveld et al. [vKvOB+97], then, with Cole-McLaughlin [PCM02], on the sweep and merge algorithm of Carr, Snoeyink & Axen [CSA03]. In doing so, they implicitly took advantage of the way the merge step sweeps a single contour at a time through isovalues, sometimes from high to low, sometimes from low to high. We will do likewise.

Pascucci & Cole-McLaughlin computed only this one topological property, not geometric functions of regions bounded by a contour. In the balance of this chapter, we will show how to do so.

10.2 Geometric Properties of Regions Bounded By A Contour

In the previous section, we saw that Bajaj, Pascucci & Schikore [BPS97] computed geometric functions of the region "above" a level set, and that Pascucci [Pas01] and Pascucci & Cole-McLaughlin [PCM02] computed topological properties of individual contours, based on the sweep and merge contour tree algorithm of Carr, Snoeyink & Axen [CSA03]. We will combine these ideas to compute geometric properties for contour-bounded regions, again using the sweep and merge contour tree algorithm as a starting point.

Bajaj, Pascucci & Schikore [BPS97] defined geometric properties in terms of a sweeping (hyper)-plane, which divides the manifold of f into two distinct sets: the region “above” and the region “below”. For individual contours, it is harder to define geometric properties. Consider Figure 10.1, which shows the contour tree and some contours of a volcanic crater lake. It is clear what is meant by the region above the 2000m level set: this consists of two regions, one consisting of the rim of the volcano, the second the island in the caldera. But what is meant by the region above the 2000m contour marked c_1? Even the region above 2000m that has c_1 as a boundary is arguable, as its boundary is defined by both c_1 and c_2.

Moreover, we do not require that contours be closed curves (or surfaces). Instead, we work with regions bounded by a single contour and (possibly) the boundary of the data set. Contours in general divide the region $\mathcal{R}$ into subregions. A contour $c = (s, h)$ on a superarc s of the contour tree divides $\mathcal{R}$ into exactly two subregions, one of which includes the region $f(s)$ restricted to isovalues above h, the other of which includes the region $f(s)$ restricted to isovalues below h. A contour that belongs to a supernode v of the contour tree divides the region $\mathcal{R}$ into one subregion for each superarc incident to v. For each such superarc s, the corresponding subregion will contain the region $f(s)$.

Note that the contour tree is well-defined even if the contours intersect the boundary, and so are the regions into which any given contour divides $\mathcal{R}$.

We compute geometric properties for these subregions. Instead of using “above” and “below”, we identify these regions by the arcs of the contour tree to which the subregions correspond. We will show in Section 10.5 that the computations of Bajaj, Pascucci & Schikore can be derived from our computations.

We assume that the properties we are computing are decomposable: that we can compute the property for a region bounded by a contour by computing the property in each cell, and combining the results [Ben79]. We assume that the combination will occur using addition. Although this assumption is not necessary, it simplifies discussion in this chapter.

Assumption 6 *Any geometric property that we compute will be decomposable by computing the property for each cell in the region of interest, then combining the results by addition.*

We also assume that the properties we are computing are piecewise-defined for each cell, and that, for each isovalue at which a property changes in a cell, the corresponding contour in that cell is inserted into the contour tree. For simplicial meshes and the properties that we consider, it suffices to work with the unaugmented contour tree. Further discussion of this will be deferred to Chapter 12.

Recall that, in the unaugmented contour tree, the edges and vertices are referred to as arcs and nodes instead of superarcs and supernodes: we will use arcs and nodes throughout this chapter as a reminder of this.

Assumption 7 *Any geometric property that we compute is piecewise-defined for each cell, and the break-points of the functional form of the property occur at isovalues of the vertices of the cell.*

10.3 Upstart and Downstart Regions

To define the regions over which we compute geometric properties, we start by recalling that from any contour c, it is possible to ascend along zero, one, or more arcs in the contour tree, and correspondingly into zero, one, or more connected components of $\mathcal{R} - c$. For each ascending arc a, we collect the set of points in $\mathcal{R} - c$ which can only be reached by passing through contours belonging to a that have isovalues $> h$. Since these points can only be reached from c by starting upwards along a, we will call the set an *upstart* region of a at c.

We also recall that any contour c that belongs to an arc a of the contour tree can be expressed as a pair (a, h), where h is the isovalue of c. For a contour c which belongs to a node μ, we cannot in general assign c to a particular arc incident to μ, and must treat them slightly differently.

Because we treat contours passing through nodes in a different fashion than contours not passing through nodes, we define upstart regions separately for each case. For convenience, we start by observing that, for any arc a of the contour tree, we can restrict the image $f(a)$ of the arc to values greater or lesser than a given isovalue h:

Definition 10.1 *Let a be an arc of the contour tree of f, and let h be any isovalue. Define the image of a restricted to isovalues above h, $f(a)|_{>h} = \bigcup\{c = (a, h') : h' > h\}$, where each c is a contour belonging to a. Similarly define $f(a)|_{<h} = \bigcup\{c = (a, h') : h' < h\}$, where each c is a contour belonging to a.*

Each arc a is an equivalence class of contours, and the image $f(a)$ is the union of the contours belonging to a, so this definition simply divides the image into the subregions with isovalues greater and lesser than a given h, respectively.

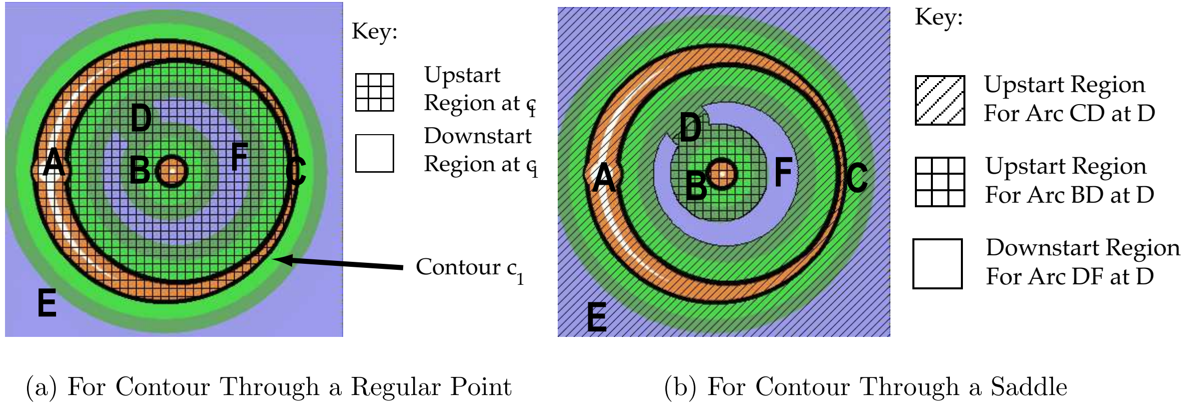
Definition 10.2 *Let $c = (a, h)$ be a contour of f at isovalue h that belongs to arc a of the contour tree for f. Define the* upstart *region of a at h to be: $R^+_a(h) = \{x \in \mathcal{R} - c : every\ path\ in\ \mathcal{R}\ from\ c\ to\ x\ intersects\ f(a)|_{>h}\}$ and the* downstart *region of a at h to be: $R^-_a(h) = \{x \in \mathcal{R} - c : every\ path\ in\ \mathcal{R}\ from\ c\ to\ x\ intersects\ f(a)|_{<h}\}$*

In Figure 10.2(a) we show sample upstart and downstart regions in the volcano example. In this figure, the hatched area is the upstart region $R^+_{CE}(h)$ for the arc CE at h, the isovalue of contour c_1. Similarly, the unhatched area is $R^-_{CE}(h)$, the downstart region for the arc CE at h. Between them, these two regions partition $\mathcal{R} - c$.

For contours that pass through nodes, we use limits to define the upstart and downstart regions:

Definition 10.3 *Let c be a contour of f at isovalue h that belongs to node μ of the the contour tree T for f, and let a be an arc of T incident to μ. Define the* upstart *region of a at h to be $\lim_{w \to h^+} R^+_a(w)$ and the* downstart *region of a at h to be $\lim_{w \to h^-} R^-_a(w)$.*

In Figure 10.2(b), we show both upstart regions and the single downstart region for the contour that passes through saddle point D in our volcano example. The diagonally hatched area is the upstart region for arc CD at h, the isovalue of saddle D, while the cross-hatched area is the upstart region for arc BD at h. In this case, there is only one downstart region, consisting of the lake surface, which is unhatched.



(a) For Contour Through a Regular Point

(b) For Contour Through a Saddle

Figure 10.2: Upstart and Downstart Regions in Volcano Example.
A: peak on crater rim. B: peak of island in lake. C and D: saddle points. F: lake surface.
(a) shows upstart region and downstart region for contour c_1 through a regular point.
(b) shows two upstart regions and one downstart region for contour through saddle D.

Note that, in this example, the downstart regions of CD and BD also exist at h. Applying the limit in Definition 10.3, we see that the downstart region $R^-_{CD}(h)$ is actually the union of the downstart region $R^-_{DF}(h)$ and the upstart region $R^+_{BD}(h)$. Although $R^+_{BD}(h)$ cannot be reached by starting downwards from the contour through D, we still include it in the downstart region of CD, to preserve the complementarity of the upstart and downstart regions of CD at h. Similarly, the downstart region $R^-_{BD}(h)$ is the union of the downstart region $R^-_{DF}(h)$ and the upstart region $R^+_{CD}(h)$. Finally, the upstart region $R^+_{DF}(h)$ is the union of the upstart regions $R^+_{BD}(h)$ and $R^+_{CD}(h)$.

We call these regions *upstart* and *downstart* instead of up and down, above and below, or inside and outside, in order to avoid confusion. Because points in an upstart region can have an isovalue below that of c, we avoid using up or above to refer to the set. Similarly, we avoid using inside and outside because "inside" can refer to regions of higher or lower isovalue. Moreover, we may have multiple inside or outside regions, depending on our boundary conditions. Upstart and downstart, however, imply that paths into the region start off in the upwards or downwards direction, but may go in the opposite direction later. In general, any contour c divides the region $\mathcal{R}$ into one upstart region for each up-arc at c and one downstart region for each down- arc at c.

Lemma 10.1 *Let $c = (a, h)$ be a contour of f. Then the upstart and downstart regions at c partition $\mathcal{R} - c$, and each upstart or downstart region is simply connected.*

Proof: We start by showing that the upstart and downstart regions at c partition $\mathcal{R} - c$, i.e. that every point $x \in \mathcal{R} - c$ belongs to an upstart or downstart region of c. We assume initially that c does not pass through a node of the tree, and can be written as $c = (a, h)$ where a is the contour tree arc to which c belongs and h is the isovalue of contour c.

Because f is continuous, for any point x in $\mathcal{R} - c$, there exists some path P_x from c to x that intersects c only at its starting point. By Theorem 6.6, there exists a corresponding path Q_x in the contour tree. But, because we have a tree, there are no cycles, and there is a unique shortest path R_x in the tree from c to x. Moreover, since there are no cycles in the tree, every point in R_x must also be in Q_x, so R_x is a subset of Q_x.

Since R_x is a shortest path, and there are no horizontal edges in the tree, R_x must either ascend or descend from c. If R_x ascends from c, then R_x must include at least one contour $d = (a, h')$ contained in $f(a)|_{>h}$. Since we know that R_x is a subset of Q_x, d belongs to Q_x, and P_x contains at least one point of d. It follows that x belongs to the upstart region of a at h. Similarly, if R_x descends from c, then x belongs to the downstart region of a at h.

To see that the upstart and downstart regions are simply connected, consider $R_a^+(h)$, the upstart region of a at c. Any point in $f(a)|_{>h}$ immediately satisfies Definition 10.2, and belongs to $R_a^+(h)$. Any other point x in $R_a^+(h)$ will be connected to c by some path P_x in $\mathcal{R} - c$, and, by the same argument just used, P_x intersects some contour d contained in $f(a)|_{>h}$. By Definition 10.2, every point on P_x between d and x must also be in $R_a^+(h)$, so x is connected to $f(a)|_{>h}$ in $R_a^+(h)$. Since this is true for every point x in the upstart region $R_a^+(h)$, the upstart region is simply connected.

This takes care of upstart regions defined by Definition 10.2, but not those defined by Definition 10.3. We now assume that c passes through a node of the tree, and apply essentially the same argument, using Definition 10.3. As w approaches h, we generate a shortest path $R_x(w)$ through the contour tree for each value of w. In the limit, even if x is very close to the contour c, all such $R_x(w)$ must share a common first arc a, and it follows that x belongs to the upstart (or downstart) region of a depending on whether the $R_x(w)$ ascend or descend along a. It follows that every point $x \in \mathcal{R} - c$ belongs to an upstart or downstart region of some arc a at c, and that these regions partition $\mathcal{R} - c$. Proof that the upstart or downstart region is simply connected follows similarly. □

10.4 Geometric Properties of Upstart and Downstart Regions

We can now compute geometric properties for upstart and downstart regions. We start by assuming that p is a geometric property that is computable for any subregion of $\mathcal{R}$. Since upstart and downstart regions are subregions of $\mathcal{R}$, we can define functions that compute the property for upstart or downstart regions for contours on any arc of the contour tree.

We call these functions *sweep functions*, since we will compute them by sweeping a contour up or down the arcs in the contour tree, and simultaneously sweeping the contour through the underlying scalar field. Since the contour c changes as the isovalue h is varied, these sweep functions take h as a parameter:

Definition 10.4 *For any arc $a = (u, v)$ and any geometric property p, we define the upward sweep function $p_a^{\uparrow}(h) = p(R_a^-(h))$ and the downward sweep function $p_a^{\downarrow}(h) = p(R_a^+(h))$.*

Note that the upward sweep function is defined in terms of the downstart region: this is because we measure the property p for the region *behind* the contour being swept. Thus, as the contour sweeps upward, the downstart region gets progressively larger. Similarly, the downward sweep function is defined in terms of the upstart region.

To compute these sweep functions efficiently, however, we need to be able to represent them analytically - for example as polynomials or piecewise polynomials, which can be stored as a set of co-efficients and evaluated for any desired isovalue h. To achieve this, we require that p be decomposable into piecewise polynomials for each cell intersecting the contour and a single polynomial for all the interior cells of the upstart or downstart region. This constraint allows us to update a piecewise polynomial for the sweep function at the vertices of the simplicial mesh that are swept through by the contour.

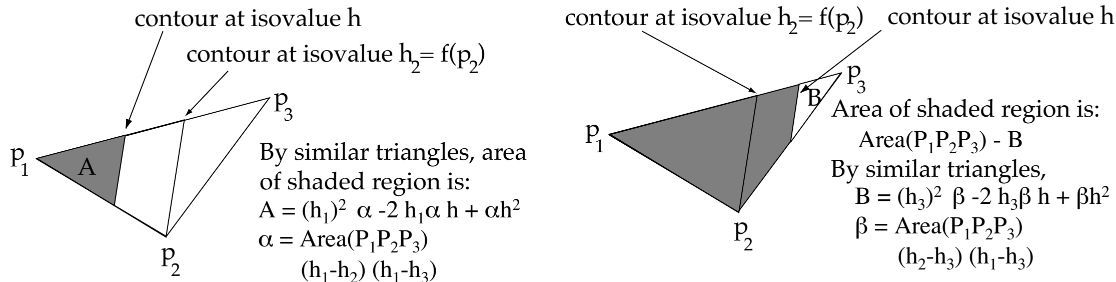


Figure 10.3: Sweep Function For Area in a Single Triangle.
Computing the area of upstart regions in a single triangle.
In each diagram, we compute the area of the upstart region as an explicit polynomial of h.
In the first diagram, the area of the shaded upstart region is given by the downward sweep function for arc p_1p_2: in the second, by the downward sweep function for arc p_2p_3.
h_1, h_2, h_3 are the isovalues at vertices p_1, p_2, p_3.

We start by briefly reviewing one of the results of Bajaj, Pascucci & Schikore [BPS97]. In a simplicial mesh, properties such as area, contour length, &c. are decomposable, and can be computed for each cell individually. In Figure 10.3, we show how the area of the upstart region can be computed as an explicit function of the isovalue h. In particular, note that the analytical form of this function is different over the intervals $(h2, h1)$ and $(h3, h2)$, but that, over each of these intervals, the coefficients of the polynomial are determined by the isovalues of the vertices p_1, p_2 and p_3.

As a plane is swept through a triangular mesh, it sweeps past the vertices of the mesh, allowing properties such as area to be computed progressively for the region above the sweep plane. As each vertex is swept past, the polynomial coefficients for the entire region are updated for the changes in the coefficient for the cells incident to the vertex.

For upstart and downstart regions, however, the situation is slightly more complex. In Figure 10.4, we show a contour being swept through the triangular mesh that we have used previously. Here, the contour is shown at an isovalue of 75: note that the region around the pit labelled 71 is included in the upstart region of the contour, even if it is below the isovalue of the sweep contour. For properties such as area and contour length, we can update the polynomial coefficients of the boundary, secure in the knowledge that the cells completely within the upstart region have been dealt with.

If, however, we are computing the volume between the manifold and the plane defined by the sweep contour, the coefficients will need updating for vertex 71 as well as for the vertices swept past by the sweep contour, even if vertex 71 has already been processed. To avoid this, we can instead compute the volume of the simplified surface obtained by pruning the edge $80-71$, as discussed in Section 11.3. If the correct volume is required for a given upstart or downstart region, we can compute it from the volume of the simplified surfaces by taking an alternating sum through the arcs of the contour tree that correspond to the upstart or downstart region of interest.

To deal with this properly, we need to impose a constraint on our decomposable property: that it be computable by decomposing the region into the *boundary cells* of the upstart region - cells that intersect the contour and the *interior cells* of the upstart region - cells that are entirely contained within the upstart region. We therefore assume the following:

Assumption 8 *Let p be any decomposable geometric property p being computed by sweeping a contour down*

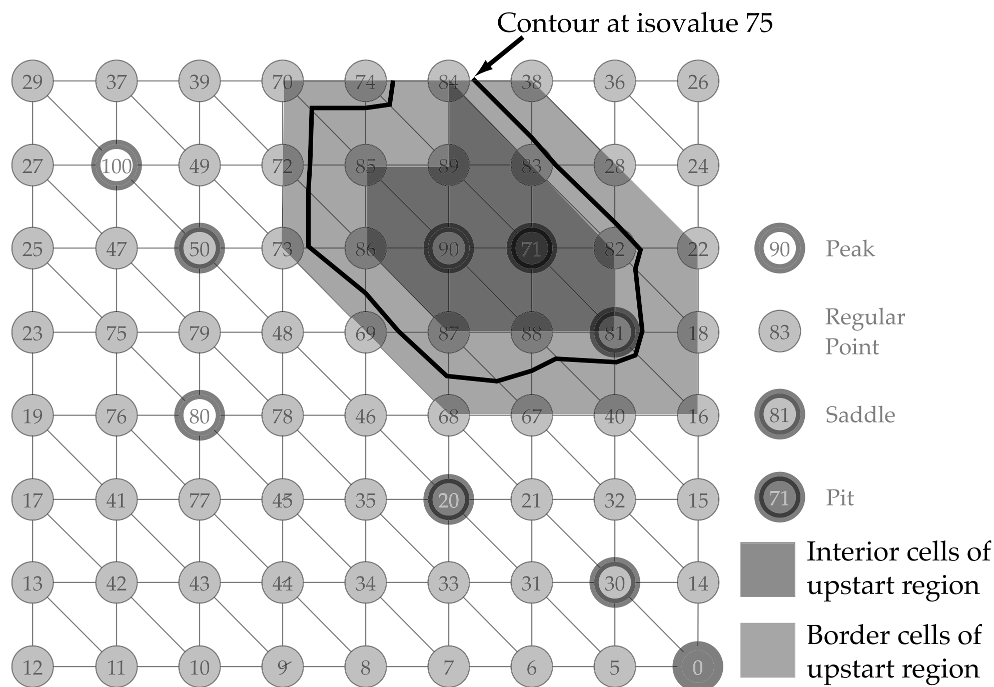


Figure 10.4: Decomposing A Downward Sweep Function. We decompose the downward sweep function into one polynomial for all of the interior cells of the corresponding upstart region, and one polynomial for each border cell of the upstart region.

an arc a of the contour tree. We assume that, for each interior cell K of the upstart region, the analytic form of $p|_K$ does not have any breakpoints over the range of isovalues corresponding to the arc a.

Combined with our assumption that no two vertices have the same isovalue, this allows us to restrict the effects of the contour sweep to those vertices past which the contour sweeps.

Lemma 10.2 *Let $c_1 = (a, h_1)$ and $c_2 = (a, h_2)$ be two contours belonging to an arc $a = (u, v)$ of the contour tree, and let p be a decomposable geometric property that satisfies Assumption 7 and Assumption 8. Then $p(R_a^+(h_1))$ and $(R_a^+(h_2))$ share the same explicit analytic function in terms of h, as do $p(R_a^-(h_1))$ and $(R_a^-(h_2))$.*

Proof: Because p is decomposable, we decompose $p(R_a^+(h))$ on a cell-by-cell basis. For any interior cell K of the upstart region, we know by Assumption 8 that $p|_K$ does not have any breakpoints between $f(u)$ and $f(v)$. Since h_1 and h_2 must also be between $f(u)$ and $f(v)$, it follows that $p|_K$ does not have any breakpoints between h_1 and h_2.

For any boundary cell K of the upstart region, $p|_K$ can only have a breakpoint between h_1 and h_2 if a vertex w of K has an isovalue between h_1 and h_2. But, if this were the case, then w would be between u and v in the contour tree, since a contour would have to sweep through it between sweeping through u and v. By contradiction, therefore, $p|_K$ has no breakpoint in the interval (h_1, h_2).

Since none of the cells into which the upstart region has been decomposed have breakpoints for p in the range (h_1, h_2), it follows that the explicit analytic form for $p(R_a^+(h_1))$ and $(R_a^+(h_2))$ must be identical, as required. □

We now consider what happens as a contour sweeps past a vertex v of the mesh - which is also a node in the contour tree. Let the cells incident to the vertex be $K_1, \ldots K_r$, and consider some cell K which is not incident to v. By our assumption that no two vertices have the same isovalue, we know that K does not have a vertex with the same isovalue as v. By Assumption 7, p restricted to K does not have a break point at $f(v)$. It follows that we can update the coefficients for the explicit analytical form of p by inspecting only the cells $K_1, \ldots K_r$.

Let us first consider what happens at a leaf of the contour tree. An upper leaf of the contour tree has no up-arcs, and we can assume that p does not exist above the leaf. A downwards sweep past the leaf therefore starts with $p = 0$ and an empty contour. Immediately below the leaf, the contour will intersect only the cells incident to the leaf, making it easy to compute:

Lemma 10.3 *At an upper leaf μ of the contour tree with incident arc b,*

$$p_b^{\downarrow} = \sum_{j=1}^{r} (p|K_j)_b^{\downarrow} \tag{10.1}$$

Proof: Since μ is an upper leaf, there is no contour above it, and only one below it. Before sweeping a contour past μ, the set of boundary cells of the upstart region will be empty, as will the set of interior cells. After sweeping a contour past μ, cells $K_1, \ldots, K_r$ will be the complete set of boundary cells, while the set of interior cells will remain empty. The result then follows from the decomposability of p. □

At lower leaves, the result is similar (proof omitted):

Lemma 10.4 *At a lower leaf μ of the contour tree with incident arc a,*

$$p_a^{\uparrow} = \sum_{j=1}^{r} (p|K_j)_a^{\uparrow} \tag{10.2}$$

At regular points, we get a slightly more complex result:

Lemma 10.5 *At a regular point μ of the contour tree with incident up-arc a and down-arc b,*

$$p_a^{\downarrow} - \sum_{j=1}^{r} (p|K_j)_a^{\downarrow} = p_b^{\downarrow} - \sum_{j=1}^{r} (p|K_j)_b^{\downarrow} \tag{10.3}$$

and

$$p_a^{\uparrow} - \sum_{j=1}^{r} (p|K_j)_a^{\uparrow} = p_b^{\uparrow} - \sum_{j=1}^{r} (p|K_j)_b^{\uparrow} \tag{10.4}$$

Proof: Let K be any interior cell of the upstart region $R_a^+(f(\mu))$. K will also be an interior cell of the upstart region $R_b^+(f(\mu))$. By our assumption that isovalues of the vertices of the mesh are unique, $p|_K$ will not have a breakpoint at $f(\mu)$, so its contribution to $p_a^{\downarrow}$ and to $p_b^{\downarrow}$ will be identical. Similarly, if K is a boundary cell of $R_a^+(f(\mu))$ that is not incident to μ in the mesh, its contribution to $p_a^{\downarrow}$ and to $p_b^{\downarrow}$ will be identical.

It then follows that $p_a^{\downarrow}$ and $p_b^{\downarrow}$ differ only in the contributions of the cells $K_1, \ldots, K_r$, and since we subtract these contributions from each side of the equation, the result follows. □

The result of sweeping past a saddle, however, is not so simple, because the contours in cells $K_1, \ldots, K_r$ do not necessarily belong to the same arcs, either above or below the saddle. We must therefore look at all arcs incident to the critical point simultaneously. For a critical point of degree δ, we will need to know $\delta - 1$ of the sweep functions to compute the remaining one. In general, we assume that we already know the downward sweep functions of the up-arcs, and the upward sweep functions of the down-arcs.

Unfortunately, there is no general rule for the relationships of the sweep functions at a saddle, so we will have to prove the relationship for each function of interest.

We will start with a simple function: the contour size (length in 2-D, surface area in 3-D):

Lemma 10.6 *Let p be a function that measures contour size, and let a be an arc of the contour tree. Then $p_a^{\downarrow} = p_a^{\uparrow}$.*

Proof: We are measuring the size of the contour - that is, the boundary of a region. The complement of the region has the same boundary, and therefore has the same contour size. □

Theorem 10.7 *Let p be a function that measures contour size. Let μ be a saddle with up-arcs $a_1, \ldots, a_{\delta^+(\mu)}$ and down-arcs $b_1, \ldots, b_{\delta^-(\mu)}$. Let μ belong to cells $K_1, \ldots, K_r$, and let these cells intersect contours belonging to arcs $\alpha_1, \ldots, \alpha_r$ above μ and arcs $\beta_1, \ldots, \beta_r$ below μ. Choose any down-arc: without loss of generality, renumber so we choose b_1. Then:*

$$p_{b_1}^{\downarrow} = \sum_{i=1}^{\delta^+(\mu)} p_{a_i}^{\downarrow} - \sum_{i=2}^{\delta^-(\mu)} p_{b_i}^{\uparrow} - \sum_{j=1}^{r} (p|K_j)_{\alpha_j}^{\downarrow} + \sum_{j=1}^{r} (p|K_j)_{\beta_j}^{\downarrow} \tag{10.5}$$

Proof: We start by taking the contour sizes for all up-arcs at μ, and updating the cells $K_1, \ldots, K_r$ to sweep downwards past μ. Since the contour length is the sum of the lengths in the individual cells, if we update for each K_j, we get the sum of the combined contour size for the contours sweeping down from the top of all of the down-arcs. We then rearrange, and apply Lemma 10.6:

$$\begin{aligned}
\sum_{i=1}^{\delta^-(\mu)} p_{b_i}^{\downarrow} &= \sum_{i=1}^{\delta^+(\mu)} p_{a_i}^{\downarrow} - \sum_{j=1}^{r} (p|K_j)_{\alpha_j}^{\downarrow} + \sum_{j=1}^{r} (p|K_j)_{\beta_j}^{\downarrow} \\
p_{b_1}^{\downarrow} &= \sum_{i=1}^{\delta^+(\mu)} p_{a_i}^{\downarrow} - \sum_{i=2}^{\delta^-(\mu)} p_{b_i}^{\downarrow} - \sum_{j=1}^{r} (p|K_j)_{\alpha_j}^{\downarrow} + \sum_{j=1}^{r} (p|K_j)_{\beta_j}^{\downarrow} \\
&= \sum_{i=1}^{\delta^+(\mu)} p_{a_i}^{\downarrow} - \sum_{i=2}^{\delta^-(\mu)} p_{b_i}^{\uparrow} - \sum_{j=1}^{r} (p|K_j)_{\alpha_j}^{\downarrow} + \sum_{j=1}^{r} (p|K_j)_{\beta_j}^{\downarrow} \qquad (10.6)
\end{aligned}$$

□

At local extrema and regular points, this collapses to the terms shown in Lemma 10.4, Lemma 10.3, and Lemma 10.5, which gives us the following useful result:

Lemma 10.8 *Let p be a function that measures contour size. Let μ be any node of the contour tree, with up-arcs $a_1, \ldots, a_{\delta^+(\mu)}$ and down-arcs $b_1, \ldots, b_{\delta^-(\mu)}$. Let μ belong to cells $K_1, \ldots, K_r$, and let these cells intersect contours belonging to arcs $\alpha_1, \ldots, \alpha_r$ above μ and arcs $\beta_1, \ldots, \beta_r$ below μ. Choose any down-arc: without loss of generality, renumber so we choose b_1. Then Equation 10.5 holds.*

Proof: For saddles, this is true by Theorem 10.7. At local minima, there is no down-arc b_1, so this is trivially true. At local maxima, the first, second, and third summations are all 0, leaving Equation 10.1. And at regular points, the second term drops out, leaving Equation 10.3. □

The dual is also true for upward sweeps: both the statement of the dual and its proof are identical to that shown, except for the sweep direction, and are omitted.

For region size (the region to the specified side of the contour, a different property holds: that the sum of the region sizes is always equal to the size of the domain of f:

Lemma 10.9 *Let p be a function that measures region size, and let a be an arc of the contour tree. Then $p_a^{\downarrow} = p(\mathcal{R}) - p_a^{\uparrow}$.*

Proof: Under our assumptions of a simplicial mesh with barycentric interpolation and unique iso-values, we know that the contour itself is of 0 size (i.e. area for 2-D data, volume for 3-D data). Since each contour divides the space into disjoint regions, their sum must always be the size of the entire region. □

Although this is convenient, we still need to state what happens at saddles:

Theorem 10.10 *Let p be a function that measures region size. Let μ be a saddle with up-arcs $a_1, \ldots, a_{\delta^+(\mu)}$ and down-arcs $b_1, \ldots, b_{\delta^-(\mu)}$. Let μ belong to cells $K_1, \ldots, K_r$, and let these cells intersect contours belonging to arcs $\alpha_1, \ldots, \alpha_r$ above μ and arcs $\beta_1, \ldots, \beta_r$ below μ. Choose any down-arc: without loss of generality, renumber so we choose b_1. Then:*

$$p_{b_1}^{\downarrow} = \sum_{i=1}^{\delta^+(\mu)} p_{a_i}^{\downarrow} + \sum_{i=2}^{\delta^-(\mu)} p_{b_i}^{\uparrow} - \sum_{j=1}^{r} (p|K_j)_{\alpha_j}^{\downarrow} + \sum_{j=1}^{r} (p|K_j)_{\beta_j}^{\downarrow} \tag{10.7}$$

Proof: Again, we start by taking the sum of the sizes for all up-arcs at μ, and updating the cells $K_1, \ldots, K_r$ to sweep downwards past μ. Unlike the contour length, however, what we have computed is not the sum of the down-sweeping functions along the down-arcs. For each b_i, $p_{b_i}^{\downarrow}$ includes the area of *each* upstart or downstart region at μ except the downstart region for b_i. Thus, if we simply add the $p_{b_i}^{\downarrow}$ terms together, we will add several areas multiple times.

Instead, we sum the areas for the up-arcs and update for each K_j, to get:

$$\sum_{i=1}^{\delta^+(\mu)} p_{a_i}^{\downarrow} - \sum_{j=1}^{r} (p|K_j)_{\alpha_j}^{\downarrow} + \sum_{j=1}^{r} (p|K_j)_{\beta_j}^{\downarrow} \tag{10.8}$$

This expresses the combined area enclosed by the contours sweeping down all of the down arcs. To

get the area enclosed by one of them, we must add the known areas enclosed by the rest:

$$p_{b_1}^{\downarrow} = \sum_{i=1}^{\delta^+(\mu)} p_{a_i}^{\downarrow} + \sum_{i=2}^{\delta^-(\mu)} p_{b_i}^{\uparrow} - \sum_{j=1}^{r} (p|K_j)_{\alpha_j}^{\downarrow} + \sum_{j=1}^{r} (p|K_j)_{\beta_j}^{\downarrow} \tag{10.9}$$

□

As with Lemma 10.8, at local extrema and regular points, this collapses to the terms shown in Lemma 10.4, Lemma 10.3, and Lemma 10.5, which gives us the following useful result:

Lemma 10.11 *Let p be a function that measures contour size. Let μ be any node of the contour tree, with up-arcs $a_1, \ldots, a_{\delta^+(\mu)}$ and down-arcs $b_1, \ldots, b_{\delta^-(\mu)}$. et μ belong to cells $K_1, \ldots, K_r$, and let these cells intersect contours belonging to arcs $\alpha_1, \ldots, \alpha_r$ above μ and arcs $\beta_1, \ldots, \beta_r$ below μ. Choose any down-arc: without loss of generality, renumber so we choose b_1. Then Equation 10.7 holds.*

Proof: For saddles, this is true by Theorem 10.10. At local minima, there is no down-arc b_1, so this is trivially true. At local maxima, the first, second, and third summations are all 0, leaving Equation 10.1. And at regular points, the second term drops out, leaving Equation 10.3. □

Again, as with contour size, the dual is also true for upward sweeps: both the statement of the dual and its proof are also identical to that shown, except for the sweep direction, and are omitted.

10.5 Algorithm For Computing Local Spatial Measures

With these results in hand, we can state an algorithm for computing local spatial measures. As may be apparent from the properties we proved in the previous section, we will compute these properties using essentially the same merge algorithm that we use to construct the contour tree from the join and split trees.

Because the details of combining sweep functions and inverting them to reverse direction depend on the geometric property being studied, we assume that *Combine()* and *Invert()* functions are available that perform these operations for us.

We start by defining a function to update sweep functions at a vertex μ in Algorithm 10.1.

Lemma 10.12 *Algorithm 10.1 computes $p_a^{\uparrow}$ correctly in $O(degree(\mu) + N(\mu))$ time, where $N(\mu)$ is the number of cells incident to μ in the mesh.*

Proof: Step 2 computes

$$\sum_{i=1}^{\delta^-(\mu)} p_{a_i}^{\uparrow}$$

Step 3 then adds

$$-\sum_{j=1}^{r} (p|c)_{\alpha_j}^{\uparrow} + \sum_{j=1}^{r} (p|K_j)_{\beta_j}^{\uparrow}$$

```
SweepUpdate(p, μ, a)
Input    : Decomposable function p
           Vertex μ
           Arc a incident to μ. Without loss of generality, a ascends from μ
           (Implicit) Functions p_b^↓ for all up-arcs b at μ except a
           (Implicit) Functions p_b^↑ for all down-arcs B at μ
Output   : Functions p_a^↑ for arc a

1 Let p_a^↑ = 0
2 for each down-arc b at μ do
  |  Let p_a^↑ = p_a^↑ + p_b^↑
  end
3 for each cell c incident to μ do
4 |  Let α be the up-arc at μ, and β the down-arc, whose contours intersect c
5 |  Let p_a^↑ = p_a^↑ − (p|c)_β^↑ + (p|c)_α^↑
  end
6 for each up-arc b at μ except a do
7 |  if p measures contour size then
8 |  |  Let p_a^↑ = p_a^↑ − p_b^↓
  |
9 |  else if p measures region size then
  |  |  Let p_a^↑ = p_a^↑ + p_b^↓
  |
  end
```

Algorithm 10.1: Computing Sweep Functions at a Vertex μ

as in the dual to Lemma 10.8 and Lemma 10.11. Step 6 then adds or subtracts

$$\sum_{i=2}^{\delta^+(\mu)} p_{b_i}^{\downarrow}$$

where $b = b_1$ without loss of generality. Step 6 subtracts this term if sweeping through contour size, adds it if sweeping through region size, as in the duals of Lemma 10.8 and Lemma 10.11.

Thus, by the duals of Lemma 10.8 and Lemma 10.11, correctness follows.

Step 2 and Step 6 execute a total of $degree(\mu) - 1$ times at a cost of $O(1)$ each. Step 3 executes a total of $N(\mu)$ times at a cost of $O(1)$ each. Therefore the time required for this algorithm is $O(degree(\mu) + N(\mu))$. □

Once we have the routine for combining results at a vertex μ, we can define the local spatial measure algorithm in Algorithm 10.2.

Theorem 10.13 *Assume that $Invert(a)$ takes $O(1)$ time. Then Algorithm 10.2 correctly computes each $p_a^{\downarrow}$ and $p_a^{\uparrow}$ in $O(N)$ time.*

Proof: As each vertex and its incident arc are processed, we reduce the size of the tree C' by one. Step 11 adds each vertex to the queue as it becomes a leaf, guaranteeing that there is always at least one vertex on the queue to process, until the algorithm halts.

```
Input  : Contour Tree C
         Decomposable function p
         Function Combine(μ, a) to compute function at vertex μ along outgoing arc a
         Function Invert(a) which converts p_a^↓ to p_a^↑ and vice versa
Output : Functions p_a^↓ and p_a^↑ for each arc a in C
1  Make a copy C′ of C
2  for each vertex μ do
3  |  if μ is a leaf of C then
4  |  |  Enqueue(μ)
   |  end
   end
5  while NumberOfArcs(C′) > 0 do
6  |  Dequeue μ and retrieve arc a = (μ, μ) from C′
7  |  Without loss of generality, assume a ascends from μ
8  |  Let p_a^↑ = Combine(μ, a)
9  |  Let p_a^↓ = Invert(a)
10 |  Delete a from C′
11 |  if degree(μ) = 1 then
12 |  |  Enqueue(μ)
   |  end
   end
```

Algorithm 10.2: Computing Local Spatial Measures from Leaves Inwards

When a is processed at μ, we know that all other incident arcs to μ have been processed and deleted from C', so we know from Lemma 10.12 that Step 8 operates correctly.

Step 1 takes $O(t)$ time to compute, as does Step 2. Step 5 executes $O(t)$ times: for the unaugmented contour tree, this is $O(n)$. Each of the sub-steps of Step 5 except Step 8 takes $O(1)$ time, while Step 8 takes $O(degree(\mu) + N(\mu)$ time by Lemma 10.12. Summed over all vertices μ in a simplicial mesh, this is $O(N)$, which gives the desired result. □

We can add several small observations to this. First, it is not difficult to combine this algorithm with the merge sweep algorithm, and compute them simultaneously. Second, because we require every node in the mesh, the divide-and-conquer algorithm of Pascucci & Cole-McLaughlin [PCM02] will not give any advantage over the sweep-and-merge algorithm of Carr, Snoeyink & Axen [CSA03], as the former discards regular points to achieve its speed gains. Third, the algorithms of Chiang & Lu [CL03], and of Chiang et al. [CLLR02] do not compute the unaugmented contour tree, and cannot be used to compute local spatial measures.

Fourth, we can use our local spatial measures to compute the isovalued sweep functions from Bajaj, Pascucci & Schikore [BPS97]. Recall that we compute the property p for upstart and downstart regions along a particular contour. Bajaj, Pascucci & Schikore computed the property p for the region above a sweep (hyper-) plane. To obtain the equivalent, we compute the property for an upstart region, then adjust our result for other contours at the same isovalue.

For example, if we are computing area, we will need to subtract out the area of any enclosed downstart region, then add the area of any upstart regions inside that, and so on. Let v be any vertex in the tree, and perform a breadth-first search from v through the tree. Every time the breadth-first search passes the isovalue h of interest in a downwards direction on an arc a, we subtract $p_a^{\downarrow}$. Every time the search passes the isovalue h in an upwards direction on an arc a, we add $p_a^{\uparrow}$. By generating an alternating sum in

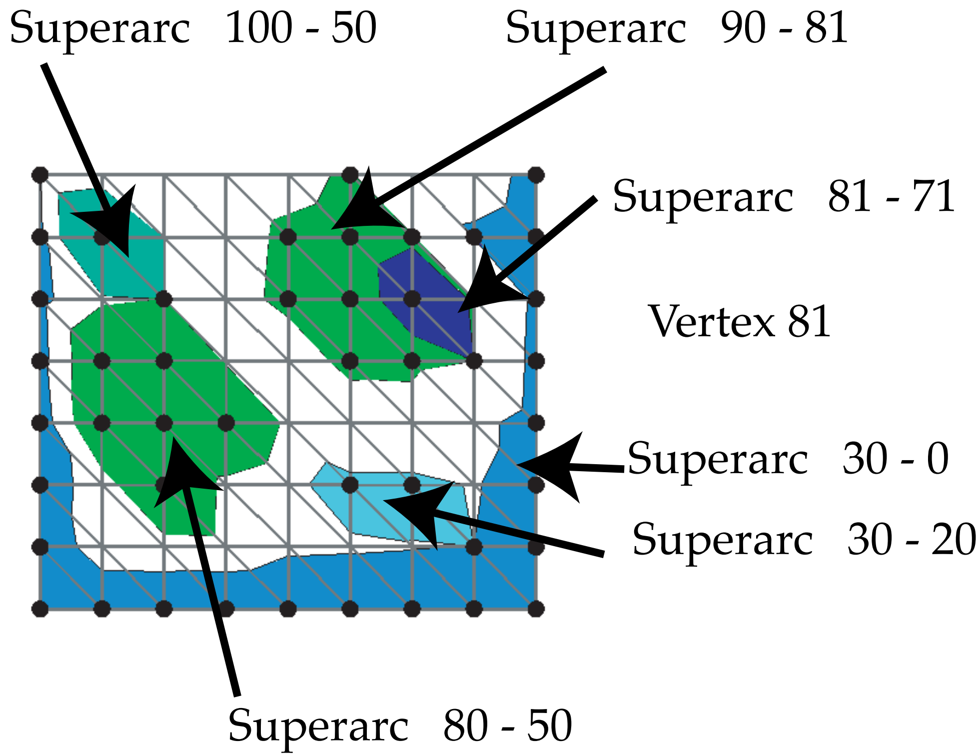


Figure 10.5: Computing the Function For Sweeping Down Superarc $81 - 50$

this fashion, we cancel out the regions below the sweep plane, giving the desired result.

We noted earlier that we could not compute volume directly with our algorithm, but could instead compute the simplified volume resulting from pruning edges of the tree that were contained in the upstart or downstart region being processed. We can then compute the correct volume with alternating sums, restricted to the subtree contained in the upstart or downstart region of interest.

10.6 Example of Local Spatial Measure Computation

Recall that in Chapter 7, we showed the progress of the merge phase in Figure 7.6: we reproduce a single step from this figure as Figure 10.5, showing the result of combining area functions at the saddle point 81. In this figure, each coloured region corresponds to the set of contours belonging to a single superarc. We are about to call Algorithm 10.1 to deal with the saddle at 81.

At 81, the areas corresponding to superarcs $90 - 81$ and $81 - 71$ are combined: the effect of Theorem 10.10 is to cancel out all the terms for the boundary between the two regions, leaving a constant representing the area of all triangles completely within the outer contour incident to vertex 81, and terms representing the outer contour's sweep downwards along $81 - 50$.

10.7 Sample Local Spatial Measures

It would be futile to attempt to list all of the geometric properties that can be computed as local spatial measures. Instead, we list some examples of properties that can be computed, starting with measures for two-dimensional scalar fields:

Isoline Length measures contour size and is computed using Theorem 10.7.

Cross-Sectional Area measures the area of the upstart region and is computed using Theorem 10.10.

Surface Area is the integral of the isoline length with respect to h and is computed using Theorem 10.10.

Volume is the integral of the cross-sectional area with respect to h and is discussed below.

Symmetric Difference is a measure of error based on volume

RMS Error is the root-mean squared error between two sets of samples, and is also discussed below.

Bounding Boxes can also be computed as local spatial measures.

Volume is the volume of the region under the function f inside the upstart or downstart region. Formally written, this is:

$$V = \int_A f(x, y)dxdy$$

where A is the upstart or downstart region in question. This is a decomposable property which can be expressed as the sum of the volumes for each cell of the mesh. Formally, the Riemann sum which defines an integral is a limit based on a decomposable property, so it is little surprise that integrals can be computed as local spatial properties.

Symmetric Difference is simply a measure of the geometric difference between two surfaces. In the case of two dimensional scalar fields, this simplifies to the volume bounded by the two surfaces. As we will see in the next chapter, the simplification method we use modifies the function by flattening out upstart or downstart regions. Thus, the symmetric difference between the original function and the simplified function will be:

$$V = \int_A f(x, y)dxdy - Area(A)h$$

for some isovalue h. Rather than computing this directly, it is simpler to compute the volume and the area separately, then compute the symmetric difference as needed.

Root Mean Squared Error measures geometric error between two sets of samples. Properly speaking, this is an approximate measure, and is discussed in the next section. For now, it suffices to note that it can be computed.

In three dimensions, each of these measures has its parallel: isoline length becomes **contour surface area**, cross-sectional area becomes **cross-sectional volume**, surface area becomes **hypersurface volume**, and volume becomes **hypervolume**.

10.8 Approximate Local Spatial Measures

We will not always want correct local spatial measures to high accuracy. For some purposes, we may be satisfied with approximations. For regular meshes, in particular, we can compute approximations as follows:

Lemma 10.14 *In regular meshes, the count of nodes along the arcs of the contour tree approximates region size.*

Proof: Recall that the area bounded by a curve can be computed using integral calculus using Riemann sums as the size of individual patches approaches 0. For a regular mesh, counting the nodes is equivalent to stopping the limit process with a patch size of 1×1 (in two dimensions). □

Lemma 10.15 *In regular meshes, the count of cells intersecting a contour approximates contour size.*

Proof: Again, recall that the correct contour size is a Riemann sum of segments (in two dimensions) or patches (in three dimensions) as segment or patch size approaches 0, or as the size of the cells into which we decompose the domain approaches 0. Again, we stop the limit process at a cell size of 1×1 (in two dimensions). □

Lemma 10.16 *In regular meshes, volume (2D) or hypervolume (3D), and therefore symmetric difference, can be approximated with the sum of the samples in an upstart or downstart region.*

Proof: In this case, the integral that defines the volume under the surface is defined to be the Riemann sum of the value of the function f as patch size approaches 0. □

Finally, we can compute Root Mean Squared Error (RMS) as follows:

Lemma 10.17 *Let f be a scalar field defined over a regular mesh, and let g be a simplified version of f with isovalue h inside an upstart or downstart region A, but which is equal to f elsewhere. Then the **Root Mean-Squared Error** of g is given by:*

$$RMS = \sqrt{\frac{\sum_{\vec{x} \in A} f(\vec{x})^2 - 2h \sum_{\vec{x} \in A} f(\vec{x}) + \sum_{\vec{x} \in A} h^2}{n}}$$

where n is the number of samples in $\mathcal{R}$, the domain of f and g.

Proof: Formally, RMS is defined in terms of the difference between the samples that define f and g. We develop the desired result as follows:

$$\begin{aligned} RMS &= \sqrt{\frac{\sum_{\vec{x} \in \mathcal{R}} (g(\vec{x}) - f(\vec{x}))^2}{n}} \\ &= \sqrt{\frac{\sum_{\vec{x} \in A} (g(\vec{x}) - f(\vec{x}))^2}{n}} \\ &= \sqrt{\frac{\sum_{\vec{x} \in A} (h - f(\vec{x}))^2}{n}} \end{aligned}$$

$$= \sqrt{\frac{\sum_{\vec{x}\in A}(h^2 - 2hf(\vec{x}) + f(\vec{x})^2)}{n}}$$

$$= \sqrt{\frac{\sum_{\vec{x}\in A} f(\vec{x})^2 - 2h\sum_{\vec{x}\in A} f(\vec{x}) + \sum_{\vec{x}\in A} h^2}{n}}$$

as required. □

Note that the second of these summation terms is the volume (or hypervolume) computed using the previous lemma, while the first summation term is the sum of the squares of the sampled values, and the last term is the area multiplied by the square of the simplified isovalue.

Each of the approximations discussed above is a piecewise function that changes only at vertices of the mesh. It follows that, with suitable minor changes, Algorithm 10.2 computes these properties correctly.

10.9 Summary

This chapter has two principal contributions: local spatial measures, and approximate local spatial measures. The first of these contributions extends work of Bajaj, Pascucci and others to compute exact geometric properties with respect to all possible isosurfaces in the contour tree, while the second approximates these properties. The principal value of these contributions will become apparent in the next chapter, in which we will use geometric properties to guide simplification of the contour tree.

