Universitat Politècnica de Catalunya
Departament de Llenguatges i Sistemes Informàtics
Informàtica Gràfica
Programa de Doctorat de Software

Extracting shapes from binary grids

Antoni Chica Calaf

Advisors: Àlvar Vinacua and Pere Brunet

Barcelona, June 2007

Acknowledgments

First, I wish to express sincere appreciation to my advisors Àlvar Vinacua and Pere Brunet for their help, supervision and encouragement. Pere's many ideas and comments and Àlvar's enthusiasm and solid math contributions were really helpful for the performance of this thesis.

I also wish to thank Jarek Rossignac for his guidance during his visit to Barcelona as well as my 3 months stay at Georgia Tech.

I would want to show my gratitude to Isabel Navazo and Carlos Andújar. Isabel contributed her insights on visibility computation and much practical advice to the point of almost being my third advisor. Carlos lent his experience to solve many problems like the appearance-preserving rendering.

My lab mates have been a source of support, ideas, coding help and many time-wasting recreations. Some have moved for jobs, others have just arrived. Thanks go to all of them.

I would like to thank people at the Institute of Robotics and at the Virtual Reality Center for making my graduate experience very pleasurable, and the lab a very good place to do research. This research was funded by a grant of the Spanish Ministry of Science throughout my work.

Finally, my biggest thank you goes to my family and my parents specially for their endless patience and constant support. To them I dedicate this thesis.

Abstract

This work explores the retrieval of shape information from binary discretizations of solids. In particular, we study binary regular grids and the associated isosurface extraction using Marching Cubes. An algorithm to identify and globally solve topology ambiguity problems is proposed, which optimizes topological properties like the resulting number of handles. Once ambiguity has been figured out, a second voting-based system is used to identify large planar regions. These planar regions are kept fixed while a constrained bilaplacian smoothing algorithm, specifically designed to deal with isosurfaces, is applied to the rest of the model. Finally, to compensate for the feature insensitivity of Marching Cubes, a visibility approach is used to identify sharp features. The resulting features are reconstructed by adding new vertices, which are placed in the intersection of planes which locally approximate surfaces. Therefore, the final model is an accurate representation of the original solid that respects the original segmentation and divides it into flat and curved faces separated by sharp features. As a result, the presented set of tools may be useful for shape recognition, simplification, reverse engineering and manufacturing applications.

Contents

1 Introduction **1**

1.1 Problem statement 1

1.2 Contributions 4

2 Related work **6**

2.1 Point cloud reconstruction 6

2.2 Isosurface extraction 8

2.3 Topology disambiguation on Marching Cubes 10

2.3.1 Topological consistency 11

2.3.2 Topological correctness 12

2.4 Feature extraction 13

2.5 Smoothing 14

2.5.1 Mesh fairing 15

2.5.2 Isosurface smoothing 15

2.6 Extraction of flats 17

2.7 Simplification 18

2.7.1 Topology simplification 18

2.7.2 Geometry simplification 19

2.7.3 Appearance-preserving simplification 22

2.8 Reverse engineering 25

2.9 Visibility computation 26

3 Common concepts **28**

4 Topology disambiguation **38**

4.1 Topology in Marching Cubes 38

4.2 Degrees of freedom 38

4.3 Math tools 41

4.4 Global structures 42
4.5 Optimization algorithm 44
4.6 Slashing criteria and combinations 45
4.7 Results 47

5 Maximal tiles extraction **50**
5.1 Problem statement 50
5.2 Plane space parameterization properties 51
5.3 Stick neighborhoods 52
5.4 Local parameterization 53
5.5 Precomputing the dictionary 55
5.6 Global parameterization 58
5.7 Voting algorithm 63
5.8 Appearance-preserving application 66
5.9 Results 67

6 Pressing **71**
6.1 Retrieving shape from binary grids 71
6.2 Detection of flats and junctions 71
6.3 Smoothing 73
6.4 Recovering sharp features 78
6.5 Results 79

7 Visibility based feature extraction **82**
7.1 Visibility inside the discrete band 82
7.2 Unconstrained visibility map 84
7.3 Constrained visibility map 90
7.4 Feature extraction 91
7.5 Results 93

8 Conclusions & future work **96**

List of Figures

1.1 Example of a very simple 3D binary image 2

2.1 MC, DMC and MT examples 9
2.2 Effects of topology inconsistency and topology incorrectness 10
2.3 Feature insensitivity of the Marching Cubes algorithm 13
2.4 Isosurface smoothing 16
2.5 Extraction of flats from a model 17
2.6 Quadric error metrics simplification 19
2.7 Appearance-preserving simplification of a taxi model using impostors 23
2.8 Reverse engineering a model into a set of basic shapes 25

3.1 Isosurface extraction from the binary grids of a medical model and a CAD model. 29
3.2 Cuberille render of the previous models 30
3.3 Extraction of the piece of surface inside a cell 30
3.4 Discrete band as the set of mixed cells 31
3.5 Marching Cubes' 15 cell types 33
3.6 Traversal of the sticks 34
3.7 Reduced set of face combinations 34
3.8 Triangulations of cell on Figure 3.3 35

4.1 Border edges for some face configurations 39
4.2 X-face possible slashings 39
4.3 Disambiguations of an X-cube 40
4.4 Sheet triangulations 40
4.5 X-face graph 43
4.6 Topology optimization of the test cube 47
4.7 Topology optimization of the ship's engine model 48

5.1 Cell's edges intersection with a plane 51

5.2 Stick neighborhood and 4D cone of stabbing planes 53
5.3 Tetrahedron used for the local parameterization of planes 54
5.4 3D Planes represented as rays in our parameterization 55
5.5 Addition of an plane to a pyramid 56
5.6 3D region of 2D separating lines 56
5.7 First slice of a $5 \times 5 \times 4$ neighborhood determining its central stick . 57
5.8 Central stick of a $5 \times 5 \times 4$ neighborhood constraining it 57
5.9 Tetrahedron used for the global parameterization of planes 59
5.10 Two neighbor cubes of the global parameterization 62
5.11 Connected Cubes parameterization and the Gauss sphere 62
5.12 Maximal tile resulting from the voting algorithm 67
5.13 First 10 tiles of two models 68
5.14 Original taxi model 68
5.15 14 tiles of an impostor model of the taxi 68
5.16 The handycam impostor model 69

6.1 2D example of the Pressing algorithm 72
6.2 Junction points of a pump model 72
6.3 The bilaplacian as two laplacian steps 73
6.4 Smoothing and snapping 74
6.5 Movement of one vertex using front tracking 74
6.6 Multiple quads front tracking 75
6.7 Isocurve leaving a ray 75
6.8 76
6.9 Arc length resampling 77
6.10 Combining two slices to smooth 3D isosurfaces 77
6.11 Smoothing algorithm behavior near junctions 78
6.12 Subdivision of chamfer triangles 78
6.13 Corners generated from chamfer triangle's subdivision 79
6.14 Subdivision of chamfer edges 79
6.15 Smoothing biomedical models 80
6.16 Pressing results on a mechanical part 81
6.17 Pressing results on the fandisk model 81

7.1 Cell-to-cell visibility of the discretization of a pentagon 83
7.2 Visibility from faces, edges and vertices 83
7.3 2D examples of cell-to-cell visibility 84

7.4 Visibility propagating like a wave 85
7.5 Erosion-based visibility concept 86
7.6 Visibility computation for the examples shown in Figure 7.3 86
7.7 Unconstrained visibility count map of a very simple model 87
7.8 Unconstrained visibility map of a cube 87
7.9 Resolution effect on the unconstrained visibility count maps 88
7.10 Unconstrained visibility count maps of two mechanical parts 88
7.11 Visibility shadows caused by holes 89
7.12 Skeleton and extracted graph of an icosahedron model 89
7.13 Skeleton and graph of a mechanical part 90
7.14 Unconstrained and constrained visibility maps comparison 90
7.15 Visibility maps of an icosahedron and a mechanical part 91
7.16 Components of an icosahedron and a mechanical part 92
7.17 Classified components of an icosahedron and a mechanical part 92
7.18 Connectivity graphs of an icosahedron and a mechanical part 93
7.19 From left to right: visibility maps, identified components, its classification and the resulting connectivity graphs. From top to bottom: the models of the FANDISK, a CROSS pipe and another mechanical part (COVER). 94
7.20 Zoom of the highlight of the COVER mechanical part in Figure 7.19. 95