Chapter 11

Contour Tree Simplification

The contour tree is vulnerable to noise in the input data. As a result, the contour tree for a large acquired data set can have millions of edges. This renders the contour tree impractical as an abstraction of acquired data, either for automatic processing, or for direct human visualization.

The principal contribution of this chapter is to show how to simplify the contour tree to a size humans can deal with, both to remove the effects of noise, and to enable large complex data sets to be visualized with the contour tree. As secondary contributions, I also show that simplifying the contour tree applies a meaningful simplification to the underlying data, and that this form of simplification can used for noise reduction.

Section 11.1 demonstrates the vulnerability of the contour tree to noise by examining the size of the contour tree for different types of data. Section 11.2 reviews previous work, discusses the stages of visualization at which simplification can be applied, and argues that the most appropriate stage is the stage at which we abstract the data and compute the contour tree. Section 11.3 then introduces the type of simplification that we apply to the contour tree, describes the corresponding simplifications to the underlying data, and states a set of rules to be respected when simplifying the contour tree. Section 11.4 then states an algorithm to perform the simplification, while Section 11.5 illustrates the results of applying different operations, rules and measures.

The balance of this chapter then looks at some practical issues: Section 11.6 describes modifications to the flexible isosurface interface to accommodate changing levels of simplification. Section 11.7 then discusses *topological noise removal*: using the simplified contour tree to remove noise in the underlying data.

Finally, Section 11.8 summarizes the contributions in this chapter and this Part.

11.1 Noise in the Contour Tree

The contour tree is impractical as a visual representation of the data if it has a large number of edges. Small details, in particular noise, cause the contour tree size to increase. In Table 11.1, we show some results for the size of the contour tree for a variety of data sets: full details on the data sets can be found in Chapter 16.

File	Type	Mesh Size	Samples(N)	Tree Size (t)	Tree/Data Ratio
marlobb (MC)	Analytical	$41 \times 41 \times 41$	68,921	912	1.32%
fuel (MC)	Simulation	$64 \times 64 \times 64$	262,144	227	0.09%
fuel (5S)	Simulation	$64 \times 64 \times 64$	262,144	299	0.11%
hipiph (MC)	Simulation	$64 \times 64 \times 64$	262,144	1,360	0.52%
neghip (MC)	Simulation	$64 \times 64 \times 64$	262,144	2,063	0.79%
neghip (5S)	Simulation	$64 \times 64 \times 64$	262,144	2,544	0.97%
f368.0.6 (MC)	Molecule	$35 \times 17 \times 51$	30,345	915	3.02%
3gap.0.8 (5S)	Molecule	$29 \times 60 \times 131$	227,940	13,164	5.78%
3gap.0.8 (MC)	Molecule	$29 \times 60 \times 131$	227,940	14,290	6.27%
1dog.0.8 (5S)	Molecule	$72 \times 64 \times 60$	276,480	17,656	6.39%
1dog.0.8 (MC)	Molecule	$72 \times 64 \times 60$	276,480	18,948	6.69%
lobster (MC)	CT scan	$34 \times 120 \times 120$	489,600	17,867	3.65%
teddybear (MC)	CT scan	$61 \times 128 \times 128$	999,424	245,588	24.57%
3dhead (MC)	MRI scan	$109 \times 256 \times 256$	7,143,424	2,231,900	30.75%
3dhead (5S)	MRI scan	$109 \times 256 \times 256$	7,143,424	2,196,594	31.24%
3dknee (MC)	MRI scan	$127 \times 256 \times 256$	8,323,072	2,751,506	32.51%
3dknee (5S)	MRI scan	$127 \times 256 \times 256$	8,323,072	2,706,019	33.06%

Table 11.1: Effects of Size and Data Type on the Contour Tree.
File: Name of the file. (5S) = contour tree computed using minimal (5-fold) simplicial subdivision of cubes. (MC) = contour tree computed using Marching Cubes (Chapter 14).
Type: Type of data.
Mesh Size: Dimensions of the input cubic mesh.
Samples: Total number of samples in the input mesh
Tree Size: Number of vertices in the contour tree, before simplification or removal of effects of perturbation.
Ratio: Ratio of tree size to number of samples.

In the table (MC) indicates that the contour tree was computed using Marching Cubes, as described in Chapter 14, while (5S) indicates that the contour tree was computed using a tetrahedral mesh obtained by subdividing each cubical cell into 5 simplices (tetrahedra). As is apparent from the table, however, this makes little difference to the size of the contour tree.

From this table, we can draw several conclusions. First, the size of the contour tree is strongly dependent on the type of data. The first group in the table is of data sets that are either defined analytically or by numerical simulation. In either case, there is unlikely to be much noise in the data, although some is possible due to numerical instabilities or round-off errors. The second group in the table is of electron distribution in protein molecules. Although this data is computed rather than acquired, it is a simulation of acquired data, and includes some simulated noise. The third group consists entirely of experimentally acquired data, in which noise is known to be a problem.

The second conclusion that we can draw is that the size of the contour tree seems to be related to the amount of noise in the data. We will see in Figure 11.2 why this might be. Note, however, that there can also be a lot of genuine small-scale detail, so we cannot conclude that noise is the cause without further study.

The third conclusion is that, even for clean simulated data, the contour tree may have several thousand edges. For noisy scanned data, it is impractical to use the contour tree, either as a representation of the data, or for manipulating surfaces in the flexible isosurface interface. In order to take advantage of the contour tree for either of these purposes, we need to simplify the data, the contour tree, or both.

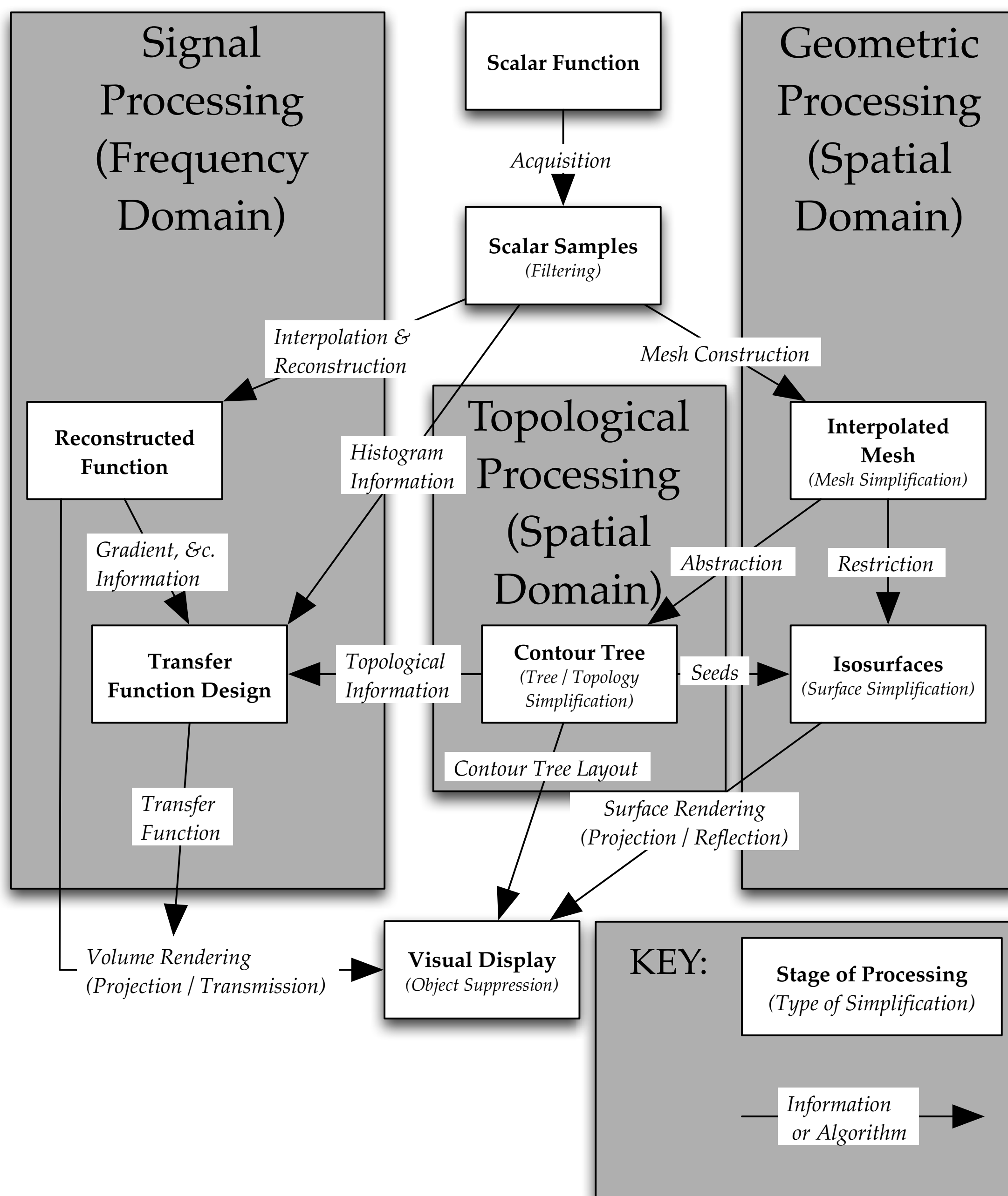


Figure 11.1: Flowchart for Scalar Field Visualization. Note how signal processing and geometric processing are essentially independent approaches to visualizing scalar fields. The contour tree, which encapsulates topological information, is closely related to geometric processing, but can also be used to aid in signal processing.

11.2 Simplification and Visualization

Before simplifying the contour tree, we must acknowledge that our computation is merely one stage of a larger process, shown in Figure 11.1. We can in principle apply simplification at almost any stage of the visualization process. However, most types of simplification are not successful at simplifying the contour tree. To see this, we briefly review the various types of simplification shown in Figure 11.1, as well as a

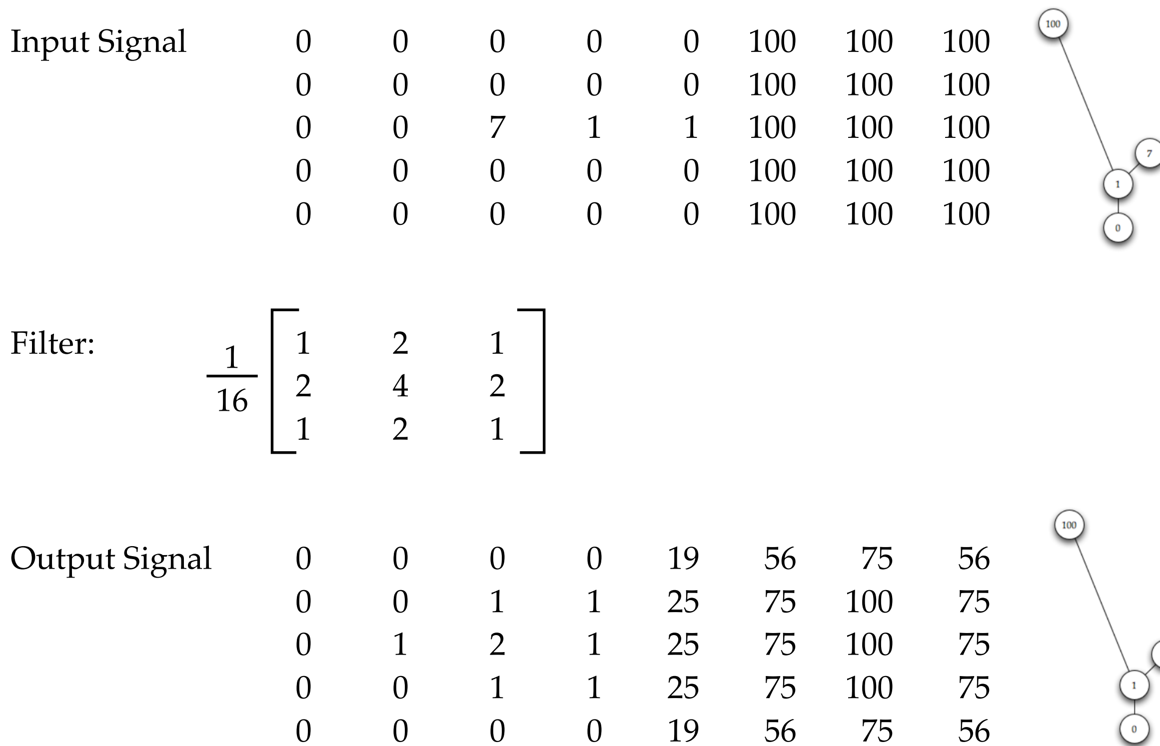


Figure 11.2: Filtering a Topologically Small Detail. In this small two-dimensional example, we see that although filtering smoothes out the peak at 7, it does not eliminate it from the contour tree.

topological approach known as *persistence*.

Filtering is performed on the original samples from the underlying function. Filtering is a generalization of the reconstruction process described in Section 4.2. It operates by reconstructing the function with a specialized kernel function, then re-sampling. It has the advantage that it is well understood and widely used for simplifying data and removing noise. An overview of filtering can be found in a text such as that by Gonzalez & Woods [GW02].

Unfortunately, as we saw in Section 4.2, kernel functions tend to be too complex for us to perform topological or geometric analysis on them. Thus, if we use filtering to simplify the data, we must do so before computing the contour tree, and hope that the kernel chosen has the desired effect of simplifying the contour tree.

Standard smoothing operations do not necessarily reduce the size of the contour tree, although they do tend to flatten out detail. A small example is shown in Figure 11.2. Before the filtering operation, the small peak at 7 causes the contour tree to have 3 edges instead of 1: without it, there is a smooth hill on the right hand side. After the filtering operation, the peak still exists, although its height has been reduced to 2, and a second smoothing operation get rids of it entirely.

Although smoothing may remove topological details, we have no guarantee that it does so, nor that it does so without eliminating features that we wish to preserve. For example, consider Table 11.2, in which we show the results of applying a simple smoothing filter twice to the UNC head data set. As we can see, it takes two smoothing operations to reduce the contour tree by an order of magnitude. In doing so, we

Smoothing Iteration	Contour Tree Size
0	2,196,594
1	483,715
2	232,230

Table 11.2: Effects of Smoothing on the Contour Tree of the 3dhead Data Set. After two smoothing operations, the contour tree is reduced by approximately an order of magnitude, but is still much too large for visual display.

lose small-scale objects such as noise, but we also lose small-scale details of large features. And the more we smooth the data, the more detail we lose from large-scale features. In comparison, simplifying the contour tree reduces the effects of noise, but without losing small-scale details of large-scale features.

Mesh Simplification reduces the input parameters N and n by replacing several small cells in the mesh by a single larger cell, usually guided by some geometric measure of the error caused by the replacement. Unless we base the simplification on the topology, we have no guarantee that the complexity of the contour tree will actually decrease.

As a rough indication, consider the data sets in the last section of Table 11.1. For experimentally acquired data sets, it appears that the contour tree is often roughly 25% of the size of the input data. If this ratio holds as the data is simplified, we would have to reduce the mesh to roughly 1,000 cells in order to reduce the contour to tree to 250 or so edges. But this would only allow us to work with $10 \times 10 \times 10$ data sets: too great a reduction for practical purposes.

Although this argument should not be relied on too heavily, we would argue that mesh simplification, like filtering, simply does not reduce the topological complexity of the data fast enough to be used to simplify the contour tree.

Mesh simplification has been used in conjunction with the contour tree, but for the reverse of the task at hand. Chiang & Lu [CL03] performed mesh simplification while preserving the contour tree. They did not, however, consider the question of simplifying the contour tree while preserving the mesh.

Tree Simplification uses graph reductions to reduce the size of the contour tree. Takahashi, Fujishiro & Takeshima [TFT01] simplify the contour tree by pruning leaves to determine the "most important" isovalues for a volume rendering transfer function. In choosing which leaves to prune, these authors concentrated on the "height" of an edge: the difference between the isovalues at the top and bottom. We generalize this approach to use, not just height, but also the local spatial measures defined in Chapter 10.

Although Takahashi, Fujishiro & Takeshima simplified the contour tree, their computations are essentially static and global. They use the contour tree as a statistical summary of the behaviour of the data. As such, this is similar in nature to the Contour Spectrum of Bajaj, Pascucci & Schikore [BPS97] or the automated detection of significant isovalues by Pekar, Wiemker & Hempel [PWH01], in which the isovalue is treated uniformly across the region of interest.

Moreover, these authors do not track which edges in the contour tree correspond to which edges in the simplified contour tree. Nor do they concern themselves with isosurface seeds. As this chapter develops, however, one of our principal concerns is to preserve seed information so that individual contour surfaces can be extracted. In fact, this is a recurrence of a common theme in this thesis: the focus on local, rather

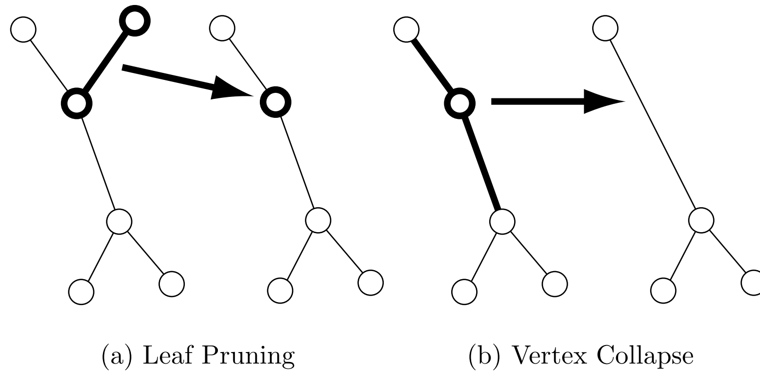


Figure 11.3: Graph Operations on the Contour Tree

than global, characteristics of the data.

Surface Simplification applies techniques similar to mesh simplification to an extracted isosurface. This reduces the output size k of the isosurfaces rendered, but not the input size n for the contour tree.

Object Suppression completely removes identifiable objects from the visual display of the data. Again, this occurs after the contour tree is computed, and does not itself reduce the contour tree's complexity. It is worth noting, however, that the flexible isosurface performs object suppression by allowing deletion of individual contours. Moreover, we will see that contour tree simplification implicitly suppresses objects.

Persistence is a general topological measure used by Edelsbrunner, Harer & Zomorodian [EHZ01]. They perform a sweep through the data, building topological features in the process. The persistence of a feature is then defined to be the number of steps of this sweep for which a feature retains its topological uniqueness. Bremer et al. [BEHP03] have used this to simplify the Morse-Smale complex for two-dimensional triangulations. These results have not yet been extended to higher dimensions, or to non-simplicial meshes, and have not yet been applied to the contour tree.

In general, persistence is based on the order in which topological features are processed, which need not be in strict order of height. For features processed in order of height, the persistence measure can be seen as a generalization of the height measure used by Takahashi, Fujishiro & Takeshima [TFT01].

In this chapter, we generalize the idea of height in a different direction. Instead of a single predetermined sweep, we use local spatial measures to determine a non-isovalued sequence of sweeps through the data similar to those used to merge the join and split trees.

11.3 Applying Graph Simplifications to the Contour Tree

To simplify the contour tree, we define two operations: *vertex collapse*, shown in Figure 11.3(b), and *leaf pruning*, shown in Figure 11.3(a). We have chosen these two operations because each has a clearly defined

effect on the contour tree and on the underlying data, because they are readily computed, because they are sufficient to simplify the contour tree to any desired number of edges, and because we have already been using them, either explicitly or implicitly.

In Section 11.3.1, we define these operations formally, and prove some useful properties. Then, in Section 11.3.2, we shall state rules for deciding which of the two operations to use at any step in the simplification process, justify our choice of operations and rules, and prove that we can always choose a simplification operation that satisfies the rules, thus enabling us to reduce the contour tree to any arbitrary size.

11.3.1 Vertex Collapses and Leaf Pruning

We start by defining the vertex collapse operation. We have already seen these in action in Section 7.6.3, where we used them to reduce the fully augmented contour tree to the unaugmented contour tree.

Definition 11.1 *A* vertex collapse *may only be performed at a vertex μ of the contour tree that is a regular point, and replaces the incident edges (λ, μ) and (μ, ρ) with an edge (λ, ρ).*

Vertex collapses have two very important properties: they have no effect on the contours of the function f, and they replace a monotone path (λ, μ, ρ) in the contour tree with a single edge (μ, ρ).

Lemma 11.1 *A vertex collapse at a vertex μ preserves every contour in the data while reducing the tree size by* 1*. Moreover, the edge (λ, ρ) corresponds to the path (λ, μ, ρ) in the unsimplified contour tree.*

Proof: We know from Definition 11.1 that μ must be a regular point, as shown in Figure 11.3(b). Thus, before the collapse, $a = (\lambda, \mu)$ represents a set of contours over the interval $(f(\lambda), f(\mu))$, μ represents a contour at $f(\mu)$, and $b = (\mu, \rho)$ represents a set of contours over the interval (ρ, μ).

Let P be a f-monotone path departing λ along edge a to the isovalue $f(\rho)$. By Corollary 6.1, this maps to a monotone path Q in the contour tree. But, since it follows edge a, there is only one possible path it can follow: (λ, μ, ρ). Since all monotone paths departing in this direction along a must follow this path, we could represent these paths instead by a single edge (λ, ρ).

Since we delete μ and replace two edges with one without disconnecting the contour tree, the result of a vertex collapse must be a tree with one edge and one vertex less than before the collapse. □

Our second operation, leaf pruning, is chosen because it is always possible to perform, especially when vertex collapses cannot be performed. Moreover, performing leaf pruning tends to make vertex collapses possible. And we have been using them implicitly since Chapter 7. Both the merge algorithm (Algorithm 7.1) and the algorithm to compute local spatial measures (Algorithm 10.2) operate on leaves of the tree, deleting them as they go - i.e. by leaf pruning.

Definition 11.2 *A* prunable leaf *of a contour tree T is a leaf vertex λ with incident edge $e = (\lambda, \mu)$ or $e = (\lambda, \mu)$ such that e is not the only up-edge or down-edge at μ.*

Although we could allow any leaf to be pruned, we chose this definition to preserve leaves for later vertex collapses where possible. We view this as desirable because leaf prunes alter f, while vertex collapses do not. We will see an example of the effects of this choice in Section 11.5.

Definition 11.3 *A* leaf prune *removes a prunable leaf, deleting a leaf vertex λ and an edge $e = (\lambda, \mu)$ or $e = (\mu, \lambda)$.*

A leaf prune reduces the degree of μ by 1: pruning a sufficient number of leaves will reduce critical points to regular points. When this happens, we can perform a vertex collapse, as for example in Figure 11.3.

Lemma 11.2 *A leaf prune reduces the size of the contour tree by* 1*.*

Proof: Follows trivially from Definition 11.3's deletion of an edge of the tree. Since one end is a leaf, this does not disconnect the tree, and it remains a tree. □

The effect of a leaf prune on the function f is easily described. It flattens out the region of the tree represented by the pruned edge.

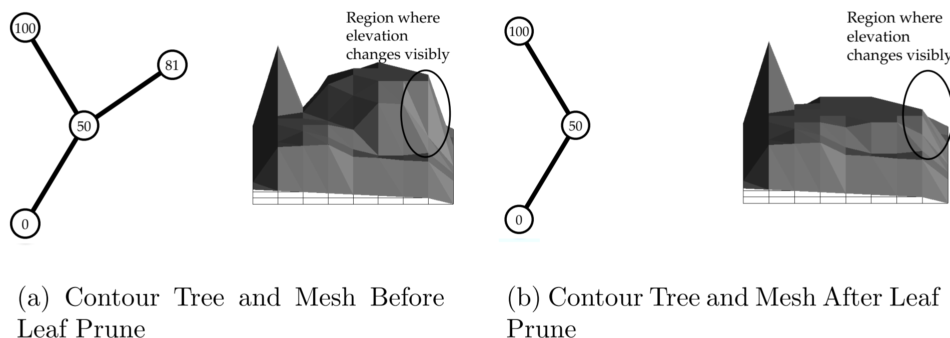
Lemma 11.3 *Let λ be a leaf of the contour tree C with incident edge $a = (\lambda, \mu)$. Without loss of generality, assume that $f(\mu) < f(\lambda)$ and that γ is the contour passing through μ at isovalue $h = f(\mu)$. Then the pruned contour tree $C - a$ is the correct contour tree for each of the following functions:*

1. *$f_1(x) = f(x)$ restricted to the region $domf - R_a^+(h)$.*
2. *$f_2(x) = f(x)$, defined over $domf$ with $R_a^+(h)$ contracted to μ.*
3. *$f_3(x)$ defined by:*
$$f_3(x) = \begin{cases} f(\mu) & \text{if} x \in R_a^+(h) \\ f(x) & \text{otherwise} \end{cases}$$
4. *$f_4(x)$ defined by barycentric interpolation over the simplicial mesh with vertex values*
$$f_4(v) = \begin{cases} f(\mu) & \text{if} v \in R_a^+(h) \\ f(v) & \text{otherwise} \end{cases}$$
for each vertex v in the mesh.

where $R_a^+(h)$ is as defined in Definition 10.2.

Proof: For $f_1(x)$, we note that $R_a^+(h)$ is the same as $f(a) + \lambda$: the union of all contours represented by either a or λ. Removing the edge a from the contour tree is equivalent to removing all of these contours from $domf$, and the result follows.

For $f_2(x)$ we use essentially the same argument, except that the contours are removed by shrinking them to μ.



(a) Contour Tree and Mesh Before Leaf Prune

(b) Contour Tree and Mesh After Leaf Prune

Figure 11.4: Flat Spot and Modified Slopes Induced by Leaf Pruning Contour Tree.
(a) shows our standard small example after several simplifications.
(b) shows the result of pruning the leaf at 90, using definition (4) from Lemma 11.3. Under any of the definitions in Lemma 11.3, we modify the upstart region of $90 - 50$ at 50.
Definition (1) would cut a hole in the domain instead of flattening out the surface.
Definition (2) would contract the upstart region to a point, distorting the entire surface.
Definition (3) would flatten the upstart region, requiring existing triangles to be subdivided.
Definition (4) allows us to change only the values stored for vertices of the mesh.

For $f_3(x)$, we note that this definition replaces $R_a^+(h)$ by a flat spot. As such, f_3 is not a Morse function. Since we have not relied on Morse theory in defining the contour tree, we use Definition 6.2 and observe that $\gamma' = \gamma \cup R_a^+(h)$ is the contour of f_3 containing μ, while every contour of $f_3(x)$ except γ' is identical to a contour in f. The result follows.

For $f_4(x)$, we contract the boundary of γ inwards until it reaches cells entirely contained in $\gamma + R_a^+(h)$. The result then follows from the proof for $f_3(x)$. $\square$

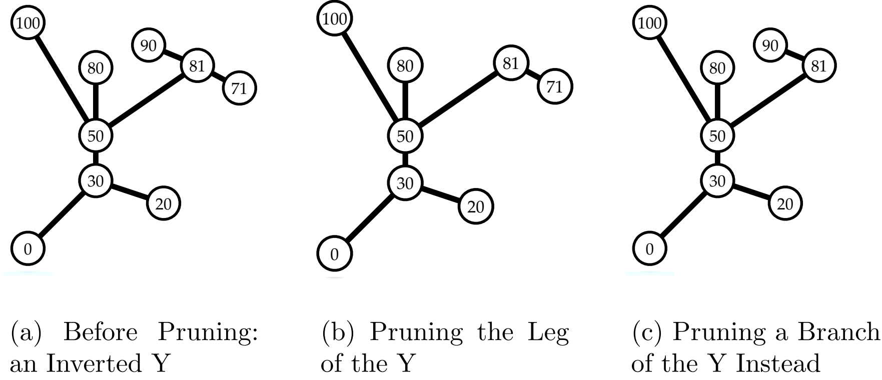
Figure 11.4 shows the effects of a leaf-prune part way through simplification of our standard example. By pruning vertex 80, we remove the corresponding contours, truncating the peak to the isovalue of vertex 50. In this figure, we have used f_4 from Lemma 11.3, as it is the easiest to compute and display. Note, however, that this choice causes small changes to the surface in the immediate neighbourhood of the contour γ: for example, the slope of the surface in the region marked in Figure 11.4 changes from the original. This variation is, however, limited to cells with at least one vertex inside $R_a^+(h)$. Topologically, however, this has no effect on the nesting relationships of the contour.

11.3.2 Collapse Rules

We have just defined two operations for simplifying the contour tree. In Figure 11.3(b), we actually have a choice whether to apply a vertex collapse or a leaf prune, and have several choices of leaf which can be pruned. In deciding which operation to apply, we will be guided by the following rules:

In collapsing the contour tree, we wish to preserve the topological and geometric utility of the structure. In particular, we wish to preserve the ability to generate isosurface seeds and the ability to track individual contours. To do so, we apply the following rules:

I Always perform a vertex collapse if one is possible. By Lemma 11.1, a vertex collapse leaves the



(a) Before Pruning: an Inverted Y

(b) Pruning the Leg of the Y

(c) Pruning a Branch of the Y Instead

Figure 11.5: Why Not All Leaves Are Prunable.
At vertex 81, there is only one up-edge, to 90. If we prune this edge, we will be unable to collapse vertex 81: instead, we would have to prune the edge $81 - 71$ as well. But, since we define $90 - 81$ to be unprunable, we will instead choose edge $71 - 81$ to prune, then collapse vertex 81, preserving more of the original function.

underlying function unchanged.

II If rule [I] does not apply, always prune the least important leaf λ and edge $a = (\lambda, \mu)$, i.e. the leaf for which $p_a^{\uparrow}(\mu^-)(f(\mu))$ is smallest for some local spatial measure p ($p_a^{\downarrow}(\mu^+)(f(\mu))$ if λ is a lower leaf). Note that, although importance is measured using a local spatial measure, we prune the globally least important leaf.

Before describing an algorithm to implement these rules, we first show that these rules always allow us to reduce a contour tree to no edges. We know from Lemma 11.1 and Lemma 11.2 that both types of operation reduce the tree size by 1. Therefore, reducing the tree to nothing takes exactly t steps, and can only fail if, at some stage, we cannot find either a regular vertex to collapse or a prunable leaf to prune.

Theorem 11.4 *Every contour tree T with t edges has either a prunable leaf or a regular point.*

Proof: Assume that T has no prunable leaves or regular points. This can only happen if every interior vertex has either two or more down-edges or two or more up-edges. Choose any interior vertex v. Without loss of generality, assume that v has two up-edges. Choose either, and call the vertex at the other end of the edge w. If w is a leaf vertex, then by Definition 11.2 it is a prunable leaf, since v has more than one up-edge. But we know T has no prunable leaves, so w cannot be a leaf.

Therefore w must be an interior vertex, and we know that it has either two or more down-edges or two or more up-edges. If w has two or more up-edges, then neither of these up-edges is the same as (w, v). Choose either of these edges, and by the same argument as before, it leads to another interior vertex. Similarly, if w has two or more down-edges, then there is at least one down-edge other than (w, v). And, again, it must lead to another interior vertex.

By this argument, we can always continue choosing edges that lead to other interior vertices without backtracking. Since T is finite, it follows that T must contain a cycle composed entirely of interior vertices. But T is a contour tree, and has no cycles.

By contradiction, our assumption is false, and there must be a prunable leaf or regular point in T.

□

11.4 Simplification Algorithms

Once we have the operations and rules stated in the previous section, an algorithm implementing simplification is straightforward. We will show how to do so, then, in Section 11.4.1, we will show how to extend the path seeds of Chapter 8 to *hierarchical path seeds* to generate contour seeds from a simplified contour tree.

Recall from Chapter 10 that we defined the sweep function $p_a^{\uparrow}$ to describe a property p computed for the downstart region $R_a^{-}(h)$ for a given arc $a = (u, v)$ of the fully augmented contour tree as a function of the isovalue h. Moreover, recall that $p_a^{\uparrow}$ was analytically uniform over a - i.e. $p_a^{\uparrow}$ had no break points at any isovalue in the range $(f(v), f(u))$.

Once we start working with superarcs of the unaugmented contour tree and with collapsed superarcs, the sweep functions will be piecewise-defined over the entire superarc. Fortunately, to simplify the contour tree, we only need to know the sweep functions at the top and bottom ends of a superarc, as these describe the entire region removed by a leaf prune. We therefore define the importance of an edge in a contour tree as follows:

Definition 11.4 *For a given geometric property p, the* importance *of an arc $a = (u, v)$ in a fully augmented contour tree is $p_e^{\uparrow}(f(u))$ if v is a prunable leaf, $p_e^{\downarrow}(f(v))$ if u is a prunable leaf, and infinite otherwise.*

Definition 11.5 *For a given geometric property p, let $s = (u, v)$ be a superarc that corresponds to the monotone path $a_1, \ldots, a_m$ in the fully augmented contour tree. The the* importance *of s is equal to the importance of a_1 if u is a prunable leaf, of a_m if v is a prunable leaf, and infinite otherwise.*

This definition of importance measures the geometric cost of pruning a leaf by computing the value of the geometric property p for the upstart or downstart region that will be flattened by pruning the leaf. During simplification, then, we always choose the leaf prune that flattens the region with the smallest value of the geometric property p. Since we do not know *a priori* in which direction a given edge $e = (u, v)$ will be pruned, we define importance values *up*(e) and *down*(e) for each edge in the contour tree. These represent the cost of pruning upwards from v and downwards from u, respectively. When a vertex becomes a prunable leaf, we determine which one of these two values is applicable, and place the leaf on a priority queue, with the priority set to the importance of the leaf edge.

As vertex collapses progressively build larger and larger paths through the contour tree, we maintain the invariant that the weights for the new edges represent the geometric property p for the upstart region by transferring the *up* and *down* values from the ends of the path to the newly created edge.

We start by implementing a vertex collapse operation in Algorithm 11.1, to reduce the complexity of the code.

As we can see, Algorithm 11.1 is principally book-keeping: the old edges are deleted, a new edge is substituted, flags are set for the old edges, and the weights are set for the new edge.

```
Function vertexCollapse(v)
Input   : A contour tree T
          Weights up(e) and down(e) for each edge e ∈ T
          A regular point v ∈ T
Output  : The contour tree with vertex v collapsed
          A new edge d replacing the two edges b, c incident to v

1 Let b, c be the two edges incident to v with endpoints u, v and v, w respectively
2 Without loss of generality, f(u) > f(v) > f(w)
3 Delete b, c from C
4 Insert new edge d from u to w
5 Store b, c on d as upperArc(d) and lowerArc(d)
6 Set alreadyCollapsed(b), alreadyCollapsed(c)
7 Set up(d) = up(b), down(d) = down(c)
8 Return d
```

Algorithm 11.1: Collapsing a Vertex in A Contour Tree. The vertex collapse creates a new edge that represents the path composed of the two old edges. Weights for the new edge are taken from the weights at then ends of the path.

For the main algorithm, shown in Algorithm 11.2, we perform all legal vertex collapses, then perform leaf prunes in ascending order of importance using a priority queue. If, after any leaf prune, the interior vertex becomes a regular point, we immediately perform a vertex collapse at that vertex.

In order to make the simplification reversible, we keep track of the simplification in the *collapseRecord* array. Leaf prunes remove an edge from the tree, while vertex collapses substitute one new edge for two old edges. On a leaf prune we store the edge that was collapsed, but on a vertex collapse we store the new edge that was substituted for two old edges. Also, we want to end up with an array which lists the collapses in reverse order, so we start at the tail end of *collapseRecord*, and work our way to the front.

Theorem 11.5 *Algorithm 11.2 correctly applies Rules I and II to simplify the contour tree in* $O(t \log t)$ *time.*

Proof: For correctness, note that Step 2 applies Rule I to all regular points initially in the contour tree. Since a vertex collapse at v decreases $degree(v)$ by two, and leaves the degree of all other vertices intact, performing a vertex collapse never creates a new regular point, so, after Step 2, there are no regular points in the tree.

New regular points can only be created as the result of a leaf-prune. Rule I is then enforced by Step 22. A leaf prune of edge u, v at Step 18 decreases the degrees of u and v by one each, leaving all other vertex degrees intact. Therefore, only v can become a regular vertex as a result of the leaf prune. As we have just noted, performing a vertex collapse at v never creates a regular point, so we have to execute Step 22 at most once for each leaf prune.

For Rule II, we claim that the priority queue always contains all prunable leaves, and may also contain unprunable leaves or previously collapsed edges. Step 13 and Step 16 ensure that unprunable leaves and previously collapsed edges are not pruned: otherwise, Rule II is enforced by the priority queue.

To see that this invariant holds, we start by observing that all leaves, prunable or otherwise, are initially placed on the queue. Edges may become leaf edges as the result of a vertex collapse, in which case

```
Input  : A contour tree T, with weights up(e) and down(e) for each edge e ∈ T
         A desired size Tree_Size or a desired importance bound Collapse_Bound
Output : A simplified contour tree with the desired size or importance bound

1  collapseRecord = ∅
2  for each vertex v in T do
3      if δ+(v) and δ−(v) are both 1 then
4          Let d = vertexCollapse(v)
5          Add d to front end of collapseRecord
       end
   end
6  for each edge e in T do
7      if e is incident to a leaf then
8          Push e on priorityQueue with priority down(e) if an upper leaf, up(e) if a lower leaf
       end
   end
9  while priorityQueue is not empty do
10     pop e = (u, v), priority(e), the least important edge, from priorityQueue. Wlog, u is a
       leaf vertex, v is not.
11     if (priority(e) > Collapse_Bound) or (size(T) < Tree_Size) then
12         Return
       end
13     if alreadyCollapsed(e) then
14         Goto Step 10
       end
15     Let u be the leaf vertex of e and v be the interior vertex of e
16     if e is the last up- or down- edge at v then
17         Goto Step 10
       end
18     Remove e from T
19     Add e to front end of collapseRecord
20     if u or v is now a leaf then
21         Push e on priorityQueue with priority down(e) if an upper leaf, up(e) if a lower leaf
       end
22     if v is now a regular vertex then
23         Let d = vertexCollapse(v)
24         Add d = (u, w) to front end of collapseRecord
25         Without loss of generality, assume f(u) > f(w)
26         if u or v is a leaf vertex then
27             Add d to priorityQueue with priority down(d) if u is leaf, up(d) if w is a leaf
           end
       end
   end
```

Algorithm 11.2: Simplifying a Contour Tree Using Local Spatial Measures. At all times, the priority queue contains all leaves, prunable or otherwise. Unprunable leaves or previously collapsed leaves are suppressed by Step 13 and Step 16.

they are placed on the queue by Step 26 or as the result of a leaf prune, in which case they are placed on the queue by Step 20.

As the tree is simplified, a prunable leaf may become unprunable if other leaves are pruned at the interior vertex. Thus, when we perform a leaf prune, we must check to see if any leaves have become

unprunable. Rather than do so immediately, we employ a lazy strategy, placing all leaves on the priority queue initially, and checking for prunability only when they are removed from the queue - i.e. in Step 16. Once a leaf becomes unprunable, it never becomes prunable again, and is never placed back on the queue.

A vertex collapse may place the new edge on the queue if it is a leaf edge: note that in this case, at least one of the edges involved in the vertex collapse must have been an unprunable leaf edge. Thus, all leaves are initially placed on the queue, but only pruned if they are prunable. New leaves only arise as the result of combining an unprunable leaf edge with another edge during a vertex collapse. A new edge is tested immediately upon creation, and placed on the priority queue if it is a leaf edge.

We also note that a prunable leaf can be involved in a vertex collapse before it is pruned. In this case, the new edge will be placed on the priority queue, and the old prunable leaf marked as already collapsed. When this leaf reaches the front of the queue, Step 13 checks to see if this is the case. Since this leaf edge is now subsumed in some other edge as the result of one or more vertex collapses, it may safely be discarded.

To see that this algorithm takes $O(t)$ time, we note that Algorithm 11.1 takes $O(1)$ time since it is essentially book-keeping. Step 2 and Step 6 are each executed at most $O(t)$ times. With suitable data structures for the contour tree, these can be done in $O(t)$ total.

We claim that Step 9 executes at most $O(t)$ times. Initially, at most $O(t)$ edges are placed on the queue. Once an edge is removed from the queue, it is never returned to the queue, either because it has been pruned, because it became unprunable, or because it has already been collapsed. We know that there are at most t leaf prunes or vertex collapses performed, and that each leaf prune or vertex collapse adds at most one edge to the queue, so there can never be more than $O(t)$ edges on the queue.

Moreover, we know that at most t leaf prunes or vertex collapses are performed in total, since each leaf prune or vertex collapse reduces the tree size by 1. And each of the operations inside the loop can be executed in $O(1)$ time with suitable data structures, except for priority queue removals and insertions, which can cost $O(\log t)$.

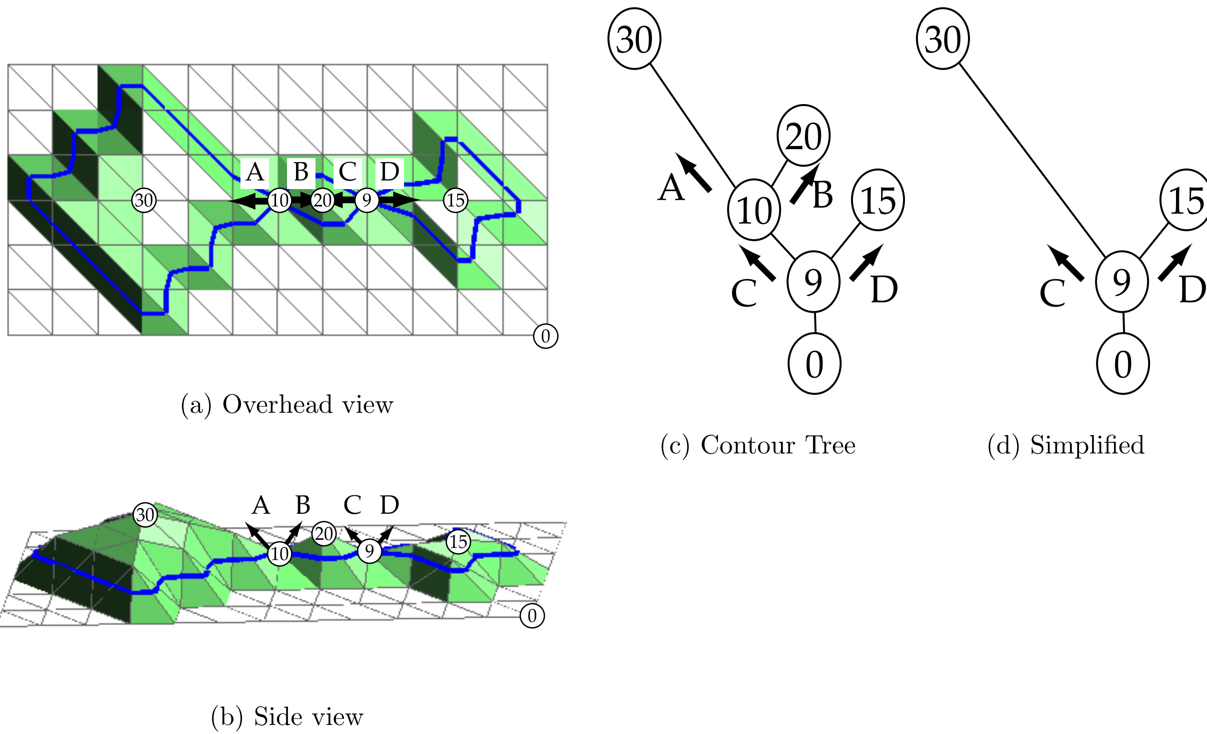
Since the loop executes $O(t)$ times at a cost of $O(\log t)$, the overall bound is therefore $O(t \log t)$. □

11.4.1 Hierarchical Path Seeds

Once we have simplified the contour tree, we will still wish to extract individual contour seeds for the continuation method. Unfortunately, the path seeds stored in the original contour tree are not automatically valid in the simplified tree. Consider the small data set shown in Figure 11.6, in which a small two-dimensional data set is shown in overhead and side views. This data set has three peaks at isovalues of 30, 20, and 15, respectively. The contour tree for this data set is shown in Figure 11.6(c), with the path seeds shown by reference to the arrows in (a) and (b): for example, edge A is the path seed for edge $30 - 10$, as it ascends from 10 towards 30 in the mesh.

Since the region corresponding to $10 - 20$ is the smallest, it is pruned, resulting in the tree shown in Figure 11.6(d). To generate a contour at isovalue 9.5 along the new superarc $9 - 30$, we follow path seed C. But, for isovalues above 10, this path seed does not generate correct contour seeds, as the path ascends up the side of peak 20 instead of peak 30.

This occurs because the path seeds (and paths) in the underlying contour tree (Figure 11.6(c)) are



(a) Overhead view

(b) Side view

(c) Contour Tree

(d) Simplified

Figure 11.6: Path Seeds in a Simplified Contour Tree. In this example, 20 - 10 is the leaf edge representing the smallest cross-sectional area, so we prune it, then simplify vertex 10 to get the simplified contour tree shown in (d). If we simply transfer the path seed at vertex 9 to the simplified edge, it will not produce the correct contours for isovalues above 10. Instead, we use the path seed at 10 for isovalues above 10. Note that no path seed is shown for edge $9 - 0$ because none is needed: any descending edge from 9 will suffice.

only guaranteed to be valid for the range of isovalues represented by the superarc. However, the new superarc $9 - 30$ represents the union of superarcs $9 - 10$ and $10 - 30$: in other words, the path $9 - 10 - 30$. Thus, to generate a correct path seed on a simplified superarc, we must search the path that it represents in the original contour tree.

Input : A contour c identified by an isovalue h and an edge s in a collapsed contour tree T

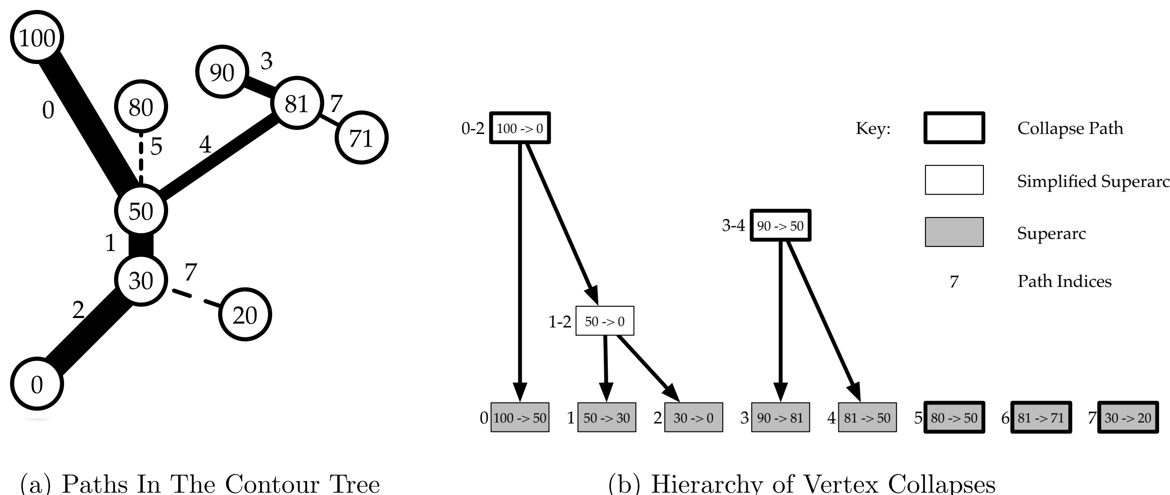
Output : A valid path seed to generate c

1 **while** $upperArc(s)$ is not $NULL$ **do**
2   Let u be the vertex at the lower end of $upperArc(s)$
3   **if** $f(u) > h$ **then**
4   Let $s = lowerArc(s)$
      else
5   Let $s = upperArc(s)$
      end
  end
6 Return PathSeed(s)

Algorithm 11.3: Algorithm for Hierarchical Extraction of Path Seeds.

A simple solution to this problem is to track which edges combine to form simplified edges. This

sequence implicitly defines a binary tree, which we can then search for the desired isovalue (and path seed). We call this approach *hierarchical path seeds*, and show it in Algorithm 11.3. Recall that, in Algorithm 11.2, when we performed a vertex collapse, we removed the two edges b and c from the tree, replaced them with a new edge d, and stored them as $upperArc(d)$ and $lowerArc(d)$.



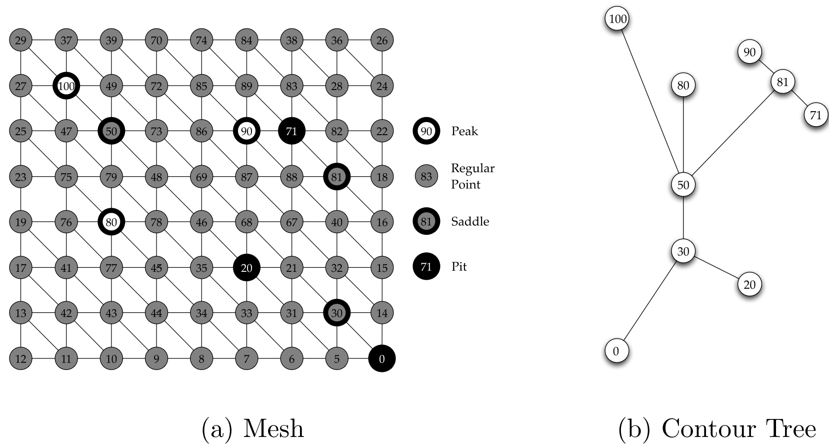
(a) Paths In The Contour Tree

(b) Hierarchy of Vertex Collapses

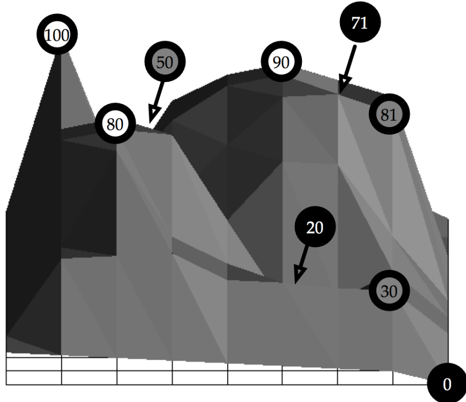
Figure 11.7: Paths Induced by Tree Pruning, And the Collapse Hierarchy. In this case, the contour tree was simplified using height as the importance measure, leaving a single arc, $100 - 0$, representing the heavy black path in the tree. Edges $90 - 81$ and $81 - 50$ collapsed to become $90 - 50$, which was then pruned. Every single edge in the contour tree belongs to a path, either generated by vertex collapses or a single edge in length. The edges can then be reindexed so that each path in the contour tree consists of consecutively-numbered edges.

Thus, we can construct a hierarchy showing how each vertex collapse contributed to building a path through the tree, as shown in Figure 11.7. To find a path seed, we start with a edge in the collapsed contour tree. This edge is either a collapsed edge or an uncollapsed edge. If the latter, it has a valid path seed attached to it. If the former, then we know that the edge was generated by a vertex collapse. We compare the isovalue of the desired contour with the isovalue of the vertex that was collapsed. If the desired contour has a higher isovalue, we use the upper of the two edges that were collapsed. If not, we use the lower of the two. We continue recursively until we reach an uncollapsed edge, and take the corresponding path seed.

We are not guaranteed that this hierarchy is balanced. However, if we collapse the contour tree completely, each collapsed edge belongs to a sub-path in the tree, as shown in Figure 11.7. To minimise the cost of path seed extraction, we can store the edges of the original contour tree in an array so that each collapse path is contiguous. The first and last indices of the subpath can then be stored on each simplified edge, as shown in Figure 11.7(b). For any given superarc, at any given level, we then know the subrange of the array in which to search, and we also know that the edges on the path are in sorted order. This allows us to use a binary search at a cost of $O(\log t)$, no matter how far we have simplified the tree. This array, of size t, can be generated in $O(t)$ time by starting at each leaf that was pruned and following the corresponding collapse path, copying it into the array and setting the subpath indices.



(a) Mesh (b) Contour Tree



(c) The Mesh as a Surface

Figure 11.8: A Triangulation in Two Dimensions, with Contour Tree

11.5 Examples of Simplification

To aid in understanding this simplification algorithm, we show in Section 11.5.1 why we chose the operations and rules described above by looking at the results of different operation and rule choices. In Section 11.5.2, we then show the results of applying Algorithm 11.2 with different spatial measures.

In both cases, our running example is the usual small 2-D triangulation shown in Figure 11.8. For clarity, the critical points in the mesh are labelled in Figure 11.8(c): since vertices 71, 50 and 20 are not visible from the perspective shown, the arrows point to the approximate location instead

11.5.1 Examples of Simplification with Different Rules

To see why we chose these particular rules and operations, we show the results of subtracting some of the rules and operations.

As our first example, we show in Figure 11.9 the results of simplification if any leaf is prunable and vertex collapses are disallowed. Note how, in Figure 11.9(e), instead of collapsing superarc $50 - 0$, we end up leaf-pruning superarcs $30 - 0$ and $50 - 30$, flattening out a large spatial region in the process. We end up with Figure 11.9(h), in which only one peak remains, surrounded by a uniform isovalue over nearly the entire domain of the function.

For our second example, we show in Figure 11.10 the results of simplification using leaf prunes and vertex collapses, but still permitting any leaf to be prunable. Thus, we permit Y-pruning (pruning the Y-leg, as described in Theorem 11.4). However, in comparison with Figure 11.9, we retain more surface detail for the same level of collapse.

For our third example, we show in Figure 11.11 the results of simplification, restricting prunes to prunable leaves, and allowing vertex collapses. In comparison with Figure 11.9, we retain more surface detail for the same level of collapse. Compared to Figure 11.10, we end up with exactly the same result. The difference occurs in subfigures (b) through (f) in each image: Figure 11.10 has flattened out the right-hand peak in the image, while Figure 11.11 has not, instead flattening out only the declivity in the side of the peak at 71.

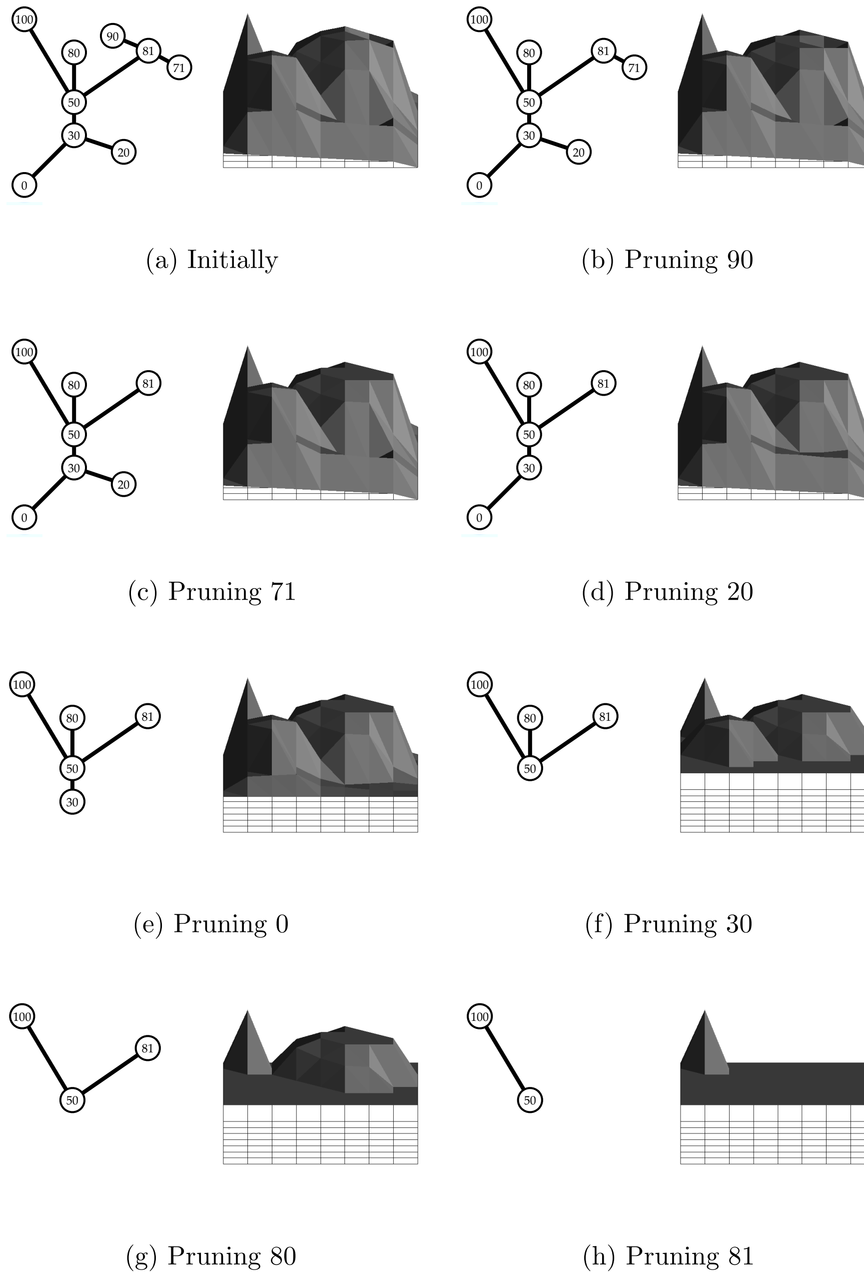
11.5.2 Examples of Simplification with Different Local Spatial Measures

We now look at the effects of applying different local spatial measures to guide the simplification algorithm. We saw in Figure 11.11 the results of simplifying with the full set of operations and rules, using the height of the superarc as the measure of importance.

In Figure 11.12, we show the results of using area to guide the simplification process. More precisely, we are using approximate area, as computed by the vertex count inside the regions. Note that the peak at 100, which ties for the smallest vertex count with the pit at 71, happened to be chosen first for pruning. Thereafter, however, the pruning order is the same as for Figure 11.11. At the end of this process, we are left with the largest peak, and a fair amount of surface detail elsewhere. In comparison with Figure 11.9, it can be argued that this approach does a worse job of simplification initially, but a better job later on.

The difficulty in Figure 11.12 occurred because we eliminated the tall, narrow peak at 100 before the short, slightly wider pit at 20, and even before the short, equally narrow pit at 71. In other words, we are now favouring short, fat features at the expense of tall, narrow features. If we wish to account for such factors, we can instead look at the volume of the region (i.e. the volume above or below the plane defining the area). Doing so results in a slightly different collapse order, in which the pits at 71 and 20 are pruned before the peak at 100: see Figure 11.13.

In this version, the three principal peaks remain in the image until a late stage, at which point, the tall, narrow peak at 100 is eliminated because it is not sufficiently tall to make up for its narrowness, where the other two peaks are both relatively tall and relatively wide. In this case, the final result is the same as Figure 11.12, but the collapse in the early stages is more satisfactory.



(a) Initially (b) Pruning 90

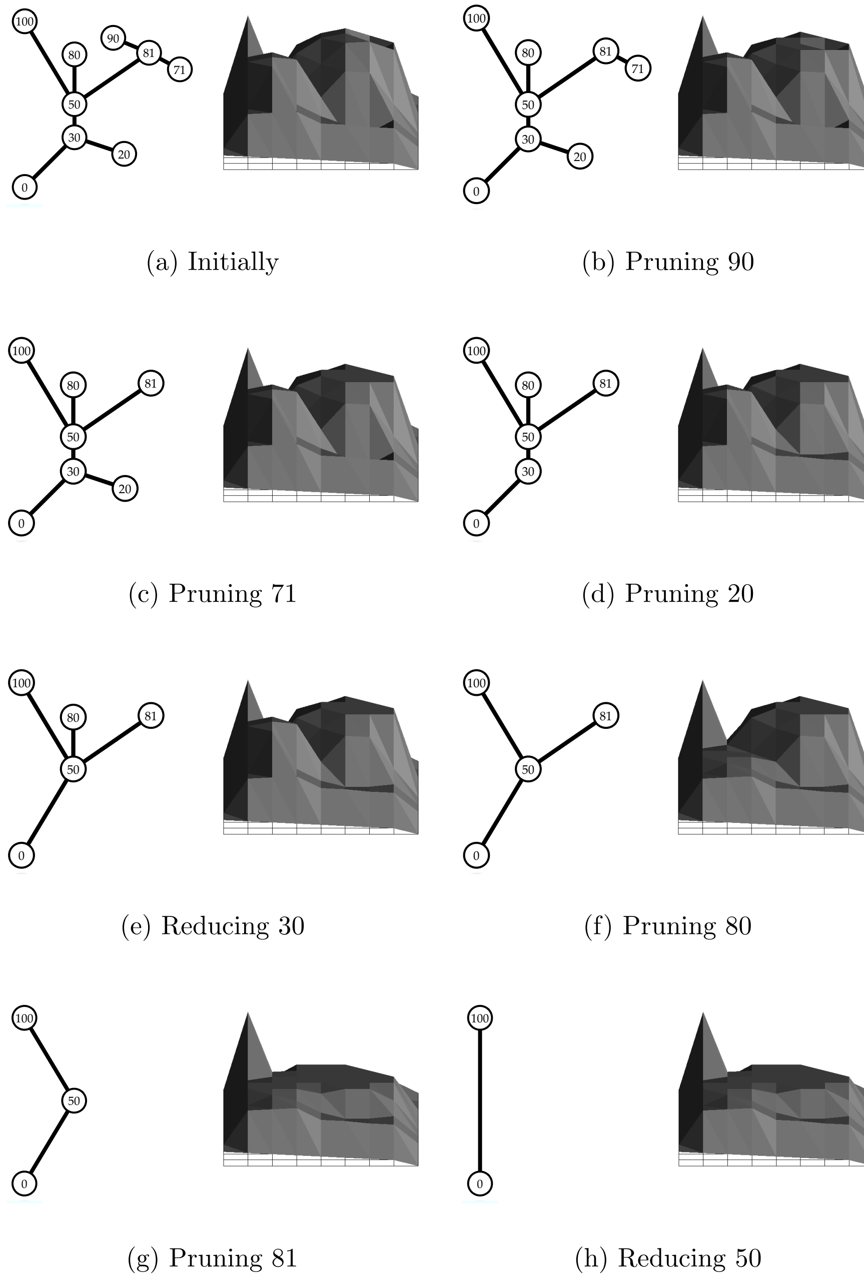
(c) Pruning 71 (d) Pruning 20

(e) Pruning 0 (f) Pruning 30

(g) Pruning 80 (h) Pruning 81

Figure 11.9: Simplifying by Height with all Leaves Prunable and no Vertex Collapses Permitted. Since every simplification step flattens part of the domain of the function, we end up with nearly all of the function set to a uniform value.

We do not claim that any one of these measures is always the correct one to use. Sometimes height will be most appropriate, sometimes area, and sometimes volume. In other cases, some other measure, such as average gradient across the contour, may be a better measure. But, for any of these measures, the algorithms for computing the measures and simplifying the contour tree will remain the same.



(a) Initially (b) Pruning 90

(c) Pruning 71 (d) Pruning 20

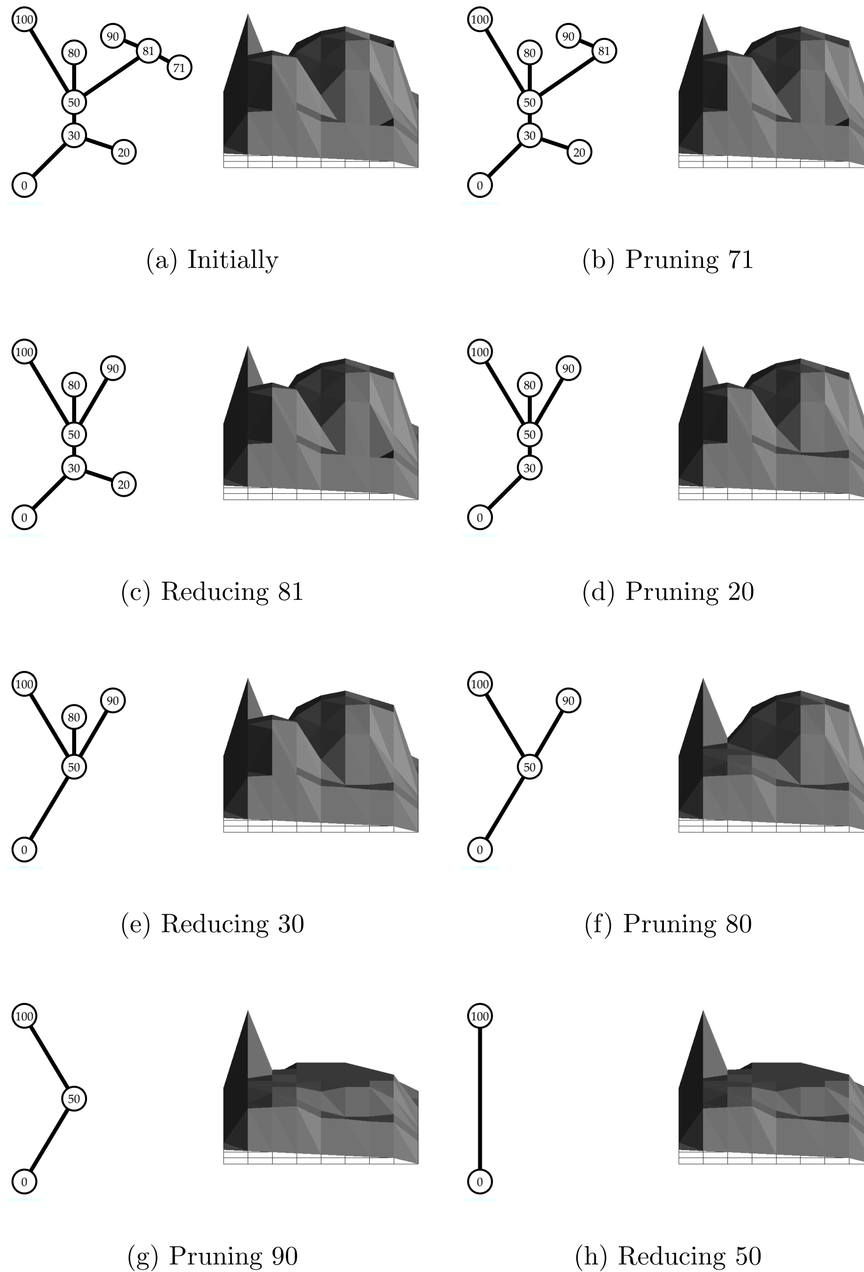
(e) Reducing 30 (f) Pruning 80

(g) Pruning 81 (h) Reducing 50

Figure 11.10: Simplifying by Height with Leaf Prunes and Vertex Collapses, allowing Y-Prunes. Compared with Figure 11.9, we now preserve much more of the original function by choosing vertex collapses whenever possible. However, the top of the right-hand peak (at 90) is still being truncated in (c): we can preserve it for use in a vertex collapse by disallowing Y-Prunes.

11.6 Simplified Flexible Isosurfaces

At this stage, we need to provide the user with control of the level of simplification applied. We do so by adding an extra set of controls to the flexible isosurface interface, which control the level of simplification applied, as shown in Figure 11.14. This figure also shows the advantage of this form of simplification, as the contour tree for this data set has over $200,000$ superarcs. The simplified contour tree shown in the



(a) Initially (b) Pruning 71

(c) Reducing 81 (d) Pruning 20

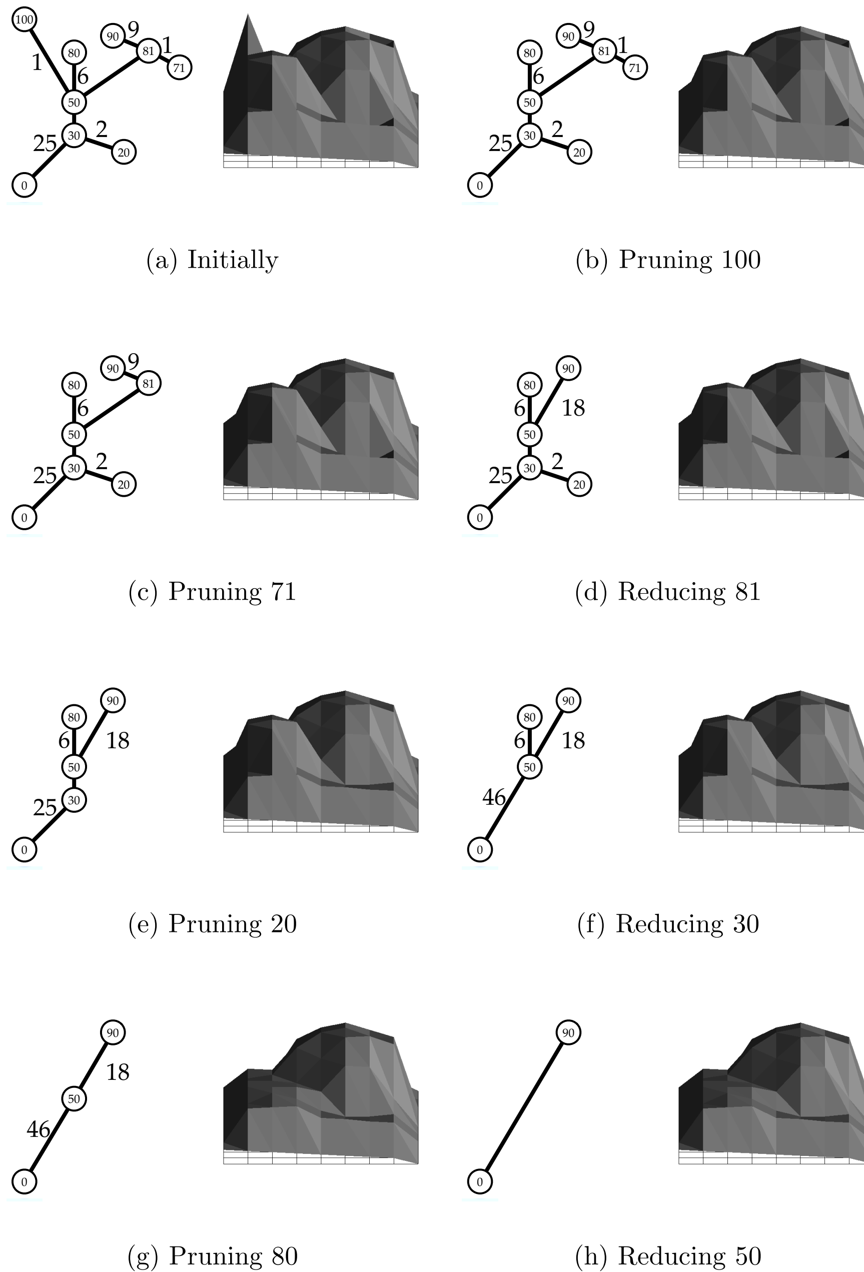
(e) Reducing 30 (f) Pruning 80

(g) Pruning 90 (h) Reducing 50

Figure 11.11: Simplifying by Height with Leaf Prunes and Vertex Collapses, without Y-Prunes. Compared to Figure 11.10, we now preserve more of the original surface between (b) and (f) in the collapse sequence. However, we still preserve the tallest peak at the expense of the two peaks that are larger by area or volume.

figure serves as an index to major features, as described in Chapter 9, without the complexity added by representing all of the small features and noise. Only a few minutes of exploration were needed to find the objects marked in this figure.

We have added several controls to the flexible isosurface: sliders to control slice distance and magnification, controls for setting the colour of individual contours and the corresponding superarcs, and buttons for invoking *dot* for layout of simplified contour trees, for saving the current flexible isosurface and graph

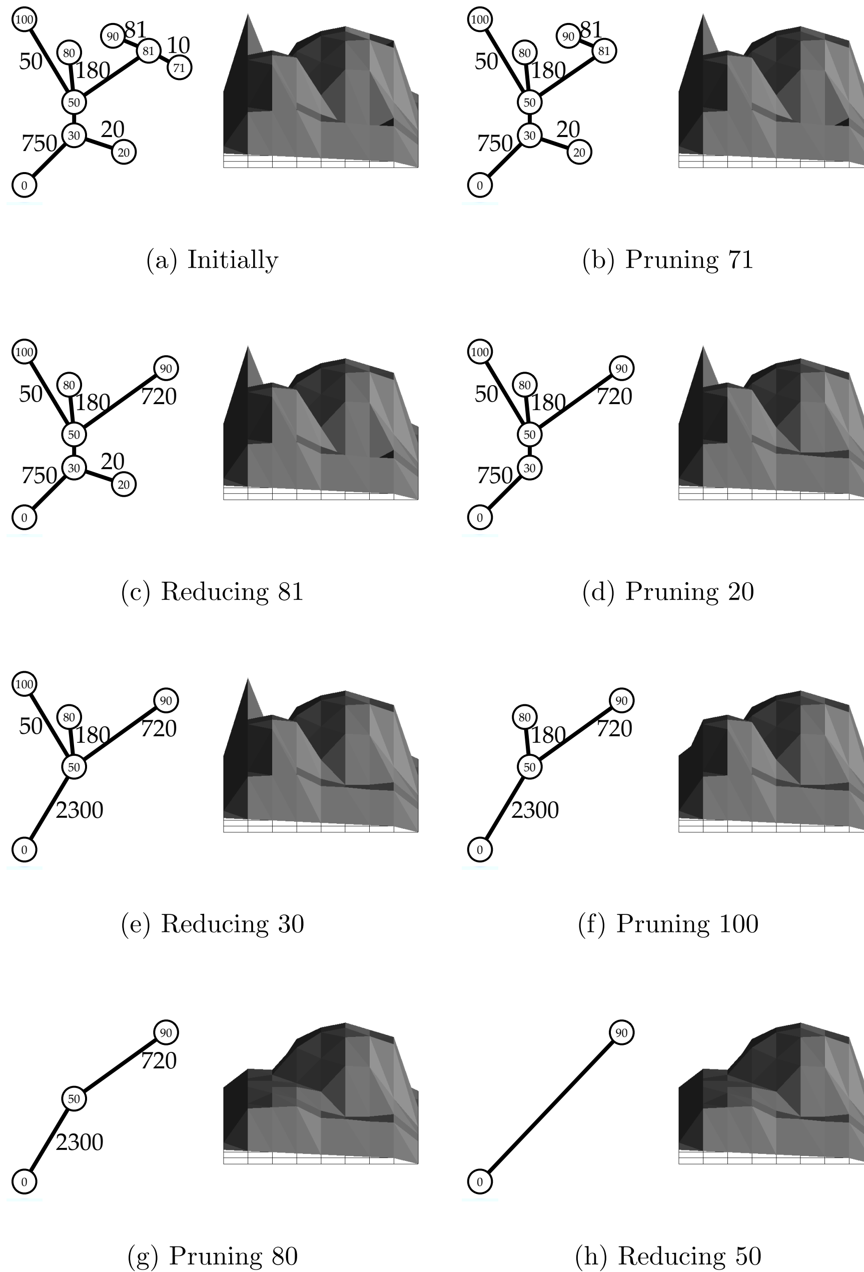


(a) Initially
(b) Pruning 100
(c) Pruning 71
(d) Reducing 81
(e) Pruning 20
(f) Reducing 30
(g) Pruning 80
(h) Reducing 50

Figure 11.12: Pruning the Contour Tree By Area. We now use cross-sectional area as our measure of importance. Compared to the previous figures, we now lose the tall peak early on in the process, preserving a small pit instead. However, the two larger peaks are preserved longer than in the previous figures.

layout, and for loading a flexible isosurface and graph layout. We have also added controls specific to simplification: collapse and uncollapse buttons, which adjust the size of the tree by 1, and a panel which plots the contour tree size against the size of the objects removed, in a log - log plot. Two sliders are provided, so that the user can adjust the simplification level either by the contour tree size, or by the object size removed.

Conveniently, we only need two additional operations:



(a) Initially (b) Pruning 71

(c) Reducing 81 (d) Pruning 20

(e) Reducing 30 (f) Pruning 100

(g) Pruning 80 (h) Reducing 50

Figure 11.13: Pruning the Contour Tree By Volume. In this figure, we prune the small pits first, keeping the three visually most prominent features (the peaks) until late in the simplification process.

1. Collapse (simplify) the tree by one vertex-collapse or leaf-prune
2. Uncollapse the tree by one vertex-collapse or leaf-prune

The two collapse sliders repeatedly call these operations as needed, to collapse or uncollapse the contour tree displayed. If large changes in the simplification level are performed frequently, it may be useful to improve on this processing. However, since we expect that the user will rarely wish to have more than $1,000$ edges in the contour tree displayed, this is unlikely to be significant.

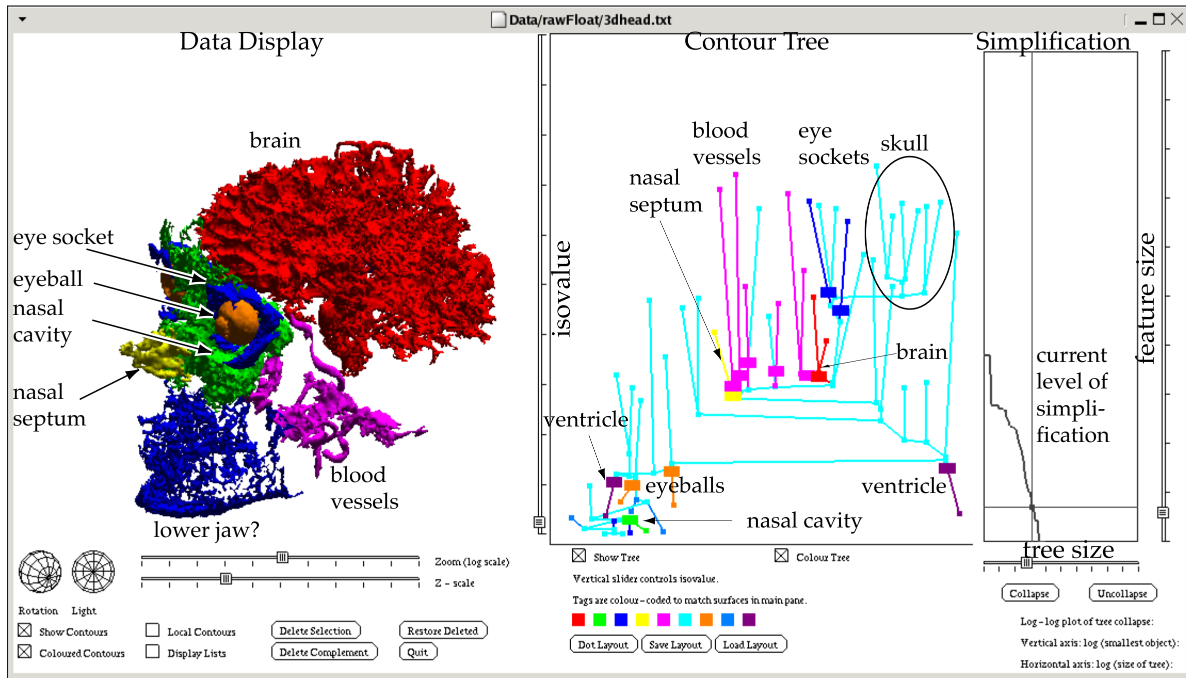


Figure 11.14: Interface to the Simplified Flexible Isosurface. Compared to the interface shown in Figure 9.4, we have added an extra panel for controlling the level of simplification with a log-log plot of tree size vs. simplification measure, and additional controls for setting the colour of superarcs and for loading and saving the layout of the simplified contour tree.

Internally, we maintain the *collapseRecord* array described in Section 11.4, and lists of the *valid* edges and vertices: those currently in the collapsed tree. Collapsing and uncollapsing consist principally of transferring vertices and edges to and from these lists.

11.6.1 Effects of Collapse on Flexible Isosurface

Since the simplification process affects the contours which can potentially be included in a flexible isosurface, we must define suitable behaviour for the flexible isosurface as collapses and uncollapses occur.

When we perform a vertex-collapse at a regular point, we collapse two edges into one. If either of these edges has a contour in the flexible isosurface, we transfer the contour to the collapsed edge. If both edges have a contour, we arbitrarily choose one: for convenience, we always choose the higher-valued. This enforces the rule that we noted for flexible isosurfaces: that we only permit one contour per edge.

When we perform a leaf-prune, we remove a edge from the tree without merging it with any other. If the flexible isosurface included a contour on that edge, it is lost. If that edge was the selection root for a contour evolution, then the evolved set of contours will no longer be valid. We have implemented a simple solution to this, by assuming that the evolution and collapse operations are mutually exclusive. When the collapse operation is requested, any existing selection is *committed*, as described in Section 9.6 before the collapse is performed.

Collapsing a single vertex can be done as shown in Algorithm 11.4. We store the *valid* lists as arrays: for each edge or vertex, we store the index in this array, or -1 if it is not currently valid. This allows us to

```
Input  : A contour tree C of size σ
         A flexible isosurface F
Output : C, reduced in size by one vertex
         F, updated to correspond with C
1  if C has only one edge then
   |  Return
   end
2  Empty the restorable array
3  Let a = collapseRecord[σ]
4  if a is valid (i.e. we are performing a leaf-prune) then
5  |  if a is the selection root then
6  |  |  Commit the selection (i.e. convert to active status)
   |  end
7  |  Clear the selection (for recomputation after the collapse)
8  |  if a is in F then
9  |  |  Remove a from F
   |  end
10 |  Remove a from valid
   else
11 |  Clear selection (for recomputation after the collapse)
12 |  if upperArc(a) is in F then
13 |  |  Remove upperArc(a) from F
   |  end
14 |  if lowerArc(a) is in F then
15 |  |  Remove lowerArc(a) from F
   |  end
16 |  Remove upperArc(a), lowerArc(a) from C
17 |  Insert a in C
18 |  if upperArc(a) was in F then
19 |  |  Add a to F with the seed value that upperArc(a) had
20 |  else if lowerArc(a) was in F then
21 |  |  Add a to F with the seed value that lowerArc(a) had
22 |  if either upperArc(a) or lowerArc(a) was the selection root then
23 |  |  Set a as selection root
   |  end
   end
24 Update selection with the current selection value
```

Algorithm 11.4: Single Interactive Collapse of Contour Tree

remove items from the array in $O(1)$ time by swapping the item to be deleted with the last item in the array.

Note that we start by emptying the restorable array, in case the edge to be deleted is currently restorable. How we treat the selection, however, depends on the details of the collapse. If we are performing a leaf-prune on the root of the current selection, we will be unable to compute the correct set of contours for the selection, so we *unselect*, and convert the current selection to part of the active set of contours. Otherwise, we simply clear the current selection, and recompute it after the collapse. If we are performing a vertex collapse, then either of the two edges collapsed could be active: if so, the combined edge will be active after the collapse. If both of the two edges collapsed are active, we arbitrarily pick the isovalue of either one, to avoid having two active contours on the same edge. Similarly, if either of these two edges was the root of the current selection, the new edge will become the correct root, and we substitute it.

With the exception of the recomputation of the selection, all of the steps here take $O(1)$ time. Selection can take as much a $O(t)$ time: if we wish to perform multiple collapses, we will want to make all the collapses first, then recompute the selection.

```
Input  : A contour tree C of size σ
         A flexible isosurface F
Output : C, increased in size by one vertex
         F, updated to correspond with C

1  if C is fully uncollapsed then
2  |  Return
   end
3  Empty the restorable array
4  Clear the selection array for recomputation after the collapse
5  Let a = collapseRecord[σ + 1]
6  if a is valid (i.e. we are reversing a vertex collapse) then
7  |  if a is in F with isovalue h then
8  |  |  Let v be the vertex we are uncollapsing
9  |  |  Remove a from F
10 |  |  if h > f(v) then
11 |  |  |  Add upperArc(a) to F with isovalue h
   |  |  end
12 |  |  Add lowerArc(a) to F with isovalue h
   |  end
13 |  Remove a from C, add upperArc(a) and lowerArc(a) to C
14 |  if a was the selection root then
15 |  |  if selection root value > f(v) then
16 |  |  |  Set selection root to upperArc(a)
   |  |  end
17 |  |  Set selection root to lowerArc(a)
   |  end
   end
18 Add a to C
19 Update selection with the current selection value
```

Algorithm 11.5: Single Interactive UnCollapse of Contour Tree

11.6.2 Effects of Uncollapse on Flexible Isosurface

To reverse the effects of a collapse, we use Algorithm 11.5. For a vertex-collapse, we expand one edge to the two edges that were merged. Any contour on the combined edge can only belong to one of the two previous edges. We remove the contour from the flexible isosurface, and add it back to the correct edge.

For leaf-pruning, since the pruned leaf was not part of the collapsed contour tree, we know that no contour will be placed on it, which simplifies processing.

11.6.3 Using Collapse to Define Layout and Colour

We noted in Chapter 9 that layout of the contour tree was potentially a problem. We partially resolve this problem using the sequence of collapses to dictate colour and location. In Lemma 11.1, we saw that vertex

collapses leave the contour tree essentially unchanged. If we are going to draw the collapsed edge as a straight line, the collapsed vertex should logically lie on that line. Thus, when we uncollapse by adding a vertex back in to the contour tree, we make the vertex collinear with the edge into which it was inserted. Leaf prunes are a different matter: there is no natural position at which to place them, and we are still exploring layout mechanisms.

Similarly, contours along the collapsed edge should retain their tag colour when uncollapsed. It follows that every edge along one of the collapse paths described in Section 11.4.1 should have the same colour. The colour of the path is determined when the path is initially created by uncollapsing a leaf prune. We have implemented a simple control in the flexible isosurface interface that sets the colour of an edge of the contour tree (and any contour to it): this colour is then inherited by any edges on the path corresponding to the edge whose colour was changed. Beyond this, any policy for setting contour colours is likely to be application-specific.

11.7 Using Topological Simplification to Remove Noise

In this chapter, we have seen how to simplify the contour tree in a meaningful fashion, based on geometric measures of the importance of a contour. Since we know that experimental noise is typically small scale and generates spurious peaks and pits in the data, we can use our simplification method to suppress noise in the data in a spatially non-uniform fashion.

We saw in Section 11.3 that leaf pruning completely eliminates the corresponding region from consideration for drawing contours. This flattens out the region so that the entire region has the isovalue of the interior vertex of the pruned edge. We claim that this has the effect of removing noise from the data without losing spatial detail in regions where less noise is present.

11.8 Summary of Contributions

In this chapter, we have introduced several contributions. In Section 11.1, we showed that the contour tree is heavily influenced by noisy data, and needs some form of simplification for practical application. Similary, Section 11.2 showed that conventional forms of simplification, whether by filtering, by mesh simplification, or by isosurface simplification, fail to simplify the contour tree sufficiently to be useful.

Section 11.3 then developed the next contribution: the application of simple graph-theoretic simplifications to the contour tree, and observed that the effect of this was identical to the simplification applied by Takahashi, Fujishiro & Takeshima [TFT01]. This thesis, however, goes beyond these authors' work. Section 11.3 also introduced the idea of using geometric measures to guide the simplification, while Section 11.4 defined a simplification algorithm, and introduced *hierarchical path seeds* for isosurface extraction after simplification. This, in turn, was used in Section 11.6 to extend the flexible isosurface interface of Chapter 9 to simplified contour trees, allowing interactive browsing not only of the isosurfaces of the contour tree, but also of the contour tree at various levels of collapse.

Finally, Section 11.7 introduced the notion of *topological filtering*: using the topological and geometric information stored in the contour tree to simplify noisy data.

This chapter also concludes Part III of this thesis, which has shown how to unify the following:

1. *topology*, in the form of the contour tree (Chapter 7)
2. *efficient isosurface extraction* using path seeds (Chapter 8)
3. *exploratory visualization* using the flexible isosurface (Chapter 9), and
4. *geometry*, in the form of local spatial measures (Chapter 10)
5. *topological simplification* using local spatial measures (Chapter 11)

to provide insight into scalar fields for scientists, doctors and engineers.

In doing so, we assumed that the input data came in the form of a simplicial mesh with no degeneracies. In the next Part of this thesis, we show how to relax this assumption and work with arbitrary meshes with potential degeneracies.

Part IV

Imperfect Data

In the previous Part, the contributions of this thesis used geometric and topological information stored in the contour tree to create new tools for exploratory visualization and data simplification. To simplify the discussion, we assumed that the function being studied is defined by barycentric interpolation on a simplicial mesh, with no degeneracies. In this Part, we show how to relax these assumptions.

The first contribution of this Part is to relax Assumption 1 and Assumption 2: that the function is defined by barycentric interpolation over a simplicial mesh. Chapter 12 extends contour tree computations to arbitrary meshes, in arbitrary dimension, with arbitrary interpolants. We show that contour tree computations need not be restricted to contours of functions that are well-defined by an interpolant, but can also be performed for surfaces derived from tessellation cases such as Marching Cubes, provided that a simple nesting condition is satisfied

In practice, most data comes on a cubic sampling grid, and isosurfaces are commonly extracted using either trilinear interpolation, or the *ad hoc* Marching Cubes tessellation cases. Chapters 13 and 14 discuss these in more detail. Chapter 14 also adds a simple shortcut for computing contour trees for the Marching Cubes cases.

Chapter 15 then relaxes Assumption 3: that no two vertices have the same isovalue. Chapter 15 discusses how to relax this assumption by perturbing the input data.

Chapter 12

Contour Trees for Non-Simplicial Meshes

12.1 Introduction

As we saw in Chapter 6, the principal algorithms for computing the contour tree have generally assumed that the data to be visualized is defined over a tetrahedral, or simplicial, mesh, using barycentric interpolation. Although geometrically convenient, this assumption is difficult to reconcile with the fact that most data in three dimensions is sampled on a cubic grid.

The contributions of this chapter extend contour tree algorithms to any arbitrary mesh, interpolant, and dimension, define a finite state machine for computing the necessary connectivity, and give an algorithm to extract the finite state machine automatically from a description of the possible cases. Furthermore, we show how to compute the contour tree even for cases such as Marching (Hyper-) Cubes, which tessellate contours in an ad hoc manner not equivalent to any known interpolant.

We also extend minimal seed sets and path seeds to these arbitrary meshes, and introduce *piecewise continuation* to allow single surfaces to be traced through the mesh, even when multiple surfaces can intersect a single cell.

Section 12.2 looks at relevant previous work, while Section 12.3 introduces *join and split graphs* to track the evolution of contours in arbitrary meshes.

Section 12.4 discusses a universal method for constructing join and split graphs for any interpolant. Section 12.5 shows how this is applied to the bilinear interpolant, while Section 12.6 discusses how to extend the contour tree to contours defined solely by tessellation methods. Section 12.7 and Section 12.8 discuss the changes required to path seeds, and to the continuation method for isosurface extraction. Finally, Section 12.9 summarizes the contributions of this chapter.

12.2 Previous Work

The algorithms described in Chapter 7 assume that f is defined over simplicial meshes, with the exception of Itoh & Koyamada's skeletonization approach [IK94, IK95, IYK01] and Pascucci & Cole-McLaughlin's work on trilinear interpolants [PCM02]. In practice, most scientific data uses cubic meshes, not simplicial meshes. For the contour tree to be useful, it is necessary either to modify the data or to modify the algorithm.

The easiest way to modify the data is to take the cubic mesh, and subdivide each cube into tetrahedra. This introduces artifacts into the interpolant function, which range from the visually unpleasant, through the directionally biased, to the topologically inaccurate. These artifacts have been analyzed by Carr, Möller & Snoeyink [CMS01], who conclude that subdividing cubes into 24 tetrahedra each is acceptable, albeit expensive.

Instead of this, we could acquire the data in such a way that simplices are the natural cells for the mesh. Carr, Theußl & Möller [CTM03] have shown, however, that doing so with body-centred cubic sampling results in more expensive, lower quality isosurfaces than working with the standard Marching Cubes cases.

The alternate approach, modifying the algorithm, can also be used, at the expense of some additional complexity. Pascucci & Cole-McLaughlin [PCM02] have already extended the contour tree algorithms to the trilinear interpolant function over cubic meshes. To do so, they identified that any given cell of the trilinear interpolant has at most 8 saddle points: 6 on the faces, 2 in the body of the cube, and that there were only 4 possible join trees (or split trees). They then constructed an oracle that, for a given set of vertices in the cell, returned the correct join (or split) tree for that cell.

In Section 12.3, we will generalize and formalize this approach, by defining *join and split graphs* which are sufficient graphs on which the sweep and merge algorithm can operate. These graphs can also be used to extend the algorithms of Takahashi, Fujishiro & Takeshima [TFT01], of Pascucci & Cole-McLaughlin [PCM02], and of Chiang et al. [CLLR02]. In all cases, this extension comes from the recognition that these algorithms are essentially graph-theoretic.

12.3 Join and Split Graphs for Arbitrary Meshes

Chapter 7 describes the sweep and merge algorithm of Carr, Snoeyink & Axen [CSA03] for computing the contour tree. This algorithm performs two sweeps through the data to construct partial trees called the join tree and split tree, then combines the join tree and the split tree in a third stage similar to that used by Takahashi et al. [TIS$^+$95] and by Takahashi, Fujishiro & Takeshima [TFT01]. Correctness of the algorithm is proven by showing that the edges of a simplicial mesh constitute a graph G such that the following properties hold:

I. G contains all joins and local maxima of the function f defined over the mesh.

II. G contains all splits and local minima of the function f defined over the mesh.

III. For any isovalue h, two vertices $u, v \in G$ are connected by a path above the isovalue h exactly when the points u, v in the mesh are also connected by a path above the isovalue h.

IV. For any isovalue h, two vertices $u, v \in G$ are connected by a path below the isovalue h exactly when the points u, v in the mesh are also connected by a path below the isovalue h.

Note that the paths in Properties III and IV need not be monotone, but that our definition in Chapter 6 used monotone paths. However, any path can always be decomposed into a sequence of monotone paths and paths lying on contours, so little turns on this. We now separate the properties required for computing the join tree from those required for computing the split tree:

Definition 12.1 *For a given function f, a* join graph *is a graph that satisfies Properties I & III, and a* split graph *is a graph that satisfies Properties II & IV.*

If we know join and split graphs for a given mesh, we can compute the contour tree, as the algorithm relies solely on these properties. This was used by Pascucci & Cole-McLaughlin [PCM02] for their divide-and-conquer algorithm.

Theorem 12.1 *Given a join graph J and split graph S for a function f, Algorithm 7.1 will correctly compute the contour tree for f in $O(sort + N + t\alpha(t))$ time, where $N = max(\sum_{v \in J} degree(v), \sum_{v \in S} degree(v))$ and the related parameter $n = max(||J||, ||S||)$.*

Proof: Proof is essentially identical to that given for Algorithm 7.1 in my M.Sc. thesis [Car00] and the corresponding journal article [CSA03], and is omitted. □

We next analyse the location of critical points of f to find out where they occur in the cells of the mesh M. We start by showing that a local maximum or minimum of f must be a local maximum or minimum of some cell of f:

Lemma 12.2 *Let M be a polyhedral mesh, and let f be a function defined over M. Let p be a local maximum of f. Then p must be a local maximum of some cell K of the mesh.*

Proof: Assume that small values of δ and ϵ are chosen to satisfy Definition 6.8, and let p be a local maximum of f. Then, by Definition 6.12, $\delta^+(p) = 0$. It follows that for any cell K to which p belongs, $\delta^+(p) = 0$ in K, and p is a local maximum of K. □

The dual to this lemma also holds:

Lemma 12.3 *Let M be a polyhedral mesh, and let f be a function defined over M. Let p be a local minimum of f. Then f must be a local minimum of some cell K of the mesh.*

Proof: Dual to proof of Lemma 12.2. □

The equivalent result is not true for saddles, which may occur at saddles in cells of the mesh, but may also occur at regular points on the boundary between two cells:

Lemma 12.4 *Let M be a polyhedral mesh, and let f be a function defined over M. Let p be a join of f. Then f must be a join of some cell K of the mesh, or a join or local maximum of the intersection $K_1 \cap K_2$ of two cells K_1, K_2 of the mesh.*

Proof: Assume that p belongs to a single cell K of f. Without loss of generality, assume that $\delta^+(p) = 2$. Then there must be at least two distinct contours γ_1, γ_2 of f at isovalue $f(p) + \delta$ that intersect $B_\epsilon(p)$. If we take the intersection of K and γ_1, we get at least one, and possibly more than one, contour of $f|K$. At least one of these contours intersects $B_\epsilon(p)$. Since this is true for both γ_1 and γ_2, we know that the number of contours of $f|K$ that intersect $B_\epsilon(p)$ is at least 2. It then follows that p is a join of $f|K$.

Now assume that p belongs to more than one cell in the mesh. Then p is a critical point of f such that $p \in K_1 \cap \ldots \cap K_m$. Without loss of generality, assume that $\delta^+(p) = 2$. Suppose there exists a cell K such that $\delta^+(p) = 2$ in K. Then p is a join of K, and satisfies the first branch of the lemma. Therefore assume, without loss of generality, that no such cell K exists.

It then follows that there must be two cells K_1, K_2 such that $\delta^+(p) = 1$ in each of K_1 and K_2, and the contours γ_1 in K_1 and γ_2 in K_2 that intersect $B_\epsilon(p)$ do not belong to the same global contour.