List of Tables

4.1 X-face graphs of some datasets 43
4.2 Number of vertices, triangles, shells, and handles of the test cube . . 48
4.3 Number of vertices, triangles, shells, and handles of the ship's engine 48

5.1 Faithfulness tests for Hough parameterization 60
5.2 Faithfulness tests for Connected Cubes parameterization 60
5.3 Times for a multigrid solution of two equal levels using $5 \times 5 \times 4$ neighborhoods 63
5.4 Times for a multigrid solution of two equal levels using $3 \times 3 \times 2$ neighborhoods 64
5.5 Times for a multigrid solution of two different size levels using $5 \times 5 \times 4$ neighborhoods 64
5.6 Times for a multigrid solution of two different size levels using $3 \times 3 \times 2$ neighborhoods 64
5.7 Times for the used multigrid solution using $5 \times 5 \times 4$ neighborhoods . 65
5.8 Times for the used multigrid solution using $3 \times 3 \times 2$ neighborhoods . 65
5.9 Tiles resulting from the test models 69
5.10 Running times of the maximal tiles retrieval 69

6.1 Running times of the Pressing algorithm 80
6.2 Error reduction of Pressing 81

7.1 Running times of visibility map computation 89
7.2 Resolution, number of mixed cells and running times of the four steps of the algorithm for the models shown in Figures 7.15 and 7.19. . . . 95

Chapter 1

Introduction

Even though continuous improvement of computer performance and computer graphics hardware allows to process larger models, many technical advances have contributed to produce ever increasingly larger data sets. Besides, algorithms that process this kind of data sets for simplification or repair depend on the number of primitives, normally triangles. This has caused many solutions to use a volumetric approach, thus making the efficiency dependent on the resolution only. Additionally, the resolution also determines the error threshold of the result.

Another advantage is that volumetric models have far fewer ambiguity problems when compared to polygonal ones. For example, non-manifold situations are a common problem on polygonal objects. Furthermore, we have access to many techniques developed for image processing since volumetric data has the same regular structure as images.

In spite of their advantages the use of volumetric models may also result in artifacts. High spatial frequencies, which indicate features in the model, result in jagged and irregular surfaces. Irregular sampling may cause a terracing effect to appear when an isosurface is extracted from a distance map. Finally, significant variations of both magnitude and direction of the image's gradient difficult a quality extraction of an isosurface.

1.1 Problem statement

The problem we are trying to solve may be described as follows: given a 3D binary image of a given model, we want to recover an error-bounded surface which matches the original "shape". As 'shape' we understand a set of sharp features which separate different smooth regions (which may be planes or other surfaces). A 3D binary image

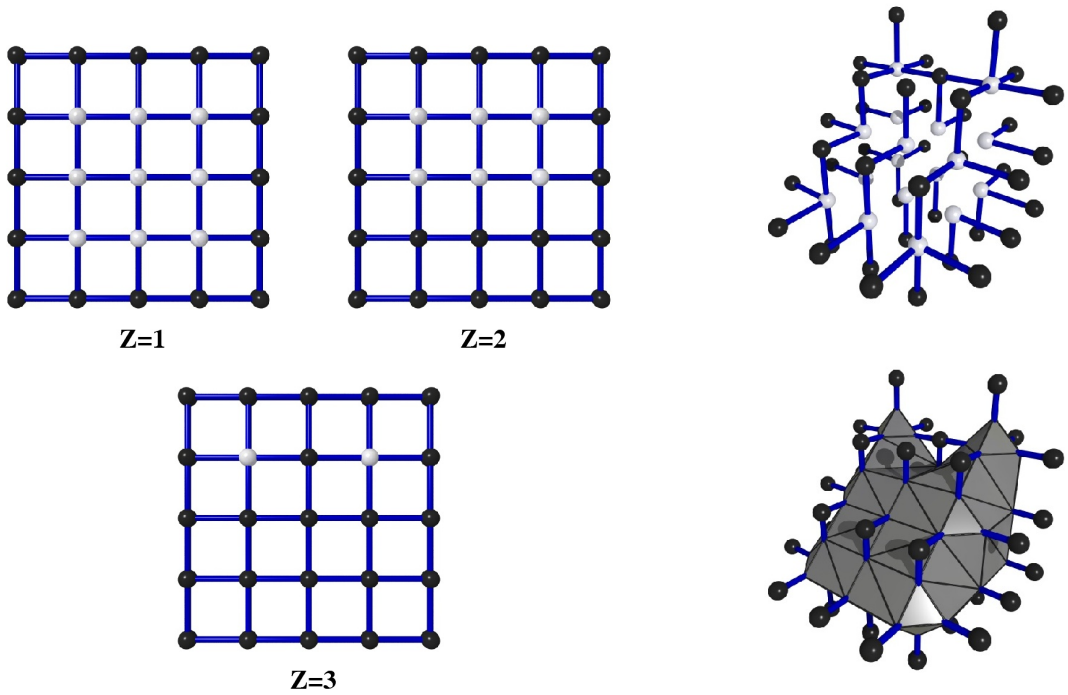


Figure 1.1: White points represent interior nodes, while black ones represent exterior nodes. The extracted surface separates black and white points.

and its extracted surface can be seen in Figure 1.1.

A reasonable question to ask is why do we limit the input to binary image data, if most applications use scalar fields. The answer is that we do not want the resulting extraction algorithm to deal with volumetric data only, but with many kinds of source data. Generating a quality 3D binary image can be easier than generating a full 3D distance field, so only segmentation results are considered. In fact, if a solution is found for the previous problem, it can be directly applied to scalar fields. Therefore, the number of considered cases is greater, even though the problem becomes more complex.

As we will see in Chapter 6, the techniques described here are mostly oriented to recover the shape of mechanical parts. Voxelizations of such models may be obtained from laser scans, which result in large point clouds. As such, point clouds could be directly processed to obtain the original shape without any need to use voxelizations. Despite that, NDT (Non-Destructive Testing) applications [HMM+03] where a complex object, made of several parts, needs to be scanned without disassembling it, yield voxelizations. This is caused by the fact that CT scanners are used to obtain the data.

There are other reasons to choose binary grids. Binary voxelizations are commonly generated from medical data segmentation when an organ or system has to be obtained. Moreover, some volume operations like splitting produce binary information on the modified regions. Even reconstruction and repair approaches use algorithms which are based on binary discretizations.

The best known algorithm to produce a surface from a given field (binary or scalar) is Marching Cubes. Using the signs of the binary image or the values of the scalar field on the vertices of the cubes, the surface is built, one cube at a time. In the case of scalar fields a threshold is used to determine if the vertices of one cube are interior or exterior, and the resulting configuration is used to obtain a surface that separates them correctly. But not every possible configuration is unambiguous, some of them have different possible solutions. This choices affect the topology of the resulting surface, but many of the variants of the Marching Cubes algorithm solve them in an arbitrary way.

Several alternatives have been proposed to give a solution to this topology ambiguity problem, but many of them rely on local criteria, making them unable to control the topological properties of the final extracted mesh. A global solution is needed, but not at the cost of the Marching Cubes efficiency. Therefore, we need an algorithm to solve this ambiguities, seeking to optimize certain topological properties like the triangle count, the total number of connected components, or the genus of the mesh.

Once the topological problem has been solved, the surface can be extracted using Marching Cubes, but this method results in an excessive data fragmentation. Simplification techniques yield good results, but do not respect the original classification of the nodes. After the simplification, some interior nodes may become exterior to the final surface, and vice versa.

The strategy followed here is to apply a face clustering simplification algorithm through the computation of large planar zones (that we call tiles), while smoothing the rest of the surface and finding the original features. Computing the largest planar region that approximates the surface with a bounded error has other applications, such as finding object signatures, and impostor-based simplification.

In the meantime, the extracted model can be used for reverse engineering applications.

1.2 Contributions

Dealing with the previous problem, this document makes the following contributions:

- Identification of the topological degrees of freedom of the Marching Cubes algorithm.
- Design of two data structures which capture the topological properties of the final extracted isosurface.
- A variant of Marching Cubes which uses those two structures to optimize the topological properties of the isosurface, while guaranteeing that its triangles are not on any of the grid's faces, thus ensuring it is a two-manifold.
- A representation for the set of planes which separate a set of black nodes from a set of white nodes.
- An algorithm that efficiently obtains a plane that separates the largest number of adjacent black and white nodes from a discrete representation of a model.
- A method which iteratively obtains the largest approximating plane, thus obtaining a set of planes that cover the model with a bounded error.
- Computation of a set of impostors from a textured model and the set of planes obtained by the previous algorithm.
- Combination of shape segmentation and smoothing, while preserving the identified features, which will be enhanced later.
- A smoothing operator is presented that may be applied in the presence of equality and inequality constraints. These constraints are used to prevent the surface from leaving an interior node outside or viceversa.
- An erosion based visibility algorithm, optimized to compute cell to cell visibility in a regular grid subdivision of space.
- Using visibility to selectively identify features and regions in a binary discretization. This functionality may deal with curved regions.
- As a result of the visibility-based feature extraction, a topological graph of the corresponding shape is generated. This graph represents the topological connection between different elements of the model: its vertices, edges and faces.