Now assume that p is *not* a join or local maximum of $K_1 \cap K_2$. Then there exists exactly one contour γ of $f|(K_1 \cap K_2)$ that intersects $B_\epsilon(p)$. This contour γ intersects both K_1 and K_2. We know that there is only one contour γ_1 of $f|K_1$ at isovalue $f(p) + \delta$ that intersects $B_\epsilon(p)$. Therefore, $\gamma \cap K_1 = \gamma_1$. By a similar argument, $\gamma \cap K_2 = \gamma_2$. But then γ_1 and γ_2 are connected by γ, and are part of the same global contour. Since this creates a contradiction, it follows that p is either a join or a local maximum of $K_1 \cap K_2$, as required. □

Again, the dual holds:

Lemma 12.5 *Let M be a polyhedral mesh, and let f be a function defined over M. Let p be a split of f. Then f must be a split of some cell K of the mesh, or a split or local maximum of the intersection $K_1 \cap K_2$ of two cells K_1, K_2 of the mesh.*

Proof: Proof is dual to that for Lemma 12.4. □

We can use these lemmas to obtain two useful corollaries:

Corollary 12.6 *Let P be the set composed of all vertices of the mesh, all local maxima and joins of the boundary of any cell, and all local maxima and joins of any cell. Then any graph including P satisfies Property I.*

Proof: Let p be a local maximum of f. From Lemma 12.2, we know that it is a local maximum of some cell K. Since all such local maxima are included in P, $p \in P$.

Now let p be a join of f. From Lemma 12.4, we know that it is a join of some cell K, or a join or local maximum of the intersection $K_1 \cap K_2$ of two cells K_1, K_2. In each case, however, $p \in P$. □

Again, the dual of this applies:

Corollary 12.7 *Let P be the set composed of all vertices of the mesh, all local minima and splits of the boundary of any cell, and all local minima and splits of any cell. Then any graph including P satisfies Property II.*

Proof: Proof is dual to that for Corollary 12.6 □

Now we can look at some straightforward ways of defining join and split graphs.

Theorem 12.8 *For a function f, the following are true:*

1. *The contour tree of f is both a join and a split graph.*
2. *The join tree of f is a join graph.*
3. *The split tree of f is a split graph.*
4. *The union of the contour trees for each polyhedral cell of the mesh is both a join and a split graph.*
5. *The union of the join trees for each cell of the mesh is a join graph.*
6. *The union of the split trees for each cell is a split graph.*
7. *If f is defined by barycentric interpolation over a simplicial mesh, then the edges of the mesh are both a join and a split graph.*
8. *The Morse complex of f (see Section 4.5.2) of the function f is both a join and a split graph.*

Proof: (1): Trivially, the contour tree satisfies Properties I & II. The contour tree satisfies Properties III & IV by decomposing the path into monotone paths and paths lying along contours, then invoking Corollary 6.1 and Lemma 6.5.

(2): Again, trivially, the join tree satisfies Property I. Property III is shown to be satisfied in [Car00] and [CSA03].

(3): Proof is dual to the proof for the join tree.

(4 - 6): Properties I and II follow from Corollary 12.6 and Corollary 12.7 respectively. For Properties III and IV, we decompose each path P into subpaths in the individual cells of the tree, and map those subpaths to the contour tree by Theorem 6.6, and thence to the join or split tree.

(7): Banchoff [Ban67] showed that the critical points of a simplicial mesh are located at the vertices of the mesh, so Properties I & II are also satisfied. Property III is shown in [Car00] and [CSA03] to be satisfied by deforming each path in f to the edges of the mesh with strictly higher values than the path, and mapping paths in the contour tree to paths in f by following edges of the mesh. Property IV follows by the usual dual relationship.

(8): Without a formal definition of the Morse complex, we can only sketch this proof. Properties I & II hold because the Morse complex contains all Morse critical points, and by Definition 6.8 and Definition 6.5, each critical point is a Morse critical point. Properties III & IV hold by a deformation similar to that used for simplicial meshes, but deforming paths instead to the boundaries of the regions of the Morse complex. □

Of these graphs, the Morse complex is theoretically attractive as a basis for computing the contour tree, but it is difficult to compute in its own right. An algorithm has been found for two-dimensional simplicial meshes [BEHP03, EHZ01], but not for three-dimensional data. Even for two dimensions, the computational cost has not yet been clearly established. The contour tree of f is sufficient to compute itself, but this is a

circular computation, and can be ignored. Similarly, using the join tree to compute itself, or the split tree to compute itself, is pointless.

The union of the cell-wise join, split or contour trees is the most practical method of dealing with non-simplicial meshes and non-barycentric interpolants. Pascucci & Cole-McLaughlin [PCM02] used the unions of join and split trees to compute the contour tree of a trilinearly interpolated mesh, and as the foundation for a divide-and-conquer variation on the sweep and merge algorithm for contour tree computation. These authors apparently use a large case table to predict all possible join and split trees for the trilinear interpolant.

We adopt a simpler approach using finite state machines that can be implemented for any possible mesh and interpolant. Since our method defines how to extract join and split graphs, it can also be used as an oracle for Pascucci & Cole-McLaughlin's divide-and-conquer approach.

12.4 Join and Split Graph Lookup Tables

In this section, we will see how to construct lookup tables that compute join and split graphs for any interpolant, based solely on previous analyses of the topologically distinct contours possible in a given cell. We observe that, no matter the interpolant used, the visualization process will end by tessellating the contour. For each such interpolant, there is typically a static analysis that defines a set of cases for the tessellation process, each valid over a specified range of isovalues. The best-known set of these cases are the Marching Cubes cases of Lorenson & Cline [LC87], as discussed in Chapter 5, but other examples include Marching Tetrahedra [Blo88, NB93, WvG90], and the trilinear interpolant [Che95, LB03].

In these analyses, the cases used to tessellate the isosurfaces fully describe the connectivity of the extracted contours in the cell. We will refer to such a set of cases as *tessellation cases*. To determine the connectivity of global contours, we start with the tessellation cases, and extract the join and split trees for each cell. Then, by Theorem 12.8 and Theorem 12.1, we can compute the contour tree for f. We will see later that this approach can even be extended to tessellation cases that are inaccurate with respect to a particular interpolant. For now, however, we assume that there is a known interpolant applied to each cell, and that tessellation cases have been defined for it.

Recall that the contour tree algorithms in Chapter 7 typically sweep through the function from high to low isovalues, or vice versa. As the sweep progresses, each cell in the mesh will change from one case to another a finite number of times. If we identify all of the possible isovalues at which these cases change, we can then define a lookup table that identifies the up- or down- arcs at each vertex in the join or split graph.

We start by developing a formal definition of tessellation cases:

For any given interpolant and mesh, there are a finite number of *potential critical points*, defined as:

Definition 12.2 *The* potential critical points *for a cell K are the vertices of the cell and the Morse critical points of the function f restricted to the cell K.*

Definition 12.3 *A* tessellator *is an algorithm that given an isovalue h, a function f, and a cell K of the mesh, approximates the contours in K at the isovalue h. We call the approximated contours generated* tessellated contours*.*

Definition 12.4 *Two contours or tessellated contours τ_1, τ_2 are* tessellation-equivalent *if they are topologically equivalent and partition the potential critical points of K in the same way.*

Definition 12.5 *A* tessellation case *for a cell K is an equivalence class of tessellated contours for all functions f sharing the same interpolant with respect to K.*

Definition 12.6 *A tessellator and its set of tessellation cases are* accurate *if, for any h, the tessellated contours at h are tessellation-equivalent to the actual contours at h.*

Definition 12.7 *A* tessellation discriminator *is an isovalue h such that there exists some $\epsilon_0 > 0$ such that for all $\epsilon < \epsilon_0$ the tessellated contours of a cell K at $h + \epsilon$ and $h - \epsilon$ are not tessellation equivalent.*

We can now establish a firm topological foundation for linking tessellation and topology:

Lemma 12.9 *For each vertex v_i of a cell K in the mesh with isovalue $f(v_i)$ distinct from the isovalue of at least one other vertex $f(v_j)$, $f(v_i)$ is a tessellation discriminator for every accurate set of tessellation cases for f.*

Proof: Without loss of generality, $f(v_i) < f(v_j)$. Choose $0 < \epsilon < f(v_j) - f(v_i)$, and let τ_1, τ_2 be the tessellated contours at $f(v_i) + \epsilon$ and $f(v_i) - \epsilon$. Then v_j must be above both τ_1 and τ_2, but v_i must be below τ_1 and above τ_2. Therefore, τ_1 and τ_2 do not partition the vertex set of the cell in the same way, and are not tessellation equivalent. □

Lemma 12.10 *If p is a Morse critical point of $f|K$, then $f(p)$ is a tessellation discriminator for K for every accurate set of tessellation cases for f.*

Proof: Since p is a Morse critical point, we know that the local topology changes at p. It follows that the tessellation contours at $f(p) + \epsilon$ and $f(p) - \epsilon$ are not topologically equivalent, and are therefore not tessellation equivalent. □

Theorem 12.11 *Let h be a tessellation discriminator for an accurate tessellator of a function f over a cell K. Then h is the isovalue of either a vertex of the cell K, or of a Morse critical point of $f|K$.*

Proof: Choose a small ϵ such that no $h - \epsilon < h' < h + \epsilon$ is also a tessellation discriminator, and let τ_1, τ_2 be the tessellated contours at $h - \epsilon, h + \epsilon$ respectively. Then, by Definition 12.7, τ_1 and τ_2 are not tessellation-equivalent.

Suppose that τ_1 and τ_2 are not topologically equivalent. Then there must exist some Morse critical point p at isovalue strictly between $h - \epsilon$ and $h + \epsilon$ at which the topology of the contours change. Since ϵ was chosen arbitrarily, we know that $f(p) = h$.

We now suppose that τ_1 and τ_2 are topologically equivalent. Since they are not tessellation equivalent, they must partition the potential critical points of K in different ways. But this can only happen if

there is some vertex v with isovalue between $h - \epsilon$ and $h + \epsilon$. Again, since ϵ was chosen arbitrarily, it follows that $f(v) = h$. □

From these lemmas, several interesting observations arise. The first observation is that it is now easy to understand why Marching Tetrahedra [Blo88, NB93, WvG90] are accurate under Definition 12.6, as are the trilinear tessellators of Chernyaev [Che95] and Lopes & Brodlie [LB03]. Both of these algorithms determine the complete set of tessellation discriminators by classifying the vertices and Morse critical points of each cell, then using this information to choose which tessellation case to render.

The second observation is that, since the potential critical points are a superset of the critical points and vertices, they satisfy Properties I & II of join and split graphs. Thus, if we can build a suitable graph using these vertices, we can build a join (or split) graph for each cell. Having done so, we can combine the join graphs for each cell to obtain a join graph for the entire function f.

The third observation is that the partition of the potential critical points by a tessellation contour holds the connected components of $\{x : f(x) \geq h\}$ and $\{x : f(x) \leq h\}$ for any isovalue h. But this is precisely the information we need to compute the join (or split) tree. Therefore, to determine whether a given potential critical point is in fact a critical point, all we need to do is consider the changes to the partition of the potential critical points as the isovalue sweep passes the potential critical point. And we can do this by comparing the tessellation cases before and after the isovalue sweep passes the potential critical point.

As the isovalue sweep passes a local maximum v, a new subset containing only v will appear in the partition. As the isovalue sweep passes a join v, two subsets S_1 and S_2 will be combined to form a new subset $S_1 + S_2 + v$. For a potential critical point v which is neither a join nor a local maximum, v will be added to an existing subset of the partition. Moreover, if v is not a local maximum, we will be able to choose a monotone path P from v to some w belonging to the same subset of the partition. If we treat this monotone path as an edge (v, w), we will be able to construct a graph satisfying properties I, II and IV: i.e. a join graph.

Since we are modelling the development of a set of tessellated contours that are topologically equivalent to the real contours, it should come as no surprise that we can prove some properties that are similar to the properties of the actual contours:

Lemma 12.12 *Let v be a local maximum of the cell K, and let Π_1, Π_2 be the partition of the potential critical points induced by accurate tessellation contours at $f(v) + \delta$ and $f(v) - \delta$ respectively, for arbitrarily small δ. Let $S_1, \ldots, S_q$ and $T_1, \ldots, T_r$ be the subsets of Π_1, Π_2 that consist of potential critical points with values greater than $f(v)$. Then $r = q + 1$, and there exist indices $k_1, \ldots, k_r$ such that $S_i = T_{k_i}$ for all $1 \leq i \leq q$, and $T_{k_r} = \{v\}$.*

Proof: We know that v changes the connectivity of sets of the form $\{x : f(x) \geq h\}$ only in the vicinity of v. Since the tessellation contours are accurate, it follows that the isovalue sweep preserves the subsets $S_1, \ldots, S_q$ of Π_1: i.e. that each S_i corresponds to some T_{k_i}. Furthermore, since v is a local maximum, it must now belong to a new subset in Π_2. □

Lemma 12.13 *Let p be a join of the cell K, and let Π_1, Π_2 be the partition of the potential critical points induced by accurate tessellation contours at $f(v) + \delta$ and $f(v) - \delta$ respectively, for arbitrarily small δ. Let $S_1, \ldots, S_q$ and $T_1, \ldots, T_r$ be the subsets of Π_1, Π_2 that consist of potential critical points with values greater*

than $f(v)$. Then $r < q$, and there exist indices $k_1, \ldots, k_r$ such that $S_i \subseteq T_{k_i}$ for all $1 \leq i \leq q$.

Proof: Let $S_1, \ldots, S_q$ and $T_1, \ldots, T_r$ be the subsets of Π_1, Π_2 that consist of potential critical points with values greater than $h = f(v)$. We know that the potential critical points in S_1 belong to the same connected component of $\{x : f(x) \geq h + \delta\}$. But that means that the potential critical points in S_1 must also belong to the same connected component of $\{x : f(x) \geq h - \delta\}$. It then follows that each S_i is contained in some T_{k_i}.

Since p is a join, it must be connected in $\{x : f(x) \geq h - \delta\}$ to two vertices v_1, v_2 that belong to different subsets S_{i_1}, S_{i_2} of Π_1, the partition at $h+\delta$. Since each of v_1, v_2 is connected to p, they must belong to the same subset T_k of Π_2, the partition at $h - \delta$. Therefore S_{i_1} and S_{i_2} are both contained in T_k, as is p, so there must be fewer subsets in Π_2 than in Π_1. □

Lemma 12.14 *Let p be a point in the cell K lying on a contour that does not pass through a local maximum or join, and let Π_1, Π_2 be the partition of the potential critical points induced by accurate tessellation contours at $f(p) + \delta$ and $f(p) + \delta$ respectively, for the usual small δ. Let $S_1, \ldots, S_q$ and $T_1, \ldots, T_r$ be the subsets of Π_1, Π_2 that consist of potential critical points with values greater than $f(p)$. Then $r = q$, and there exist indices $k_1, \ldots, k_r$ such that $S_i = T_{k_i}$ for all $1 \leq i \leq q$ except that if p is a potential critical point, then there exists some j for which $S_j + \{p\} = T_{k_j}$.*

Proof: As in Lemma 12.13, each S_i must be a subset of some T_{k_i}. p is not a join or local maximum, so there is only one contour γ at $h + \delta$ that intersects $B_\epsilon(p)$. Because p is not on a contour that passes through a join, no two of the S_i belong to the same T_k. And because p is not a local maximum, it cannot belong to a new subset of the partition. It follows that if p is a potential critical point, it must be added to some S_j, i.e. that $S_j + \{p\} = T_{k_j}$. □

These lemmas have shown that the subsets of the partition induced by a tessellation discriminator reflect the connectivity of the sets $\{x : f(x) \geq h\}$ that are tracked by the join tree. In order to build a join graph, we now need to build a set of edges that satisfy Property III. We start by generating these edges, then prove that Property III holds.

Lemma 12.15 *Let p be a point in a cell K, and let $S_1, \ldots, S_q$ be the subsets of the partition of potential critical points induced by an accurate tessellation contour at $f(p) + \delta$, for the usual small delta. Then, for each connected component γ_i of $\{x : f(x) \geq f(p) + \delta\}$ into which a f-monotone path in $\mathcal{R}$ from P leads, there exists a path P_i from p to some potential critical point v_i of the corresponding subset S_i.*

Proof: For each such connected component γ_i, we generate an ascending path from p until we reach a local maximum. This local maximum must belong to the corresponding subset S_i. And since each local maximum is at a potential critical point, we know that this local maximum is v_i. □

We will build a join graph using these paths, represented as individual edges (v_i, p):

Theorem 12.16 *Let (V, E) be a graph with vertex set V, the set of potential critical points of some cell K in the mesh. For each vertex $v \in V$, let Π_v, P_v be the partitions of the potential critical points induced by accurate tessellation contours at $f(v) + \delta$ and $f(v) - \delta$, respectively. Then, for each subset S_i of Π_v that is contained in the same subset T_v of P_v to which v belongs, choose a monotone path from v to some potential*

critical point $v_i \in S_i$, and let (v_i, v) be an edge of E. If E contains only these edges, then (V, E) is a join graph of $f|K$.

Proof: First note that, by Lemma 12.15, we will always be able to choose a suitable monotone path vv_i, so E is well-defined.

We know that the set of potential critical points of K contains the set of vertices, joins and local maxima of K. It follows that Property I is satisfied.

To prove Property III, assume that we have vertices p and q in the graph, with $f(p) < f(q)$, and let E' denote the graph E restricted to edges with vertices at isovalues $\geq f(p)$.

($\Leftarrow$): Let Q be a path in E' between p and q. Then Q consists of a sequence of edges $e_1, \ldots, e_m$, each of which was defined by choosing a f-monotone path in $\mathcal{R}$. Note that each e_i is an edge between two vertices at isovalues $\geq f(p)$, and that the corresponding f-monotone path therefore cannot drop below the isovalue $f(p)$. Let P be the concatenation of these f-monotone paths. P is not a f-monotone path, but our definition of Property III does not require f-monotone paths, merely paths that do not drop below $f(p)$. Since this is true for each of the component paths, it is also true for P, and the result follows.

($\Rightarrow$): Without loss of generality, each v_i to which a path is traced is a local maximum. To see that this is true, assume not. Then v_i is not a local maximum, and there exists a further ascending path from v_i to some v_j, and so on. Since we are always ascending, and there are a finite number of vertices, this ascent will eventually end at a local maximum.

Let p, q be vertices of the graph that are connected in $f|K$ by some path $P \subset \{x : f(x) \geq f(p)\}$. From Lemma 12.15, we can generate paths from p and q to local maxima m_p and m_q. We now show that we can trace a path in $f|K$ between v_p and v_q, using a counting argument. Without loss of generality, $v_p \neq v_q$.

Now assume that there are M local maxima and N joins contained in E'. We chose one edge to add to E' for each distinct direction of ascent from each join, so the total number of edges in E' is equal to the sum of the updegrees of all joins in E'.

If we were to compute the join tree of E', all M local maxima and N joins would belong to it. Assign each arc in the join tree to the upper of the two nodes it connects. But the down-degree of every node in E' except the lowest join must be 1, so we have $M + N - 1$ arcs in the join tree of E'. Since each local maximum has an up-degree of 0, it follows that the sum of the updegrees of all joins in E' must also be $M + N - 1$.

Returning to E' itself, we now see that it has $M + N - 1$ edges in it. Moreover, we claim that there are no cycles in E'. Suppose that there were such a cycle C, and let s be the lowest vertex in the cycle. Since each edge in E' has a join at its lower end, s must be a join. Then the ascending paths corresponding to the edges rs, st in the cycle are ascending paths into distinct connected components of $\{x : f(x) > f(s)$, because of how we chose our edges. The rest of the cycle $C - \{s\}$ is a sequence of paths in $\{x : f(x) > f(s)\}$, because each edge in $C - \{s\}$ is a monotone path between two vertices with values higher than $f(q)$. Therefore r and t belong to the same connected component of $\{x : f(x) \geq f(s)\}$. Since this is a contradiction, no cycle C exists in E'.

We now know that E' is a cycle-free graph over $M + N$ nodes containing $M + N - 1$ edges. But any cycle-free graph with one more node than edges is a tree. It follows that E' is a tree, and that all of the

vertices of E' are connected to each other by paths in E', as required.

Let P be the path between v_p and v_q in E'. Then pPq connects p and q in E as required.

Since we have now shown that we can create a path P in f for every path Q in E' and vice versa, Property III holds, and it now follows that E is a join graph for $f|K$. □

We have now shown that a join graph can be constructed for any cell in the mesh, based purely on the tessellation cases used to extract the contours, and that we can take the union of these join graphs to generate a join graph for the entire mesh. Rather than explicitly constructing the join graph for each cell, however, we will compute the edges adjacent to a given vertex v as required.

```
    Input  : A potential critical point v of a mesh M
             A lookup table ascent of ascending edges given a potential critical point and
             tessellation case
    Output : The set of higher-isovalued neighbours for v in a valid join graph for M
1   for each cell containing v do
2   |   Compute C, the tessellation case for the cell, at an isovalue of f(v) + ε
3   |   for each edge e = vw stored in ascent[C, v] do
4   |   |   Add (v, w) to the join graph for M (i.e. treat v, w as connected)
    |   end
    end
```

Algorithm 12.1: Using a Lookup Table to Find Join Graph Neighbours

We do so by defining an algorithm to find the neighbours of a given potential critical point in the join graph of a cell, by pre-computing a lookup array indexed by the tessellation case immediately before the join sweep reaches the potential critical point, and the identity of the potential critical point. For each tessellation case, and each potential critical point p, the array specifies the ascending edges at p in the join graph, as shown in Algorithm 12.1.

The algorithm for computing the lookup table itself is shown as Algorithm 12.2.

Theorem 12.17 *Algorithm 12.2 and Algorithm 12.1 correctly compute the edges in the join graph for the mesh M.*

Proof: By Theorem 12.8, we know that the union of the join graphs of each cell is a join graph for the mesh, so all we must prove is that Algorithm 12.2 and Algorithm 12.1 correctly compute the edges in the join graph for any cell K in M.

Recall that, in Theorem 12.16, we started by choosing uphill paths from each join j in $f|K$. Step 10 implements this step, while the surrounding loop establishes the components of Π_1 into which we need to choose uphill paths, where Π_1 is the partition of the potential critical points of K induced by the contour through the join j.

Since this computation is combinatorial in nature, Algorithm 12.1 computes it in advance and stores it in *ascent*[], to be recovered later in Algorithm 12.1. □

Analyzing the runtime of Algorithm 12.2 depends on two parameters: N_{cases}, the number of tessellation cases, and N_{pcp}, the maximum number of potential critical points in a cell. Then:

```
Input    : A set of tessellation cases for a given interpolant
Output   : A lookup table ascent of ascending edges for each potential critical point and
           tessellation case
1  for each tessellation case C do
2  |  Let verticesAbove be the set of potential critical points above the contour in C
3  |  for each potential critical point p that is not in verticesAbove do
4  |  |  Let D be the case where verticesAbove ∪ {p} are above the contour
5  |  |  if no such case D exists then
6  |  |  |  Report that the interpolant is inconsistent, and halt.
   |  |  end
   |  |  else
7  |  |  |  Let δ be the region above the contour in D to which p belongs
8  |  |  |  for each region γ above the contour in C do
9  |  |  |  |  if γ ⊂ δ then
10 |  |  |  |  |  Choose a vertex w in γ such that a f-monotone path P in f|K connects p
   |  |  |  |  |  and w
11 |  |  |  |  |  Add edge pw to ascent[C, p]
   |  |  |  |  end
   |  |  |  end
   |  |  end
   |  end
   end
```

Algorithm 12.2: Computing the Join Graph Lookup Table

Theorem 12.18 *Algorithm 12.2 takes $O(N_{cases}N_{pcp}^3)$ time to execute.*

Proof: Step 1 will execute N_{cases} times. In each iteration of this loop, Step 3 will execute N_{pcp} times. Step 8 will execute at most N_{pcp} times. If γ and δ are stored as arrays of bits representing which potential critical points belong to each set, then Step 9 executes in at most N_{pcp} time. Step 10 can take the form of checking all other potential points to find a suitable edge, in which case it, too, takes at most N_{pcp} time. The result then follows. □

The payoff to this procedure is that the cost of generating the join graph is amortized in the precomputation of the *ascent*[] lookup table, which is determined solely by the tessellation cases. It follows that:

Theorem 12.19 *For any potential critical point v, Algorithm 12.1 takes $\Theta(\delta^+(v))$ time to compute the up-arcs of v in the join graph.*

Proof: Follows immediately from the fact that we simply lookup the edges in an array with access cost of $O(1)$. □

Note that the lookup table we describe here can be used as an oracle for Pascucci & Cole-McLaughlin's divide-and-conquer contour tree computation algorithm. The only difficulty we may face is choosing suitable edges p and w at Step 10 in Algorithm 12.2. This, however, only needs to be done once for each interpolant, and can be done by hand if needed.

If, instead of ascending paths, we choose the edge from v to the lowest-valued vertex v_i in each

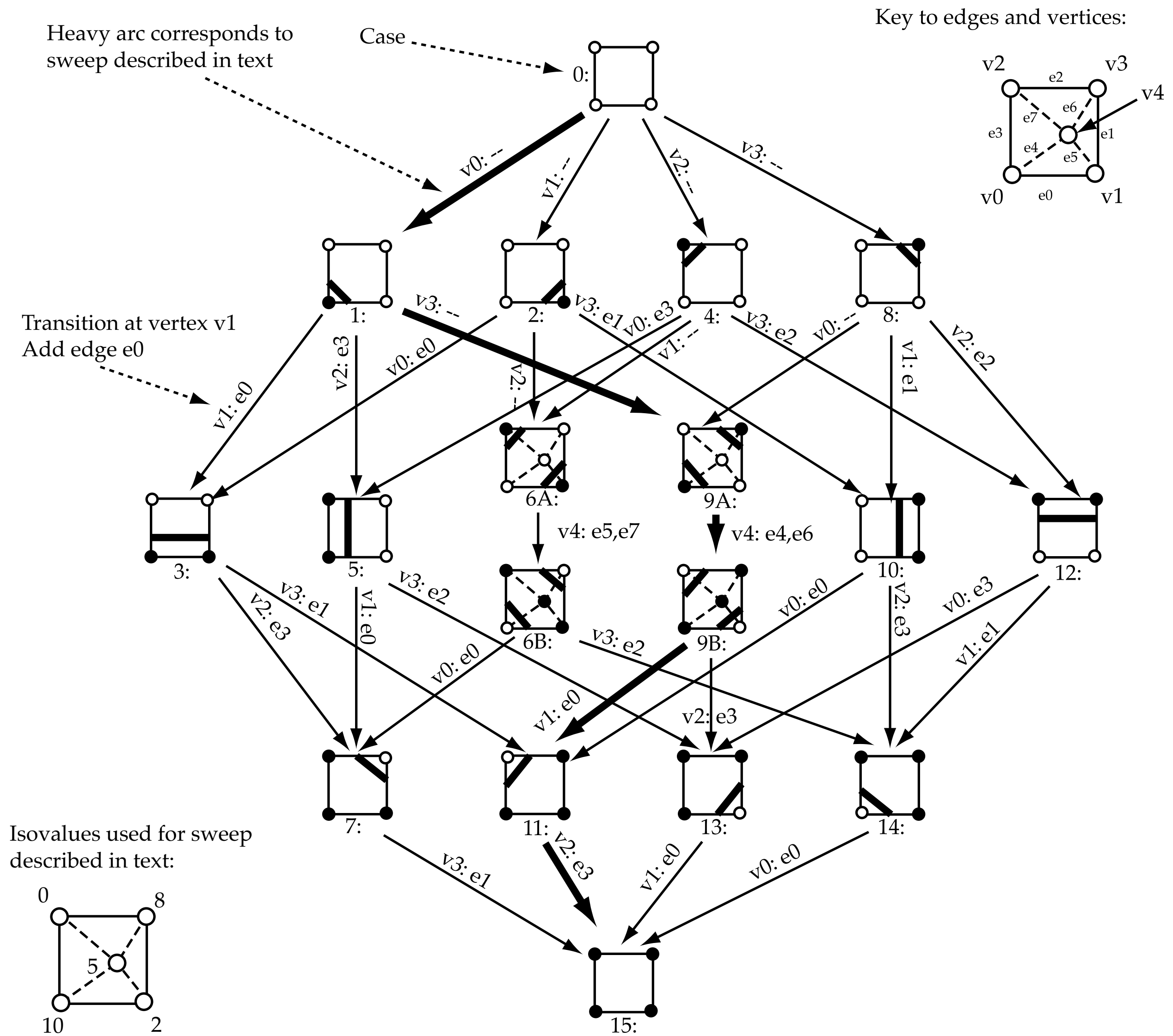


Figure 12.1: Finite State Machine To Compute Join Graph for Bilinear Interpolant. //In this figure, each state corresponds to one of the tessellation cases. As we sweep from high isovalues to low in any given cell, we sweep past each potential critical point in turn. Each sweep past a potential critical point corresponds to a transition in the finite state machine. Any edges listed along the transition are added to the join graph for the cell to represent the changes in connectivity at the potential critical point past which we swept.

connected component, then we will compute the join tree directly for the cell.

12.5 An Example: the Bilinear Interpolant

Conveniently, these lookup tables can be displayed as finite state machines, in which each state corresponds to a single tessellation case, and each transition corresponds to a sweep of the isosurface past a a single potential critical point. As an example, Figure 12.1 As an example of this approach, Figure 12.1 shows the join graph lookup table for the bilinear interpolant as a finite state machine. In this example, assume that

the isovalues are those shown in the lower left hand corner of the diagram. The potential critical points will be the set of vertices of the cell plus the body saddle, if present.

Initially, we start in state 0. Since $v0$ has the highest isovalue of the potential critical points, it is processed first, corresponding to a transition from state 0 to state 1. Since $v0$ is a local maximum in this cell, no edge is added to the join graph during the transition. The next potential critical point to be processed is $v3$ at an isovalue of 8, corresponding to a transition to state $9A$. Again, no edges are added to the join graph.

Once we reach state $9A$, we note that the isovalue 5 of the saddle $v4$ must always be higher than the remaining two isovalues 2, 0 of the vertices $v1$, $v2$ in order for the saddle to be in the interior of the cell, so the only possible transition is to state $9B$, corresponding to a sweep past the saddle. Since the saddle connects to both $v0$ and $v3$, we add edges $e4$ and $e6$ to the join graph.

Next, we sweep past $v1$ with an isovalue of 2, adding edge $e0$ as we make a transition to state 11. Finally, we sweep past $v2$, adding edge $e3$ as we make a transition to state 15 and terminate.

In general, since the transitions represent sweeps in a particular direction through the cell, we know that the finite state machine must always halt, and that the set of transitions passes through all possible tessellations to be extracted from that cell. We also note that the finite state machine for the split graph uses the same transitions pointing in the opposite direction. Each transition will use the same vertex in both directions, but the edge to be added may change. For example, during the split sweep, a transition from state 15 to state 7 will not add any edges to the split graph.

In the next section, we see how to extend this approach to tessellation cases that do not correspond strictly to any particular interpolant.

12.6 Contour Trees for Tessellation Cases

Recall from Chapter 6 that we defined the contour tree using concepts derived from Morse Theory, which studies properties of continuously differentiable functions. But, in practice, we rely on the assumptions stated in Part III to ensure that there are only a finite number of critical points. We then apply a combinatorial algorithm to a finite superset of the critical points, relying on the fact that the topology can change at most a finite number of times.

Once we have recognized that we are not studying all possible properties of an arbitrary continuously differentiable function, but rather the finite number of changes possible to contours extracted from a mesh defined over a finite set of points, we can start extending the contour tree algorithms to contours defined in other ways.

Recall also that Boyell & Ruston [BR63] defined the contour tree in terms of the nesting relationship of a finite number of explicit polygonal contours. This can trivially be extended to a finite number of classes of equivalent contours, such as the cases used for tessellation.

As a concrete example, let us consider the contours extracted in square cells by the cases shown in Figure 12.2, after the usual symmetric reductions. Note that this is not the same as the bilinear interpolant discussed in the previous section: in particular, the presence of saddles is ignored.

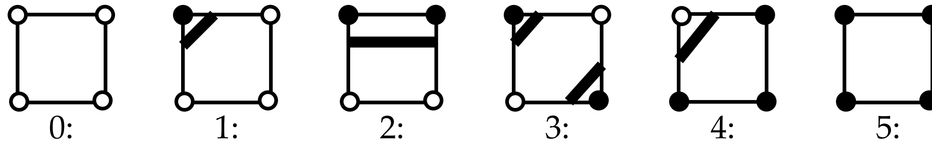
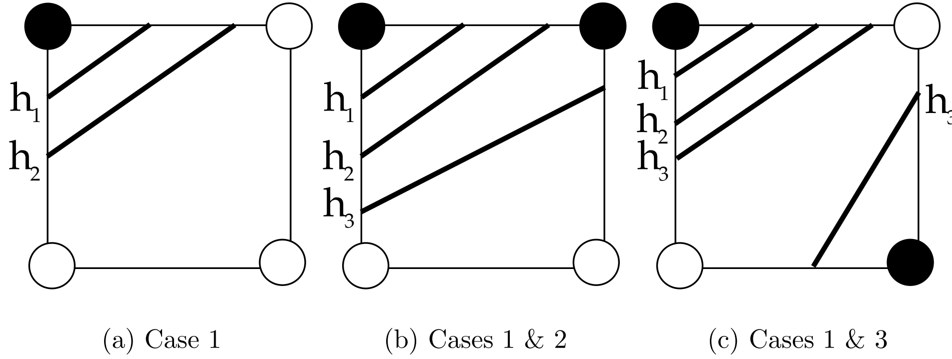


Figure 12.2: Sample Cases for Tessellation in Two Dimensions. As usual, black vertices are above the contour, white vertices below.



(a) Case 1 (b) Cases 1 & 2 (c) Cases 1 & 3

Figure 12.3: Nesting of Some Tessellation Cases.
(a) shows two contours of case 1, one inside the other.
(b) shows two contours of case 1 nested inside a contour of case 2.
(c) shows two contours of case 1 nested inside one of the contours of case 3.

Notice that each case nests inside the previous case in the following sense. In Figure 12.3(a), as we reduce the isovalue, the endpoints of the contour move linearly away from the black vertex. Thus for any $h_1 > h_2$ that falls into case 1, the contour at h_1 will nest inside the contour at h_2. As the endpoints slide along the edges, the next vertex passed by the contour may be adjacent or diagonally opposite. In the first case, we transition to case 2, as for the bilinear interpolant. But, for any $h_3 < h_2$ for which h_2 uses case 1 and h_3 uses case 2, the contour at h_2 will nest inside the contour at h_3, as shown in Figure 12.3(b). Similarly, Figure 12.3(c) shows a transition to case 3. Here, case 1 at h_1 will nest inside case 3 at $h_3 < h_1$. Finally, case 3 nests inside case 4 by a similar argument.

Also notice that we can discuss sweeping a contour through the cell in terms of transition from one case to another. This allows us to describe the join graph (or split graph) concisely by using finite state machines. Figure 12.4 shows the finite state machine that computes the join graph using the tessellation cases of Figure 12.2, complete with a sample sweep through the cell. For this sweep, we assume that the vertices are ordered $v2, v1, v3, v0$, i.e. that $f(v2) > f(v1) > f(v3) > f(v0)$.

During this sweep, we transition from case 0 to case 4 when we sweep past $v2$, then to case 6, case 14, and finally case 15. In the first step, we add no edges, as $v2$ is a local maximum for the cell. Similarly, in the second step, we add no edges. In the third step, from 6 to 14, we add two edges because $v3$ is connected to both $v2$ and $v1$. Finally, we add $e0$ to connect $v0$ to the other vertices.

Notice also that not all points in the cell have contours passing through them. Figure 12.5 shows an example of a square in which case 1, 3 and 4 are swept through: the area marked 1 in the figure is the area through which contours of case 1 sweep, and so on. Since the contours never sweep through the white region in the centre, it is clear that we are not working with contours defined by a function f that is defined for the entire cell. Yet we can still make statements about the connectivity of the contours generated, and

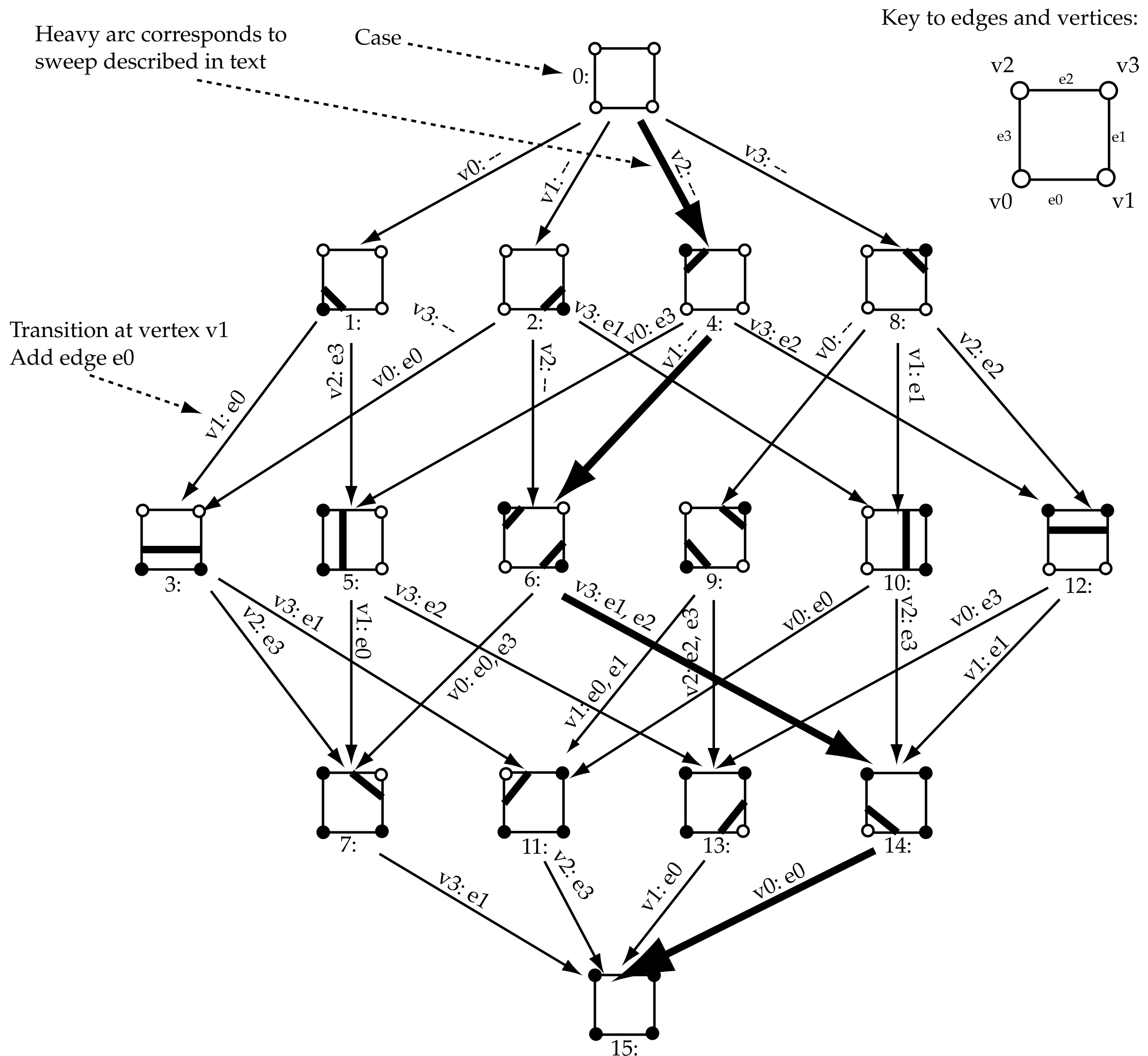


Figure 12.4: Finite State Machine To Compute Join Graph for Simple 2-D Tessellation Cases. //As with Figure 12.1, each state corresponds to one of the tessellation cases, and each transition to a sweep past a potential critical point. In this case, the potential critical points are the vertices of the cell.

the geometric properties of those contours.

In terms of theoretical justification, we can, for example, achieve this by rejecting the Morse theoretic definition of the contour tree, and redefining it in terms solely of nesting properties. Alternately, we can define f for the white region to be equal to the isovalue of the top right corner which was swept past, then use perturbation or other techniques to ensure that critical points are unique, as required for the Morse theory. Either approach will lead to the same result: a contour tree that correctly represents the nesting of the contours that we can extract.

We will use this later, in Chapter 13, to generate join and split graphs for the Marching Cubes cases of Montani, Scateni & Scopigno [MSS94a].

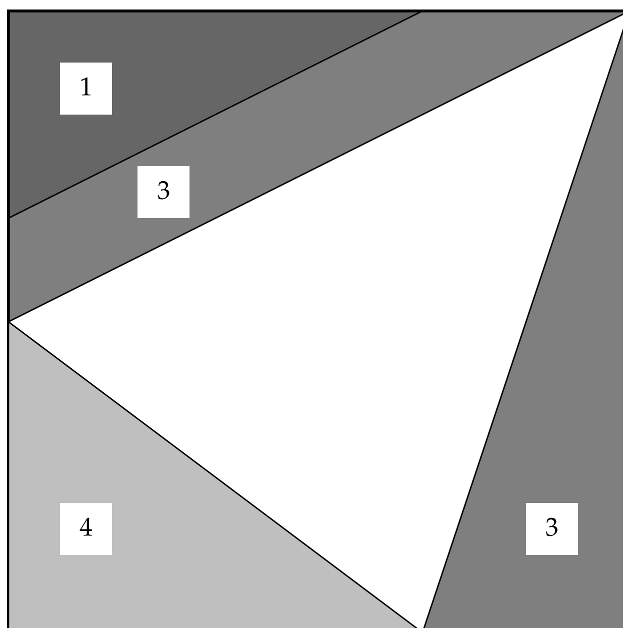


Figure 12.5: Areas Swept Through By Contours. In this figure, the white region in the centre does not belong to any contour, whereas the shaded regions belong to contours in particular cases.

12.7 Generating Isosurface Seeds

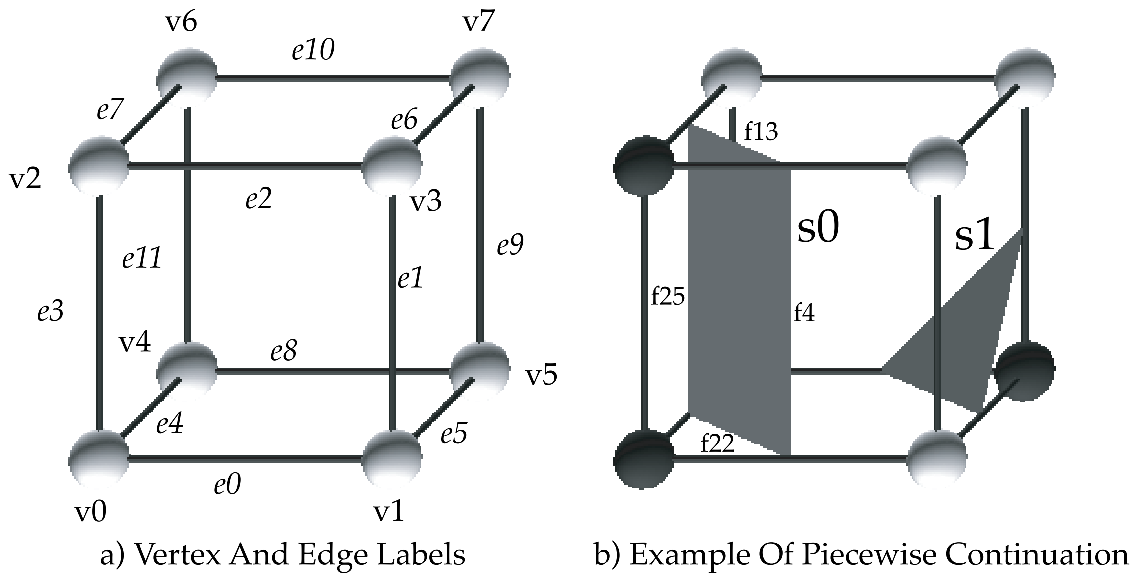
One of the major uses of the contour tree is for extracting isosurfaces. For this, we need to define how to extract seeds, and how to use the continuation method to extract individual contours.

To find seeds, we use the path seeds described in Chapter 8. Recall that a path seed is an edge by which we leave a critical point to generate an f-monotone path that intersects the desired contour. Each path seed is detected during the join or split sweep. Since edges in the join graph represent sets of ascending paths from joins, we follow edges in the join graph when we need an ascending path. Similarly, we follow edges in the split graph when we need descending paths.

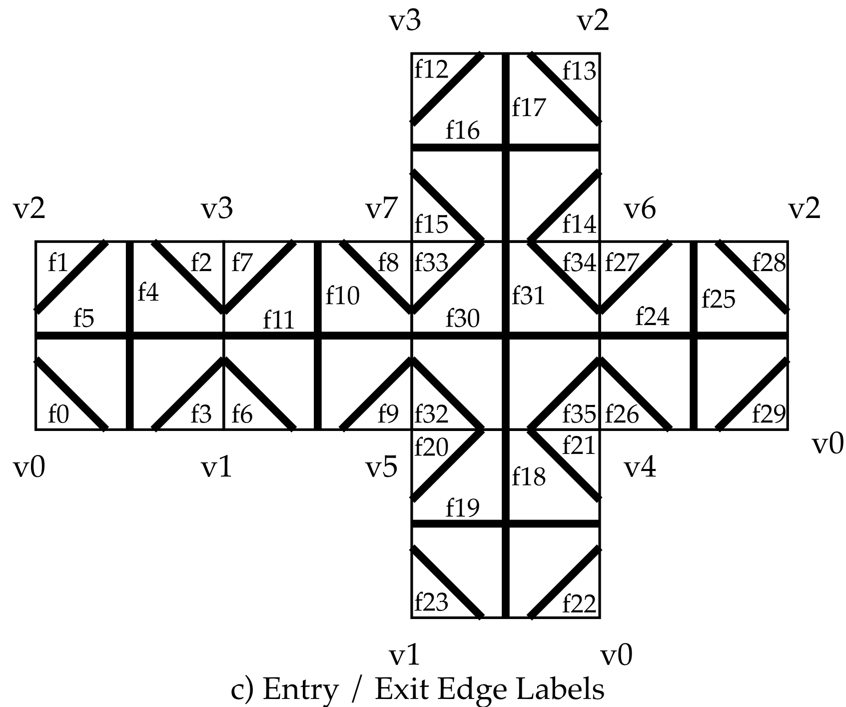
12.8 Piecewise Continuation

Once we have found a seed edge for each contour, we extract the contour itself using the continuation method of Wyvill, McPheeters & Wyvill [WMW86a]. For simplicial meshes, this method is well-behaved, as at most one contour surface intersects any given tetrahedron. For arbitrary meshes, it is possible for multiple contour surfaces to intersect a given cell. For example, in Figure 12.6(b), two surfaces, labelled $s0$ and $s1$, intersect the cell. When the continuation method enters a cell of type 6, it queues up adjacent cells, and extracts both surfaces simultaneously.

To deal with individual surfaces one at a time, we must generate seeds for individual surfaces, and modify continuation so that it follows one surface at a time. Seeds for individual surfaces are easy to obtain if we are using seed edges rather than seed cells. We know that seed edges correspond to monotone paths in the function, so it is only possible for a seed edge to intersect one surface at a given isovalue.



a) Vertex And Edge Labels b) Example Of Piecewise Continuation



c) Entry / Exit Edge Labels

Figure 12.6: Labels For Piecewise Continuation, With Example

To follow only one surface at a time, we assign a label to each surface in the cell, as shown in Figure 5.1. When we queue up adjacent cells, we queue up only those to which our current surface connects. We call this method *piecewise continuation*, and show it in Algorithm 12.3.

In general, we do not know which surface we connect to in the adjacent cell, but we do know that a triangle in the current cell shares an edge with a triangle in that cell. We call this edge an *entry edge* or *exit edge*, depending on whether we are entering or exiting the cell across the edge. Thus, instead of entering the cell, as is implicit in the continuation method given above, we enter the cell *across a given edge*. These entry and exit edges can connect any pair of edges on a given face, and are labelled $f0 - f35$ in Figure 12.6(c).

For example, in Figure 12.6(b), suppose we enter the cell across edge $f4$. We use a lookup table to determine which surface $f4$ belongs to: in this case, it belongs to $s0$. After extracting $s0$, we exit the cell across edges $f4, f13, f22, \& f25$ (possibly excepting $f4$, our entry edge). Each exit edge is also an entry edge in an adjacent cell: for example, $f13$ leads to the adjacent cell above this cell. From Figure 12.6(c), the exit edge $f13$ becomes the entry edge $f22$ in that cell. Thus, to continue along this surface only, we take each exit edge, convert it to an entry edge in an adjacent cell, and queue up both adjacent cell and entry edge.

As usual for Marching Cubes, we define the relevant lookup tables only for the base cases shown in Figure 5.1, and use rotations to generate all the other cases. The code to generate these tables is included in Appendix A, and is available at `http://www.cs.ubc.ca/\~hcarr/downloads/mcCases.c`. For completeness, the code implementing Algorithm 12.3 is also included in Appendix B.

```
Input  : A seed edge e
         An isovalue H
Output : The unique contour surface that intersects e

1  Find a cell that includes e.
2  Compute the Marching Cubes case for that cell.
3  Look up the surface the seed edge intersects in this cell.
4  Choose an exit edge for the surface as the entry edge.
5  Queue the seed cell and entry edge
6  while the queue is non-empty do
7     Remove a cell and edge from the queue
8     Compute the Marching Cubes case
9     Find the surface to which the edge belongs
10    if surface is marked then
          Goto 6
      else
11        Extract surface
12        Mark surface
      end
13    for each exit edge belonging to the surface do
14        Convert exit edge to entry edge in neighbour
15        Queue neighbour and entry edge
      end
   end
```

Algorithm 12.3: The Piecewise Continuation Method of Isosurface Extraction

12.9 Summary

This chapter started by identifying the fact that data is rarely presented on a simplicial mesh, cubic grids being far more common for visualization purposes. This chapter has extended the work of Pascucci & Cole-McLaughlin [PCM02] to provide a general framework for performing contour tree computations on arbitrary meshes, using any interpolant, and in any dimension.

Instead of analysing the individual interpolant by hand, as in Pascucci & Cole-McLaughlin [PCM02], we have seen how to compute suitable *join and split graphs* using a lookup table extracted automatically from any set of tessellation cases.

Because we constructed the join and split graphs from the tessellation cases instead of from static

analysis of the interpolant, we were then able to extend contour tree computations to sets of tessellation cases which do not correspond to well-defined interpolants, provided that the tessellation cases satisfy a criterion based on the nesting relationship of individual tessellated contours.

We will see in Chapter 14 that this latter result is useful where contours are being extracted on an *ad hoc* basis, such as the Marching Cubes cases of Montani, Scateni & Scopigno [MSS94a], or where analysis of a given interpolant is difficult or incomplete.

Even if the analysis of the interpolant is incorrect, we will still be able to generate contour trees that are consistent with the surfaces that are actually extracted, provided that the tessellation cases satisfy a simple nesting condition. That this is an advantage is apparent from the sequence of papers analysing the trilinear interpolant [LC87, Dür88, WvG90, NH91, Mat94, MSS94a, Nat94, Che95, CGMS00, CMS01, LB03]. These papers all attempt to generate isosurfaces from the trilinear interpolant, but, except for [Che95, LB03], have various errors in them. Most of them, however, generate isosurfaces that satisfy the nesting condition, allowing us to use the contour tree to extract and manipulate individual contours.

Even now that there is a correct analysis of the trilinear interpolant [Che95, LB03], it may still be desirable to use a simpler tessellator for speed, for consistency with other tools, or simplicity of coding. Again, the ability to compute the contour tree for the contours actually extracted is a contribution made by this chapter.

In addition, in order to facilitate using non-simplicial meshes with the flexible isosurface interface of Chapter 9, we have shown how to extend *path seeds* and *minmal seed sets* for isosurface extraction, and defined *piecewise continuation* to extract the individual isosurfaces.

In the next two chapters, we will see how to apply the techniques from this chapter to the most common meshes: (hyper)-cubic meshes with either multilinear interpolation, or the non-interpolating Marching (Hyper)-Cubes tessellation cases.

Chapter 13

Contour Trees for Multilinear Interpolants

In the previous chapter, we showed how to extend the contour tree algorithm to any arbitrary mesh and interpolant, using *join and split graphs.* This chapter deals principally with computing join and split graphs for bilinear, trilinear, and higher-dimensional multilinear interpolants.

Section 13.1 starts with the example of the bilinear interpolant, and shows several different methods of computing suitable join and split graphs, then discusses local spatial measures. Section 13.2 repeats the exercise for three dimensions, and Section 13.3 looks at higher dimensions.

13.1 Join and Split Graphs for the Bilinear Interpolant

There are several ways we might consider to construct join and split graphs for the bilinear interpolant. We can, for example, construct a lookup table, as described in Algorithm 12.2. Alternately, we can classify all possible contour trees, join trees or split trees, for the bilinear interpolant, and construct a lookup table for the entire cell. The merit of doing so is that we get the entire contour tree for each cell at once, rather than one edge at a time of a join graph. Static analysis of the interpolant, however, is required to compute the set of possible contour trees, for example, by using Algorithm 12.2 to compute the classes of join trees.

Other methods include subdividing each cell into smaller cells with simpler topology, such as simplices, or building a join graph by adding edges to the edges of the mesh. In this section, we will see examples of how to apply each of these methods to the bilinear interpolant.

Bilinear Lookup Tables: The first way to construct join and split graphs is to use the lookup arrays computed in Section 12.4. This captures exactly the same surface as will be generated when a contour is extracted, because we use exactly the same tests for determining which case we are in as we do for the contour extraction.

In the previous chapter, we used the bilinear interpolant as an example in Figure 12.1, showing the

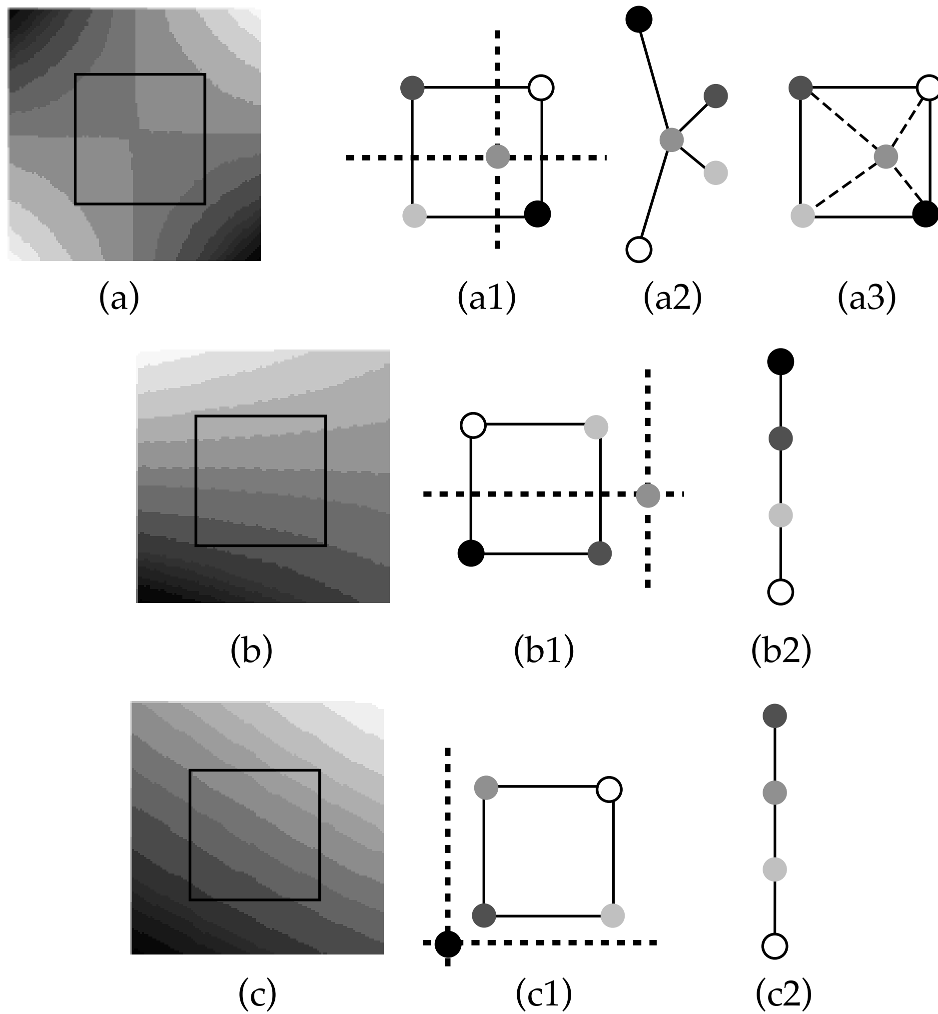


Figure 13.1: Possible Contour Trees for Bilinear Interpolant. For the bilinear interpolant, the saddle is located at the intersection of the horizontal and vertical asymptotes. Depending on whether the saddle is inside the cell (a), off to one side (b), or diagonally off the corner of the cell (c), the contour tree can only by of the form (a2), (b2) or (c2). (a3) shows how to subdivide a cell containing a saddle to obtain simplicial subcells with equivalent topology.

lookup table in the form of a finite state machine.

Cell Classification and Lookup: Figure 13.1 classifies all possible types of cell for the bilinear interpolant, and creates a lookup table of all possible contour trees. Assuming that the vertices have distinct isovalues, Nielson & Hamann [NH91] showed that the contours are a family of hyperbolae sharing horizontal and vertical asymptotes passing through a saddle. For the bilinear interpolant, the contour tree will then depend on whether this saddle is inside the cell, as shown in case (a), to one side, as shown in case (b), or off in a diagonal direction, as shown in case (c).

For the bilinear interpolant, this analysis is straightforward. For the trilinear interpolant, Pascucci & Cole-McLaughlin [PCM02] have shown how to classify the possible join and split trees. Their method, however, relies on a large lookup table to deal with all the possible orderings of potential critical points. For higher dimensions, this approach is likely to prove intractable, since the number of possible situations increases rapidly, and since it becomes much more difficult to analyse the contours by inspection, as we have

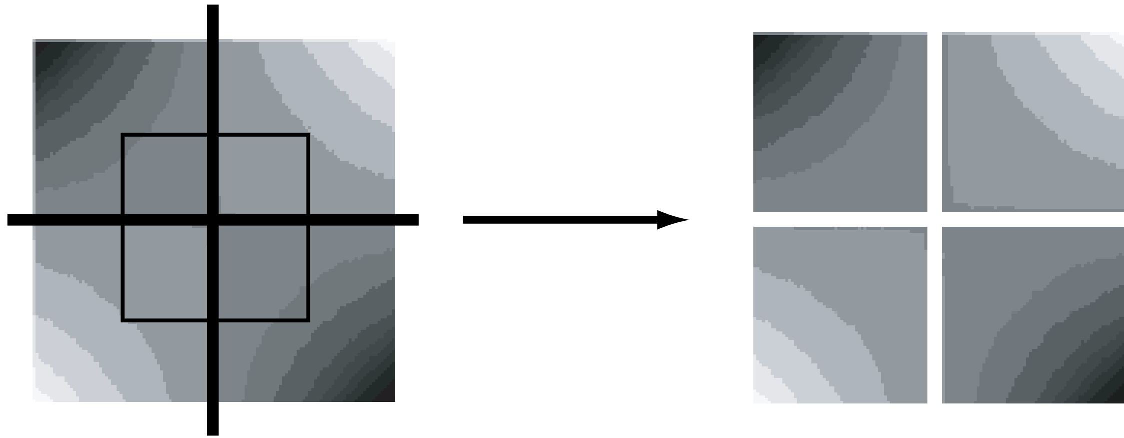


Figure 13.2: Subdividing a Bilinear Cell With Vertical and Horizontal Asymptotes. We can split a cell along the horizontal and vertical asymptotes through the saddle to obtain four subcells, each of which is guaranteed to have simpler topology.

done here.

Cell Subdivision: The third way to construct a topology graph is to divide the cell into four by adding the saddle point, the vertical and horizontal asymptotes through the saddle, and vertices at the edges, as shown in Figure 13.2. This divides the cell into four axis-aligned subcells. Conveniently, the bilinear interpolant in each subcell is the restriction to the subcell of the bilinear interpolant in the entire cell. Thus, any critical point in the subcell must be a critical point in the original cell. But we know that the saddle is the only critical point of the original cell. Since this saddle occurs at a vertex of each of the subcells, the subcells cannot have critical points in their interior, and fall into case (c) of Figure 13.1. Thus, the subcells have simple join, split, and contour trees, which can be computed from the union of the join, and split trees for the subcells.

In two dimensions, however, it is simpler to classify all cells and use a lookup, as described in Section 13.1. However, in higher dimensions, this method becomes important, as we will see below.

Cell Augmentation: The fourth way to construct a topology graph is to insert the saddle point, if present, and connect it to all four corners, as shown in Figure 13.1(a3): this explicitly represents the contour trees for the four subcells. If the saddle point is not present, we have case (b) and (c) of Figure 13.1. Such cells are not inspected further: instead, the edges of the cells are used, as they sufficiently represent the cell's topology.

We can also add a vertex in the middle of cells of the types shown in Figure 13.1 (b) and (c). Since the topology of these cells is simple, this causes no problems. But this allows us to have a consistent set of edges in every cell: the edges of the cell, plus connections from the inserted vertex to all four corners. For the bilinear interpolant, this gives us a division of the square into triangular cells, in each of which the topology is equivalen to that generated by the barycentric interpolant.

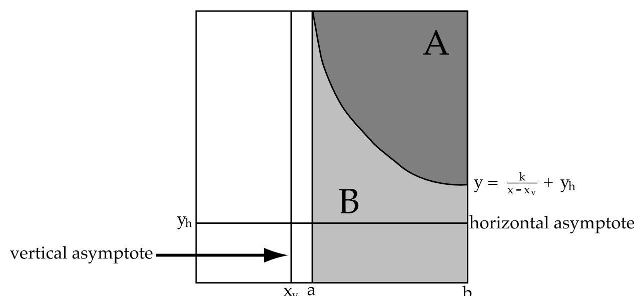


Figure 13.3: Computing Area in a Bilinear Cell. Since bilinear interpolants generate hyperbolic contours, computing local spatial measures involves logarithmic terms.

13.1.1 Bilinear Spatial Measures

Once we know how to compute the topology for the bilinear interpolant, we still need to know how to compute geometric properties such as area enclosed by the contour. Recall that Nielsons & Hamann [NH91] showed that the contours of a bilinear cell are hyperbolae.

In Figure 13.3, we show a hyperbolic contour for which we wish to compute the area of region A. Computing this area requires computing area B under the hyperbolic curve $y = \frac{k}{x-x_v} + y_h$, where x_v is the x coordinate of the vertical asymptote, y_h is the y coordinate of the horizontal asymptote, and k is a scaling constant. The area of B is then given by the integral:

$$\int_a^b \frac{k}{x - x_v} + y_h \, dx = k \ln |b - x_v| - k \ln |a - x_v| + (b - a) y_h \tag{13.1}$$

The appearance of a logarithmic term is problematic, because the method described in Chapter 10 for computing local geometric properties tracks the sum of terms for each cell intersecting a sweep line, collapsing the sum into a single polynomial of $O(1)$ size.

We can sum logarithmic terms using the identity:

$$\log(a) + \log(b) = \log(ab) \tag{13.2}$$

For an extended sequence of log terms, however, we will have to worry about the numerical precision of performing a large number of multiplications. Moreover, for trilinear and higher interpolants, it is not easy to define the integral, and we expect further difficulties in exact computation of properties.

13.2 Trilinear Topology Graphs

We have just discussed four methods to compute bilinear join and split graphs. We can use the same methods to compute trilinear join and split graphs.

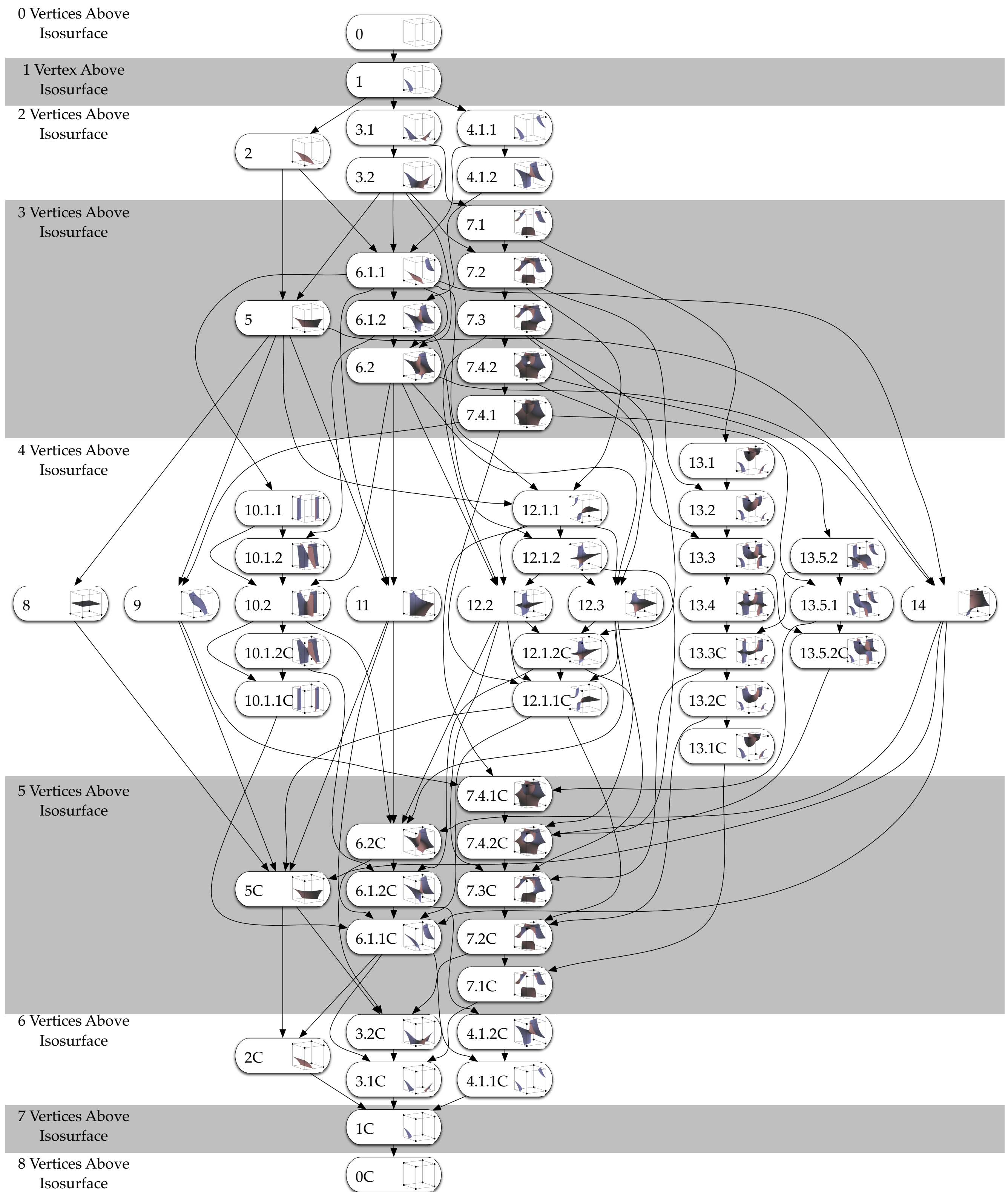


Figure 13.4: Finite State Machine for Join Graph of the Trilinear Interpolant. As with Figure 12.1, each state corresponds to one of the tessellation cases, and each transition to a sweep past a potential critical point corresponds to a transition in the finite state machine. The states have been reduced by symmetry, and are numbered following Chernyaev [Che95].

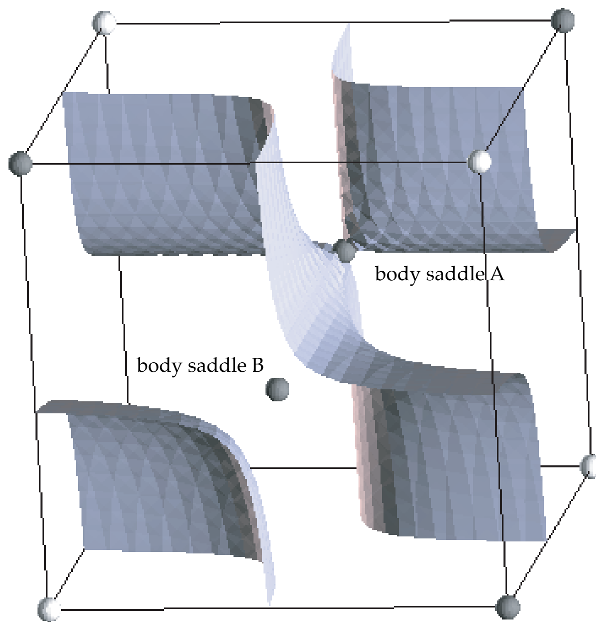


Figure 13.5: A Trilinear Cell with Two Body Saddles. An example of a trilinear interpolant with two body saddles. Note how the contour through body saddle A prevents the existence of any monotone path from one vertex of the mesh to body saddle B.

Trilinear Lookup Tables: We know from Chapter 12 that we can construct lookup tables for join and split graphs automatically once we have an accurate set of tessellation cases. Chernyaev [Che95] and Lopes & Brodlie [LB03] have reported accurate tessellation cases for the trilinear interpolant. It follows that we can use the algorithms in Chapter 12 to compute suitable lookup tables.

Figure 13.4 shows the finite state machine representing the lookup table

Cell Classification and Lookup: Alternately, we can compute lookup tables that give the join and split trees for every possible ordering of the potential critical points of the trilinear interpolant, as Pascucci & Cole-McLaughlin [PCM02] have done.

Cell Subdivision: As we did for the bilinear case, we can also subdivide each cell into smaller cells, by passing planes perpendicular to each axis through each body saddle point. The trilinear interpolant in each subcell is simply the restriction to that subcell of the trilinear interpolant in the original cell. As with the bilinear case, we know that the body saddles of the original cell are now at the vertices of the subcell, so no subcell has a body saddle in its interior. This drastically simplifies the cases: we can then use the Asymptotic Decider of Nielson & Hamann [NH91] to adjudicate face saddles. This approach, of dividing the cells into topologically simpler subcells, is essentially the same as the *Dividing Cubes* approach pioneered by Cline et al. [CLL$^+$88] for isosurface rendering, and used by Cignoni et al. [CGMS00] and Lopes & Brodlie [LB03].

Cell Augmentation We can augment a trilinear cell with additional edges to produce a join or split graph. Each face with a face saddle must have the face saddle connected to all four vertices of the face. If there is one body saddle, we add edges to connect it to each vertex and each face saddle of the cell. If

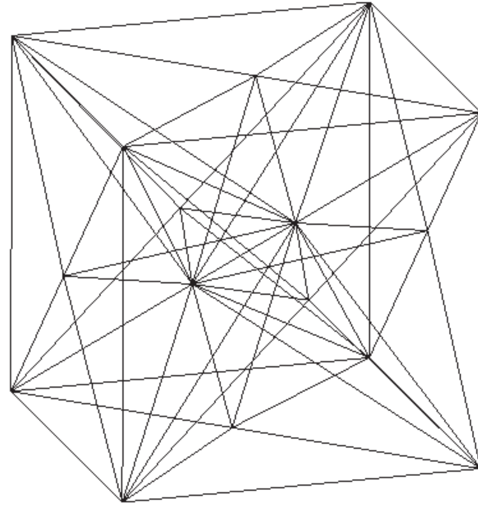


Figure 13.6: Augmenting a Trilinear Cell with Two Body Saddles. The graph shown here, with edges connecting both body saddles to each other, to the face saddles, and to all but one of the vertices, is a sufficient join and split graph for computing the contour tree.

there are two body saddles, they will be roughly aligned along a major diagonal of the cell, as shown in Figure 13.5. The isosurface passing through body saddle A precludes a monotone path between body saddle B and the vertex opposite body saddle B on the major diagonal. Monotone paths, however, exist between each body saddle and each vertex or face saddle, and between the two body saddles.

Remembering that we construct our topology graph to represent all possible monotone paths, we take:

1. the edges of the cube,
2. edges from each face saddle to the corners of the face,
3. edges from each body saddle to each vertex of the cube *except* the vertex diagonally opposite along the main diagonal,
4. edges from each body saddle to each face saddle, and
5. an edge between the two body saddles

Thus, depending on how many saddle points a cell has, we may need to add as many as 51 additional edges, as shown in Figure 13.6. This is unsatisfactory, as it adds great complexity for relatively little practical gain.

13.2.1 Trilinear Local Spatial Measures

Computing bilinear spatial measures already involved logarithmic terms. Trilinear spatial measures can be expected to be even worse: for example, defining the integral to compute the area enclosed by the

surfaces in Figure 13.5 will be extremely complex. We regard the exact computation of these measures as computationally impractical for large data sets.

We expect, however, that for large data sets, the approximate local spatial measures described in Section 10.8 will suffice for most practical purposes.

13.2.2 Summary

For the trilinear interpolant, then, there are several practical ways of computing the contour tree. At present, we recommend using the cell classification and lookup oracle described by Pascucci & Cole-McLaughlin [PCM02], because it is known to work, and is relatively straightforward. Alternately, the finite state machine approach could be used, although we have yet to construct the relevant finite state machine.

No matter which method is used, however, dealing with the trilinear interpolant is inherently complex. Since we have mostly worked with noisy acquired data, we have chosen to avoid the additional complexity, since we cannot rely on any sub-voxel details, and have adopted the approach described in Chapter 14, instead.

13.3 Higher Dimensions

As we continue to higher dimensions, it is clear that the multilinear interpolation will retain the properties we have discussed for bilinear and trilinear interpolants, but that constructing suitable cases, lookup tables, and finite state machines will become more and more complex.

A hypercube in four dimensions has 8 faces, each of which is a cube in three dimensions. Over each such cube, the quadrilinear interpolant reduces to the trilinear interpolant. We know that each cube can have two body saddles and six face saddles. This implies that the hypercube can have as many as $8 \times (2+6) = 64$ critical points shared with adjacent hypercubes.

Moreover, the body of the hypercube may have additional saddles: since the two-dimensional case has 1 and the three-dimensional case has 2, the four dimensional case is expected to have at least 3. Adding in the 16 vertices of the hypercube, we find that we may need to inspect as many as $64 + 3 + 16 = 83$ values.

Of the general approaches described above, the most practical appear to be the lookup table defined in Section 12.4 and cell subdivision. The lookup table relies on having established suitable isocontouring cases, which have not yet been worked out. Moreover, given the large number of papers published [LC87, Dür88, WvG90, NH91, Mat94, MSS94a, Nat94, Che95, CGMS00, CMS01, LB03] that it took to analyse the trilinear interpolant and its cases correctly, we expect that it will be some time before this is practical.

The cell subdivision approach, however, is feasible, as it relies on static analysis of simpler cases. If we can construct a set of cases for the quadrilinear interpolant, assuming that there are no saddle points whatsoever, then we can compute the contour tree correctly. To do so, we compute any body saddles of the hypercube, and pass axis-aligned hyperplanes through each one. This then reduces the problem to one involving simple cases in the body of the hypercube, and the trilinear interpolant in each cubic "face" of the hypercube, for which we know we can find the contour tree. However, this approach is likely to be

moderately complex to implement.

Because of this, and because of the difficulty of tessellating the quadrilinear interpolant, we believe that an approach based on tessellation cases such as Marching Hypercubes [BWC00] is likely to be the most practical approach to computing contour trees for higher dimensions.

13.4 Summary

In this chapter, we have described how to determine suitable topology graphs for computing the contour tree over multilinearly interpolated cells in two and three dimensions. These solutions generally involve exhaustive description of all possible surface configurations, breaking the cell down into less complex subcells, or constructing a graph directly that captures all classes of monotone paths.

The drawback to all of these solutions is that they are complex. In two dimensions, the complexity is manageable, as it involves at most one additional vertex to process, and a small number of graph edges. In three dimensions, however, the complexity rapidly increases. And, in four or higher dimensions, the complexity passes the bounds of the practical.

For this reason, the next chapter will describe a simpler approach: working with the cases of the well-known *Marching Cubes* [LC87].

Chapter 14

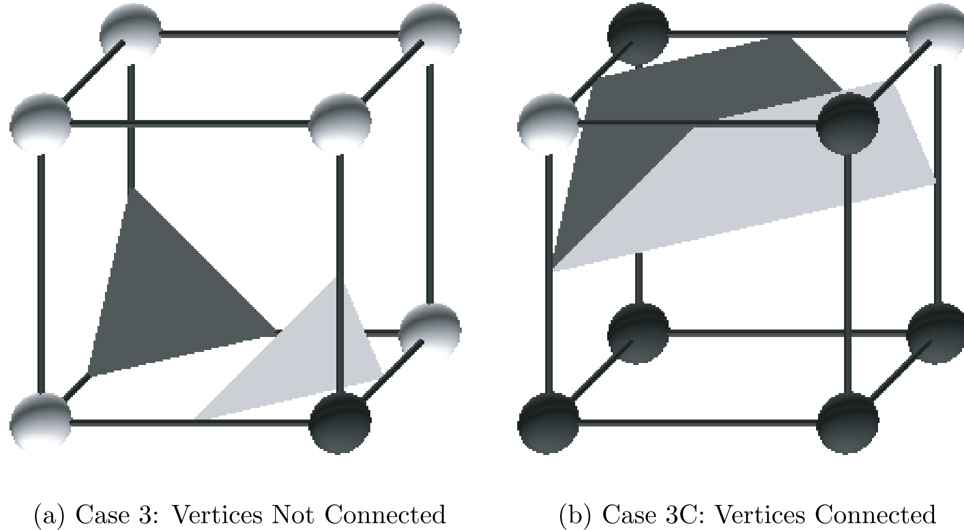
Contour Trees for Marching Cubes

As we saw in the previous chapter, computing the topology for the trilinear and higher-dimensional multilinear interpolants is a complex task. Multilinear interpolation, however, is not always the method of choice, for a variety of reasons.

1. *Complexity*: generating the topologically correct trilinear surface is difficult to do accurately, as can be seen by the sequence of papers refining Marching Cubes [CGMS00, Che95, Dür88, LB03, LC87, Mat94, MSS94a, Nat94, NH91]. In higher dimensions, this complexity will only increase.
2. *Cost*: computing correct saddles adds additional time and expense even when extracting a single isosurface.
3. *Quantization*: where data is stored as an 8- or 16- bit integer, with implicit error of ± 0.5, it is unwise at best to infer topological details at computed isovalues such as 12.345.
4. *Noise*: experimental volumetric data is subject to noise, which has two effects:
   (a) Fine topological detail cannot be assumed to be accurate.
   (b) Noise drastically increases the size of the contour tree, so that most of the fine detail will not be of interest to the user in any event.
5. *Instability*: since the position of the saddle(s) in a cube is a degree-3 polynomial, accuracy is a concern, as contour tree algorithms are vulnerable to numerical errors, since they rely on a correct sorting order.
6. *Local Spatial Measures*: we saw in the previous chapter that multilinear interpolants make it difficult to compute local spatial measures accurately.

This is not to say that trilinear interpolation is always inappropriate. In particular, it is preferred for simulation data, where noise can be assumed to be non-existent, and fine topological detail often indicates important events and / or problems with the simulation.

However, for noisy, experimentally sampled data, we claim that the complexity attendant on correct trilinear interpolation is unnecessary. Instead, in this chapter, we show how to compute contour trees directly for the commonly used Marching Cubes cases of Lorenson & Cline [LC87], as corrected by Montani, Scateni & Scopigno [MSS94a] and for the related Marching Hypercubes of Bhaniramka, Wenger, & Crawfis [BWC00].



(a) Case 3: Vertices Not Connected (b) Case 3C: Vertices Connected

Figure 14.1: Marching Cubes Case 3 and its Converse Case 3C

14.1 Join and Split Graphs for Marching Cubes

As we noted in Chapter 12, we can compute the contour tree correctly even if we have a set of tessellating surfaces that do not define a continuous interpolant. The best-known such set of surfaces are the Marching Cubes cases of Lorenson & Cline [LC87], which we discussed in Chapter 5. In this section we describe simple join and split graphs that allow easy and efficient computation of the contour tree, not for these cases, but for the corrected cases of Montani, Scateni & Scopigno [MSS94a].

For these cases, there is a simple and convenient shortcut for computing join and split graphs. Consider the cases shown in Figure 14.1. In Figure 14.1(a), two vertices (in black) are "above" the contour, and six (in white) are "below" the contour. In Figure 14.1(b), the reverse is true. The rule adopted by Montani, Scateni & Scopigno [MSS94a] is that two black vertices that are diagonally opposed on a face, they are assumed to be inside distinct surfaces, unless connected by edges of the cube. Thus, two black vertices are *never* connected only by a face diagonal. Moreover, two black vertices are assumed never to be connected only by a body diagonal of the cube.

Thus, in a situation such as shown in Figure 14.1(a), as we sweep past the second of the two black vertices, we do not need to concern ourselves with the possibility of connecting to a component by a face or body diagonal. And in Figure 14.1(b), as we sweep past the second of the two black vertices on the upper face of the cell, we do not need the face diagonal, as the black vertices are already connected by edges of the cube.

It then follows that the edges of the cube suffice to represent the nesting of these surfaces during the join sweep: i.e. that the edges of the cube form a join graph.

For the split tree, we wish to track the white vertices. In Figure 14.1(b), we can see that we need the opposite rule: two white vertices are always connected by a face diagonal. Body diagonals, however, are not used. In Figure 14.1(a), when we sweep past the second of the two white vertices on the lower face, adding the face diagonal is redundant, but not incorrect.

This gives a simple way to find a split graph: always add all edges and face diagonals. Again, we ignore major diagonals, as vertices are never connected only by a major diagonal of the cube. We end up with simple join and split graphs for Marching Cubes:

1. The join graph uses the edges of the cubic mesh
2. The split graph uses the edges and face diagonals.

If we wish to compare this to using the lookup table in Section 12.4, we observe that the lookup table algorithm will add $n-1$ edges as it sweeps past the n vertices of a cell. For a cube, this means that we add 7 edges total per cube. Since each vertex participates in 8 cubes on average, and there are 8 vertices per cube, it follows that we add 7 edges per vertex in each of the join and split sweeps. In comparison, the join graph we have just defined adds 6 edges per vertex, while the split graph adds $6+24=30$ edges. Notionally, then, this approach adds twice as many edges as the lookup table would, in exchange for simpler implementation and cheaper edge construction.

We saw in Chapter 13 that the lookup table could be expressed as a finite state machine. In Figure 14.2, we show the finite state machine for the join graph for the Marching Cubes cases of Montani, Scateni & Scopigno [MSS94a]. The finite state machine for the split graph is identical, except that transitions run in the opposite direction, and the edges to be added to the join graph are different.

14.2 Local Spatial Measures for Marching Cubes

As with simplicial meshes, computing local spatial measures for the Marching Cubes cases of Montani, Scateni & Scopigno [MSS94a] is relatively straightforward, albeit a little messy. In each of these cases, the upstart and downstart regions are bounded by the extracted isosurface, and include vertices of the cube. Thus, each upstart or downstart region is a polyhedral solid, whose vertices are either vertices of the cube or vertices interpolated along the edges of the cube. Since the interpolation is linear, these latter vertices are linear combinations of the original vertices of the cube, parameterized by h, the isovalue.

To compute geometric measures such as volume, we decompose each such polyhedral solid into tetrahedra, using only the vertices of the polyhedron. For each tetrahedron, we compute the desired measure, then combine the results.

As an example, consider Figure 14.3, which shows the decomposition of an upstart region in case 2. This upstart region has 6 vertices, four of which are linear combinations of vertices of the cube. We decompose this polyhedral region into the tetrahedra v0v02v04v15, v0v1v02v15 and v1v02v15v13, compute the geometric measure for each tetrahedron separately, then recombine them.

Alternately, as we did in Chapter 11, we can use approximate local spatial measures.

14.3 Summary

In this chapter, we observed that the trilinear interpolant is often overkill, especially for noisy, experimentally acquired data. For this reason, we have shown how to compute suitable join and split graphs for the more

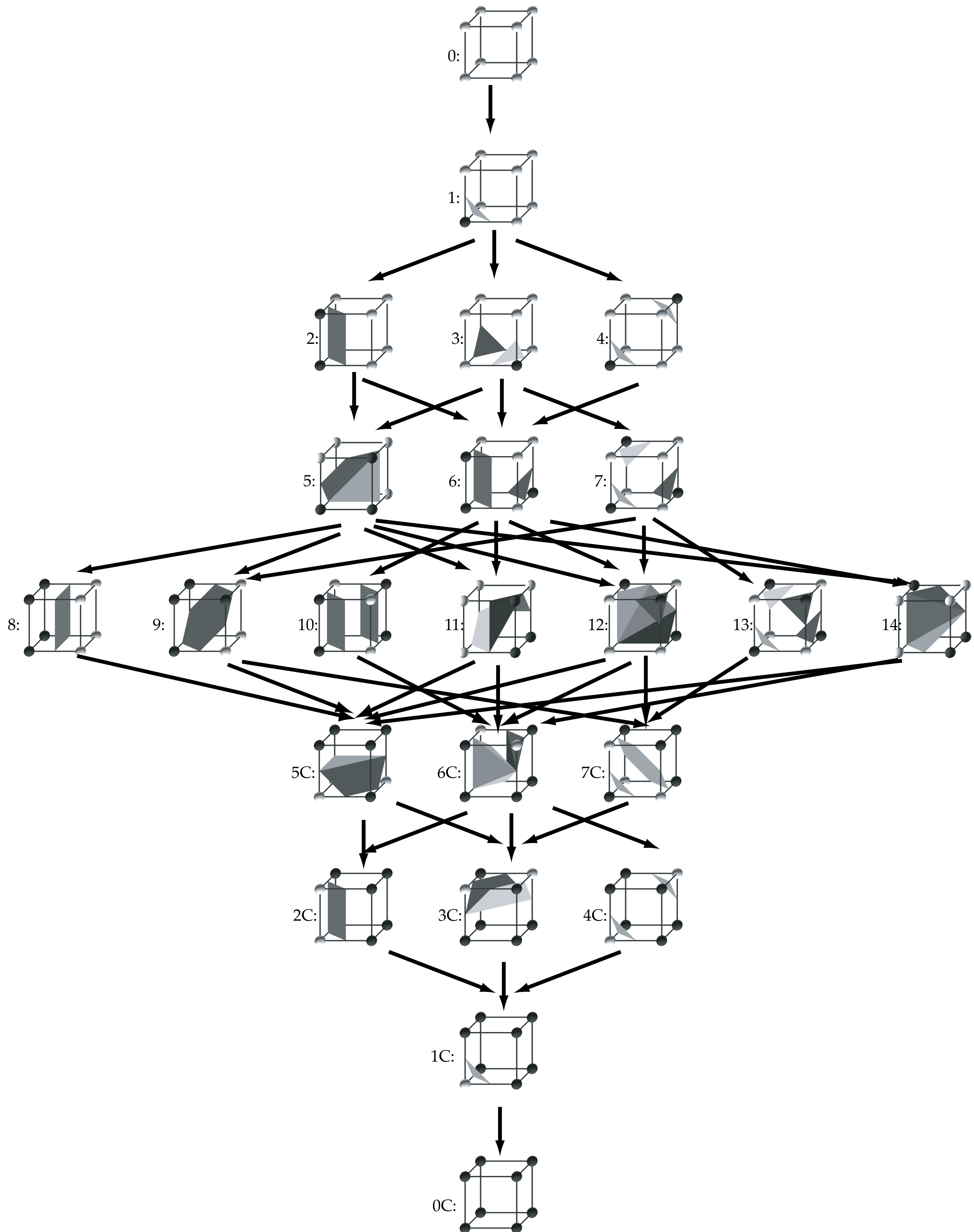


Figure 14.2: Finite State Machine for Marching Cubes cases of Montani, Scateni, & Scopigno [MSS94a]. As with Figure 13.4, we show only the basic cases after symmetry reduction. This finite state machine can be obtained by merging states from Figure 13.4: for example, case 3 here can be obtained by merging cases 3.1 and 3.2 in the trilinear case.

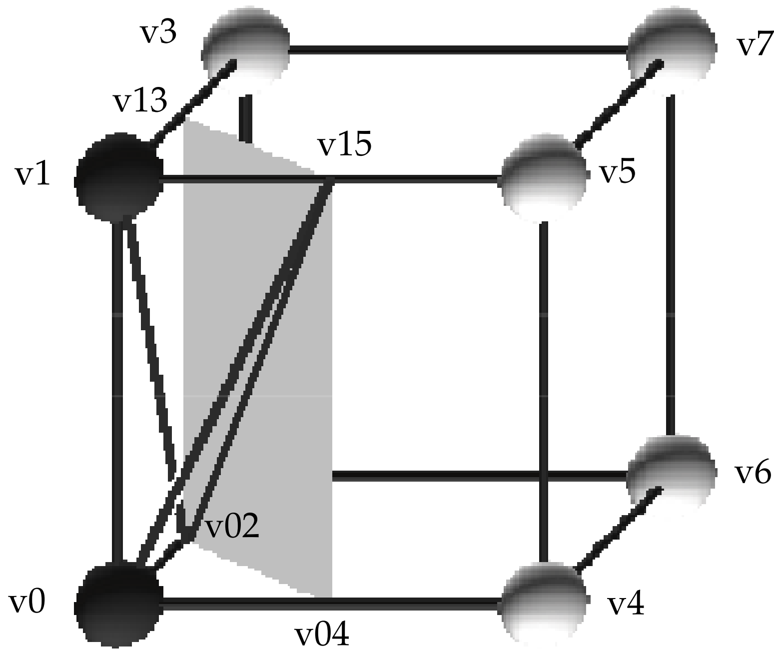


Figure 14.3: Computing Geometric Measures for Case 2 of Marching Cubes. In this figure, vertices of the form v04 are linear combinations of two vertices, in this case v0 and v4. The upstart region for the contour shown is the polyhedron defined by v0, v1, v02, v04, v13, and v15. Geometric properties are computed by decomposing this region into the tetrahedra v0v02v04v15, v0v1v02v15 and v1v02v15v13.

commonly used tessellation cases of Marching Cubes [LC87, MSS94a]. Moreover, we have contributed a simple shortcut for computing join and split graphs for these cases.

Chapter 15

Perturbation

In this chapter, we show how to relax our assumption that no two vertices of the mesh share the same isovalue.

In practice, this assumption is rarely satisfied. The input data is usually stored as either 8 or 16-bit integers, and usually has at least $64 \times 64 \times 64 = 262,144$ samples. Since 8- and 16- bit integers only have 256 and $65,536$ distinct values respectively, it is not possible to have a data set of this size without repeated values. Even if we restrict the assumption to critical points, for a contour tree of size $65,537$ or greater, the assumption must be false.

We had this assumption in the first place because the Morse theoretic underpinnings assume that changes in topology occur at critical points, not critical regions. Or, in simpler words, that there are no flat spots, no plains, no plateaus, and that there is always a single point as a saddle.

We handle this problem by applying symbolic perturbation. This is equivalent to tilting the data so that flat spots become ever so slightly slanted. Edelsbrunner & Mücke [EM90] have formalized this idea as *Simulation of Simplicity*. To simulate simplicity, we add a different ϵ value at each sample, choosing the ϵ values so that the largest of them is still smaller than the smallest difference between two existing non-unique isovalued samples. This guarantees that no two samples have exactly the same isovalue.

Since this is difficult to arrange using fixed-precision input data, it is customary to perturb symbolically instead of literally. Thus, when two isovalues are compared, we use the usual comparison unless the two isovalues are identical, in which case we compute the ϵ perturbations and compare them instead. Provided that we have a finite number of samples, this works out nicely, as we can always choose sufficiently small ϵ values.

In practice, we can take advantage of the fact that the samples are stored in a regular array in the computer's memory. Choose a single sufficiently small δ such that δ multiplied by the largest legal memory address is still smaller than any difference between two unique isovalues. Then, for a sample stored at memory address M, let the corresponding ϵ value be: $\epsilon(M) = \delta \times M$. Using this, it is easy to test which of two samples is greater after perturbing. Algorithm 15.1 shows the algorithm for the test. We must be careful using this, because algorithms such as quicksort can enter an infinite loop if the samples are moved in memory. We therefore assume that the samples always remain fixed in memory.

Input : Pointers $d1$ and $d2$ to two isovalues stored at fixed memory addresses.
Output : Result of comparing the isovalues at $d1$ and $d2$ using symbolic perturbation.

```
1  if content of memory address d1 is less than content of memory address d2 then
2  |  Return "less than";
3  else if content of memory address d1 is greater than content of memory address d2 then
4  |  Return "greater than";
5  else if address d1 is less than address d2 then
6  |  Return "less than";
7  else if address d1 is greater than address d2 then
8  |  Return "greater than";
9  else
10 |  Return "equal";
   end
```

Algorithm 15.1: Comparing Vertices with Memory-Address Perturbation

Thus, adding perturbation to the algorithm to compute the contour tree is quite straightforward. Using the contour tree thereafter, however, becomes more complex due to the presence of the perturbation, as we will see in the following sections.

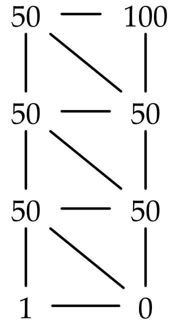
15.1 Removing Perturbation

The major difficulty with perturbation is that it adds topological detail to the contour tree that should not be present. Consider the mesh shown in Figure 15.1(a). Six of the vertices have the same isovalue 50. If we apply perturbation as described in the previous section, we add the symbolic values $\alpha < \beta < \gamma < \delta < \epsilon < \zeta < \eta < \theta$ to get Figure 15.1(b). Using this as input, we get the contour tree shown in Figure 15.1(d), when what we want to work with is the tree shown in Figure 15.1(c). Most of the superarcs in Figure 15.1(d) are between pairs of vertices whose isovalues differ only by the imposed perturbation. For convenience, let us refer to these superarcs as ϵ-superarcs.

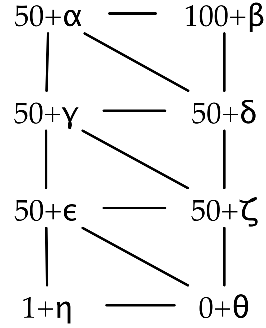
To give an idea of the consequences of the perturbation, assume that we wish to track the evolution of the contour at isovalue 0 as the isovalue is increased. In the unperturbed mesh in Figure 15.1(a), we will see a sequence of contours rising eventually to 100. If we follow through the perturbed tree in Figure 15.1(d), however, we get a sequence of contours rising only to $50 + \zeta$. This can be avoided if we modify our evolution-tracking so that, instead of strictly ascending, we are permitted to take downwards arcs, provided that the descent is only the result of perturbation. But this requires us to have two definitions of "less than": one for sorting and generating the tree, the other for ascending / descending in the tree.

A preferable solution is to remove the perturbation by collapsing any ϵ-superarc, as discussed in Section 11.3. In this instance, edge collapses are permissible, as the objection stated in Section 11.3 does not apply. Recall that the objection stated was that edge-collapses resulted in misrepresenting the number of contours at isovalues spanned by the superarc. For ϵ-superarcs, we have the reverse problem. The ϵ-superarc represents contours that do not actually exist. Thus, edge-collapsing the ϵ-superarc actually removes the effects of the perturbation.