This combination of contributions leads to an error-bounded reconstruction which is segmented into planar and smooth regions, separated by sharp features. Furthermore, the shape segmentation may ease subsequent shape identification for reverse engineering applications.

Chapter 2

Related work

Recovering models through reconstruction techniques is an important problem in reverse engineering, manufacturing and shape recognition. Input data for such techniques may be obtained from a variety of sources, one of them being laser scans. As a result large point clouds need to be reconstructed to obtain a model which faithfully represents the scanned object. Another possibility is using CT or MRI scanners to obtain regular grids and apply isosurface reconstruction methods. This approach is particularly useful for biomedical models or mechanical models which cannot be disassembled prior to the scanning process (Non-Destructive Testing [HMM+03]). Both kinds of techniques benefit greatly from feature detection, as shape may be more properly recovered near them, if they have been previously extracted. In order to detect the model's features we will use a novel method which applies visibility computation.

Once a model has been obtained, it may need to be postprocessed to remove noise inherent to the scanning process. Multiple smoothing algorithms have been introduced to deal with this problem. Over-tesselation also needs to be solved with the aid of simplification methods. Finally, planar and curved shapes are identified for reverse engineering applications.

All this related work will be presented before introducing our contributions.

2.1 Point cloud reconstruction

Point cloud reconstruction is a classic problem in the computer graphics field which has been well studied. The multiple solutions given to this problem may be classified according to the internal representations used during the process.

Methods that use implicit functions to represent the surface of models were pio-

neered by the work of Hoppe et al [HDD*92]. PCA (Principal Component Analysis) is used to estimate normals whose directions are then unified by means of neighborhood graph search. These normals define an implicit signed distance function from which the model is obtained applying Marching Cubes (the zero levelset is extracted).

Regular implicit solutions use weighted sums of basis functions which are radially simetric, but this assumes that the surface is locally simetric everywhere. As a result sharp features are not reconstructed properly. Anisotropic basis functions are an alternative. Work by Dinh et al [DTS01] identifies the direction of anisotropy using principal component analysis of the data points in a local neighborhood. Tensor filtering is then applied to smooth the obtained field of principle directions.

Earlier work on implicit functions [TO99] used globally defined basis functions, which leads to quadratic complexity. This complexity may be reduced using a fast multipole method [CBC*01]. A greedy algorithm reduces the number of radial basis functions centers which results in a compression and computational advantages. The polyharmonic splines used fill holes and extrapolate smooth surfaces which allows to apply this strategy to automatic repair of meshes.

Another possibility is to locally approximate the surface using Moving Least Squares (MLS) [ABC*02]. MLS techniques define a projection operator which only needs a local subset of the point cloud to be computed. The reconstructed surface is defined as the invariant set of the projection operator and it is a smooth manifold with a bounded sampling error with respect to the original surface. Fleishman et al [FCO*05] apply robust statistics to identify outliers and exclude them from the fitting process. Feature lines are thus preserved, but their topology is not extracted explicitly.

The reconstruction problem may also be solved from a computational geometry point of view. In [ABK98] the 2D Voronoi diagram of a 2D sample point set is used to generate a planar graph called the crust. This graph is defined to be composed of the edges in the Delaunay triangulation that can be enclosed by circles empty not only of sample points, but of Voronoi vertices as well. This resulting crust is guaranteed to contain all the edges between adjacent sample points, but only if the curve was "well-sampled". A sample is good if its sampling density is at least inversely proportional to the distance to the medial axis. Thus, the resulting algorithm is heavily influenced by noise.

The Power Crust [ACK01] also follows this direction. It approximates the medial axis transform (MAT) of 3D object with a finite union of balls. This approximate MAT is then used to define a piecewise-linear approximation of the object surface,

which is called the power crust. The approach guarantees a geometrically and topologically correct reconstruction of the original surface if it is sufficiently sampled. An improved version of the Power Crust algorithm which produces correct outputs in the presence of noise is also presented in [MAV*05].

All of the previous techniques do not explicitly extract the features of the model. Gumhold et al [GWM01] propose extracting the features, as a preprocessing step, to improve the reconstruction process. Their technique does not need a reconstruction, but it works on a neighborhood graph that connects nearby points. Each of the edges on this graph is given a weight that matches the unlikelihood of it being on a feature. Then, a subgraph is selected by minimizing the edge penalty weights. Finally, the features are extracted by fitting wedges and corners to crease lines and junctions, respectively.

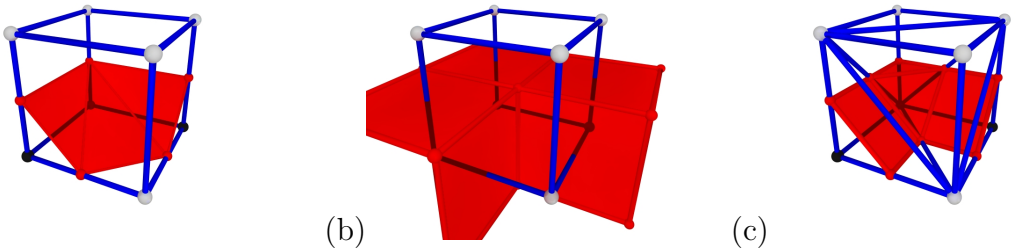
The Bayesian point cloud reconstruction method by Jenke et al [JWB+06] uses statistical techniques to recover surfaces. A Gaussian error model is combined with a set of prior assumptions about the surface to be reconstructed. These prior assumptions are expressed as probability distributions, which are used to infer a maximum probability reconstruction by means of Bayes' rule. This process is based in two fundamental steps: a numerical optimization that maximizes the probability neglecting the discrete components (features) and a discrete optimization which optimizes the position of edge, region and corners points. As a result, both topology and geometry are extracted.

2.2 Isosurface extraction

The use of the Marching Cubes (MC) technique (Figure 2.1), originally proposed by Lorensen and Cline [LC87], is nowadays considered to be the standard approach to the problem of extracting isosurfaces from a volumetric dataset. The algorithm works by defining cells as the space between 8 voxels, and the whole isosurface is extracted incrementally from those cubes. This step is called marching and it only extracts topology, while a second step positions the vertices along the edges that connect positive and negative voxels, interpolating their values linearly.

Despite its extensive use in many applications, some defects have been pointed out regarding topological consistency and correctness, the excessive amount of data generated as output, and its inability to restore sharp features.

As the fragmentation of the final isosurface may prevent the interactive rendering, adaptive solutions have been proposed to deal with this problem. Discretized



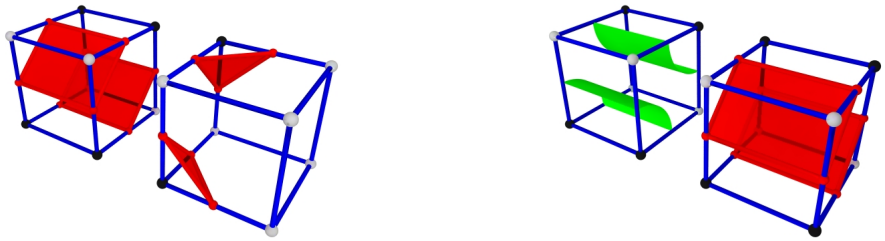
(a) (b) (c)

Figure 2.1: Three strategies used to extract isosurfaces: (a) Marching Cubes, (b) Dual Marching Cubes, and (c) Marching Tetrahedra

Marching Cubes (DMC, [MSS*94], [MSS00]) is one of them. Where Marching Cubes applies two steps: marching and interpolating, DMC applies first a discrete marching step, then it merges the resulting primitives, and finally it positions the remaining vertices using linear interpolation. During the marching step, all the configurations consider the vertices to be at the midpoints of their sticks, thus classifying all the generated primitives in different planes. Those planes are used to merge primitives into larger faces. Reductions of 50% of the number of primitives are obtained on average.