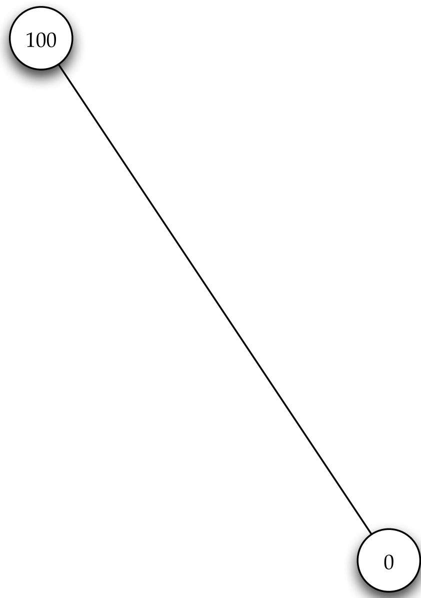
Figure 15.2 shows the results of removing the perturbation in this way. Notice that, after removing



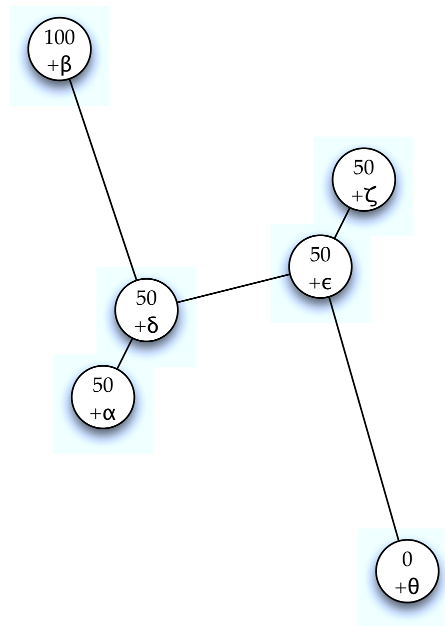
(a) Unperturbed Mesh



(b) Perturbed Mesh



(c) Unperturbed Contour Tree



(d) Perturbed Contour Tree

Figure 15.1: Effects of Perturbation on A Small Mesh

all of the ϵ-superarcs, we were left with Figure 15.2(d), in which $50 + \epsilon$ had become a regular point, so we performed a vertex-collapse at $50 + \epsilon$ to remove it.

Note that the flat region is in fact the *sweep region* of the ϵ-superarc, as defined in Chapter 6, and removing the ϵ-superarcs is equivalent to simplifying those edges. Since we will never extract an isosurface exactly at the isovalue of these vertices, removing ϵ-superarcs will not affect extracted isosurfaces.

In Algorithm 15.2, we show a naïve algorithm to remove perturbation from the contour tree. Unfortunately, Step 3 is bounded by $O(t)$, since either u or v may be of arbitrary degree, giving a total runtime bound of $O(t^2)$. Instead, we substitute Algorithm 15.3.

Input : A contour tree $C = (V, E)$ with edges due to perturbation
Output : C with the perturbation edges removed

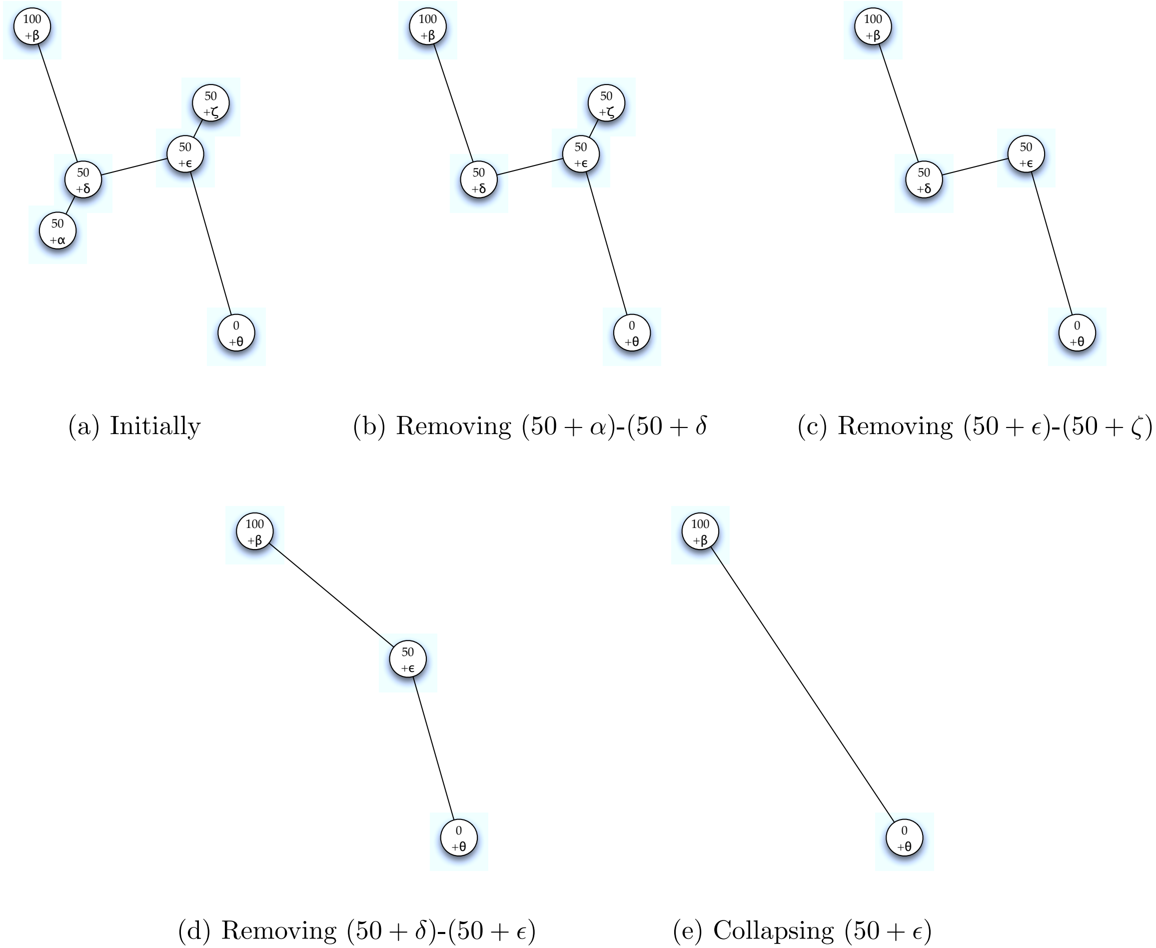
```
1  for each edge e = (u, v) in E do
2  |  if f(u) = f(v) then
3  |  |  Identify u and v (transfer all edges from v to u)
   |  end
   end
4  for each vertex v in V do
5  |  if v is now a regular point then
6  |  |  Perform vertex collapse at v
   |  end
   end
```

Algorithm 15.2: Naïve Removal of Perturbation from Contour Tree

Input : A contour tree $C = (V, E)$ with edges due to perturbation
Output : C with the perturbation edges removed

```
1  for each vertex v in V do
2  |  if wasChecked(v) then
3  |  |  Skip v.
   |  end
4  |  Set wasChecked(v)
5  |  for each arc e = (u, v) incident to v do
6  |  |  if f(u) = f(v) then
7  |  |  |  Delete e
8  |  |  |  if not wasChecked(u) then
9  |  |  |  |  Add u to queue
   |  |  |  end
   |  |  end
   |  end
10 |  while queue not empty do
11 |  |  pop u from queue
12 |  |  set wasChecked(u)
13 |  |  for each edge e = (u, w) incident to u do
14 |  |  |  if f(u) = f(w) then
15 |  |  |  |  Delete e
16 |  |  |  |  if not wasChecked(u) then
17 |  |  |  |  |  Add u to queue
   |  |  |  |  end
   |  |  |  else
18 |  |  |  |  Transfer e from u to v
   |  |  |  end
   |  |  end
19 |  |  Delete u
   |  end
20 |  if v is now a regular point then
21 |  |  Perform vertex collapse at v
   |  end
   end
```

Algorithm 15.3: Improved Removal of Perturbation from Contour Tree



(a) Initially (b) Removing $(50+\alpha)$-$(50+\delta$ (c) Removing $(50+\epsilon)$-$(50+\zeta)$

(d) Removing $(50+\delta)$-$(50+\epsilon)$ (e) Collapsing $(50+\epsilon)$

Figure 15.2: Removing Perturbation from a Contour Tree

In this improved algorithm, we perform a local breadth-first search at each vertex v to determine which other nodes are linked to it purely by ϵ-height edges. Each edge incident to v is checked for ϵ height: if so, the *other* end is added to a queue of vertices known to be ϵ-linked to v, and the edge is deleted from the tree. As each vertex u is removed from the queue, its incident arcs are similarly checked. Edges of ϵ height are deleted, with their other ends being added to the queue, while non-ϵ edges are transferred to v. Since edges are always transferred *to* v, each edge is transferred at most once.

After all vertices equivalent to v have been identified and had their incident arcs transferred to v, if v has become a regular point, it is collapsed. This can happen, for example, at a join vertex with two upwards arcs and one downwards, if one of the upwards arcs is a leaf arc of ϵ height.

Each edge in the tree is checked at most twice for ϵ height (once from each end, in the case of non-ϵ edges), and transferred between vertices at most once. Thus, this improved version is bounded by $O(t)$.

Removing perturbation in this way, however, makes it more difficult to use the path seeds described in Section 8.2, as we will see in the next section.

15.2 Perturbation and Path Seeds

In the previous section, we discussed removing perturbation with the simplification from Chapter 11. If, however, we wish to use path seeds to extract contours, as described in Section 8.2, we will have to work with the perturbed contour tree.

To see why this is true, consider the mesh in Figure 15.3(a), which shows all ascending edges as arrows. If we remove the perturbation in the contour tree, we end up with path seeds only at the ends of the superarc $100 - 0$. Suppose that we use the path seed at 0, and wish to extract the contour at 75. If we attempt to ascend from 0 in the mesh to find a suitable seed edge, we immediately hit the flat spot at 50. To leave the flat spot and continue ascending, we first have to cross the flat spot, as shown by the heavy edges in Figure 15.3(a).

But, without knowing which direction to go, we will potentially need to explore the entire flat spot to find the correct "exit". Since this flat spot can be arbitrarily large, this imposes significant penalties on the path seed method. This can be alleviated in part if we mark every vertex in a flat spot with the direction of ascent or descent. This, however, requires additional processing.

Alternately, and more simply, we can leave the perturbation in. In Figure 15.3(b), we show an example of this, where the perturbation actually helps to find a suitable path to ascend. However, this is not always the case, as we can get stuck in a local maximum caused by the perturbation, as shown in Figure 15.3(c).

In practice, since we are using hierarchical path seeds to deal with simplification, the easiest solution is to leave the perturbation in and use hierarchical path seeds. Thus, in Figure 15.3(d), we use a path seed from $50 + \delta$ for the isovalue range $50 - 100$, and a path seed from 0 for the isovalue range $0 - 50$.

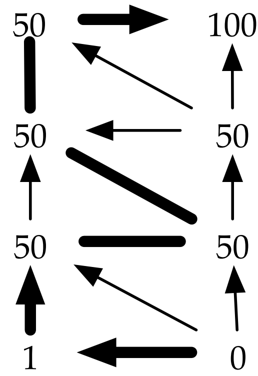
15.3 Perturbation and Non-Simplicial Contour Trees

Once we start working with non-simplicial meshes and interpolants other than the barycentric interpolant, we need to be cautious with perturbation. Consider for example the trilinear interpolant discussed in Chapter 13. We compute the isovalue of body saddles based on the values of the corners of the cell. Without going into detail, it is possible that one of the body saddles may have the same isovalue as not one, but two vertices of the cell. Since the isovalue of the saddle depends on the isovalues of those vertices, careful attention must be paid to ensure that the perturbation does not disturb the correct sorting order.

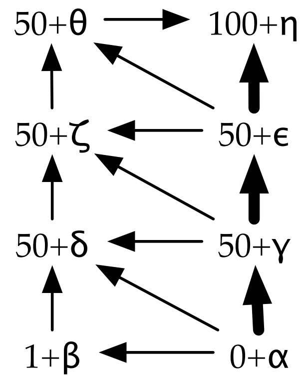
We do not examine this further in this thesis, as we have chosen to implement the Marching Cubes approach discussed in Chapter 14, instead. Since there are no derived saddles, the perturbation scheme discussed above works quite smoothly.

15.4 Summary

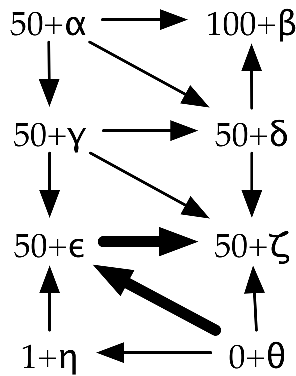
In this chapter, we have discussed using symbolic perturbation to satisfy the assumption in Part III that no two vertices have the same isovalue. The principal contribution we have made with respect to perturbation



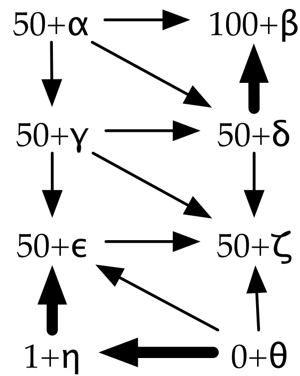
(a) Flat Spots



(b) An Easy Case



(c) Getting Stuck



(d) Hierarchical Seeds

Figure 15.3: Effects of Perturbation on Path Seeds. Note how perturbation can make path seeds work, as in (b), or prevent them from working by creating artificial flat spots, as in (c). Using hierarchical path seeds, as in (d), avoids this problem.

is to show that removing the perturbation is a form of simplification. It follows from this that we can address the side-effects of perturbation by using the hierarchical path seeds discussed in Chapter 8, provided that we account for the perturbation when following path seeds through the mesh. Moreover, we have observed that we have chosen to use approximate local spatial measures, which are unaffected by the perturbation.

Part V

Results and Conclusions

In previous Parts, we showed how the contour tree can be used to unify topological and geometric information for exploratory manipulation of individual isosurfaces in scalar fields, and how to extend existing contour tree algorithms to non-simplicial meshes.

In this part, we give some numerical results in Chapter 16 on the performance of these algorithms, and on the effectiveness of topological simplification. We then review the contributions in this thesis and state some conclusions in Chapter 17, and speculate on future research directions in Chapter 18.

Chapter 16

Results

In this chapter, we provide some performance results for various algorithms described in this thesis. Unless otherwise specified, timing results were obtained using a Macintosh G4 computer with dual 1GHz processors, 1 GB of main RAM, and a 64 MB GeForce 4 video card, running Mac OS X 10.1.5 or 10.2.6. Although the machine has dual processors, none of the code written was parallelized.

This chapter starts with a section discussing the types of data tested and their sources in Section 16.1, followed by sections discussing individual results for different parts of this thesis.

Thus, Section 16.2 gives some indication of the performance of the contour tree computation algorithms, including removing perturbation and performing simplification. Section 16.3 then compares the cost of computing the contour tree using Marching Cubes and the minimal 5-fold simplicial subdivision described in [CMS01]. Section 16.4 gives some results for isosurface extraction with path seeds. Finally, Section 16.5 shows some images produced using the flexible isosurface interface with simplification.

16.1 Sources of Data

For testing the flexible isosurface, we used data from a number of sources, shown in Table 16.1. These data sets range from analytical data (Marschner/Lobb), through numerical simulations and molecular simulations, to MRI, CT and X-ray data. The first group consists of molecular datasets provided to us by Alan Ableson and Janice Glasgow of Queen's University, Kingston. The second group consists of data sets obtained from the Volvis website (www.volvis.org), made available by various researchers. Finally, the data sets in the last group are of miscellaneous origins, principally from UNC-CH.

Broadly speaking, we expect the noise to increase in the order stated: analytical data is expected to be accurate to machine precision. Numerical simulations can be expected to be largely free of noise, but there may be some numerical instability or round-off error. The molecular data are simulated X-ray crystallographic data, based on resolved proteins from the Protein Data Bank (www.pdb.org). Although simulated, the simulation includes water molecules presumed to be near the proteins in question, and the data is somewhat noisy. Finally, experimentally acquired data can be expected to have a significant amount of noise.

Name	Type	Size	Courtesy of:
f368.0.6	Molecular	$35 \times 17 \times 51$	Ableson / Glasgow
3gap.0.8	Molecular	$29 \times 60 \times 131$	Ableson / Glasgow
1dog.0.8	Molecular	$72 \times 64 \times 60$	Ableson / Glasgow
Aneurysm	X-ray	$256 \times 256 \times 256$	Philips Research
Bonsai	CT scan	$256 \times 256 \times 256$	S. Roettger
Boston Teapot	CT scan	$256 \times 256 \times 178$	Terarecon Inc. et al.
Engine	CT scan	$256 \times 256 \times 128$	General Electric
Foot	X-ray	$256 \times 256 \times 256$	Philips Research
Fuel	Simulation	$64 \times 64 \times 64$	German Research Council
Hydrogen Atom	Simulation	$128 \times 128 \times 128$	German Research Council
Lobster	CT scan	$301 \times 324 \times 56$	SUNY Stony Brook
Marschner/Lobb	Analytical	$41 \times 41 \times 41$	Marschner / Lobb [ML94]
Neghip	Simulation	$64 \times 64 \times 64$	SUNY Stony Brook
Nucleon	Simulation	$41 \times 41 \times 41$	German Research Council
Shockwave	Analytical	$64 \times 64 \times 512$	Unknown
Silicium	Simulation	$98 \times 34 \times 34$	SUNY Stony Brook
Skull	X-ray	$256 \times 256 \times 256$	Siemens Medical Systems
Statue Leg	CT scan	$341 \times 341 \times 93$	BAM, Germany
Hipiph	Simulation	$64 \times 64 \times 64$	Scripps Clinic, La Jolla
3dknee	MRI	$256 \times 256 \times 127$	Siemens / UNC-CH
3dhead	MRI	$256 \times 256 \times 109$	Siemens / UNC-CH
Teddybear	MRI	$128 \times 128 \times 61$	Univ. of Erlangen-Nuremberg
Rat	MRI	$240 \times 256 \times 256$	Whole Frog Project, LBL

Table 16.1: Characteristics and Sources of Test Data

Data Set	Size (n)	Contour Tree Construction (s)	Tree Size (t) (Superarcs)	Ratio (t/n)
Nucleon	68,921	0.34	676	0.98%
Marschner-Lobb	68,921	0.32	1,620	2.35%
Silicium	113,288	0.56	399	0.35%
Fuel	262,144	1.16	227	0.09%
Neghip	262,144	1.26	2,063	0.79%
Shockwave	2,097,152	9.26	722	0.03%
Hydrogen	2,097,152	9.58	12,729	0.61%
Lobster	5,461,344	32.97	348,369	6.38%
Engine	8,388,608	54.84	509,397	6.08%
Statue	10,814,133	56.84	448,815	4.15%
Teapot	11,665,408	61.26	87,388	0.75%
Aneurysm	16,777,216	81.40	65,626	0.39%
Bonsai	16,777,216	92.30	187,437	1.12%
Foot	16,777,216	114.77	866,482	5.16%
Skull	16,777,216	422.46	2,190,897	13.06%

Table 16.2: Construction Times for the Contour Tree. Times do not include perturbation removal. Note that the clean data sets tend to have small contour trees relative to the input data size, while noisy acquired data sets tend to have quite large contour trees.

16.2 Contour Tree Computation

In Table 16.2, we show some results for computing the contour tree, using the Marching Cubes method from Chapter 14 and simplification using Local Spatial Measures.

Data Set	Size (n)	Tree Size (t) (Superarcs)	Perturbation Removal (s)	Unperturbed Size (t')	Ratio (t'/n)	Ratio (t'/t)
Nucleon	68,921	676	< 0.01	49	0.07%	7.24%
Marschner-Lobb	68,921	1,620	< 0.01	695	1.01%	42.90%
Silicium	113,288	399	< 0.01	225	0.35%	56.40%
Fuel	262,144	227	< 0.01	129	0.04%	56.83%
Neghip	262,144	2,063	< 0.01	248	0.09%	12.02%
Shockwave	2,097,152	722	< 0.01	31	< 0.01%	4.29%
Hydrogen	2,097,152	12,729	0.01	8	< 0.01%	0.06%
Lobster	5,461,344	348,369	0.47	77,349	1.42%	22.20%
Engine	8,388,608	509,397	0.68	134,642	1.61%	26.43%
Statue	10,814,133	448,815	0.57	120,668	0.01%	26.89%
Teapot	11,665,408	87,388	0.10	20,777	0.18%	23.78%
Aneurysm	16,777,216	65,626	0.08	36,667	0.22%	55.87%
Bonsai	16,777,216	187,437	0.24	82,876	0.50%	44.22%
Foot	16,777,216	866,482	1.31	508,854	3.03%	58.73%
Skull	16,777,216	2,190,897	11.15	931,348	5.55%	42.51%

Table 16.3: Sizes of Contour Trees after Perturbation Removal, with Times Required. Although the ratio varies, perturbation generally accounts for 1/4 to 1/2 of the edges in the contour tree. In extreme cases, such as the hydrogen dataset, the underlying simulation is exceptionally clean, with topologically complex flat regions which require many perturbed superarcs to represent.

We draw two conclusions from these results. First, the size of the contour tree is generally related to the size of the input data, but tends to be smaller for analytical and simulated data sets than for experimentally acquired data sets. Given our comments in Chapter 11, this is hardly surprising. Second, we see that these computations are well within the range of a modern computer for data sets with up to 16 million sample point. For larger data sets, batch processing and storage of the contour tree may be required, but is still feasible.

Table 16.3 shows that the cost associated with the symbolic perturbation of Chapter 15 is significant. Removing the perturbation from the tree typically halves the number of superarcs in the contour tree. One reason for this may be that the data sets are 8-bit, so quantization is likely to introduce many flat spots that cause the contour tree to have extra edges. However, we can also see that the cost of stripping out the perturbation is a small fraction of the cost of computing the contour tree in the first place.

Finally, we show the time required to simplify the contour tree using approximate local spatial measures in Table 16.4. In this case, we use hypervolume as our spatial measure, by analogy to the volume measure we used to collapse the 2-D sample data set in Figure 11.13. We approximate this by taking the product of the height of the edge and the volume of the isosurface at the saddle point. The volume is turn approximated by the number of mesh vertices as in Section 10.8.

Compared to the cost of constructing the contour tree in the first place, we see that simplification is relatively cheap to perform. We note, however, that computation of the local spatial measure is included in the initial contour tree construction. Since these measures involve counting samples as they are processed, the overhead is quite low.

In each case, the simplification takes as input the contour tree after perturbation removal, and simplifies it down to a single edge, to allow it to be used in the simplified flexible isosurface interface described in Chapter 11.

Data Set	Size (n)	Tree Size (Superarcs)	Simplification Time (s)
Nucleon	68,921	49	< 0.01
Marschner-Lobb	68,921	695	< 0.01
Silicium	113,288	225	< 0.01
Fuel	262,144	129	< 0.01
Neghip	262,144	248	< 0.01
Shockwave	2,097,152	31	< 0.01
Hydrogen	2,097,152	8	< 0.01
Lobster	5,461,344	77,349	0.22
Engine	8,388,608	134,642	0.39
Statue	10,814,133	120,668	0.32
Teapot	11,665,408	20,777	0.05
Aneurysm	16,777,216	36,667	0.08
Bonsai	16,777,216	82,876	0.23
Foot	16,777,216	508,854	1.75
Skull	16,777,216	931,348	3.43

Table 16.4: Time Required to Simplify Contour Trees After Removing Perturbation.

		Minimal Simplicial Subdivision				
File	Data Size	Load	Sort	Join/Split	Merge	Tree Size
3dknee	127 x 256 x 256	54.36s	22.99s	141.34s	32.28s	2,751,506
3dhead	109 x 256 x 256	45.71s	17.69s	89.05s	23.94s	2,231,900
1dog.0.8	72 x 64 x 60	2.18s	0.34s	1.63s	0.19s	18,498
3gap.0.8	29 x 60 x 131	1.80s	0.27s	1.31s	0.15s	14,290
neghip	64 x 64 x 64	1.59s	1.63s	1.00s	0.08s	2,544
fuel	64 x 64 x 64	1.55s	0.63s	0.91s	0.06s	299
		Marching Cubes				
File	Data Size	Load	Sort	Join/Split	Merge	Tree Size
3dknee	127 x 256 x 256	54.17s	23.03s	129.70s	31.12s	2,706,019
3dhead	109 x 256 x 256	45.67s	17.63s	81.62s	22.86s	2,196,594
1dog.0.8	72 x 64 x 60	2.20s	0.34s	1.61s	0.18s	17,656
3gap.0.8	29 x 60 x 131	1.81s	0.27s	1.28s	0.15s	13,164
neghip	64 x 64 x 64	1.59s	1.63s	0.99s	0.08s	2,063
fuel	64 x 64 x 64	1.55s	0.63s	0.94s	0.06s	227

Table 16.5: Comparison of Construction Times Using Simplicial Subdivision and Marching Cubes. Broadly speaking, there is no speed advantage to either basis for computing the contour tree. Note that these times are not directly comparable with the results in Table 16.2, as they result from an older, less well-optimized implementation of the algorithm. But, since the implementation of the simplicial subdivision has not been optimized, comparing the equivalent marching cubes implementation is more appropriate.

16.3 Results for Marching Cubes

In Table 16.5, we show some times for constructing the contour tree using the minimal 5-fold subdivision scheme described in [CMS01], in comparison with times for constructing the contour tree using the Marching Cubes method described in Chapter 14. The contour tree sizes in this table are before removing perturbation, and are different between the two sets of figures. This was expected, as Marching Cubes imposes a different topology from that imposed by minimal simplicial subdivision.

The size of tree and the speed of computation are still similar, as the large-scale topology of the

File	Size	Isovalue	Five Simplices	Marching Cubes	Ratio
3dknee	127 x 256 x 256	1639.2	3,662,308	1,634,744	2.24
3dhead	109 x 256 x 256	1639.2	1,043,892	441,998	2.36
1dog.0.8	72 x 64 x 60	0.30	393,450	165,964	2.37
3gap.0.8	29 x 60 x 131	0.65	194,568	81,798	2.38
neghip	64 x 64 x 64	101.9	49,484	20,360	2.43
fuel	64 x 64 x 64	101.9	7,564	2,946	2.57

Table 16.6: Sample Isosurface Sizes for Simplicial Subdivision and Marching Cubes. Marching Cubes outperforms simplicial subdivision by a factor of about 2.3, roughly in accordance with the estimates in [CTM03], while avoiding the directional biases in [CMS01].

data is unaffected by local decisions. Both versions process the same set of vertices and approximately the same number of edges, so we expected this to be true.

Using Marching Cubes, however, results in a clear gain in rendering speed, as shown in Table 16.6. We took the minimum and maximum sampled isovalues, and arbitrarily chose an isovalue at 40% of this range. We then extracted the isosurface using the minimal (five simplices) subdivision and the Marching Cubes implementations. In each case, the simplicial subdivision resulted in roughly two to two-and-a-half times as many triangles, which we expected from the results of Carr, Theußl and Möller [CTM03].

Moreover, we know from [CMS01] that simplicial subdivision produces isosurfaces with visible directional biases, where Marching Cubes does not.

Marching Cubes is both faster and better quality than simplicial subdivision, which is the main reason why we have used it. In practice, the bottleneck for exploratory visualization is the cost of rendering to the screen. For isosurfaces, this cost is principally k: the number of primitives (usually triangles) generated. In particular, when adjusting an isosurface, the major cost is that of transferring roughly 100 bytes to the video card for each triangle generated. For datasets such as 3dknee, which generate surfaces in the millions of triangles, this dominates almost any other cost. For the flexible isosurface interface, we generate new isosurfaces interactively, and do not have sufficient time to simplify the surface before sending it to the video card. Thus, using Marching Cubes rather than simplices is a major advantage.

A correct trilinear interpolated surface should also be faster than simplicial subdivision, but involves significant overhead both in computing the contour tree and in tessellating topologically complex cells. Moreover, the additional topological complexity is all at small scales. Since we remove small scale topological detail during the simplification process, it is questionable how much is gained by trilinear interpolation for the noisy acquired datasets we have been studying.

16.4 Isosurface Extraction Using Path Seeds

Now that we have looked at computation times for the contour tree algorithm, we consider the cost of using path seeds for isosurface extraction. In Table 16.7, we show some results, again using Marching Cubes for speed. These results are summaries of trying various isosurfaces interactively, giving ranges rather than precise measurements. So, for example, in f368, we tried a number of isosurfaces, with triangle counts ranging from 10,000 to 17,000, and found that the length of paths used to find isosurfaces was in the range $4 - 6$.

Data Set	Size (n)	Superarcs (t)	Longest Path	Triangle Count	Frame Rate (fps)
f368	30,345	915	4-6	10K - 17K	15 - 60
marlobb	68,921	912	1-23	10K - 35K	24 - 65
fuel	262,144	227	2-3	3K - 11K	25 -180
hipiph	262,144	1,360	5-29	2K - 22K	15 -150
neghip	262,144	2,063	7-10	10K - 30K	20 - 60
lobster	489,600	17,867	4	19K - 96K	7 - 32
teddybear	1,015,808	245,588	4-5	19K -250K	3 - 21
3dhead	7,143,424	2,231,900	2-5	85K -500K	0.5 - 3.5
3dknee	8,323,072	2,751,506	1-6	325K -1634K	0.5 -1.64

Table 16.7: Some Results for Path Seed Isosurface Extraction. In general, the larger the contour tree, the shorter the paths, and vice versa. Moreover, the path lengths are much smaller than the isosurface sizes, as predicted.

Broadly speaking, these results indicate that the overhead for using path seeds to extract isosurfaces is negligible compared to the cost of extracting and rendering triangles. Moreover, the noisier the data set, the shorter the paths. But this also illustrates once more why it is inadvisable to use simplicial subdivisions, as the larger datasets usually dropped below a frame rate which can reasonably be called interactive: halving this frame rate for simplicial subdivision was a penalty we were not prepared to accept.

16.5 Images Produced Using Simplified Contour Trees

In this section, we show some images produced using the flexible isosurface interface with the simplified contour tree. In each case, the ability to focus on large-scale topology allowed us to isolate what appeared to be important contours relatively quickly.

Figure 16.1 shows an image generated from the UNC Head data set. No expert knowledge was used to select the isosurfaces shown: instead, only a few minutes with the interface were required to identify the major features shown. The image shown took about half an hour in total to produce, much of which time was spent in laying out the contour tree by hand.

To produce this image, we reduced the contour tree to a single edge, then uncollapsed one edge at a time. As each new leaf was added to the tree, we placed a contour on that leaf and manipulated it to determine what it was, then assigned a suitable colour: for example, all identifiable skull fragments were assigned the light blue colour in order that the corresponding edges in the contour tree not be obtrusive. Adjusting the vertex positions to get the groupings of the leaves shown was done manually.

In terms of particular structures, we note that the ventricular structures inside the brain are shown in the contour tree, but are not visible in the data display. Also, several of the structures shown are defined as cavities - i.e. contours surrounding pits of low intensity rather than peaks of high intensity. The ventricles, eyeballs and nasal / sinus cavity all fit into this category. So does the structure which we have labelled the lower jaw. Since the lower jaw is made of bone, it ought not to be a cavity, and this structure may represent tooth pulp and nerves, or perhaps the gap between teeth and cheeks. This structure is, however, clearly associated with the jaw. Other structures are given by contours around peaks of high intensity, such as the blood vessels, eye sockets, nasal septum, and brain.

This image can be compared with Figure 16.2, which shows a conventional isosurface of the same

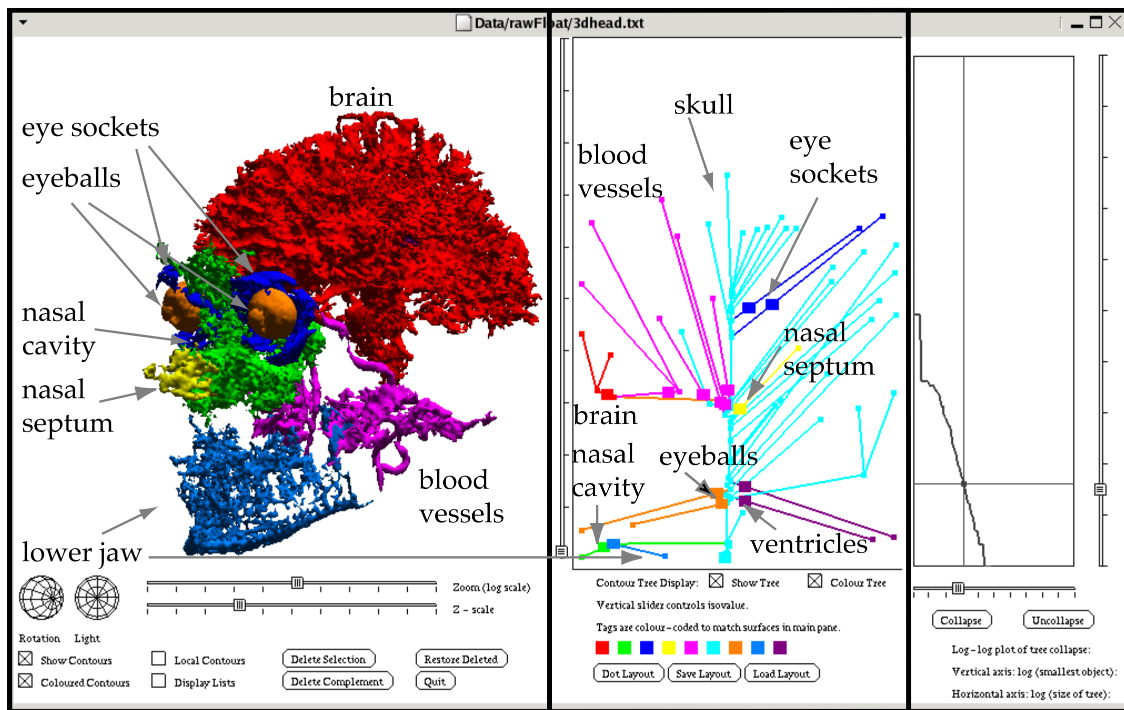


Figure 16.1: Topological Simplification of UNC Head Data Set. A number of structures are shown, some defined in terms of local minima, others in terms of local maxima.

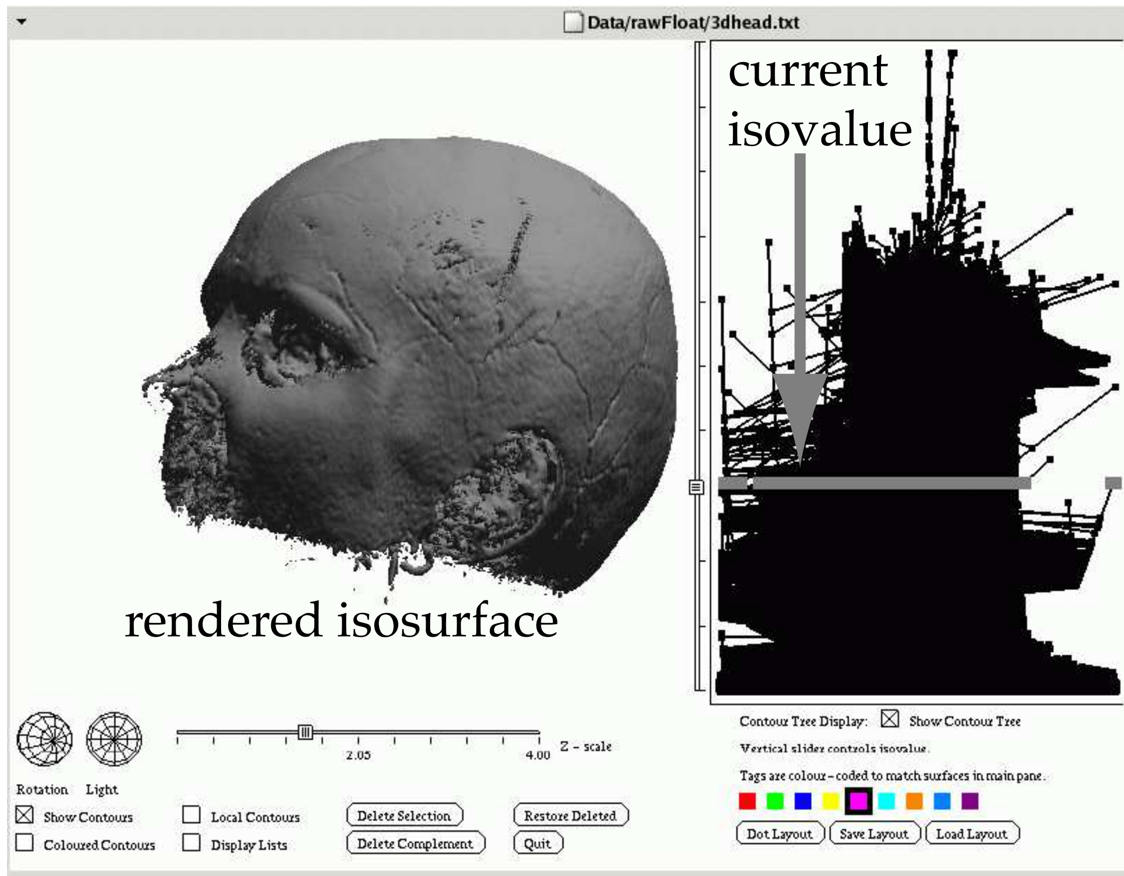


Figure 16.2: Conventional Isosurface of UNC Head Data Set Without Contour Tree Simplification.

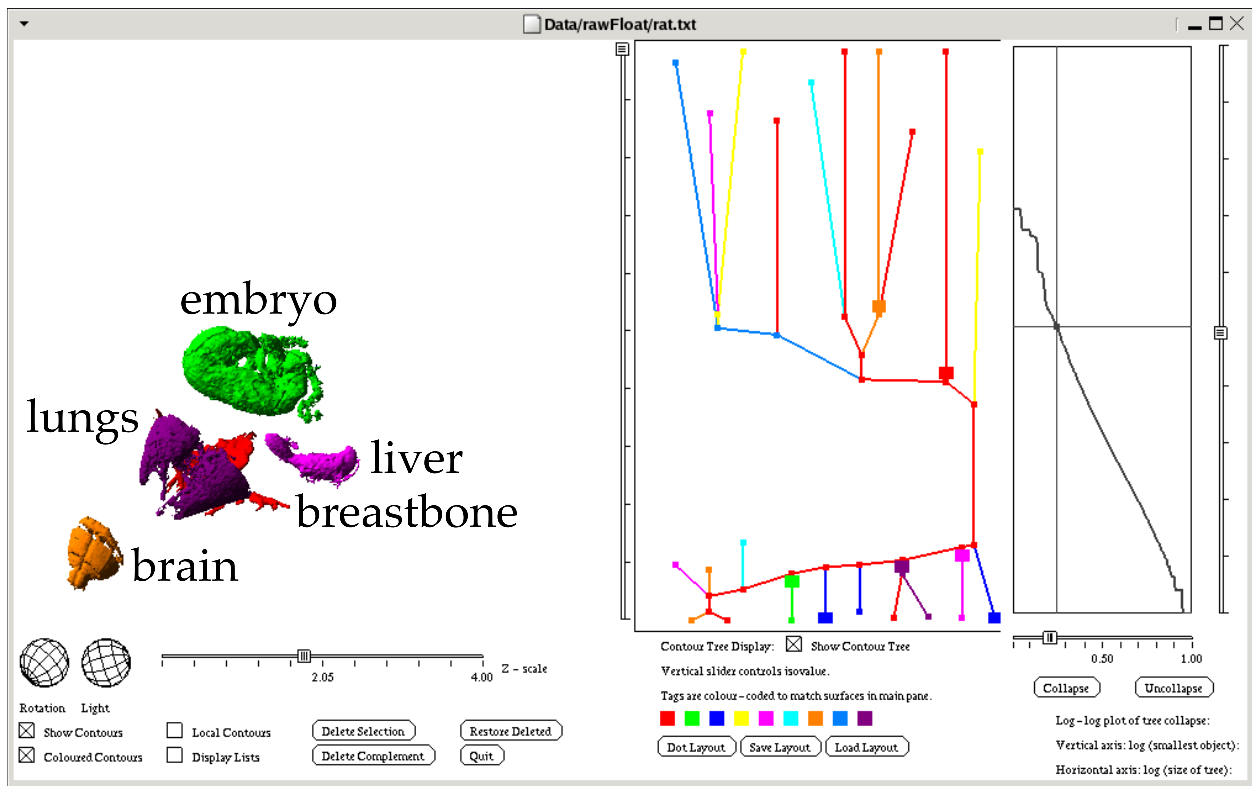


Figure 16.3: Structures Visible in a Noisy MRI Scan of a Pregnant Rat.

data set. Note how the isosurface for the skull completely conceals all other contours.

Figure 16.3 shows the results of a similar exploration of a 240 × 256 × 256, low-quality MRI data set of a pregnant rat, taken from the *Whole Frog Project* at `http://www-itg.lbl.gov/ITG.hm.pg.docs/Whole.Frog/Whole.Frog.html`. Again, simplification reduces the contour tree to a size that can be used to explore the data.

In this case, the contour tree shown was laid out using the *dot* tool described in Chapter 9 rather than manual manipulation. As a result, this image took less than five minutes to produce. Before exploring this data set I was unaware that the rat in question was pregnant, but after finding an obvious embryo, I confirmed that the rat was indeed pregnant by checking the source of the data. This illustrates one of the great advantages of this type of exploration of the data: one need not know in advance precisely what one is looking for.

In Figure 16.4, we show several structures in the CT head data set, first with the skull, then without. Since the ventricular structures are among the largest structures in this data set as well as the UNC head, we hope to be able to perform automated extraction of these in the future. Again, tree layout was performed using *dot*, and these images took only a few minutes to produce.

Finally, Figure 16.5 shows an example of how topological simplification using the contour tree and local spatial measures can help to suppress noise in isosurface renderings.

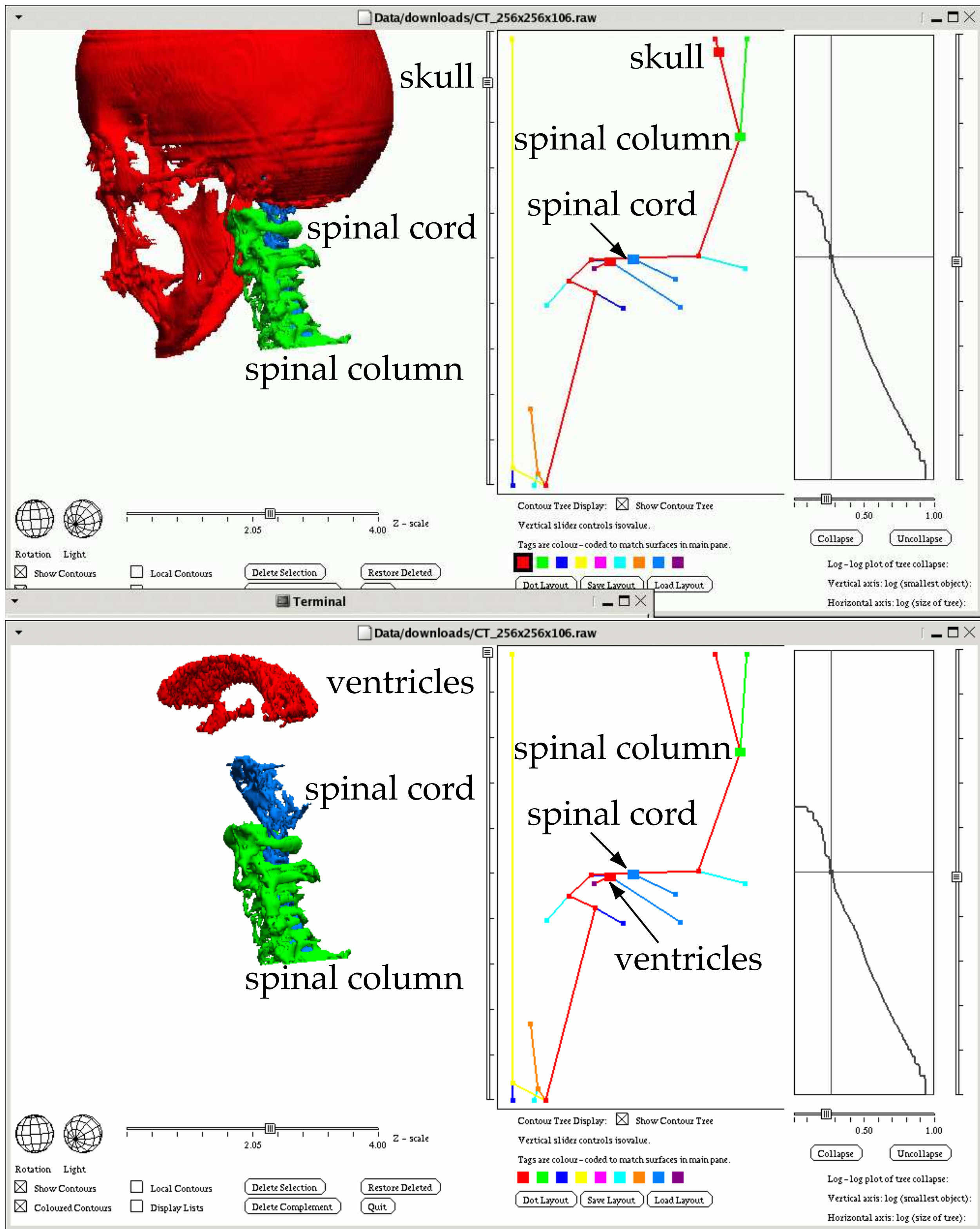


Figure 16.4: Structures in the CT Head data set.
These images show the spine, spinal cord, skull and ventricular structures.

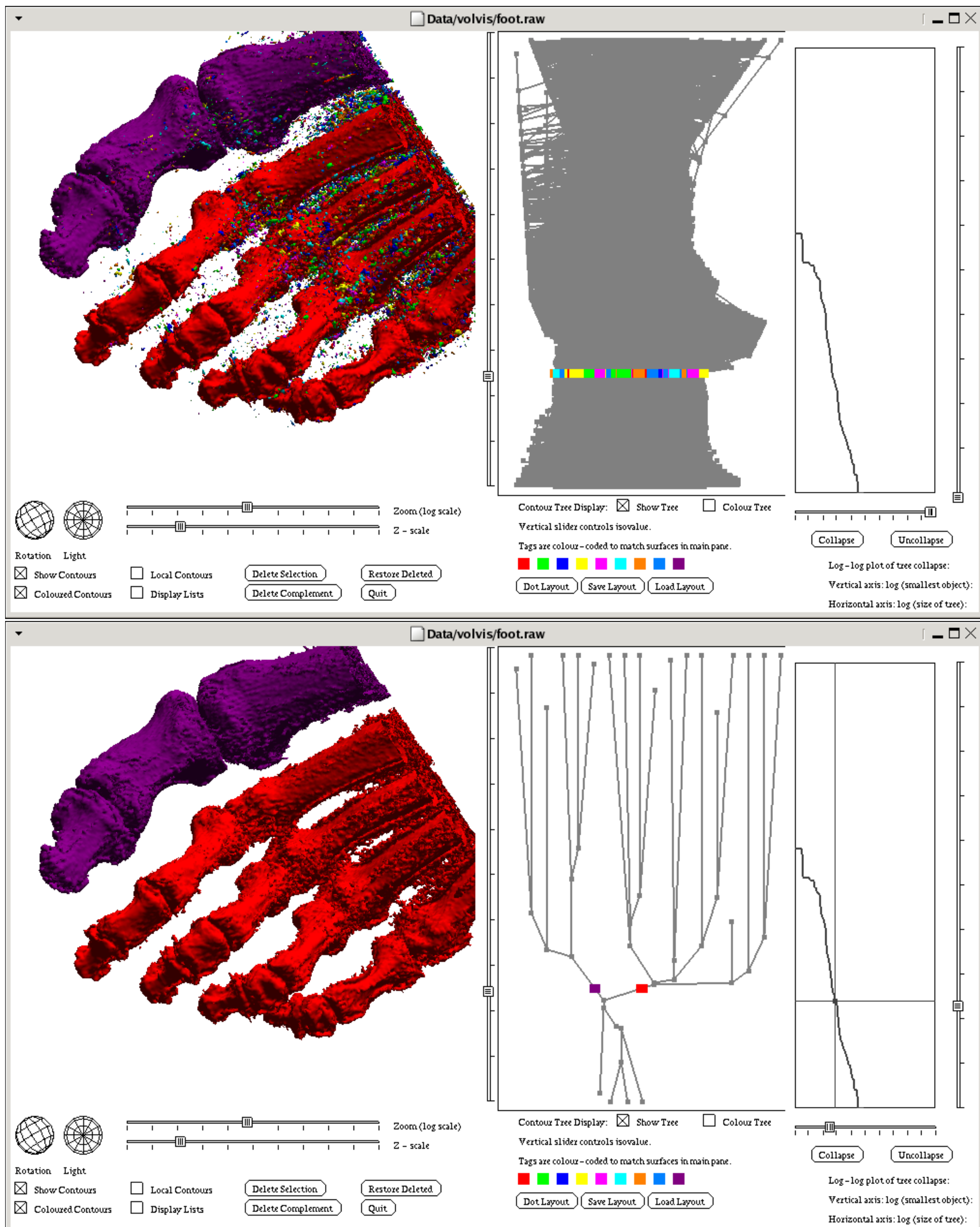


Figure 16.5: Removing Noise Topologically.
In these two images, we show the effects of simplification on conventional isosurfaces. Note that, by simplifying the contour tree, we automatically remove all contours of small importance, drastically improving the display of the bones of the foot without losing any detail. Also note the way that the five-fold structure of the contour tree reflects the five toes in the foot.