Other strategies try to propose solutions which solve all the MC problems with one single algorithm. Such is the case of Cubical Marching Squares [HWCC05], which solves the topological inconsistency problem, removes cracks that appear due to adaptive resolution representations, and restores sharp features. In order to do that, the authors use an adaptive grid structure, an octree, where transition faces (those between different resolution cells) are subdivided using the higher resolution of their neighbors, and subsequently used to compute the final surface edges. Those edges are arranged into cycles for every cell, and triangulated afterwards. 2D features are recovered during the edges computation and 3D features during cell's surface extraction. Even though the technique is very powerful, its needs Hermite data as input (exact intersection points and their normals).

Dual Marching Cubes techniques have been proposed as an alternative to MC (also shown in Figure 2.1). One vertex is extracted from each cell and quadrilateral faces are extracted such that they stab edges connecting positive and negative voxels. As a drawback, non-manifolds may result from Dual MC algorithms. Ju et al [JLSW02] present a method which uses Hermite data as input, computing an isosurface by applying adaptive dual contouring. Their solution can handle multi-material discretizations and obtain a simplified mesh using quadric error metrics to



(a) (b)

Figure 2.2: (a) Inconsistent decisions on two neighboring cells regarding their connecting face may lead to holes, while (b) incorrect decisions on a given cell may result in an improper isosurface extraction. The green curved surface represents a piece of the original model, while the red ones represent the result obtained from Marching Cubes.

position vertices and control how to collapse the associated octree. Furthermore, the simplification can be modified to apply topology safety (no topological changes).

On the other hand, implicit representations have become increasingly common in geometric modeling for operations like Booleans, Minkowski sums, simplification and remeshing. They are typically defined as an isosurface extracted from a scalar field, which needs to accurately approximate the boundary of the final solid. Varadhan et al [VKSM04] work on the problem of obtaining a topologically correct isosurface from an implicit representation. Such a surface is extracted in an adaptive fashion. Cells which are considered complex are subdivided to solve their topological imprecision. That is, if the piece of the surface which is inside a given cell is not topologically equivalent to a disk, that cell is considered complex. An isosurface is extracted from the adaptive grid using MC combined with crack patching techniques. Finally, adaptive subdivision methods are applied to improve the quality of the resulting mesh.

2.3 Topology disambiguation on Marching Cubes

As it has been previously introduced, Marching Cubes extracts an iso-surface from a scalar field cube by cube. This gives rise to a problem of consistency when ambiguous faces (those with two edges to extract) are found (such a case is shown in Figure 2.2). They need to be processed consistently on the two neighboring cubes that meet on a same ambiguous face. This problem was first noted by Dürst [D88], and solutions

have been proposed in [HR95] and [NB93].

All the mentioned techniques solve ambiguity problems applying local decisions, thus being unable to deal with global topological properties of the extracted isosurface. In fact, topological correctness techniques also need image data, being unable to use binary images.

Important tools for the manipulation of binary images are available. These come from the digital geometry field [KR04], where the objects that resulted in a certain digitalization are recovered if the resolution of the image is high enough. In order to achieve good extraction results, a digital topology is introduced which defines how pixels are connected. Once the connectivity has been defined morphology operators may be used to solve inconsistencies.

This kind of alternative approaches appear when a certain topology is wanted, as in [BK02]. For example, the interface of a certain organ normally has a known topology (like the brain being homeomorphic to a sphere), but simple thresholding yields a different one. Knowing which topology changes are produced by adding or removing a voxel from the set of solid voxels enables us to perform morphological operations, while guaranteeing that the resulting set is homeomorphic to the initial one. This morphological operations include erosion, dilation, openings, closings and inflation.

2.3.1 Topological consistency

Topological consistency solutions avoid holes on the extracted surface by choosing coherently how to solve every ambiguous face of the model. In order to achieve it, those methods do not need image data values, only inside/outside information, resulting into very simple implementations that do not solve the correctness problem.

Preferred polarity methods consist in taking an uniform decision over all the ambiguous faces of the model, and can be implemented either by using a table of possible cube configurations or algorithmically. Lachaud [L96] solves the ambiguity problems of MC methods by defining two different neighborhood adjacencies for white voxels and black voxels, from the 6, 18, and 26 adjacency relations. A MC table is deduced from that selection, and it is applied to models to obtain an isosurface with coherent topology. Montani et al [MSS94] also precompute a MC look-up table to obtain continuous isosurfaces. Bloomenthal [B94] solves the coherency problem during the extraction. His approach polygonizes an implicit function using regular grids. It needs a set of starting points because it applies a continuation method to obtain the

mixed cubes or tetrahedra (by traversing them through mixed faces). Root-finding by binary subdivision is used to obtain the vertices' position (instead of using linear interpolation), and those positions are cached.

Another family of methods subdivide every cube into five or six tetrahedra ([Z92], [NFHL91]), solving the problem of ambiguity given that the algorithm that extracts surface for a tetrahedra is unambiguous. The resulting disambiguation depends on how a cube is subdivided into tetrahedra.

2.3.2 Topological correctness

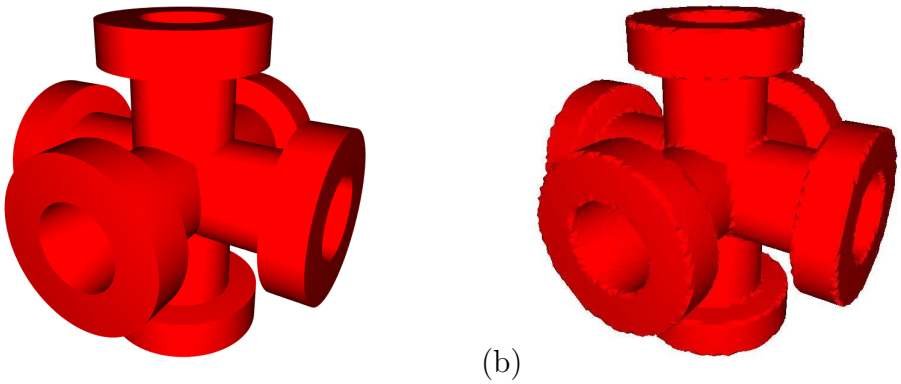
Meanwhile, topological correctness methods try to extract the proper surface from a complete image scalar field (in Figure 2.2). Those are divided into two groups, the first trying to obtain correctness on the boundary of ambiguous faces, while the other also tries to recover the original topology in ambiguous cubes.

In the first group there are methods like face center resampling ([WMW86], [W91]), which solve the ambiguity problem obtaining the value at the center of the face by interpolation. Bilinear interpolation techniques are represented by the Asymptotic Decider [NH91] which applies a bilinear function to the four values on an ambiguous cell face. Additionally, a multi-entry table is used. Each case is divided into 2^i subcases, where i is the number of ambiguous faces on that configuration (thus cases without X-faces remain untouched). The bilinear function used on each face is an hyperbola, and the disambiguation of the X-face follows from the value interpolated at the crossing of the hyperbola's two asymptotes. Gradient contributions may also be helpful ([WG90]).

Critical point analysis ([SH97]) and trilinear interpolation ([CGMS00], [N03]) techniques belong to the second group.

In [SH97], a topologically correct polygonization algorithm is obtained by finding and tracking surface's critical points. If the surface is represented by an implicit function f, critical points can be found at its gradient's zeros. Furthermore, they can be distinguished by computing the signs of the eigen values of the stability matrix (as the Jacobian of the gradient is called in catastrophe theory - which studies the qualitative effect of the different parameters -) of f. Changes of those eigen values' signs determine topological changes, which means that they can be followed by tracking the evolution of the critical points of f.

Trilinear interpolation of the 8 values of a cell allows us to obtain values where they are not present. For example, Cignoni et al [CGMS00] extract an isosurface using a



(a) (b)

Figure 2.3: (a) Discretizing the distance scalar field of a certain shape like this cross pipe, and (b) extracting from it an isosurface using Marching Cubes results in artifacts. These artifacts are caused by the feature insensibility of MC.

trilinear local approach (one cell at a time), and subdividing it until an user-selected approximation accuracy is achieved. In order to obtain the first approximation of the isosurface, a table with all the possible polygonizations of the trilinear interpolant is used. Afterwards, every triangle is checked to decide if subdivision is needed. This process is guided by the user defined accuracy.

The contribution by Nielson [N03] analyzes all the possible shapes of the trilinear interpolant in a cell, depending on the values of its vertices. Then, a three level table is built from the result, where the first level corresponds to the MC cases table, the second one corresponds to the asymptotic decider additional cases, and the third one completes the table to take all the trilinear cases into account.

Lopes and Brodlie [LB03] also enumerate all possible topology cases that result from the trilinear interpolant, and apply a new triangulation scheme to improve the accuracy. The main drawback is the large increase in the number of triangles of the final isosurface.

2.4 Feature extraction

The mesh resulting from MC extraction still has some errors produced by the feature insensitivity of uniform grids (this artifact may be seen in Figure 2.3). Those features need to be recovered along the boundaries of regions of smooth curvature change.

In [KBSS01], Kobbelt et al improve on two problems present in regular MC. First, the approximate computation of samples through linear interpolation which results in incorrect estimations near features. Second, the mentioned feature insensitivity of

regular grid sampling methods. In order to solve the first problem, a modified version of the discretized scalar distance function is proposed. Storing three axis-oriented distances for each grid point, samples can be linearly interpolated on sticks (grid edges which connect black and white grid points) and yield good approximations for surface intersections with the grid. Finally, an extended version of MC is described. Mixed cells which do not contain features are processed using regular MC techniques (case tables), while other cells need to distinguish between edge features and corner features. A new vertex is computed for those, and a triangle fan is used to represent the sheet on that cell. The last step consists in flipping edges which result in connecting two feature points (those added in the previous step). Surface normals need to be computed to compute the position of new feature points.

Attene et al [AFRS04] work with regular meshes instead, being able to recover features in scanned models as well as in regular grids. Edge-sharpener deals with this problem by identifying regions where the normal smoothly changes, and recovering features at the boundaries between those smooth regions. Edges that connect different smooth regions are called chamfer edges, while triangles with one or more chamfer edges are called chamfer triangles. Subdividing all the chamfer triangles, while positioning the added vertices to be on the intersection between smooth regions yields the desired result. This approach is then applied to recover sharp features from MC isosurface extractions, and to improve the decompression results of the SwingWrapper compression algorithm.

In Sharpen & bend [AFSR05] the authors improved on Edge-Sharpener. As the MC techniques are usually feature insensitive, most of the error can be found at sharp features. Thus, the Edge Sharpener algorithm may be used to reduce that error, resulting in a more distributed error diffusion. Subdivision techniques applied to meshes reduce noise, but they also round off features. In order to use such a method after an edge sharpening process, it needs to be adapted to respect the already detected features. As a result, the combination reduces the error between the original models and their remeshed counterparts.

2.5 Smoothing

The main objective of smoothing algorithms is to reduce undesirable noise and high-frequency details while retaining important geometrical features. Most approaches apply iteratively a smoothing operator, which is normally derived from discrete versions of the Laplacian.

Although, most fairing methods work on general meshes, some of them have been specially adapted to deal with isosurfaces extracted from binary grids.

2.5.1 Mesh fairing

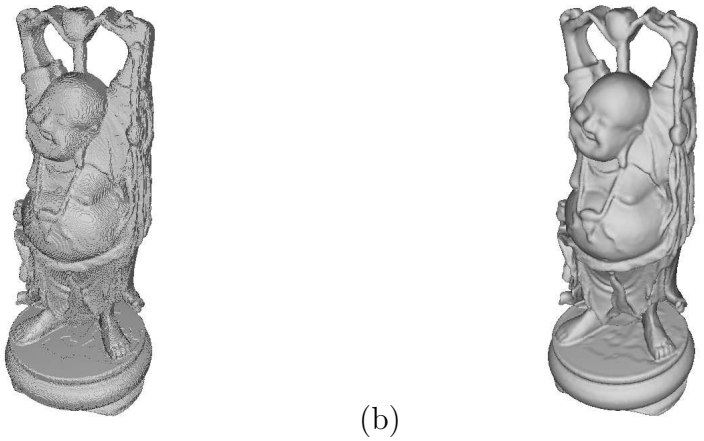
Taubin's lambda-nu smoothing algorithm is presented in [T95]. Each iteration is composed of a forward laplacian step scaled by lambda, followed by a backward laplacian step scaled by nu. The laplacian operator used corresponds to its discrete approximation (the umbrella operator) applying different weights (like area based weights that take into account non-uniform discretization). This algorithm results from applying low-pass filters theory on discrete surface signals. As a consequence, the shrinkage effect present in Gaussian filtering (like a simple laplacian) disappears. Subdivision is presented as a possible field of application for the new filtering approach. Several formulations of neighborhoods are also discussed to introduce interpolatory, smooth and hierarchical constraints.

The signal processing approaches to the fairing of meshes introduced by Taubin lead to a variety of artifacts. Some of them include geometry distortion during smoothing and numerical instability due to irregularity in the sampling. Highly sampled meshes also lead to a slow convergence. In order to solve these problems, Desbrun et al [DMSB99] modified the discrete laplacian operator to get a scale-dependent version. Besides, convergence is improved by means of an implicit integration method. The resulting sparse linear system leads to an unconditionally stable algorithm which allows to increase the step size of each iteration. As additional features, exact volume preservation is enforced automatically, and hard and soft constraints on the position of points of the mesh may be added.

Kobbelt [K97] also addresses the problem of smoothing arbitrary topology meshes. Meshes are refined applying an uniform subdivision method to the original triangles. The position of new vertices is determined by minimizing a functional which measures a discrete approximation of bending energy.

2.5.2 Isosurface smoothing

Most smoothing techniques can be adapted to apply equality constraints, but inequality restrictions are needed for isosurfaces resulting from MC methods. In Marching Cubes we are interested in constraining vertices' movement to the edges that generated them. In the same fashion, in Dual Marching Cubes, vertices cannot leave their respective cells. These conditions guarantee that the smoothed surface contains the



(a) (b)

Figure 2.4: In this example, (a) an isosurface extracted from a binary grid is (b) smoothed in order to improve its appearance.

voxels that were classified as interior, while excluding those which were considered exterior.

Isosurface smoothing (in Figure 2.4) can be useful to remove some artifacts, which manifest as terracing effects due to high spatial frequencies present in volumetric models (like medical ones). Applying convolution smoothing algorithms like the Gaussian does not yield good results, as too large kernels are needed to remove terraces, which at the same time eliminate significant detail. The solution chosen by Gibson [G98] is to use dual marching cubes to compute the isosurface, and then apply iteratively an operator that diminishes a certain energy function. Choosing the operator as the one which moves a point to a new position, equidistant of its neighbors, reduces the energy function, which is computed as the sum of the squared lengths of all the links in the surface net. Finally, the computed isosurface is used to compute distances on the grid points, which will be used to compute gradients and normals for each vertex of the resulting mesh. Even though the result is smoother than what is obtained from applying a Gaussian kernel, the imposed constraints break the smoothness of this laplacian-like approach.

Whitaker [W00] suggests performing the isosurface smoothing volumetrically, thus level set methods are applied. The level set function is initialized to -1 at exterior voxels and +1 at interior ones, and then mean curvature flow is performed on the narrow band. This operation is equivalent to applying the laplacian, resulting in a minimal area surface which stretches over the angular constraints.

In [NGH+03] and [N04], two smoothing methods based on Marching Cubes are presented. The first approach defines an energy for a 2D version of MC at every

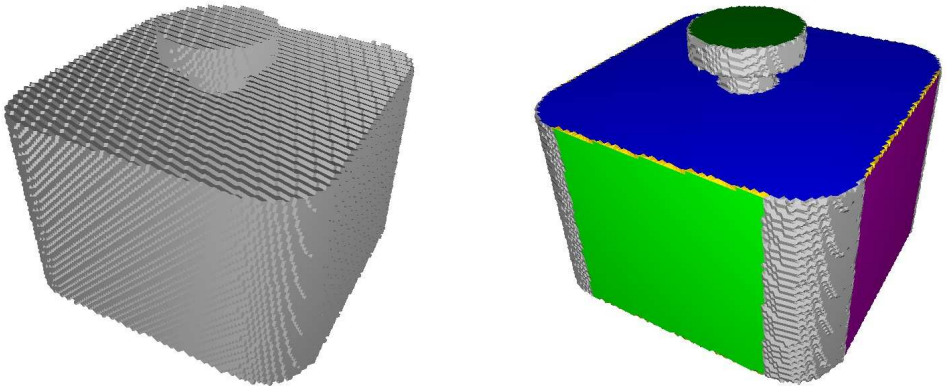


Figure 2.5: Planar regions have been extracted from a pump model, which can be used as signatures to identify the object or to simplify it.

vertex (using the curvature of the quadratic curve passing through that vertex and its neighbors), and minimizes it for each vertex, thus producing a movement for each of them. This is extended to 3D by optimizing that energy on the planar contours of the isosurface. Several iterations are applied to smooth it.