Chapter 17

Conclusions

In this thesis, we have shown how to use the topological information encoded in the contour tree to enable interactive manipulation of individual contour surfaces in an isosurface scene, using an interface called the flexible isosurface.

The major research contributions embodied in this thesis are the *flexible isosurfaces* of Chapter 8, which generalize level sets and permit interactive exploration of individual contours, the *local spatial measures* of Chapter 10, which define geometric properties with respect to regions bounded by contours, and the *topological simplification* of Chapter 11, which uses local spatial measures to simplify both the contour tree and the underlying function.

In addition, there are a range of secondary contributions, without which the major contributions would not have been possible. These include:

1. the *path seeds* in Chapter 8: an efficient, optimally sized method for generating isosurfaces directly from the contour tree.
2. the *contour evolution* methods in Chapter 8 that use the contour tree to track the evolution of an individual isosurface as the isovalue is varied.
3. the demonstration in Chapter 8 that both the *largest contour segmentation* of Manders et al. [MHS+96] and the *object-oriented visualization* of Silver [Sil95] are special cases of the flexible isosurface.
4. the *approximate local spatial measures* in Chapter 10 that approximate local spatial measures by enumerating samples
5. the generalization of contour tree algorithms in Chapter 12 to *non-simplicial meshes* in any dimension, consisting of any type of cell, with any type of interpolant.
6. the further generalization of contour tree algorithms in Chapter 12 to non-interpolating tessellations such as Marching Cubes [LC87].
7. the extension in Chapter 12 of the continuation method of Wyvill, McPheeters & Wyvill [WMW86a] to *piecewise continuation*, which generates individual contours one at a time.
8. the discussion in Chapter 13 of techniques for computing the contour tree for quadrilinear interpolation in four dimensions.

9. the simple shortcut in Chapter 14 for computing contour trees for the Marching Cubes cases of Montani, Scateni & Scopigno [MSS94a].
10. the application in Chapter 15 of simplification to remove perturbation that was applied to the data to avoid degeneracies.

Finally, Chapter 16 gave some results that show that these techniques can feasibly be implemented for typical data sets using modern desktop hardware, even without heavy optimization of code.

Chapter 18

Future Work

The possibilities inherent in the topological and geometric information stored in the contour tree have not yet been exhausted. There are a variety of directions that could be pursued:

1. Improvements to Contour Tree Algorithms
2. Topology of Non-Isovalued Surfaces
3. Variations on the Flexible Isosurface Interface
4. Local Spatial Measures
5. Applications of Topological Simplification and Filtering
6. Non-Simplicial Meshes and Computational Geometry
7. Time-Varying Three Dimensional Data

18.1 Improvements to Contour Tree Algorithms

Although the algorithms described in Chapter 7 are efficient, and work smoothly for medium-sized data sets, we expect them to bog down when sizes exceed in-core memory. Pascucci & Cole-McLaughlin [PCM02] have shown a parallel algorithm for computing the un-augmented join tree and split tree, but the merge phase is apparently not parallelized. Since the local spatial measures of Chapter 10 require processing the entire augmented contour tree, their parallelization method may need adaptation.

The bottleneck at present for interactive visualization is isosurface extraction and rendering. Although the continuation method is guaranteed not to spend time in empty cells, it follows the surface it is extracting, no matter how convoluted. As a result, its memory access pattern rarely coincides with the layout of the data in memory. It would be useful to see if the memory access pattern could be optimized.

Moreover, except for parallel extraction of multiple surfaces, the continuation algorithm does not naturally parallelize, because of the need to check whether cells have been visited previously. It would also be useful to find ways to use parallelism for rendering.

Finally, the simplification performed in Chapter 11 is also serial at present, but may be parallelizable: this is not, however, a great concern, as the simplification step is relatively fast. For large data sets, however, it would be useful to be able to parallelize it.

Both Pascucci & Cole-McLaughlin [PCM02] and Chiang et al. [CLLR02] have shown $O(n + t\log(t))$ algorithms to compute the reduced (un-augmented) contour tree for regular data, which they achieve by avoiding processing regular points. It would be useful to determine lower bounds for contour tree computations, both for augmented and un-augmented contour trees.

Finally, we saw in Chapter 15 that path seeds computed with perturbation need to retain that perturbation when used to extract isosurfaces. We would like to find a way around this, so that the perturbation is only used to compute the contour tree. Since we saw in Chapter 16 that over half of the edges in the contour tree can be due to perturbation, this would reduce the memory footprint in a natural way.

18.2 Topology of Non-Isovalued Surfaces

In Chapters 12 and 14, we showed how to compute contour trees for contours extracted by non-interpolating tessellation algorithms, based on the observation that the contours at different isovalues nest within each other.

However, not all methods for extracting significant boundaries rely on isovalue surfaces (i.e. contours). Instead, many techniques extract boundaries based on locating sharp boundaries, even if they are not isovalued. These techniques include the *active contour* methods of Kass, Witkin & Terzopoulos [KWT87] and the *level set* methods of Osher & Sethian [OS88], both of which require an initial starting point or seed, which is often provided manually. We have seen in Chapter 8 that the contour tree contains sufficient information to provide seeds for isosurface extraction. We would like to use the contour tree along with local spatial measures to find suitable propagation seeds for such methods.

A variation on this idea would be to generate seeds automatically from the contour tree and run these boundary generation algorithms for a large number of the seeds. If the underlying algorithm is well-behaved, these boundaries are likely to nest inside each other, and we may be able to index them with a structure similar to the contour tree.

18.3 Variations in the Flexible Isosurface Interface

The flexible isosurface interface is extremely powerful for exploratory visualization through individual contour evolution. There are a number of issues that we have left unresolved, and a number of new directions that we feel could be usefully pursued.

We have noted that drawing the contour tree in a visually pleasing fashion is difficult. We would like to explore suitable ways of doing so, and also explore whether the implicit isovalue axis in the contour tree is useful to end users, or whether a more arbitrary layout of the contour tree might be useful. We would also like to explore layout algorithms optimized for contour trees, and in particular for the hierarchy of contour trees generated by simplification. And we would like to explore methods for choosing colours to assign to individual edges based on local spatial measures.

One of the most powerful aspects of the contour tree is the ability to use it to index and annotate individual contours. We would like to add the ability for users to define labels for edges, and have them propagate through levels of simplification if necessary. One potential use of this is in continuing medical education: an instructor could use the contour tree to annotate important features in the data set. The student could then explore the data until the annotations were found, thus guiding the exploratory and learning processes. Similarly, we could add local spatial measure information to the interface, so that the user would be given the volume, surface area, &c. of any surface as they selected it in the main scene or the contour tree.

We have chosen to implement a simple, single-measure simplification: we would like to explore alternate methods, such as user-guided collapses, multi-measure collapses, or spatially-sensitive collapses. For user-guided collapses, it would be necessary to define how the user might specify collapses, and the data structures to support such actions. Multi-measure collapses, by comparison, might use one measure to remove noise, then a second measure to simplify away features that, although genuine features, add too much detail to a scene. Finally, spatially-sensitive collapses would involve making greater use of the assignment of individual samples to superarcs of the contour tree. In this way, each superarc represents a region of space. Thus, if a user is more interested in one region of space, it should be feasible to choose a region of space (perhaps a ball of fixed radius), and guide the collapse in such a way that objects intersecting that region were made immune to simplification.

A related direction also takes advantage of the contour tree's subdivision of the data into regions for classification and segmentation. Since the contour tree encodes the topology of the isovalue and distinguishes between different contours, we would like to use this to affect how individual samples are classified. One of the simplest examples of this can be seen in Figure 16.1, where we chose not to display any isosurfaces corresponding to the skull in order to focus attention on the internal organs. We believe that classifying samples according to the superarc to which they belong could significantly improve transfer functions for volume rendering. We would like to modify the flexible isosurface interface so that locally supported transfer functions are defined with respect to contours in a flexible isosurface.

We also note that in the UNC head data set the skull and the brain contours join at an isovalue below which the individual surfaces continue to develop. A narrow bridge between the two surfaces connects them before they have finished their development. As a result, we may be missing some features of the brain because we cannot see through the skull. If we could edit the data to keep these two major features separate over a wider range of isovalues, we would have another valuable tool for isosurface visualization.

18.4 Local Spatial Measures

In Chapter 10, we observed that we could compute exact polynomials for simplicial meshes. In Chapter 13, we identified that doing so for multilinear interpolants is likely to be complex and costly. Nevertheless, we would like to explore doing so, if only for completeness. Similarly, in Chapter 14, we observed that local spatial measures for the Marching Cubes cases could be obtained by suitable tessellation of the regions bounded by the contour. In this thesis, we chose to use approximate local spatial measures, but there may be some merit in defining the equations for volume and surface area for these tessellation cases.

We also note that the local spatial measures we have discussed have been general in application. For any given field of research, however, there may be more specific geometric measures of importance, and we would like to collaborate with researchers in a variety of fields to determine whether this is in fact the case.

Finally, we note that the contour tree also provides a convenient data structure for specific queries to be carried out in a data set. With local spatial measures one could efficiently answer queries such as "Find all contours that have an enclosed volume of larger than 10 units and an approximate surface-area-to-volume ration of 5." If information such as bounding boxes can be computed as a local spatial measure, then we can also include spatial constraints. And inverse problems can also be stated. For example, given examples of tumors, what should the query constraints be to find other such features.

18.5 Applications of Topological Simplification and Filtering

In this thesis, we have scratched the surface of the uses of topological simplification. Given that we progressively remove larger and larger features, the simplified contour tree could be used as a canonical representation of data, as it is in principle translation-, rotation-, and scale- invariant. Potential applications of this include graph matching for automated annotation and automated segmentation based on local spatial measures.

We note that Chiang & Lu [CL03] used the contour tree to guide mesh simplification in engineering data sets. We would like to combine this topology-preserving form of simplification with our topology-discarding form of simplification to find mesh simplifications that correspond to given contour tree simplifications.

Finally, in Figure 11.9 and the following figures, we constructed equivalent surfaces to the simplified surface by hand. This is straightforward for simplicial meshes, where we can change the isovalues at vertices without altering the contour tree. It is less trivial to do this for non-simplicial meshes with complex interpolants, and we would like to examine this problem in more detail.

18.6 Non-Simplicial Meshes and Computational Geometry

We have shown how to extend contour tree algorithms to arbitrary meshes in a computationally efficient manner. Our motivation in so doing was principally to cope with the dilemma that computational geometry algorithms commonly require simplicial meshes, while acquired data is commonly on cubic meshes. We would like to explore whether techniques similar to the finite state machine approach in Chapter 12 would be applicable to other algorithms, and in particular, to the Morse and Morse-Smale complex algorithms of Edelsbrunner, Harer & Zomorodian [EHZ01] and Bremer et al. [BEHP03].

Further, although we have defined a general approach, there are a number of specific cases we are interested in pursuing in detail, including the Spiderweb algorithm of Cox, Karron & Mishra [CKM93], and cubic and polycubic splines. For the trilinear case, we would also like to work out the details of the lookup tables for join graphs, of piecewise continuation, and of local spatial measures. For the quadrilinear case, we would like to implement the spatial subdivision approach to contour tree construction, and to analyse the quadrilinear tessellation cases. Finally, we would like to build the lookup tables for Marching Cubes, and confirm that the Marching Cubes shortcuts generalizes to higher dimensions.

18.7 Time-Varying Three Dimensional Data

Although all of the material discussed in this thesis is applicable to data of arbitrary dimensions, time-varying data is not always best dealt with as a function defined over four equivalent dimensions. In practice, most four-dimensional data consists of time-slices of three-dimensional data. It would be useful to be able to construct contour trees quickly for arbitrary or interpolated time-slices, using some form of abstraction that tracks the evolution of the contour tree over time.

Bibliography

[AAW00] Anagnostou, K., Atherton, T. J., and Waterfall, A. E. 4D Volume Rendering with the Shear Warp Factorisation. In *Proceedings of Volume Visualization 2000*, pages 129–137, 2000.

[ADM92] Arnaud, Y., Desbois, M., and Maizi, J. Automatic Tracking and Characterization of African Convective Systems on Meteosat Pictures. *Journal of Applied Meteorology*, 31:443–453, 1992.

[AFH81] Artzy, E., Frieder, G., and Herman, G. T. The Theory, Design, Implementation and Evaluation of a Three-Dimensional Surface Detection Algorithm. *Computer Graphics and Image Processing*, 15:1–24, 1981.

[Art79] Artzy, E. Display of Three-Dimensional Information in Computed Tomography. *Computer Graphics and Image Processing*, 9:196–198, 1979.

[Ban67] Banchoff, T. F. Critical Points and Curvature for Embedded Polyhedra. *Journal of Differential Geometry*, 1:245–256, 1967.

[BEHP03] Bremer, P.-T., Edelsbrunner, H., Hamann, B., and Pascucci, V. A Multi-resolution Data Structure for Two-dimensional Morse-Smale Functions. In *Proceedings of IEEE Visualization 2003*, pages 139–146, 2003.

[Ben75] Bentley, J. L. Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM*, 18(9):509–517, 1975.

[Ben79] Bentley, J. L. Decomposable Searching Problems. *Information Processing Letters*, 8:244–251, 1979.

[Bli82] Blinn, J. F. A Generalization of Algebraic Surface Drawing. *ACM Transactions on Graphics*, 1(3):235–256, 1982.

[Blo88] Bloomenthal, J. Polygonization of implicit surfaces. *Computer Aided Geometric Design*, pages 341–355, 1988.

[BP99] Bajaj, C. L. and Pascucci, V. Time Critical Adaptive Refinement and Smoothing. Technical Report 99-36, Texas Institute for Computational and Applied Mathematics, Austin, Texas, 1999.

[BPS97] Bajaj, C. L., Pascucci, V., and Schikore, D. R. The Contour Spectrum. In *Proceedings of IEEE Visualization 1997*, pages 167–173, 1997.

[BPS99] Bajaj, C. L., Pascucci, V., and Schikore, D. R. Seed Sets and Search Structures for Optimal Isocontour Extraction. Technical Report 99-35, Texas Institute for Computational and Applied Mathematics, Austin, Texas, 1999.

[BR63] Boyell, R. L. and Ruston, H. Hybrid Techniques for Real-time Radar Simulation. In *Proceedings of the 1963 Fall Joint Computer Conference*, pages 445–458. IEEE, 1963.

[BSRF00] Bemis, K. G., Silver, D., Rona, P. A., and Feng, C. Case Study: A Methodology for Plume Visualization with Application to Real-Time Acquisition and Navigation. In *Proceedings of IEEE Visualization 2000*, pages 481–484, 2000.

[BWC00] Bhaniramka, P., Wenger, R., and Crawfis, R. A. Isosurfacing in Higher Dimensions. In *Proceedings of IEEE Visualization 2000*, pages 267–273, 2000.

[Car00] Carr, H. Efficient Generation of 3D Contour Trees. Master's thesis, University of British Columbia, Vancouver, BC, Canada, 2000.

[CGMS00] Cignoni, P., Ganovelli, F., Montani, C., and Scopigno, R. Reconstruction of topologically correct and adaptive trilinear surfaces. *Computers And Graphics*, 24:399–418, 2000.

[Che95] Chernyaev, E. Marching Cubes 33: Construction of Topologically Correct Isosurfaces. Technical report, CERN, 1995.

[CKF03] Cox, J., Karron, D., and Ferdous, N. Topological Zone Organization of Scalar Volume Data. *Journal of Mathematical Imaging and Vision*, 18:95–117, 2003.

[CKM93] Cox, J., Karron, D., and Mishra, B. The SpiderWeb Surface rendering algorithm. *Innovation and Technology in Biology and Medicine*, 14(6):634–655, 1993.

[CL03] Chiang, Y.-J. and Lu, X. Progressive Simplification of Tetrahedral Meshes Preserving All Isosurface Topologies. *Computer Graphics Forum*, 22(3):to appear, 2003.

[CLL+88] Cline, H. E., Lorenson, W. E., Ludke, S., Crawford, C., and Teeter, B. Two algorithms for the three-dimensional reconstruction of tomograms. *Medical Physics*, 15(3):320–327, 1988.

[CLLR02] Chiang, Y.-J., Lenz, T., Lu, X., and Rote, G. Simple and Output-Sensitive Construction of Contour Trees Using Monotone Paths. Technical Report ECG-TR-244300-01, Institut für Informatik, Freie Universtät Berlin, 2002.

[CM97] Chiueh, T.-c. and Ma, K.-L. A Parallel Pipelined Renderer for Time-Varying Volume Data. In *Proceedings of Parallel Architectures, Algorithms, and Networks 1997*, pages 9–15. IEEE, 1997.

[CMM+97] Cignoni, P., Marino, P., Montani, C., Puppo, E., and Scopigno, R. Speeding Up Isosurface Extraction Using Interval Trees. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):158–169, 1997.

[CMS01] Carr, H., Möller, T., and Snoeyink, J. Simplicial Subdivisions and Sampling Artifacts. In *Proceedings of IEEE Visualization 2001*, pages 99–106, 2001.

[CS03] Carr, H. and Snoeyink, J. Path Seeds and Flexible Isosurfaces: Using Topology for Exploratory Visualization. In *Proceedings of Eurographics Visualization Symposium 2003*, pages 49–58, 285, 2003.

[CSA00] Carr, H., Snoeyink, J., and Axen, U. Computing Contour Trees in All Dimensions. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 918–926, January 2000.

[CSA03] Carr, H., Snoeyink, J., and Axen, U. Computing Contour Trees in All Dimensions. *Computational Geometry: Theory and Applications*, 24(2):75–94, 2003.

[CTM03] Carr, H., Theußl, T., and Möller, T. Isosurfaces on Optimal Regular Samples. In *Proceedings of Eurographics Visualization Symposium 2003*, pages 39–48, 284, 2003.

[dBETT99] di Battista, G., Eades, P., Tamassia, R., and Tollis, I. G. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice-Hall Inc., Englewood-Cliffs, NJ, 1999.

[DCH88] Drebin, R. A., Carpenter, L., and Hanrahan, P. Volume Rendering. *Computer Graphics*, 22(4):65–74, 1988.

[DCK+98] Dobashi, Y., Cingoski, V., Kaneda, K., Yamashita, H., and Nishita, T. A Fast Volume Rendering Method for Time-Varying 3-D Scalar Field Visualization Using Orthonormal Wavelets. *IEEE Transactions on Magnetics*, 34(5):3431–3434, 1998.

[Dee96] Deering, M. Geometry Compression. *Computer Graphics*, 30:13–20, 1996.

[DH94] Durkin, J. W. and Hughes, J. F. Nonpolygonal Isosurface Rendering for Large Volume Datasets. In *Proceedings of IEEE Visualization 1994*, pages 293–300, 1994.

[dLvLV+00] de Leeuw, W. C., van Liere, R., Verschure, P. J., Visser, A. E., Manders, E. M. M., and van Driel, R. Visualization of Time Dependent Confocal Microscopy Data. In *Proceedings of IEEE Visualization 2000*, pages 473–476, 2000.

[Dür88] Dürst, M. Letters: Additional Reference to "Marching Cubes". *Computer Graphics*, 22(4):65–74, 1988.

[ECS00] Ellsworth, D., Chiang, L.-J., and Shen, H.-W. Accelerating Time-Varying Hardware Volume Rendering Using TSP Trees and Color-Based Error Metrics. In *Proceedings of Volume Visualization 2000*, pages 119–129, 2000.

[Ede80] Edelsbrunner, H. Dynamic Data Structures for Orthogonal Intersection Queries. Technical report, Inst. Informationsverarb, Tech. Uniz. Graz, Graz, Austria, 1980.

[EHZ01] Edelsbrunner, H., Harer, J., and Zomorodian, A. Hierarchical Morse Complexes for Piecewise Linear 2-Manifolds. In *Proceedings of the 17th ACM Symposium on Computational Geometry*, pages 70–79. ACM, 2001.

[EM90] Edelsbrunner, H. and Mücke, E. P. Simulation of Simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Transactions on Graphics*, 9(1):66–104, 1990.

[FKU77] Fuchs, H., Kedem, Z., and Uselton, S. Optimal Surface Reconstruction from Planar Contours. *Communications of the ACM*, 20:693–702, 1977.

[FM67] Freeman, H. and Morse, S. On Searching A Contour Map for a Given Terrain Elevation Profile. *Journal of the Franklin Institute*, 284(1):1–25, 1967.

[Gal91] Gallagher, R. S. Span Filtering: An Optimization Scheme For Volume Visualization of Large Finite Element Models. In *Proceedings of IEEE Visualization 1991*, pages 68–75. IEEE, 1991.

[Gar90] Garrity, M. P. Raytracing Irregular Volume Data. *Computer Graphics*, 24(5):35–40, 1990.

[GC86] Gold, C. and Cormack, S. Spatially Ordered Networks and Topographic Reconstruction. In *Proceedings of the 2nd International ACM Symposium on Spatial Data Handling*, pages 74–85, 1986.

[GW02] Gonzalez, R. C. and Woods, R. E. *Digital Image Processing (2d ed.)*. Prentice-Hall Inc., Englewood-Cliffs, NJ, 2002.

[HB94] Howie, C. and Blake, E. H. The Mesh Propagation Algorithm for Isosurface Construction. *Computer Graphics Forum*, 13:65–74, 1994.

[HH92] Hansen, C. D. and Hinker, P. Massively Parallel Isosurface Extraction. In *Proceedings of IEEE Visualization 1992*, pages 77–83, 1992.

[HL79] Herman, G. T. and Lun, H. K. Three-Dimensional Display of Human Organs from Computed Tomograms. *Computer Graphics and Image Processing*, 9:1–21, 1979.

[IK94] Itoh, T. and Koyamada, K. Isosurface Extraction By Using Extrema Graphs. *IEEE Transactions on Visualization and Computer Graphics*, 1:77–83, 1994.

[IK95] Itoh, T. and Koyamada, K. Automatic Isosurface Propagation Using an Extrema Graph and Sorted Boundary Cell Lists. *IEEE Transactions on Visualization and Computer Graphics*, 1(4):319–327, 1995.

[IYK01] Itoh, T., Yamaguchi, Y., and Koyamada, K. Fast Isosurface Generation Using the Volume Thinning Algorithm. *IEEE Transactions on Visualization and Computer Graphics*, 7(1):32–46, 2001.

[Kaj86] Kajiya, J. T. The rendering equation. *Computer Graphics*, 20(4):143–150, 1986.

[KK94] Kweon, I. S. and Kanade, T. Extracting Topographic Terrain Features from Elevation Maps. *CVGIP: Image Understanding*, 59:171–182, 1994.

[KRS01] Kettner, L., Rossignac, J., and Snoeyink, J. The Safari Interface for Visualizing Time-Dependent Volume Data Using Iso-surfaces and Contour Spectra. *Computational Geometry: Theory and Applications*, 25(1-2):97–116, 2001.

[KWT87] Kass, M., Witkin, A., and Terzopoulos, D. Snakes: Active Contour Models. In *Proceedings of the 1st International Conference on Computer Vision*, pages 259–268. IEEE, 1987.

[KWTM04] Kindlmann, G., Whitaker, R., Tasdizen, T., and Möller, T. Curvature-Based Transfer Functions for Direct Volume Rendering: Methods and Applications. In *Proceedings of IEEE Visualization 2003*, pages 513–520, 2004.

[LB03] Lopes, A. and Brodlie, K. Improving the robustness and accuracy of the marching cubes algorithm for isosurfacing. *IEEE Transactions on Visualization and Computer Graphics*, 9(1):16–29, 2003.

[LC87] Lorenson, W. E. and Cline, H. E. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *Computer Graphics*, 21(4):163–169, 1987.

[Lev88] Levoy, M. Volume Rendering: Display of Surfaces from Volume Data. *IEEE Computer Graphics and Applications*, 8(3):29–37, 1988.

[LKS+98] Lanzagorta, M., Kral, M., Swan, II, J., Spanos, G., Rosenberg, R., and Kuo, E. Three-Dimensional Visualization of Microstructures. In *Proceedings of IEEE Visualization 1998*, pages 487–490, 1998.

[LMC01] Lum, E. B., Ma, K.-L., and Clyne, J. Texture Hardware Assisted Rendering of Time-Varying Volume Data. In *Proceedings of IEEE Visualization 2001*, pages 263–270, 2001.

[LSJ96] Livnat, Y., Shen, H.-W., and Johnson, C. R. A Near Optimal Isosurface Extraction Algorithm Using the Span Space. *IEEE Transactions on Visualization and Computer Graphics*, 2(1):73–84, 1996.

[Mat94] Matveyev, S. V. Approximation of Isosurface in the Marching Cube: Ambiguity Problem. In *Proceedings of IEEE Visualization 1994*, pages 288–292, 1994.

[Max95] Max, N. L. Optical Models for Direct Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, 1995.

[MC00] Ma, K.-L. and Camp, D. M. High Performance Visualization of Time-Varying Volume Data over a Wide-Area Network. In *Supercomputing 2000*, 2000.

[MCW93] Max, N. L., Crawfis, R. A., and Williams, D. Visualization for Climate Modelling. *IEEE Computer Graphics and Applications*, 13(4):34–41, 1993.

[MDB87] McCormick, B. H., DeFanti, T. A., and Brown, M. D. Visualization in Scientific Computing. *Computer Graphics*, 21(6), 1987.

[MHS+96] Manders, E. M. M., Hoebe, R., Strackee, J., Vossepoel, A., and Aten, J. Largest Contour Segmentation: A Tool for the Localization of Spots in Confocal Images. *Cytometry*, 23:15–21, 1996.

[Mil63] Milnor, J. *Morse Theory.* Princeton University Press, Princeton, NJ, 1963.

[ML94] Marschner, S. R. and Lobb, R. J. An Evaluation of Reconstruction Filters for Volume Rendering. In *Proceedings of IEEE Visualization 1994*, pages 100–107, 1994.

[MSS94a] Montani, C., Scateni, R., and Scopigno, R. A modified look-up table for implicit disambiguation of Marching Cubes. *Visual Computer*, 10:353–355, 1994.

[MSS94b] Montani, C., Scateni, R., and Scopigno, R. Discretized Marching Cubes. In *Proceedings of IEEE Visualization 1994*, pages 281–287, 1994.

[Nat94] Natarajan, B. On generating topologically consistent isosurfaces from uniform samples. *Visual Computer*, 11:52–62, 1994.

[NB93] Ning, P. and Bloomenthal, J. An Evaluation of Implicit Surface Tilers. *IEEE Computer Graphics and Applications*, 13:33–41, 1993.

[NH91] Nielson, G. M. and Hamann, B. The Asymptotic Decider: Resolving the Ambiguity in Marching Cubes. In *Proceedings of IEEE Visualization 1991*, pages 83–91. IEEE, 1991.

[OS88] Osher, S. and Sethian, J. A. Fronts Propagating with Curvature Dependent Speed: Algorithms Based on Hamilton-Jacobi Formulations. *Journal of Computational Physics*, 79:12–49, 1988.

[Pas01] Pascucci, V. On the Topology of the Level Sets of a Scalar Field. In *Abstracts of the 13th Canadian Conference on Computational Geometry*, pages 141–144, 2001.

[PCM02] Pascucci, V. and Cole-McLaughlin, K. Efficient Computation of the Topology of Level Sets. In *Proceedings of IEEE Visualization 2002*, pages 187–194, 2002.

[PWH01] Pekar, V., Wiemker, R., and Hempel, D. Fast Detection of Meaningful Isosurfaces for Volume Data Visualization. In *Proceedings of IEEE Visualization 2001*, pages 223–230, 2001.

[Ree46] Reeb, G. Sur les points singuliers d'une forme de Pfaff complètement intégrable ou d'une fonction numérique. *Comptes Rendus de l'Acadèmie des Sciences de Paris*, 222:847–849, 1946.

[RG94] Roth, M. and Guritz, R. Case Study: Visualization of Volcanic Ash Clouds. In *Proceedings of IEEE Visualization 1994*, pages 386–390, 1994.

[RG00] Robbins, K. A. and Gorman, M. Fast Visualization Methods for Comparing Dynamics: A Case Study in Combustion. In *Proceedings of IEEE Visualization 2000*, pages 433–436, 2000.

[RHC94] Rushmeier, H. E., Hamins, A., and Choi, M.-Y. Case Study: Volume Rendering of Pool Fire Data. In *Proceedings of IEEE Visualization 1994*, pages 382–385, 1994.

[Sab88] Sabella, P. A Rendering Algorithm for Visualizing 3D Scalar Fields. *Computer Graphics*, 22(4):51–55, 1988.

[SC86] Sircar, J. K. and Cebrian, J. A. Application of Image Processing Techniques to the Automated Labelling of Raster Digitized Contour Maps. In *Proceedings of the 2nd International ACM Symposium on Spatial Data Handling*, pages 171–184, 1986.

[SCM99] Shen, H.-W., Chiang, L.-J., and Ma, K.-L. A Fast Volume Rendering Algorithm for Time-Varying Fields Using a Time-Space Partitioning (TSP) Tree. In *Proceedings of IEEE Visualization 1999*, pages 371–376, 1999.

[SH99] Sutton, P. M. and Hansen, C. D. Isosurface Extraction in Time-Varying Fields Using a Temporal Branch-on-Need Tree (T-BON). In *Proceedings of IEEE Visualization 1999*, pages 147–152, 1999.

[SH00] Sutton, P. M. and Hansen, C. D. Accelerated Isosurface Extraction in Time-Varying Fields. *IEEE Transactions on Visualization and Computer Graphics*, 6(2):98–107, 2000.

[She98] Shen, H.-W. Isosurface Extraction in Time-varying Fields Using a Temporal Hierarchical Index Tree. In *Proceedings of IEEE Visualization 1998*, pages 159–167, 1998.

[SHLJ96] Shen, H.-W., Hansen, C. D., Livnat, Y., and Johnson, C. R. Isosurfacing in Span Space with Utmost Efficiency (ISSUE). In *Proceedings of IEEE Visualization 1996*, pages 287–294, 1996.

[Sil95] Silver, D. Object-Oriented Visualization. *IEEE Computer Graphics and Applications*, 15(3):55–62, 1995.

[SJ94] Shen, H.-W. and Johnson, C. R. Differential Volume Rendering: A Fast Volume Visualization Technique for Flow Animation. In *Proceedings of IEEE Visualization 1994*, pages 180–187, 1994.

[SJ95] Shen, H.-W. and Johnson, C. R. Sweeping Simplices: A fast iso-surface extraction algorithm for unstructured grids. In *Proceedings of IEEE Visualization 1995*, pages 143–150, 1995.

[SK91] Shinagawa, Y. and Kunii, T. L. Constructing a Reeb Graph Automatically from Cross Sections. *IEEE Computer Graphics and Applications*, 11(6):45–51, 1991.

[SKK91] Shinagawa, Y., Kunii, T. L., and Kergosien, Y. L. Surface Coding Based on Morse Theory. *IEEE Computer Graphics and Applications*, 11:66–78, September 1991.

[SSZC94] Samtaney, R., Silver, D., Zabusky, N., and Cao, J. Visualizing Features and Tracking Their Evolution. *Computer*, 27(7):20–27, 1994.

[SW96] Silver, D. and Wang, X. Volume Tracking. In *Proceedings of IEEE Visualization 1996*, pages 157–164, 1996.

[SW98] Silver, D. and Wang, X. Tracking Scalar Features in Unstructured Datasets. In *Proceedings of IEEE Visualization 1998*, pages 79–86, 1998.

[Tar75] Tarjan, R. E. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22:215–225, 1975.

[TFT01] Takahashi, S., Fujishiro, I., and Takeshima, Y. Topological Volume Skeletonization and its Application to Transfer Function Design. Technical Report OCHA-IS 2000-3, Department of Information Sciences, Faculty of Science, Ochanomizu University, Ochanomizu, Japan, February 2001.

[Til24] Tilley, C. The Facies Classification of Metamorphic Rocks. *Geological Magazine*, 61:167–171, 1924.

[TIS+95] Takahashi, S., Ikeda, T., Shinagawa, Y., Kunii, T. L., and Ueda, M. Algorithms for Extracting Correct Critical Points and Constructing Topological Graphs from Discrete Geographical Elevation Data. *Computer Graphics Forum*, 14(3):C–181–C–192, 1995.

[TLM01] Tenginakai, S., Lee, J., and Machiraju, R. Salient Iso-Surface Detection with Model-Independent Statistical Signatures. In *Proceedings of IEEE Visualization 2001*, pages 231–238, 2001.

[TO91] Thune, N. and Olstad, B. Visualizing 4-D Medical Ultrasound Data. In *Proceedings of IEEE Visualization 1991*, pages 210–215. IEEE, 1991.

[TV98] Tarasov, S. P. and Vyalyi, M. N. Construction of Contour Trees in 3D in $O(n \log n)$ steps. In *Proceedings of the 14th ACM Symposium on Computational Geometry*, pages 68–75, 1998.

[UK88] Upson, C. and Keeler, M. V-BUFFER: Visible Volume Rendering. *Computer Graphics*, 22(4):59–64, 1988.

[vK94] van Kreveld, M. Efficient Methods for Isoline Extraction from a Digital Elevation Model Based on Triangulated Irregular Networks. In *Proceedings of the 6th International Symposium on Spatial Data Handling*, pages 835–847, 1994.

[vKvOB+97] van Kreveld, M., van Oostrum, R., Bajaj, C. L., Pascucci, V., and Schikore, D. R. Contour Trees and Small Seed Sets for Isosurface Traversal. In *Proceedings of the 13th ACM Symposium on Computational Geometry*, pages 212–220, 1997.

[Wes95] Westermann, R. Compression Domain Rendering of Time-Resolved Volume Data. In *Proceedings of IEEE Visualization 1995*, pages 168–175, 1995.

[WMW86a] Wyvill, B., McPheeters, C., and Wyvill, G. Animating Soft Objects. *Visual Computer*, 2:235–242, 1986.

[WMW86b] Wyvill, G., McPheeters, C., and Wyvill, B. Data Structure for Soft Objects. *Visual Computer*, 2:227–234, 1986.

[WvG90] Wilhelms, J. and van Gelder, A. Topological Considerations in Isosurface Generation. *Computer Graphics*, 24(5):79–86, 1990.

[WvG92] Wilhelms, J. and van Gelder, A. Octrees for Faster Isosurface Generation. *ACM Transactions on Graphics*, 11(3):201–227, 1992.

[Wyv94] Wyvill, B. Explicating Implicit Surfaces. In *Proceedings of Graphics Interface 1994*, pages 165–173, 1994.

[Zyd88] Zyda, M. J. A Decomposable Algorithm for Contour Surface Display Generation. *ACM Transactions on Graphics*, 7(2):129–148, 1988.

Appendix A

Code for Generating Piecewise Continuation Tables

The code in this appendix generates the piecewise continuation tables for the Marching Cubes cases of Montani, Scateni & Scopigno [MSS94a] by performing rotations on the symmetry-reduced cases specified. We use *surface fragment* to specify a single distinct surface in a cell, because it will typically form a fragment of a final surface. This code generates the following tables:

Array	Purpose
mcFollowVertexCoords[8][3]	Table showing the location of each vertex in the cell.
vertex2Edge[8][8]	Table mapping pairs of vertex IDs to an edge ID.
edgeVertices[24][2]	Table showing which vertices each edge connects.
caseName[23]	The name of each base (symmetry-reduced) marching cubes case.
baseCase[256]	The symmetry-reduced base case for a given marching cubes case.
nSurfaces[256]	The number of surface fragments for each marching cubes case (maximum 4).
nTriangles[256][4]	The number of triangles for each surface fragment (maximum 5).
mcFollowTriangles[256][4][15]	The actual triangles as triplets of points identified by the edge to which they belong.
seedEdge2Surface[256][24]	Which surface fragment each possible seed edge intersects in a given case (at most 1).
nExitEdges[256][4]	The number of exit edges (i.e. edges on cube faces) for each surface fragment (max 7).
surface2ExitEdges[256][4][7]	The identities of the exit edges for each surface fragment.
exit2EntryEdge[36]	Which exit edge in one cube corresponds to which entry edge in another.
exitDirection[36][3]	Which direction the exit leads (e.g. (-1, 0, 0) leads to a cube in the -x direction).
entryEdge2Surface[256][36]	Which surface fragment each entry edge intersects in a given case.

A.1 The Code

```
#include <stdio.h>

#define N_CASES 23 //   # of cases in MC table (including mirror-images)
                                                                        //   case 10M is the mirror image of case 10 (i.e. 10M = 14)

int vertex2edge[8][8] =                                                 //   table defining the possible edges
    { // vertex2edge                                                    //   -1: no edge
    {   -1,  0,   3,   12,      4,   20,  19,  -1  },                   //   edges starting from 000
    {   0,   -1,  13,  1,       21,  5,   -1,  14  },                   //   edges starting from 001
    {   3,   13,  -1,  2,       18,  -1,  7,   16  },                   //   edges starting from 010
    {   12,  1,   2,   -1,      -1,  15,  17,  6   },                   //   edges starting from 011

    {   4,   21,  18,  -1,      -1,  8,   11,  22  },                   //   edges starting from 100
    {   20,  5,   -1,  15,      8,   -1,  23,  9   },                   //   edges starting from 101
    {   19,  -1,  7,   17,      11,  23,  -1,  10  },                   //   edges starting from 110
    {   -1,  14,  16,  6,       22,  9,   10,  -1  }                    //   edges starting from 111
    }; // vertex2edge

int edge2faceEdge[12][12] =                                             //   which face edge connects each pair of edges
    { // edge2faceEdge
    {   -1,  3,   4,   0,       22,  23,  -1,  -1,      18,  -1,  -1,  -1},     //   from edge 000-001
    {   3,   -1,  2,   5,       -1,  6,   7,   -1,      -1,  11,  -1,  -1},     //   from edge 001-011
    {   4,   2,   -1,  1,       -1,  -1,  12,  13,      -1,  -1,  17,  -1},     //   from edge 011-010
    {   0,   5,   1,   -1,      29,  -1,  -1,  28,      -1,  -1,  -1,  24},     //   from edge 010-000

    {   22,  -1,  -1,  29,      -1,  19,  -1,  25,      21,  -1,  -1,  26},     //   from edge 001-101
    {   23,  6,   -1,  -1,      19,  -1,  10,  -1,      20,  9,   -1,  -1},     //   from edge 011-111
    {   -1,  7,   12,  -1,      -1,  10,  -1,  16,      -1,  8,   15,  -1},     //   from edge 010-110
    {   -1,  -1,  13,  28,      25,  -1,  16,  -1,      -1,  -1,  14,  27},     //   from edge 000-100

    {   18,  -1,  -1,  -1,      21,  20,  -1,  -1,      -1,  32,  31,  35},     //   from edge 100-101
    {   -1,  11,  -1,  -1,      -1,  9,   8,   -1,      32,  -1,  33,  30},     //   from edge 101-111
    {   -1,  -1,  17,  -1,      -1,  -1,  15,  14,      31,  33,  -1,  34},     //   from edge 111-110
    {   -1,  -1,  -1,  24,      26,  -1,  -1,  27,      35,  30,  34,  -1}      //   from edge 110-100
    }; // edge2faceEdge

int edgeVertices[24][2] =                                               //   table with the vertices each edge connects
    { // edgeVertices
    0,   1,       1,   3,       3,   2,       2,   0,                   //   e0 - e3
    0,   4,       1,   5,       3,   7,       2,   6,                   //   e4 - e7
    4,   5,       5,   7,       7,   6,       6,   4,                   //   e8 - e11

    0,   3,       1,   2,       1,   7,       3,   5,                   //   d0 - d3 (e12 - e15)
    2,   7,       3,   6,       4,   2,       0,   6,                   //   d4 - d7 (e16 - e19)
    0,   5,       1,   4,       4,   7,       5,   6                    //   d8 - d11 (e20 - e23)
    }; // edgeVertices

int faceEdge2Edges[36][2] =                                             //   map from faceEdge to edge
    { // faceEdge2Edges
    0,   3,       3,   2,       2,   1,       1,   0,       0,   2,       3,   1,
    5,   1,       1,   6,       6,   9,       9,   5,       5,   6,       1,   9,
    6,   2,       2,   7,       7,   10,      10,  6,       6,   7,       2,   10,

    8,   0,       4,   5,       5,   8,       8,   4,       4,   0,       0,   5,
    11,  3,       7,   4,       4,   11,      11,  7,       7,   3,       3,   4,
    9,   11,      10,  8,       8,   9,       9,   10,      10,  11,      11,  8
    }; // faceEdge2Edges

int nSurfaces[N_CASES] =                                                //   # of distinct surfaces in each case
    { // nSurfaces
    0,                                                                  //   0 vertices above
    1,                                                                  //   1 vertex above
    1,   2,   2,                                                        //   2 vertices above
    1,   2,   3,                                                        //   3 vertices above
    1,   1,   1,   1,   2,   2,   4,                                    //   4 vertices above (remember mirroring)
    2,   1,   1,                                                        //   5 vertices above
    2,   1,   1,                                                        //   6 vertices above
    1,                                                                  //   7 vertices above
    0                                                                   //   all vertices above
    }; // nSurfaces

int nTriangles[N_CASES][4] =
    { // nTriangles
    //   no vertices above
    {   0    },                                                         //   case 0 has no triangles
    //   one vertex above
    {   1    },                                                         //   case 1 has 1 triangle
    //   two vertices above
    {   2    },                                                         //   case 2 has 2 triangles
    {   1,   1    },                                                    //   case 3 has 2 surfaces, each with 1 triangle
    {   1,   1    },                                                    //   case 4 has 2 surfaces, each with 1 triangle
    //   three vertices above
    {   3    },                                                         //   case 5 has 3 triangles
    {   2,   1    },                                                    //   case 6 has 2 surfaces, with 2 & 1 triangles
    {   1,   1,   1    },                                               //   case 7 has 3 surfaces, each with 1 triangle
    //   four vertices above
    {   2    },                                                         //   case 8 has 1 surface with 1 triangle
    {   4    },                                                         //   case 9 has 1 surface with 4 triangles
    {   4    },                                                         //   case 10 has 1 surface with 4 triangles
    {   4    },                                                         //   case 10M (mirrored) has 1 surface with 4 triangles
    {   3,   1    },                                                    //   case 11 has 2 surfaces with 3 & 1 triangles
    {   2,   2    },                                                    //   case 12 has 2 surfaces with 1 triangle each
    {   1,   1,   1,   1    },                                          //   case 13 has 4 surfaces, each with 1 triangle
    //   five vertices above
    {   1,   4    },                                                    //   case 7C has 2 surfaces, with 1 & 4 triangles
    {   5    },                                                         //   case 6C has 1 surface with 5 triangles
```

```
    {   3   },                                                                  //  case 5C has 1 surface with 3 triangles
    //  six vertices above
    {   1,  1   },                                                              //  case 4C has 2 surfaces with 1 triangle each
    {   4   },                                                                  //  case 3C has 1 surface with 4 triangles
    {   2   },                                                                  //  case 2C has 1 surface with 2 triangles
    //  seven vertices above
    {   1   },                                                                  //  case 1C has 1 surface with 1 triangle
    //  eight vertices above
    {   0   }                                                                   //  case 0C has no triangles
    }; // nTriangles

int baseTriangles[N_CASES][4][15] =                                             //  array listing the actual triangles
    { // baseTriangles
    //  no vertices above: case 0
    {   {   0   }   },                                                          //  0:  make sure that the array is non-empty
    //  1 vertex above: case 1
    {   {   0,  3,  4   }   },                                                  //  1:  1 surface: 1 triangle
    //  2 vertices above: cases 2 - 4
    {   {   3,  4,  5,      1,  3,  5   }   },                                  //  2:  1 surface: 2 triangles
    {   {   0,  3,  4   },  {   1,  6,  2   }   },                              //  3:  2 surfaces: 1 triangle each
    {   {   0,  3,  4   },  {   6,  9,  10  }   },                              //  4:  2 surfaces: 1 triangle each
    //  3 vertices above: cases 5 - 7
    {   {   1,  3,  4,      1,  4,  8,      1,  8,  9   }   },                  //  5:  1 surface: 3 triangles
    {   {   3,  4,  5,      1,  3,  5   },  {   6,  9,  10  }   },              //  6:  2 surfaces: 2, 1 triangles
    {   {   0,  3,  4   },  {   1,  6,  2   },  {   7,  10, 11  }   },// 7:  3 surfaces: 1 triangle each
    //  4 vertices above: cases 8 - 13
    {   {   1,  3,  11,     1,  11, 9   }   },                                  //  8:  1 surface: 2 triangles
    {   {   1,  2,  7,      1,  7,  11,     1,  11, 5,                          //  9:  1 surface: 4 triangles
            5,  11, 8   }   },
    {   {   1,  3,  4,      1,  4,  8,      1,  8,  10,                         //  10: 1 surface: 4 triangles
            1,  10, 6   }   },
    {   {   3,  4,  5,      3,  5,  9,      3,  9,  10,                         //  10M: 1 surface: 4 triangles
            3,  10, 2   }   },
    {   {   1,  3,  4,      1,  4,  8,      1,  8,  9   },                      //  11: 2 surfaces: 3, 1 triangles
        {   7,  10, 11  }   },
    {   {   0,  2,  7,      0,  7,  4   },                                      //  12: 2 surfaces: 2, 2 triangles
        {   5,  8,  6,      8,  10, 6   }   },
    {   {   0,  3,  4   },  {   1,  6,  2   },  {   7,  10, 11  },              //  13: 4 surfaces: 1 triangle each
        {   5,  8,  9   }   },
    //  5 vertices above: cases 7C - 5C
    {   {   7,  2,  3   },                                                      //  7C: 2 surfaces: 1, 4 triangles
        {   1,  0,  4,      1,  4,  11,     1,  11, 6,      6,  11, 10  }   },
    {   {   3,  1,  10,     1,  6,  10,     10, 9,  5,                          //  6C: 1 surface: 5 triangles
            10, 5,  4,      10, 4,  3   }   },
    {   {   3,  1,  11,     11, 1,  5,      11, 5,  8   }   },                  //  5C: 1 surface: 3 triangles
    //  6 vertices above: cases 4C - 2C
    {   {   0,  4,  3   },  {   6,  10, 9   }   },                              //  4C: 2 surfaces: 1 triangle each
    {   {   3,  2,  4,      4,  2,  6,      4,  6,  1,                          //  3C: 1 surface: 4 triangles
            4,  1,  0   }   },
    {   {   3,  5,  4,      3,  1,  5   }   },                                  //  2C: 1 surface: 2 triangles
    //  7 vertices above: case 1C
    {   {   0,  4,  3   }   },                                                  //  1C: 1 surface: 1 triangle
    //  all vertices above: case 0C
    {   {   0   }   }                                                           //  make sure array is non-empty
    }; // baseTriangles

char *caseName[N_CASES] = {   "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "10", "10M", "11", "12", "13",
                              "7C", "6C", "5C", "4C", "3C", "2C", "1C", "0C" };

int baseCase[N_CASES] =                                                         //  base case for each class of cases
    { // baseCase
    0,                                                                          //  case 0
    1,                                                                          //  case 1
    3,  9,  129,                                                                //  cases 2 - 4
    35, 131, 73,                                                                //  cases 5 - 7
    51, 23, 163, 139, 99, 165, 105,                                             //  cases 8 - 13
    182, 124, 236,                                                              //  cases 7C - 5C
    126, 246, 252,                                                              //  cases 4C - 2C
    254,                                                                        //  case 1C
    255                                                                         //  case 0C
    }; // baseCase

int nAboveVert[N_CASES][4] =                                                    //  # of vertices "above" each surface
    { // nAboveVert
        {   0   },                                                              //  case 0
        {   1   },                                                              //  case 1
        {   2   },  {   1,  1   },  {   1,  1   },                              //  cases 2 - 4
        {   3   },  {   2,  1   },  {   1,  1,  1   },                          //  cases 5 - 7
        {   4   },  {   4   },  {   4   },  {   4   },                          //  cases 8 - 10M
        {   3,  1   },  {   2,  2   },  {   1,  1,  1,  1   },                  //  cases 11 - 13
        {   1,  4   },  {   5   },  {   5   },                                  //  cases 7C - 5C
        {   6,  6   },  {   6   },  {   6   },                                  //  cases 4C - 2C
        {   6   },                                                              //  case 1C
        {   0   }                                                               //  case 0C
    }; // nAboveVert

int nBelowVert[N_CASES][4] =                                                    //  # of vertices "below" each surface
    { // nBelowVert
        {   0   },                                                              //  case 0
        {   6   },                                                              //  case 1
        {   6   },  {   5,  5   },  {   6,  6   },                              //  cases 2 - 4
        {   5   },  {   5,  5   },  {   4,  4,  4   },                          //  cases 5 - 7
        {   4   },  {   4   },  {   4   },  {   4   },                          //  cases 8 - 10M
        {   4,  4   },  {   4,  4   },  {   3,  3,  3,  3   },                  //  cases 11 - 13
        {   3,  3   },  {   3   },  {   3   },                                  //  cases 7C - 5C
        {   1,  1   },  {   2   },  {   2   },                                  //  cases 4C - 2C
        {   1   },                                                              //  case 1C
        {   0   }                                                               //  case 0C
    }; // nBelowVert
```

```
int baseAboveVert[N_CASES][4][6] =                                          //  which vertices are "above" a given surface
    { // aboveVert
    //  no vertices above: case 0
    {   {   0   }   },                                                      //  0:  make sure that the array is non-empty
    //  1 vertex above: case 1
    {   {   0   }   },                                                      //  1:  1 surface: 1 vertex above it
    //  2 vertices above: cases 2 - 4
    {   {   0,  1   }   },                                                  //  2:  1 surface: 2 vertices above it
    {   {   0   },  {   3   }   },                                          //  3:  2 surfaces: 1 vertex above each
    {   {   0   },  {   7   }   },                                          //  4:  2 surfaces: 1 vertex above each
    //  3 vertices above: cases 5 - 7
    {   {   0,  1,  5   }   },                                              //  5:  1 surface: 3 vertices above
    {   {   0,  1   },  {   7   }   },                                      //  6:  2 surfaces: 2, 1 vertices above
    {   {   0   },  {   3   },  {   6   }   },                              //  7:  3 surfaces: 1 triangle each
    //  4 vertices above: cases 8 - 13
    {   {   0,  1,  4,  5   }   },                                          //  8:  1 surface: 4 vertices above it
    {   {   0,  1,  2,  4   }   },                                          //  9:  1 surface: 4 vertices above it
    {   {   0,  1,  5,  7   }   },                                          //  10: 1 surface: 4 vertices above it
    {   {   0,  1,  3,  7   }   },                                          //  10M: 1 surface: 4 vertices above it
    {   {   0,  1,  5   },  {   6   }   },                                  //  11: 2 surfaces: 3, 1 vertices above
    {   {   0,  2   },  {   5,  7   }   },                                  //  12: 2 surfaces: 2, 2 vertices above
    {   {   0   },  {   3   },  {   6   },  {   5   }   },                  //  13: 4 surfaces: 1 vertex above each
    //  5 vertices above: cases 7C - 5C
    {   {   2   },  {   1,  4,  5,  7   }   },                              //  7C: 2 surfaces: 1, 4 vertices above each
    {   {   2,  3,  4,  5,  6   }   },                                      //  6C: 1 surface: 5 vertices above it
    {   {   2,  3,  5,  6,  7   }   },                                      //  5C: 1 surface: 5 vertices above it
    //  6 vertices above: cases 4C - 2C
    {   {   1,  2,  3,  4,  5,  6   },                                      //  4C: 2 surfaces: 6 vertices above each
        {   1,  2,  3,  4,  5,  6   }   },
    {   {   1,  2,  4,  5,  6,  7   }   },                                  //  3C: 1 surface: 6 vertices above it
    {   {   2,  3,  4,  5,  6,  7   }   },                                  //  2C: 1 surface: 6 vertices above it
    //  7 vertices above: case 1C
    {   {   1,  2,  3,  4,  5,  6   }   },                                  //  1C: 1 surface: 6 vertices above (plus one major diag)
    //  all vertices above: case 0C
    {   {   0   }   }                                                       //  make sure array is non-empty
    }; // aboveVert

int aboveVert[256][4][6];                                                   //  running copy

int baseBelowVert[N_CASES][4][6] =                                          //  which vertices are "below" a given surface
    { // belowVert
    //  8 vertices below: case 0
    {   {   0   }   },                                                      //  0:  make sure that the array is non-empty
    //  7 vertex below: case 1
    {   {   1,  2,  3,  4,  5,  6   }   },                                  //  1:  1 surface: 6 vertices below it
    //  6 vertices below: cases 2 - 4
    {   {   2,  3,  4,  5,  6,  7   }   },                                  //  2:  1 surface: 6 vertices below it
    {   {   1,  2,  4,  5,  6   },  {   1,  2,  5,  6,  7   }   },          //  3:  2 surfaces: 5 vertices below each
    {   {   1,  2,  3,  4,  5,  6   },                                      //  4:  2 surfaces: 6 vertices below each
        {   1,  2,  3,  4,  5,  6   }   },
    //  5 vertices below: cases 5 - 7
    {   {   2,  3,  4,  6,  7   }   },                                      //  5:  1 surface: 5 vertices below
    {   {   2,  3,  4,  5,  6   },  {   2,  3,  4,  5,  6   }   },          //  6:  2 surfaces: 5 vertices below each
    {   {   1,  2,  4,  5   },  {   1,  2,  5,  7   },                      //  7:  3 surfaces: 1 triangle each
        {   2,  4,  5,  7   }   },
    //  4 vertices below: cases 8 - 13
    {   {   2,  3,  6,  7   }   },                                          //  8:  1 surface: 4 vertices below it
    {   {   3,  5,  6,  7   }   },                                          //  9:  1 surface: 4 vertices below it
    {   {   2,  3,  4,  6   }   },                                          //  10: 1 surface: 4 vertices below it
    {   {   2,  4,  5,  6   }   },                                          //  10M: 1 surface: 4 vertices below it
    {   {   2,  3,  4,  7   },  {   2,  3,  4,  7   }   },                  //  11: 2 surfaces: 4 vertices below each
    {   {   1,  3,  4,  6   },  {   1,  3,  4,  6   }   },                  //  12: 2 surfaces: 4 vertices below each
    {   {   1,  2,  4   },  {   1,  2,  7   },  {   2,  4,  7   },          //  13: 4 surfaces: 3 vertices below each
        {   1,  4,  7   }   },
    //  3 vertices below: cases 7C - 5C
    {   {   0,  3,  6   },  {   0,  3,  6   }   },                          //  7C: 2 surfaces: 3 vertices below each
    {   {   0,  1,  7   }   },                                              //  6C: 1 surface: 3 vertices below it
    {   {   0,  1,  4   }   },                                              //  5C: 1 surface: 3 vertices below it
    //  2 vertices below: cases 4C - 2C
    {   {   0   },  {   7   }   },                                          //  4C: 2 surfaces: 1 vertex below each
    {   {   0,  3   }   },                                                  //  3C: 1 surface: 2 vertices below it
    {   {   0,  1   }   },                                                  //  2C: 1 surface: 2 vertices below it
    //  1 vertices below: case 1C
    {   {   0   }   },                                                      //  1C: 1 surface: 1 vertex below
    //  no vertices below: case 0C
    {   {   0   }   }                                                       //  make sure array is non-empty
    }; // belowVert
int belowVert[256][4][6];                                                   //  running copy

int nExitEdges[256][4];                                                     //  how many exit edges each surface has
int exitEdges[256][4][8];                                                   //  what those exit edges are

int bitFlag[8] = { 1, 2, 4, 8, 16, 32, 64, 128 };                           //  bit flags for extracting bits

int rotX[8] = { 2, 0, 3, 1, 6, 4, 7, 5 };
int rotY[8] = { 4, 0, 6, 2, 5, 1, 7, 3 };
int rotZ[8] = { 2, 3, 6, 7, 0, 1, 4, 5 };

int invRotX[8] = { 1, 3, 0, 2, 5, 7, 4, 6};
int invRotY[8] = { 1, 5, 3, 7, 0, 4, 2, 6};
int invRotZ[8] = { 4, 5, 0, 1, 6, 7, 2, 3};

int edgeRotX[24] = {    3, 0, 1, 2,         7, 4, 5, 6,         11, 8, 9, 10,       //  cube edges
                        13, 12, 20, 21,     15, 14, 17, 16,     18, 19, 23, 22 };   //  face diagonals
int edgeRotY[24] = {    4, 3, 7, 11,        8, 0, 2, 10,        5, 1, 6, 9,         //  cube edges
                        18, 19, 12, 13,     17, 16, 23, 22,     21, 20, 15, 14 };   //  face diagonals
int edgeRotZ[24] = {    2, 6, 10, 7,        3, 1, 9, 11,        0, 5, 8, 4,         //  cube edges
                        16, 17, 15, 14,     23, 22, 19, 18,     13, 12, 20, 21 };   //  face diagonals

int invEdgeRotX[24] = { 1, 2, 3, 0,         5, 6, 7, 4,         9, 10, 11, 8,       //  cube edges
```

```
                        13, 12, 17, 16,     19, 18, 20, 21,     14, 15, 23, 22 };   //   face diagonals
int invEdgeRotY[24] = {  5, 9, 6, 1,         0, 8, 10, 2,        4, 11, 7, 3,        //   cube edges
                        14, 15, 23, 22,     17, 16, 12, 13,     21, 20, 19, 18 };   //   face diagonals
int invEdgeRotZ[24] = {  8, 5, 0, 4,         11, 9, 1, 3,        10, 6, 2, 7,        //   cube edges
                        21, 20, 15, 14,     12, 13, 19, 18,     22, 23, 17, 16 };   //   face diagonals

int cases[256];                                                                     //   array holding type for each case
int triangles[256][4][15];                                                          //   array storing the triangles
int seedEdge2Surface[256][24];                                                      //   table for looking up surfaces from seed edges
int entryEdge2Surface[256][36];                                                     //   table to go from entry edge to surface ID

int queue[256]; int qSize = 0;                                                      //   queue for processing cases

int Permute(int oldCase, int *perm)                                                 //   permutes the 8 vertices by the perm vector
     { // Permute()
     int result = 0;                                                                //   the result we end with
     int i;                                                                         //   loop index
     for (i = 0; i < 8; i++)                                                        //   for each bit
          { // i loop
          result |= ((oldCase & bitFlag[perm[i]]) ? 1 : 0) << i;                    //   extract the appropriate bit & stuff back in
          } // i loop
     return result;                                                                 //   and return what we started with
     } // Permute()

void ProcessCase(int oldCase, int *perm, int *edgePerm, int *invPerm)               //   computes perm, and sets it to same type
     { // ProcessCase()
     int newCase = Permute(oldCase, perm);                                          //   compute the permutation
     int whichCase;                                                                 //   which of the major cases it is
     int i, j, k;

     int whichAbove, whichBelow;                                                    //   which vertex we look at above / below surface
     int whichEdge;                                                                 //   which edge the above / below pair specifies

     if (cases[newCase] == -1)                                                      //   if this is the first time the case is reached
          { // first time case reached
          whichCase = cases[oldCase];                                               //   figure out which case it is
          cases[newCase] = whichCase;                                               //   set the cases to match
          queue[qSize++] = newCase;                                                 //   add the case to the queue
          for (i = 0; i < nSurfaces[whichCase]; i++)                                //   loop through the surfaces
               { // per surface
               for (j = 0; j < 3*nTriangles[whichCase][i]; j++)                     //   and through the # of triangles
                    { // loop through each triangle
                    triangles[newCase][i][j] = edgePerm[triangles[oldCase][i][j]];  //   copy the triangle vertex
                    } // loop through each triangle
               for (j = 0; j < nAboveVert[whichCase][i]; j++)
                    { // above loop
                    whichAbove = invPerm[aboveVert[oldCase][i][j]];                 //   compute above vert from old one
                    aboveVert[newCase][i][j] = whichAbove;                          //   which vertex is above
                    for (k = 0; k < nBelowVert[whichCase][i]; k++)
                         { // above / below pair
                         whichBelow = invPerm[belowVert[oldCase][i][k]];            //   compute above vert from old one
                         belowVert[newCase][i][k] = whichBelow;                     //   which vertex is above
                         whichEdge = vertex2edge[whichAbove][whichBelow];           //   retrieve the edge from the table
                         if (whichEdge == -1) continue;                             //   skip -1 edges
                         seedEdge2Surface[newCase][whichEdge] = i;                  //   flag the case to tie to this surface
                         } // above / below pair
                    } // above loop
               } // per surface
          } // first time case reached
     else if (cases[newCase] != cases[oldCase])                                     //   if they don't match
          printf("Major error.  Case %d permutes to %d, but has different type (%d vs. %d)\n", oldCase, newCase, cases[oldCase], cases[newCase]);
     } // ProcessCase()

int main()                                                                          //   main routine
     { // main()
     int theBaseCase, theCase, theCase2, theSurface, aboveID, belowID;              //   indices used throughout
     int qNext;                                                                     //   next item on queue
     int whichCase;                                                                 //   case being processed
     int whichAbove, whichBelow;                                                    //   which vertex we look at above / below surface
     int whichEdge;                                                                 //   which edge the above / below pair specifies
     int whichTri;                                                                  //   index for triangles
     int triVert0, triVert1, triVert2;                                              //   three vertices of a triangle
     FILE *outFile;                                                                 //   file handle for output

     for (theCase = 0; theCase < 256; theCase++)                                    //   initialize arrays to -1 or 0
          { // array initialization
          cases[theCase] = -1;                                                      //   sets the base case for each to -1

          for (theSurface = 0; theSurface < 4; theSurface++)                        //   for each possible surface
               { // for each surface
               nExitEdges[theCase][theSurface] = 0;                                 //   set the number of exit edges
               for (whichEdge = 0; whichEdge < 6; whichEdge++)                      //   for each possible exit edge
                    exitEdges[theCase][theSurface][whichEdge] = -1;                 //   make it predictable
               } // for each surface

          for (whichEdge = 0; whichEdge < 24; whichEdge++)                          //   sets the seed edge -> surface table
               seedEdge2Surface[theCase][whichEdge] = -1;

          for (whichEdge = 0; whichEdge < 36; whichEdge++)                          //   sets the entry edge -> surface table
               entryEdge2Surface[theCase][whichEdge] = -1;
          } // array initialization

     for (theBaseCase = 0; theBaseCase < N_CASES; theBaseCase++)                    //   loop to set the base cases
          { // set base cases
          whichCase = baseCase[theBaseCase];
          cases[whichCase] = theBaseCase;                                           //   store the base case in the array
          for (theSurface = 0; theSurface < nSurfaces[theBaseCase]; theSurface++)   //   loop through the surfaces
               { // loop through surfaces
               for (whichTri = 0; whichTri < 3*nTriangles[theBaseCase][theSurface]; whichTri++)
                                                                                    //   and through the # of triangles
```

```
                { // loop through each triangle
                triangles[whichCase][theSurface][whichTri] = baseTriangles[theBaseCase][theSurface][whichTri];
                } // loop through each triangle

            for (aboveID = 0; aboveID < nAboveVert[theBaseCase][theSurface]; aboveID++)
                { // above loop
                whichAbove = baseAboveVert[theBaseCase][theSurface][aboveID];    //   which vertex is above
                aboveVert[whichCase][theSurface][aboveID] = whichAbove;          //   store it in the new array
                for (belowID = 0; belowID < nBelowVert[theBaseCase][theSurface]; belowID++)
                    { // above / below pair
                    whichBelow = baseBelowVert[theBaseCase][theSurface][belowID];//  which vertex is below
                    belowVert[whichCase][theSurface][belowID] = whichBelow;     //   store it in the new array
                    whichEdge = vertex2edge[whichAbove][whichBelow];            //   retrieve the edge from the table
                    if (whichEdge == -1) continue;                              //   skip -1 edges
                    seedEdge2Surface[whichCase][whichEdge] = theSurface;        //   flag the case to tie to this surface
                    } // above / below pair
                } // above loop
            } // loop through surfaces
        queue[qSize++] = whichCase;                                             //   and add it to the queue
        } // set base cases

    for (qNext = 0; qNext < qSize; qNext++)                                     //   walk through queue
        { // loop through all cases on queue
        ProcessCase(queue[qNext], rotX, invEdgeRotX, invRotX);                  //   do an x-rotation
        ProcessCase(queue[qNext], rotY, invEdgeRotY, invRotY);                  //   do a y-rotation
        ProcessCase(queue[qNext], rotZ, invEdgeRotZ, invRotZ);                  //   do a z-rotation
        } // loop through all cases on queue

//  now we need to figure out the exit / entry edges
    for (theCase = 0; theCase < 256; theCase++)                                 //   for each case
        { // loop through all cases
        for (theSurface = 0; theSurface < nSurfaces[cases[theCase]]; theSurface++) //   for each surface
            { // for each surface
            for (whichTri = 0; whichTri < 3*nTriangles[cases[theCase]][theSurface]; whichTri+= 3)
                { // for each triangle
                triVert0 = triangles[theCase][theSurface][whichTri];            //   find triangle vertex (i.e. cube edge) IDs
                triVert1 = triangles[theCase][theSurface][whichTri+1];
                triVert2 = triangles[theCase][theSurface][whichTri+2];
                whichEdge = edge2faceEdge[triVert0][triVert1];                  //   look up face edge ID
                if (whichEdge != -1)                                            //   if it's a legal edge
                    { // good edge
                    exitEdges[theCase][theSurface][nExitEdges[theCase][theSurface]++] = whichEdge;
                    entryEdge2Surface[theCase][whichEdge] = theSurface;
                    } // good edge
                whichEdge = edge2faceEdge[triVert0][triVert2];                  //   look up face edge ID
                if (whichEdge != -1)                                            //   if it's a legal edge
                    { // good edge
                    exitEdges[theCase][theSurface][nExitEdges[theCase][theSurface]++] = whichEdge;
                    entryEdge2Surface[theCase][whichEdge] = theSurface;
                    } // good edge
                whichEdge = edge2faceEdge[triVert1][triVert2];                  //   look up face edge ID
                if (whichEdge != -1)                                            //   if it's a legal edge
                    { // good edge
                    exitEdges[theCase][theSurface][nExitEdges[theCase][theSurface]++] = whichEdge;
                    entryEdge2Surface[theCase][whichEdge] = theSurface;
                    } // good edge
                } // for each triangle
            } // for each surface
        } // loop through all cases

    outFile = fopen("./FollowCubeTables.h", "w");
    if (outFile == NULL)
        { // file open failed
        printf("Unable to open FollowCubeTables.h for writing.  Dumping to standard out.\n");
        outFile = stdout;
        } // file open failed

    fprintf(outFile, "// FollowCubeTables.h\n");
    fprintf(outFile, "// Copyright H. Carr 2003\n");
    fprintf(outFile, "// Marching Cube cases surface-specific contour-following (continuation)\n");
    fprintf(outFile, "// Generated %s, %s\n", __TIME__, __DATE__);

    fprintf(outFile, "extern int nSurfaces[256];\n");
    fprintf(outFile, "extern int nTriangles[256][4];\n");
    fprintf(outFile, "extern int mcFollowTriangles[256][4][15];\n");
    fprintf(outFile, "extern int seedEdge2Surface[256][24];\n");
    fprintf(outFile, "extern int nExitEdges[256][4];\n");
    fprintf(outFile, "extern int surface2exitEdges[256][4][7];\n");
    fprintf(outFile, "extern int exit2entryEdge[36];\n");
    fprintf(outFile, "extern int exitDirection[36][3];\n");
    fprintf(outFile, "extern int entryEdge2Surface[256][36];\n");
    fprintf(outFile, "extern int vertex2edge[8][8];\n");
    fprintf(outFile, "extern int mcFollowVertexCoords[8][3];\n");
    fprintf(outFile, "extern int edgeVertices[24][2];\n");
    fprintf(outFile, "extern char *caseName[23];\n");
    fprintf(outFile, "extern int baseCase[256];\n");

    fclose(outFile);

    outFile = fopen("./FollowCubeTables.cpp", "w");
    if (outFile == NULL)
        { // file open failed
        printf("Unable to open FollowCubeTables.cpp for writing.  Dumping to standard out.\n");
        outFile = stdout;
        } // file open failed

    fprintf(outFile, "// FollowCubeTables.cpp\n");
    fprintf(outFile, "// Copyright H. Carr 2003\n");
    fprintf(outFile, "// Marching Cube cases surface-specific contour-following (continuation)\n");
    fprintf(outFile, "// Generated %s, %s\n", __TIME__, __DATE__);
```

```
fprintf(outFile, "#include \"FollowCubeTables.h\"\n\n");

fprintf(outFile, "int nSurfaces[256] =\n\t{ // nSurfaces\n");
for (theCase = 0; theCase < 256; theCase+= 16)
     { // theCase loop
     for (theCase2 = theCase; theCase2 < theCase+16; theCase2++)
          fprintf(outFile, "\t%d%s", nSurfaces[cases[theCase2]], theCase2 == 255 ? "" : ",");
     fprintf(outFile, "\n");
     } // theCase loop
fprintf(outFile, "\t}; // nSurfaces\n\n");

fprintf(outFile, "int nTriangles[256][4] =\n\t{ // nTriangles\n");
for (theCase = 0; theCase < 256; theCase++)
     { // theCase loop
     fprintf(outFile, "\t\t//\tCase %d (base case %s)\n", theCase, caseName[cases[theCase]]);
     fprintf(outFile, "\t\t{");
     for (theSurface = 0; theSurface < nSurfaces[cases[theCase]]; theSurface++)
          fprintf(outFile, "\t%d%s", nTriangles[cases[theCase]][theSurface], theSurface == nSurfaces[cases[theCase]] - 1 ? "\t" : ",");
     fprintf(outFile, "}%s\n", theCase == 255 ? "" : ",");
     } // theCase loop
fprintf(outFile, "\t}; // nTriangles\n\n");

fprintf(outFile, "int mcFollowTriangles[256][4][15] =\n\t{ // mcFollowTriangles\n");
for (theCase = 0; theCase < 256; theCase++)
     { // theCase loop
     fprintf(outFile, "\t//\tCase %d (base case %s)\n", theCase, caseName[cases[theCase]]);
     fprintf(outFile, "\t\t{");
     for (theSurface = 0; theSurface < nSurfaces[cases[theCase]]; theSurface++)
          { // loop through surfaces
          fprintf(outFile, "\t{");
          for (whichTri = 0; whichTri < 3 * nTriangles[cases[theCase]][theSurface]; whichTri++)
               fprintf(outFile, "\t%d%s", triangles[theCase][theSurface][whichTri], (whichTri % 3) == 2 ? (whichTri == 3 * nTriangles[cases[theCase]][theSurface] - 1 ? "" : ",\t") : ",");
          fprintf(outFile, "\t}%s", theSurface == nSurfaces[cases[theCase]] - 1 ? "\t" : ",");
          } // loop through surfaces
     fprintf(outFile, "}%s\n", theCase == 255 ? "\t" : ",");
     } // theCase loop
fprintf(outFile, "\t}; // mcFollowTriangles\n\n");

fprintf(outFile, "int seedEdge2Surface[256][24] =\n\t{ // seedEdge2Surface\n");
for (theCase = 0; theCase < 256; theCase++)
     { // case loop
     fprintf(outFile, "\t//\tCase %d (base case %s)\n", theCase, caseName[cases[theCase]]);
     fprintf(outFile, "\t\t{");
     for (whichEdge = 0; whichEdge < 24; whichEdge++)
          { // whichEdge loop
          fprintf(outFile, "\t%d%s", seedEdge2Surface[theCase][whichEdge], whichEdge == 23 ? "\t" : ",");
          } // whichEdge loop
     fprintf(outFile, "}%s\n\n", theCase == 255 ? "\t" : ",");
     } // case loop
fprintf(outFile, "\t}; // seedEdge2Surface\n\n");

fprintf(outFile, "int nExitEdges[256][4] = \n\t{ // nExitEdges\n");
for (theCase = 0; theCase < 256; theCase++)
     { // theCase loop
     fprintf(outFile, "\t\t//\tCase %d (base case %s)\n", theCase, caseName[cases[theCase]]);
     fprintf(outFile, "\t\t{");
     for (theSurface = 0; theSurface < nSurfaces[cases[theCase]]; theSurface++)
          fprintf(outFile, "\t%d%s", nExitEdges[theCase][theSurface], theSurface == nSurfaces[cases[theCase]] - 1 ? "\t" : ",");
     fprintf(outFile, "}%s\n", theCase == 255 ? "" : ",");
     } // theCase loop
fprintf(outFile, "\t}; // nExitEdges\n\n");

fprintf(outFile, "int surface2exitEdges[256][4][7] =\n\t{ // surface2exitEdges\n");
for (theCase = 0; theCase < 256; theCase++)
     { // theCase loop
     fprintf(outFile, "\t\t//\tCase %d (base case %s)\n", theCase, caseName[cases[theCase]]);
     fprintf(outFile, "\t\t{");
     for (theSurface = 0; theSurface < nSurfaces[cases[theCase]]; theSurface++)
          { // loop through surfaces
          fprintf(outFile, "\t{");
          for (whichEdge = 0; whichEdge < nExitEdges[theCase][theSurface]; whichEdge++)
               fprintf(outFile, "\t%d%s", exitEdges[theCase][theSurface][whichEdge], whichEdge == nExitEdges[theCase][theSurface] - 1 ? "\t" : ",");
          fprintf(outFile, "\t}%s", theSurface == nSurfaces[cases[theCase]] - 1 ? "\t" : ",");
          } // loop through surfaces
     fprintf(outFile, "}%s\n", theCase == 255 ? "\t" : ",");
     } // theCase loop
fprintf(outFile, "\t}; // surface2exitEdges\n\n");

fprintf(outFile, "int exit2entryEdge[36] = \n");
fprintf(outFile, "\t{ //\texit2entryEdge\n");
fprintf(outFile, "\t35,\t34,\t33,\t32,\t31,\t30,\n");
fprintf(outFile, "\t29,\t28,\t27,\t26,\t25,\t24,\n");
fprintf(outFile, "\t23,\t22,\t21,\t20,\t19,\t18,\n");
fprintf(outFile, "\t17,\t16,\t15,\t14,\t13,\t12,\n");
fprintf(outFile, "\t11,\t10,\t9,\t8,\t7,\t6,\n");
fprintf(outFile, "\t5,\t4,\t3,\t2,\t1,\t0\n");
fprintf(outFile, "\t}; //\texit2entryEdge\n\n");

fprintf(outFile, "int exitDirection[36][3] = \n");
fprintf(outFile, "\t{ //\texitDirection\n");
fprintf(outFile, "\t -1,  0,  0,\t -1,  0,  0,\t -1,  0,  0,\t -1,  0,  0,\t -1,  0,  0,\t -1,  0,  0,\n");
fprintf(outFile, "\t  0,  0,  1,\t  0,  0,  1,\t  0,  0,  1,\t  0,  0,  1,\t  0,  0,  1,\t  0,  0,  1,\n");
fprintf(outFile, "\t  0,  1,  0,\t  0,  1,  0,\t  0,  1,  0,\t  0,  1,  0,\t  0,  1,  0,\t  0,  1,  0,\n");
fprintf(outFile, "\t  0, -1,  0,\t  0, -1,  0,\t  0, -1,  0,\t  0, -1,  0,\t  0, -1,  0,\t  0, -1,  0,\n");
fprintf(outFile, "\t  0,  0, -1,\t  0,  0, -1,\t  0,  0, -1,\t  0,  0, -1,\t  0,  0, -1,\t  0,  0, -1,\n");
fprintf(outFile, "\t  1,  0,  0,\t  1,  0,  0,\t  1,  0,  0,\t  1,  0,  0,\t  1,  0,  0,\t  1,  0,  0,\n");
fprintf(outFile, "\t}; //\texitDirection\n\n");

fprintf(outFile, "int entryEdge2Surface[256][36] = \n\t{ // entryEdge2Surface\n");
for (theCase = 0; theCase < 256; theCase++)
```

```
	{ // theCase loop
	fprintf(outFile, "\t\t//\tCase %d (base case %s)\n", theCase, caseName[cases[theCase]]);
	fprintf(outFile, "\t\t{\n\t\t");
	for (whichEdge = 0; whichEdge < 36; whichEdge++)
		fprintf(outFile, "\t%2d%s", entryEdge2Surface[theCase][whichEdge], (whichEdge % 12) != 11 ? "," : whichEdge == 35 ? "\n" : ",\n\t\t");
	fprintf(outFile, "\t\t}%s\n", theCase == 255 ? "\t" : ",");
	} // theCase loop
fprintf(outFile, "\t}; // entryEdge2Surface\n\n");

fprintf(outFile, "int vertex2edge[8][8] = \t\t\t\t\t\t\t\t\t\t\t\t\t//\ttable defining the possible edges\n");
fprintf(outFile, "\t{ // vertex2edge\t\t\t\t\t\t\t\t\t\t\t\t\t//\t-1: no edge\n");
fprintf(outFile, "\t{\t-1,\t0,\t3,\t12,\t\t4,\t20,\t19,\t-1\t},\t\t\t\t\t\t\t//\tedges starting from 000\n");
fprintf(outFile, "\t{\t0,\t-1,\t13,\t1,\t\t21,\t5,\t-1,\t14\t},\t\t\t\t\t\t\t//\tedges starting from 001\n");
fprintf(outFile, "\t{\t3,\t13,\t-1,\t2,\t\t18,\t-1,\t7,\t16\t},\t\t\t\t\t\t\t//\tedges starting from 010\n");
fprintf(outFile, "\t{\t12,\t1,\t2,\t-1,\t\t-1,\t15,\t17,\t6\t},\t\t\t\t\t\t\t//\tedges starting from 011\n");
fprintf(outFile, "\n");
fprintf(outFile, "\t{\t4,\t21,\t18,\t-1,\t\t-1,\t8,\t11,\t22\t},\t\t\t\t\t\t\t//\tedges starting from 100\n");
fprintf(outFile, "\t{\t20,\t5,\t-1,\t15,\t\t8,\t-1,\t23,\t9\t},\t\t\t\t\t\t\t//\tedges starting from 101\n");
fprintf(outFile, "\t{\t19,\t-1,\t7,\t17,\t\t11,\t23,\t-1,\t10\t},\t\t\t\t\t\t\t//\tedges starting from 110\n");
fprintf(outFile, "\t{\t-1,\t14,\t16,\t6,\t\t22,\t9,\t10,\t-1\t}\t\t\t\t\t\t\t//\tedges starting from 111\n");
fprintf(outFile, "\t}; // vertex2edge\n");

fprintf(outFile, "int mcFollowVertexCoords[8][3] =\n");
fprintf(outFile, "\t{//\tmcFollowVertexCoords\n");
fprintf(outFile, "\t\t{0, 0, 0},\n");
fprintf(outFile, "\t\t{0, 0, 1},\n");
fprintf(outFile, "\t\t{0, 1, 0},\n");
fprintf(outFile, "\t\t{0, 1, 1},\n");
fprintf(outFile, "\t\t{1, 0, 0},\n");
fprintf(outFile, "\t\t{1, 0, 1},\n");
fprintf(outFile, "\t\t{1, 1, 0},\n");
fprintf(outFile, "\t\t{1, 1, 1}\n");
fprintf(outFile, "\t}; //\tmcFollowVertexCoords\n\n");

fprintf(outFile, "int edgeVertices[24][2] = \t\t\t\t\t\t\t\t\t\t\t\t//\ttable with the vertices each edge connects\n");
fprintf(outFile, "\t{ // edgeVertices\n");
fprintf(outFile, "\t0,\t1,\t\t1,\t3,\t\t3,\t2,\t\t2,\t0,\t\t\t\t\t\t\t//\te0 - e3\n");
fprintf(outFile, "\t0,\t4,\t\t1,\t5,\t\t3,\t7,\t\t2,\t6,\t\t\t\t\t\t\t//\te4 - e7\n");
fprintf(outFile, "\t4,\t5,\t\t5,\t7,\t\t7,\t6,\t\t6,\t4,\t\t\t\t\t\t\t//\te8 - e11\n");
fprintf(outFile, "\t\n");
fprintf(outFile, "\t0,\t3,\t\t1,\t2,\t\t1,\t7,\t\t3,\t5,\t\t\t\t\t\t\t//\td0 - d3 (e12 - e15)\n");
fprintf(outFile, "\t2,\t7,\t\t3,\t6,\t\t4,\t2,\t\t0,\t6,\t\t\t\t\t\t\t//\td4 - d7 (e16 - e19)\n");
fprintf(outFile, "\t0,\t5,\t\t1,\t4,\t\t4,\t7,\t\t5,\t6\t\t\t\t\t\t\t//\td8 - d11 (e20 - e23)\n");
fprintf(outFile, "\t}; // edgeVertices\n\n");

fprintf(outFile, "char *caseName[23] = {\t\"0\", \"1\", \"2\", \"3\", \"4\", \"5\", \"6\", \"7\",\n");
fprintf(outFile, "\t\t\t\t\t\t\"8\", \"9\", \"10\", \"10M\", \"11\", \"12\", \"13\",\n");
fprintf(outFile, "\t\t\t\t\t\t\"7C\", \"6C\", \"5C\", \"4C\", \"3C\", \"2C\", \"1C\", \"0C\" };\n\n");

fprintf(outFile, "int baseCase[256] =\n\t{ // baseCase\n");
for (theCase = 0; theCase < 256; theCase+= 16)
	{ // theCase loop
	for (theCase2 = theCase; theCase2 < theCase+16; theCase2++)
		fprintf(outFile, "\t%d%s", cases[theCase2], theCase2 == 255 ? "" : ",");
	fprintf(outFile, "\n");
	} // theCase loop
fprintf(outFile, "\t}; // baseCase\n\n");

fclose(outFile);

return 0;
} // main()
```

Appendix B

Code for Piecewise Continuation

The code in this appendix implements piecewise continuation as described in Section 12.8, using the tables generated by the code in Appendix A.

There are three principal routines:

PiecewiseContinuation() takes a surface fragment as an argument, then uses a queue-based implementation of Algorithm 12.3 to extract the contour surface passing through that surface fragment.

SeedPiecewiseContinuation() takes a seed edge, converts it into a surface fragment, invokes **PiecewiseContinuation()** to perform the actual extraction, then invokes **RemoveFlags()** to remove the flags set during the contour extraction.

RemoveFlags(), like **PiecewiseContinuation()**, follows the surface. Unlike **PiecewiseContinuation()**, **RemoveFlags()** does not render the surface. Instead, it resets the flags that mark which fragments have been visited.

Several other routines are assumed, but not shown: **RenderTriangle()**, which renders a single triangle, **height(x, y, z)**, which retrieves an isovalue of a sample, **Visited(x, y, z, s)**, which tests whether surface fragment s has been visited in cell (x, y, z), and **Visit(x, y, z, s)** and **UnVisit(x, y, z, s)** which set and clear flags for surface fragments.

B.1 The Code

```
#include <queue>

//   include the tables generated by the code in Appendix A
#include "FollowCubeTables.h"

//   a little class defining a single surface in a cell
class SurfaceFragment
     {    // class SurfaceFragment
     public:
     long x, y, z, entryFaceEdge;
     SurfaceFragment(long X, long Y, long Z, long EntryFaceEdge)
          { x = X; y = Y; z = Z; entryFaceEdge = EntryFaceEdge; }
     };   // class SurfaceFragment

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

```
//                                                                                                                                  //
//    PiecewiseContinuation() implements piecewise continuation, seeded by a given surface in a given cell                         //
//                                                                                                                                  //
//    It does so by converting the seed edge to a seed cell + a surface ID, then invoking a routine called PiecewiseContinuation() //
//    to generate the surface itself.  Finally, it calls a routine called RemoveFlags() to remove the flags marking which cells    //
//    and surfaces have been visited.                                                                                               //
//                                                                                                                                  //
//    The following subroutines are assumed:                                                                                        //
//         height(x, y, z)          retrieves the isovalue at sample (x, y, z)                                                      //
//         Visited(x, y, z, s)      tests whether surface s in cell (x, y, z) has already been visited                              //
//         Visit(x, y, z, s)        marks surface s in cell (x, y, z) as already visited                                            //
//         RenderTriangle()         renders a single triangle belonging to the surface fragment                                     //
//                                                                                                                                  //
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void PiecewiseContinuation(double ht, long x, long y, long z, int theEntryFaceEdge)                     //   starts drawing cube at specified edge
     { // PiecewiseContinuation()
     long theCase = 0;                                                                                  //   the marching cubes case
     long theSurface;                                                                                   //   the surface the edge belongs to
     long whichExitFaceEdge;                                                                            //   index for exit edges
     long theExitFaceEdge;                                                                              //   the exit edge & entry in next cube
     double cubeVert[8];                                                                                //   local array holding vertices of cube
     long whichVertex;                                                                                  //   index for loop through vertices
     long whichTri;                                                                                     //   index for loop through triangles
     queue<SurfaceFragment> theQueue;                                                                   //   queue of surfaces to be processed.

     //   initialize the queue
     SurfaceFragment theSurfaceFragment = SurfaceFragment(x, y, z, theEntryFaceEdge);                   //   cell surface being processed
     theQueue.push(theSurfaceFragment);                                                                 //   push the initial cell surface on queue

     //   loop through the queue, processing surface fragments
     while (not theQueue.empty())                                                                       //   while the queue is non-empty
          { // loop to empty queue
          theSurfaceFragment = theQueue.front();                                                        //   grab the front item
          theQueue.pop();                                                                               //   and pop to remove it
          x = theSurfaceFragment.x; y = theSurfaceFragment.y; z = theSurfaceFragment.z;                 //   store these in local variables
          if ((x < 0) || (y < 0) || (z < 0)) continue;                                                  //   check for out-of bounds
          if ((x > xDim-2) || (y > yDim-2) || (z > zDim-2)) continue;

          //   compute the marching cubes case
          cubeVert[0] = height(x, y, z);             cubeVert[1] = height(x, y, z+1);                   //   copy the vertex heights
          cubeVert[2] = height(x, y+1, z);           cubeVert[3] = height(x, y+1, z+1);
          cubeVert[4] = height(x+1, y, z);           cubeVert[5] = height(x+1, y, z+1);
          cubeVert[6] = height(x+1, y+1, z);         cubeVert[7] = height(x+1, y+1, z+1);

          theCase = 0;
          for (whichVertex = 0; whichVertex < 8; whichVertex++)                                         //   loop through corners, computing facet lookup
               if (ht < cubeVert[whichVertex])                                                          //   if the corner is above desired height
                    theCase |= (1 << whichVertex);                                                      //   set bit flag

          //   find which surface the entry edge belongs to, and check whether it has already been processed
          theSurface = entryEdge2Surface[theCase][theSurfaceFragment.entryFaceEdge];                    //   find the surface we are on
          if (Visited(x, y, z, theSurface)) continue;                                                   //   if it's been flagged, skip it
          Visit(x, y, z, theSurface);                                                                   //   mark the surface as "visited"

          //   render the surface fragment
          for (whichTri = 0; whichTri < 3*nTriangles[theCase][theSurface]; whichTri+= 3)                //   walk through the triangles of the surface
               RenderTriangle(ht, x, y, z, cubeVert,
                              mcFollowTriangles[theCase][theSurface][whichTri + 0],
                              mcFollowTriangles[theCase][theSurface][whichTri + 1],
                              mcFollowTriangles[theCase][theSurface][whichTri + 2]);

          //   now follow the contour out each face
          for (whichExitFaceEdge = 0; whichExitFaceEdge < nExitEdges[theCase][theSurface]; whichExitFaceEdge++)
               { // for each exit edge
               theExitFaceEdge = surface2exitEdges[theCase][theSurface][whichExitFaceEdge];             //   find the exit edge's ID

               theQueue.push(SurfaceFragment(
                    x + exitDirection[theExitFaceEdge][0],                                              //   first entry in row gives delta x
                    y + exitDirection[theExitFaceEdge][1],                                              //   second entry gives delta y
                    z + exitDirection[theExitFaceEdge][2],                                              //   third entry gives delta z
                    exit2entryEdge[theExitFaceEdge]));                                                  //   convert from exit to entry
               } // for each exit edge
          } // loop to empty queue
     } // PiecewiseContinuation()

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//                                                                                                                                  //
//    SeedPiecewiseContinuation() generates a single contour at isovalue "ht", starting from the seed edge                          //
//    (x1, y1, z1) -> (x2, y2, z2)                                                                                                  //
//                                                                                                                                  //
//    It does so by converting the seed edge to a seed cell + a surface ID, then invoking a routine called PiecewiseContinuation() //
//    to generate the surface itself.  Finally, it calls a routine called RemoveFlags() to remove the flags marking which cells    //
//    and surfaces have been visited.                                                                                               //
//                                                                                                                                  //
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void SeedPiecewiseContinuation(double ht, long x1, long y1, long z1, long x2, long y2, long z2)         //   routine to follow surface from seed edge
     { // SeedPiecewiseContinuation()
     long xm, ym, zm;                                                                                   //   minimum x, y, z of the two endpoints
     long xc, yc, zc;                                                                                   //   the ID of a cell shared by the endpoints
     int i1, i2;                                                                                        //   the vertex IDs of the endpoints in the cell
     long whichCubeEdge, theSurface, whichFaceEdge;                                                     //   used to figure out where to start
     long theCase = 0;                                                                                  //   the marching cubes case
     long whichVertex;                                                                                  //   loop index for vertices
     double cubeVert[8];                                                                                //   local array holding vertices of cube

     //   find the common cell to which the endpoints belong and make sure it's in bounds
     xm = x1 < x2 ? x1 : x2;        ym = y1 < y2 ? y1 : y2;        zm = z1 < z2 ? z1 : z2;              //   take the minimum in each dimension
     xc = xm;                       yc = ym;                       zc = zm;                             //   and work out which cell we are in
```

```
    if (xc == xDim - 1) xc--;     if (yc == yDim - 1) yc--;     if (zc == zDim - 1) zc--;              //   make sure we stay inside a legal cell

    //   compute indices of vertices with respect to this cell
    i1 = ((x1 - xc) << 2) + ((y1 - yc) << 1) + (z1 - zc);       i2 = ((x2 - xc) << 2) + ((y2 - yc) << 1) + (z2 - zc);
                                                                                                       //   compute indices
    //   look up the edge ID in the cell
    whichCubeEdge = vertex2edge[i1][i2];                                                               //   find which edge we are on in the cell
    if (whichCubeEdge == -1)                                                                           //   error-check for bad edge
        { printf("Major problem: %ld to %ld is not a valid seed edge\n", i1, i2); return;}

    //   now figure out which marching cube case we have
    if ((xc < 0) || (yc < 0) || (zc < 0)) return;                                                      //   do nothing if any index negative
    if ((xc > xDim-2) || (yc > yDim-2) || (zc > zDim-2)) return;                                       //   ditto for last face worth of vertices

    cubeVert[0] = height(xc, yc, zc);       cubeVert[1] = height(xc, yc, zc+1);
    cubeVert[2] = height(xc, yc+1, zc);     cubeVert[3] = height(xc, yc+1, zc+1);
    cubeVert[4] = height(xc+1, yc, zc);     cubeVert[5] = height(xc+1, yc, zc+1);
    cubeVert[6] = height(xc+1, yc+1, zc);   cubeVert[7] = height(xc+1, yc+1, zc+1);

    theCase = 0;
    for (whichVertex = 0; whichVertex < 8; whichVertex++)                                              //   loop through corners, computing facet lookup
        if (ht < cubeVert[whichVertex])                                                                //   if the corner is above desired height
            theCase |= (1 << whichVertex);                                                             //   set bit flag

    //   use a lookup table to find which surface the seed edge intersects
    theSurface = seedEdge2Surface[theCase][whichCubeEdge];                                             //   find the surface to which be belong
    if (theSurface == -1)                                                                              //   error check for bad surface
        { printf("Major problem: Edge %ld in case %ld is not a valid seed edge\n", whichCubeEdge, theCase); return;}

    //   and set the edge we "entered" the cell through to any valid edge
    whichFaceEdge = surface2exitEdges[theCase][theSurface][0];                                         //   take the first exit edge (arbitrarily)


    PiecewiseContinuation(ht, xc, yc, zc, whichFaceEdge);                                              //   start the surface off with the given edge
    RemoveFlags(ht, xc, yc, zc, whichFaceEdge);                                                        //   clean out the flags
    } // SeedPiecewiseContinuation()

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//                                                                                                                                      //
//   RemoveFlags() resets the flags set by piecewise continuation                                                                       //
//                                                                                                                                      //
//   It does so following the surface in the same way as PiecewiseContinuation(), resetting visitation flags instead of rendering //
//   triangles.                                                                                                                         //
//                                                                                                                                      //
//   The following subroutines are assumed:                                                                                             //
//       height(x, y, z)          retrieves the isovalue at sample (x, y, z)                                                           //
//       Visited(x, y, z, s)      tests whether surface s in cell (x, y, z) has already been visited                                   //
//       UnVisit(x, y, z, s)      marks surface s in cell (x, y, z) as already visited                                                 //
//                                                                                                                                      //
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void RemoveFlags(double ht, long x, long y, long z, int theEntryFaceEdge)                              //   starts drawing cube at specified edge
    { // RemoveFlags()
    long theCase = 0;                                                                                  //   the marching cubes case
    long theSurface;                                                                                   //   the surface the edge belongs to
    long whichExitFaceEdge;                                                                            //   index for exit edges
    long theExitFaceEdge;                                                                              //   the exit edge & entry in next cube
    double cubeVert[8];                                                                                //   local array holding vertices of cube
    long whichVertex;                                                                                  //   index for loop through vertices
    queue<SurfaceFragment> theQueue;                                                                   //   queue of surfaces to be processed.

    //   initialize the queue
    SurfaceFragment theSurfaceFragment = SurfaceFragment(x, y, z, theEntryFaceEdge);                   //   cell surface being processed
    theQueue.push(theSurfaceFragment);                                                                 //   push the initial cell surface on queue

    //   loop through the queue, processing surface fragments
    while (not theQueue.empty())                                                                       //   while the queue is non-empty
        { // loop to empty queue
        theSurfaceFragment = theQueue.front();                                                         //   grab the front item
        theQueue.pop();                                                                                //   and pop to remove it
        x = theSurfaceFragment.x; y = theSurfaceFragment.y; z = theSurfaceFragment.z;                  //   store these in local variables
        if ((x < 0) || (y < 0) || (z < 0)) continue;                                                   //   check for out-of bounds
        if ((x > xDim-2) || (y > yDim-2) || (z > zDim-2)) continue;

        //   compute the marching cubes case
        cubeVert[0] = height(x, y, z);             cubeVert[1] = height(x, y, z+1);                    //   copy the vertex heights
        cubeVert[2] = height(x, y+1, z);           cubeVert[3] = height(x, y+1, z+1);
        cubeVert[4] = height(x+1, y, z);           cubeVert[5] = height(x+1, y, z+1);
        cubeVert[6] = height(x+1, y+1, z);         cubeVert[7] = height(x+1, y+1, z+1);

        theCase = 0;
        for (whichVertex = 0; whichVertex < 8; whichVertex++)                                          //   loop through corners, computing facet lookup
            if (ht < cubeVert[whichVertex])                                                            //   if the corner is above desired height
                theCase |= (1 << whichVertex);                                                         //   set bit flag

        //   find which surface the entry edge belongs to, and check whether it has already been processed
        theSurface = entryEdge2Surface[theCase][theSurfaceFragment.entryFaceEdge];                     //   find the surface we are on
        if (not Visited(x, y, z, theSurface)) continue;                                                //   if it's been unflagged, skip it
        UnVisit(x, y, z, theSurface);                                                                  //   mark the surface as "visited"

        //   now follow the contour out each face
        for (whichExitFaceEdge = 0; whichExitFaceEdge < nExitEdges[theCase][theSurface]; whichExitFaceEdge++)
            { // for each exit edge
            theExitFaceEdge = surface2exitEdges[theCase][theSurface][whichExitFaceEdge];               //   find the exit edge's ID

            theQueue.push(SurfaceFragment(
                x + exitDirection[theExitFaceEdge][0],                                                 //   first entry in row gives delta x
                y + exitDirection[theExitFaceEdge][1],                                                 //   second entry gives delta y
                z + exitDirection[theExitFaceEdge][2],                                                 //   third entry gives delta z
                exit2entryEdge[theExitFaceEdge]));                                                     //   convert from exit to entry
```

```
        } // for each exit edge
    } // loop to empty queue
} // RemoveFlags()
```