The second method applies the dual of the dual of MC iteratively to smooth isosurfaces. The main drawback is that computing the dual of the dual of MC is equivalent to applying the laplacian, which pushes the isosurface against the constraints. As a result, the final surface does not really look smooth.

Another approach [EBV05] is to approximate the discrete band with a piecewise algebraic surface. This surface is smoothed by increasing its continuity from C^2 to C^3 on the boundary faces of the cells. At the same time, it is constrained to intersect a set of "hard" cells by pulling it to their centers. As a result, this method may be used to obtain smoother versions of a discrete model, and to remove noise on medical applications.

2.6 Extraction of flats

The detection of large planar regions (Figure 2.5) is important in simplification and reverse engineering. One way of doing it is fitting planes to the model. This problem has been studied by Faugeras et al [FH86] in the field of computer vision for recognizing 3D objects.

Decoret et al [DDSD03] extend the Hough transform to 3D. This allows them to vote in a discretization of the Hough plane space for the planes which "capture" (a

triangle is "captured" by a plane if its vertices are nearer than a certain distance) every triangle of a given model. A plane is extracted from the voting space, captured triangles are projected onto that plane to generate a billboard, and their contribution is removed. This process is repeated greedily, until all the geometry has been projected onto billboards.

Uniform discretizations of the Hough transform space lead to non-uniform distributions of planes, which produce high running times. Furthermore, it can easily choose planes which intersect the object producing poor texture performance, instead of generating them to approximate the surface of the object.

In the field of simplification Andujar et al [ABE+03] present a solution which detects large planar regions, adjusts polygonal faces to them and renders the result in conjunction with a set of oriented points. A tetrahedral volumetric representation is generated first, followed by the computation of a measure of flatness for the faces of all mixed tetrahedra (those on the discrete band). This measure is the radius of the largest disc which can be adjusted into the neighboring part of the discrete band. Then, a region growing algorithm is applied from the tetrahedra with higher planarity values to obtain the largest tiles, which are clipped against each other. Finally, the resulting tiles are converted into polygonal faces and oriented points are generated to sew them.

2.7 Simplification

Simplification algorithms are also related to the stated problem. They focus on reducing geometry complexity while preserving as much detail as possible.

Simplification literature has focused mainly on highly tesselated and topologically simple triangle meshes. Most methods apply iterative operators such as vertex removal, edge collapse, pair contraction or face removal which reduce the number of geometric primitives. These operators may be guided by error metrics based solely on geometry or be directed by the position of the viewer. Extreme simplification techniques also exist that replace distant geometry by impostors.

2.7.1 Topology simplification

Acquisition of 3D models yields objects with topological noise due to inaccuracies during the scanning process and the merging operations. This noise complicates other operations like remeshing, parameterization and simplification subsequently

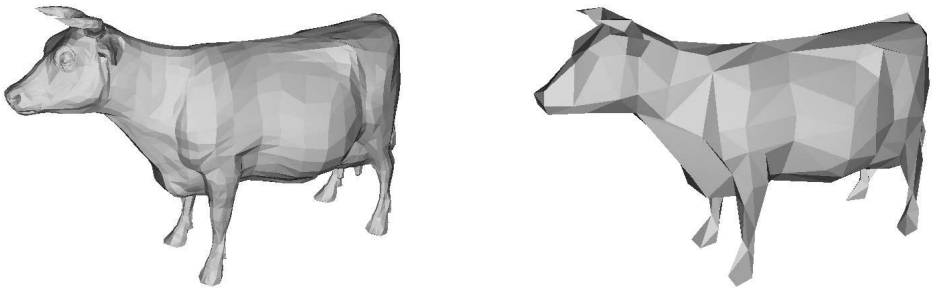


Figure 2.6: Results of applying the quadric error metrics algorithm to a cow model.

applied. The approach by Guskov and Wood [GW01] removes unnecessary non-trivial topological noise from surfaces.

In order to detect topological noise, small regions are grown around seed faces. If its genus turns out to be different from zero, small topological noise has been found. Then, a non-separating cut must be found and used to repair that part of the mesh. The region is cut and both new holes are sealed. Of course, the size of the grown region influences which handles are found and which are not.

2.7.2 Geometry simplification

Many techniques have been developed for the geometry simplification of triangles meshes. One of them is vertex clustering which was introduced in [RB93] by Rossignac and Borrel. Given a triangle mesh, its vertices are grouped into clusters by geometrical proximity. Each cluster results into a representative vertex which is compute by weighting its associated vertices. Vertices with higher probability of lying on the object's silhouette from an arbitrary direction, and those bounding large faces that should not be affected by the simplification, receive larger weights. Finally, some triangles may disappear, while other ones may be collapsed into edges. As a result, vertex clustering techniques can yield non-manifolds models.

Low and Tan [LT97] also use a vertex clustering approach and their main contribution comes from the way vertices are clustered. Where previous approaches, used uniform vertex clustering, this paper selects vertices by grade and builds cells around them. This technique allows cells to be of different shapes, and it adapts better to the model.

In [GH97] Garland and Heckbert simplify by collapsing pairs of vertices, selecting

only certain pairs for contraction. Edge connected vertices and those which are not connected but at a certain distance less than some threshold are considered candidates for edge collapse operations. The algorithm iteratively contracts a pair selected as to minimize an error metric. Despite many error metrics may be selected, the authors apply a quadric error metric, as it allows them to track the accumulated error on each vertex with only 10 floating point numbers. Boundaries can be preserved by weighting quadrics appropriately. An example result of this technique may be seen in Figure 2.6.

View-dependent simplification may be applied by means of a vertex octree (like in [LE97]). By selecting different depths for each node according to view-dependent error metrics, a vertex clustering approach may be used. Useful metrics include screenspace error thresholding and silhouette preservation measures. Actually, the two of them can be easily combined by tightening the screenspace threshold for those nodes of the octree which may potentially represent a silhouette. Projecting the bounding sphere of the vertices in a given node approximates the screenspace error produced when that node is simplified. Additionally, potential silhouettes are detected by maintaining a cone that contains the normals of the triangles at each node of the octree. This cone is used to determine if the triangles in that node are completely front-facing, completely back-facing, or potentially in the silhouette. The octree is subdivided to fulfill a certain triangle budget, exploiting temporal coherence. Its real-time efficiency may be further improved by asynchronously running the simplify and render tasks.

El-Sana and Varshney [EV99] also propose a view dependent simplification method that combines edge collapse, vertex split, virtual edge collapse, spline-based new vertex positioning and level of detail handling. Using virtual edges derived from the computation of an associated Voronoi diagram, the initial view-dependence tree is generated by collapsing those edges based on a distance-based metric. New vertices are positioned by estimating them using Hermite interpolation, while taking care that no fold-overs occur. The active front is computed by applying a series of collapses and splits which are managed by a refinement metric. This refinement metric comes from computing the length of the two splines approximated from a given vertex to the light source and to the view point. Thus, the same metric is used when positioning new vertices and determining the level of detail.

Yet another group of methods cluster faces into connected components, which are transformed into single faces, thus simplifying the original mesh. One such a case is presented in Superfaces [KT96] by Kalvin and Taylor.

The goal in this contribution is to group mesh triangles into regions which are

called superfaces. The boundaries of these superfaces are simplified and retriangulated to yield a simplified mesh. In fact, the superfaces do not need to be planar. In order to build these superfaces we initially select a random set of seeds from which to grow. Each triangle imposes constraints on the feasible set of planes which may approximate a superface. These constraints restrict the planes to be at a maximum distance from the triangle and to have a compatible normal. Thus, each constraint removes a halfspace of planes, which result in a convex polytope for a set of them. Instead of computing this polytope, a conservative approximating ellipsoid is used, and triangles are added until that addition results in an empty ellipsoid.

While Superfaces ([KT96]) builds hierarchical clusterings of faces directed by an error metric that guarantees a maximum deviation on any point, Garland et al [GWH01] use a combined error metric. This metric minimizes the average deviation by using quadrics, while guarantying that the global fit for the normal does not deviate too much from the normals of the cluster, and maximizing the regularity (it prefers circles to other shapes) of the resulting clusters. Some possible applications include distance, intersection and collision queries, which use the fact that hierarchies of oriented bounding boxes can be computed from the face clustering. Surface simplification may also be achieved applying the "Superfaces" algorithm. Finally, radiosity computation obtains large efficiency benefits from hierarchical face clusterings.

Sheffer [S01] also uses a face clustering approach while preserving sharp boundaries, trying to get as simple as possible boundary shapes and creating regions with small curvature change. Finally, it tests the possibility of symmetrically splitting regions to divide them into two adjacent ones. The method is intended for CAD model simplification.

Most face clustering approaches apply error metrics based on positional information (like the L_2). Cohen-Steiner et al [CAD04] introduce a new metric, the $L_{2,1}$ shape metric, which is based on the normal field. Their algorithm works in two steps which are applied iteratively after selecting a random initial set of seed triangles. First, the geometry is partitioned into non-overlapping clusters which minimize the new shape metric. Second, a representative called proxy is computed for every cluster. Every proxy consists in an average point and an average normal which approximate a given partition minimizing the error metric. The points associated with each proxy are used in the next iteration as new seeds for the partitioning algorithm. Thus, selecting the number of seeds will limit the final number of proxies, but new ones can be added very easily and they can be removed as well. The main application presented in their work is to use the resulting partition to remesh the model. Points where three or

more partitions meet generate anchor vertices, and edges result from the fact that we know which regions converge at each of them. Following the boundary of each cluster, yields a chain of edges between its anchor vertices, but they need to be recursively subdivided using a chord-length criterion. The resulting faces are triangulated using a discrete delaunay triangulation method, and useless edges are removed from the mesh to obtain the final polygons.

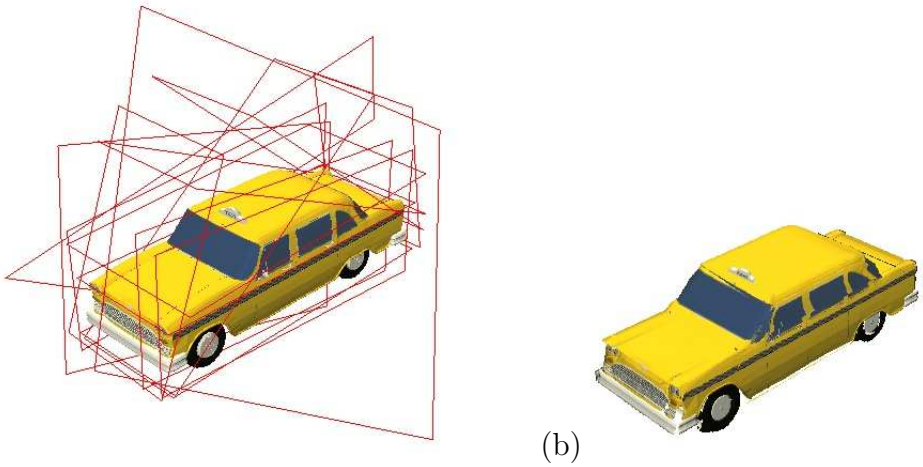
A different approach comes from using volumetric techniques to apply simplification and repair to conventional meshes. Nooruddin and Turk [NT03] present such a method that is able to deal with holes, double walls and intersecting parts. The simplification component of their algorithm derives from a topology-altering method based on morphological operators. In short, their algorithm takes a polygonal mesh as input, scan converts it into a volumetric model, modifies its topology to simplify it and converts the voxelization back to polygons. Finally, a topology-preserving simplification method is applied. The whole process always results in a manifold mesh.

Many simplification algorithms are focused on reducing the complexity of highly tessellated objects, thus failing to obtain good simplifications of topologically rich assemblies. Andujar et al [ABA02] overcome this difficulty by devising a simplification algorithm, which reduces topology complexity in a natural way. First, the object is discretized to obtain an octree representation that may be viewed as a multiresolution hierarchy of grid models. From this hierarchy, a series of polyhedral surfaces are extracted with decreasing accuracy and complexity. These surfaces are further simplified by a face reduction process based on the edge collapse operator. As a result of this process, topological simplification and aggregation is achieved.

2.7.3 Appearance-preserving simplification

Models may be further simplified if we are only interested in the final appearance. This new condition enables us to use texture maps, as well as impostors (obtaining results such as those in Figure 2.7). Methods that apply this kind of approaches are known as appearance-preserving.

For example, in [COM98] maps are used to store attribute information (texture, bump, and normal maps) and then the model is simplified while taking into account position, curvature, and color deviation. First, the model is divided into several patches (as few as possible), assigning each of them a texture parameterization. In order to do that, the vertices of one patch are distributed as evenly as possible in



(a) (b)

Figure 2.7: (a) The original taxi model with the 14 quads which will be used as impostors. (b) The rendering which results from using an impostor-based technique.

their corresponding map, while ensuring a one to one mapping (that is, no folds in the texture plane). Second, the initial maps are computed by drawing the triangles in their corresponding texture space using Gouraud shading to interpolate the attributes. Finally, the textured model is simplified by applying a weighted average of the regular positional error metric and a texture deviation metric. This texture deviation metric measures the 3D distance between points with the same texture coordinates in a simplified model and a previous model. To simplify its tracking, a set of axis aligned boxes is used.

While others try to preserve attribute values during the simplification process (by taking into account attribute discontinuities on candidate elements to be simplified), Cignoni et al [CMRS98] use a post process (thus independent from the chosen simplification algorithm) to produce a texture map for the final model. Each triangle is sampled in two directions according to its size. For each of these samples a corresponding one is computed in the original surface at minimal Euclidean distance. Finally, the attribute is obtained and stored into a texture. The texture patch corresponding to a given triangle is not deformed into a half-square triangle (they are easier to pack in a texture), but only sheared along the X axis to reduce the wasted space between the previous patch and the current one. Artifacts caused by insufficient sampling rate and discontinuities along borders are overcome using oversampling (like in antialiasing) and slightly wider texture patches, respectively. The approach is used to compute texture, bump and displacement maps, as well as to obtain 2D texture maps for a model that uses procedural 3D textures.

A different group of image based techniques replace the original geometry by some

well-placed textured polygons, known as impostors. Sillion et al [SDB97] subdivide urban scenery models into regions (using the streets), and then precompute a set of impostors for the rest of the city model from each of those regions. Then, during the visualization, the model is divided into the local model (the two adjacent regions), and the distant one, which is rendered using the impostors. To reduce the number of impostors and make their "cache life" longer, 3D information is associated to them. The impostors are applied to meshes extracted from the depth buffer obtained during their render, instead of the typical quadrilateral. Moreover, this mesh reduces problems of "cracking" along the boundaries of the local model.

Regular image based techniques simplify the distant part of the model into a single mesh impostor (SMI). Decoret et al [DSSD99] use MMI (multiple mesh impostors), subdividing the distant model into multiple impostors according to their mutual overlapping, as seen from the current cell. This strategy contributes to remove cracks, incomplete representation and rubber sheet effects. Furthermore, they strive to simplify the complexity of the scene to have extra time per frame, which is used to dynamically update the impostors. Thus, deformation problems and resolution mismatches are considerably reduced.

A similar strategy is presented by Aliaga and Lastra in [AL99], but their approach may be applied to general scenes. Scene complexity is reduced by precomputing LDIs from a set of grid points. Thus, LDIs are used to substitute far geometry with warped layered images (images with associated depth values and one or more layers per pixel). The system guarantees a maximum number of rendered primitives, allowing to control the frame rate.

Wilson's contribution ([W02]) adds an error-bounded scheme for the computation of sample locations. This sample locations are used to compute image-based simplifications which are useful for rendering acceleration. The resulting impostors are spatially encoded using a generalization of video compression methods to handle three-dimensional data. Finally, this collection of impostors is used to improve interactive walkthrough frame rates.

Impostors may also be used as a simplification of a complex single object. In particular, a set of planes can be chosen to represent the model, projecting its geometry onto those planes to obtain a set of textured impostors. The resulting set is known as a billboard cloud ([DDSD03]).

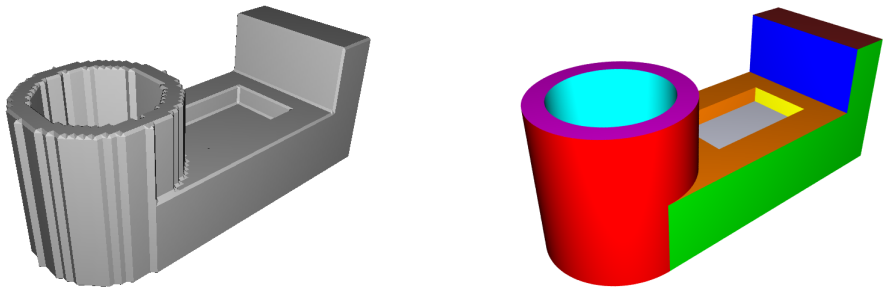


Figure 2.8: Reverse engineering techniques recover basic shapes from a given model. Left, the isosurface extracted from a binary grid is shown. The right image shows the model as the combination of several basic shapes, like boxes and cylinders.

2.8 Reverse engineering

Reverse engineering applications try to fit parametric surfaces that approximate unstructured data, thus recovering shape information (as in Figure 2.8).

The survey by Varady et al [VMC97] introduces the subject of reverse engineering, describing its basic phases. Different data acquisition techniques are analyzed, dividing them into tactile and non-contact methods. Before any segmentation, the incoming data may be preprocessed to combine multiple views and reduce noise. Then, the resulting points are grouped into regions to which an appropriate single surface is adjusted. These surfaces are normally either planes or natural quadrics (spheres, cylinders or cones), however free-form surfaces are also used. Finally, accurate and consistent boundary models need to be built. Identifying sharp edges, adding blends, providing continuity where it is needed, and enforcing constraints are important parts of this last step. Two main approaches to segmentation may be considered, edge-based and face-based methods. Edge-based techniques try to identify sequences of edges that bound the different surfaces present in the model. If we are looking for sharp edges, angles between normals may be estimated to detect sharp changes in the normal direction. If smooth edges are also sought, changes in higher order derivatives (like curvature) need to be traced. Their main drawback is that they consider only a small part of the data to segment the model. Face-based methods cluster faces into regions which may be approximated by a single surface, deriving edges from the boundaries of the final clusters. One common drawback of this kind of approaches is that they may need to change the surface being used to approximate the region while growing it.

The segmentation step is particularly important as it determines the regions that will be fit using different surfaces. Normally, shape segmentation approaches adjust natural quadrics to the regions resulting from their algorithms. Gelfand and Guibas ([GG04]) apply a different kind of surfaces to approximate those regions. Slippable surfaces are those that remain unaltered after applying certain rigid motions. For example, a plane can be rotated around its normal and translated perpendicularly to it, and result in the same surface. The same can be applied to spheres, cylinders, linearly extruded curves, surfaces of revolution and helices. Those surfaces are used in a bottom-up segmentation approach, which first computes a similarity score between points. This score allows to compare the slippable shape between the neighborhoods of two points, being useful during the region growing step. In order to get a robust algorithm, a multi-pass approach is applied, and a post-processing step is used to reduce non-slippable regions.

2.9 Visibility computation

In Chapter 7, we will see how to extract the features of a given model using visibility computations. This process is guided by a measure of visibility computed for the mixed cells of a voxel model. Cell-to-cell visibility needs to be computed for a scenario where the cells are cubes of the same size and the occluders are the union of a set of such cubes.

Some of the from-region visibility algorithms are only useful for certain classes of scenes. Others provide an elegant solution to the exact cell-to-cell visibility problem, but require complex calculations [CCS03].

Nirensen et al [NBG02] present an exact method which represents 3D polygons and the visibility stabbing lines in a 5D Euclidean space derived from Plücker space. Geometric substractions of occluded lines are then performed from the set of potential visibility lines. This approach works well for scenes where the number of total occluders is relatively small, which is not our case (every homogeneous cell adjacent to a mixed one is an occluder in our method).

Seeking to achieve temporal exact visibility, Durand et al [DDP02] introduced the 3D Visibility Complex. Maximal free segments (segments that have no intersection with the occluders and whose length is maximal) are grouped according to which objects they see. As a result, the boundaries of these groups represent visual events where the visibility changes. The structure that stores all this information is what it is called the Visibility Complex, and it allows to track exact visibility as the view-

point changes its location. The main drawback of this approach in our setting it is complexity (worst case bound of $O(n^4)$).

Erosion-based methods, introduced by Wonka et al [WWS00], are more applicable to our problem. In [WWS00], the scene is subdivided into cells for which cell-to-object visibility is precomputed. The precomputation is carried on by sampling the view cells, eroding the occluders and computing point-to-object visibility using the shrunk occluders. The result is a conservative visibility algorithm that, due to the view cells' sampling, is only efficiently applied to 2D occluders (which is useful in urban environments).

Decoret et al [DDS03] extended the erosion-based method by computing cell-to-cell visibility instead of cell-to-object. This modification allows them to shrink the source cell, the target cell and the occluders, transforming the cell-to-cell problem into a point-to-point visibility query (which only takes an intersection test between the shrunk occluders and the segment that connects the source and the target cells). The problem remains as to how to accurately shrink the occluders, and this is solved in [DDS03] computing a conservative estimate of the shrinking. In Chapter 7 we will see that in our setting we are able to compute the shrunk occluders exactly, which leads to an even tighter potentially visible set (PVS).

Chapter 3

Common concepts

This chapter introduces some basic concepts regarding our implementation of Marching Cubes. It is intended to establish the terminology we will be using during the following chapters.

Binary grids

There are many ways of representing volumes, each of them with their own set of advantages and disadvantages. Among them, discrete volumes are one possible representation very well suited to boolean and morphological operations. Furthermore, many algorithms require "clean" models, which isosurface extraction algorithms can easily obtain from volume representations.

Commonly, volume representations take the form of scalar fields. As such, they may represent distance to the surface or some physical property of the model, but they are commonly discretized before being used in subsequent algorithms. This discretization may take the form of a sampling, if what we have is a continuous formula, or be caused by the acquisition process itself in case the data is being obtained from sensors. Additionally, the discretization may be regular or irregular.

Anyway, before any surface can be extracted from such scalar field discretizations, a segmentation process has to take place. This segmentation usually results in a binarization, where one set of samples are considered to be inside the volume and another set has to be outside.

Due to the above reasons, the simplest volume representation, the binary grid, is an intermediate step in the process of extracting a triangle mesh. It contains some of the shape information of the isosurface (see Figure 3.1). For the rest of this document, we will be using uniformly sampled binary grids, where those samples are positioned

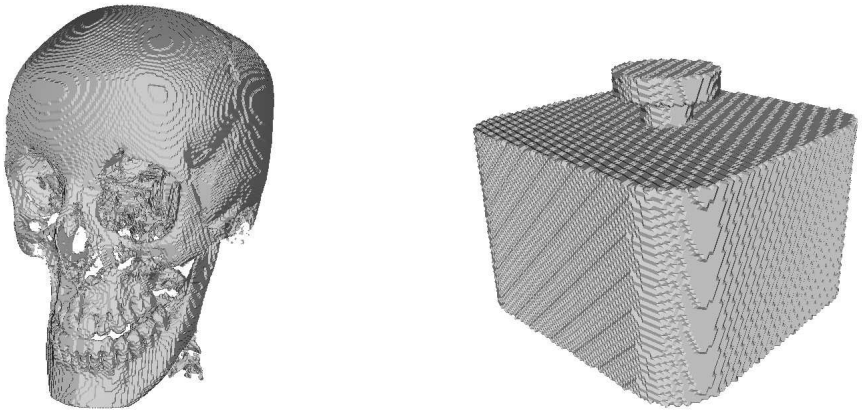


Figure 3.1: These images show the result of extracting a surface from a binary grid. Part of the original shape is lost due to the binarization. The left picture shows a skull model, while the right one shows a CAD model.

at the grid's crossings.

Voxels

Each of the uniformly positioned samples influences the value of the region which is nearest to it. The shape of this region is that of a cube centered at the sample, whose side's size is equal to the sampling size. This cube is called a voxel and it is considered black or white depending on whether it is inside or outside the volume respectively.

Now, if we are only using binary values, each voxel can only be inside or outside the surface. As a consequence, a first approximation to the problem of visualizing the binary volume results from drawing the black voxels only. These voxels are drawn as cubes but result in a very poor result like that of Figure 3.2.

Cells

In order to extract a surface from our binary volume representation, we will apply Marching Cubes. To simplify the extraction process, Marching Cubes divides volume space into cells, and extracts a piece of the final surface on each of them. Each of these cells is a cube which has 8 samples or grid points as corners (as shown in Figure 3.3). Due to this fact, cells may be classified as mixed if they have black and white corners, or non-mixed if they only have black corners or white ones. This classification is important because we can consider that the surface only traverse mixed cells.

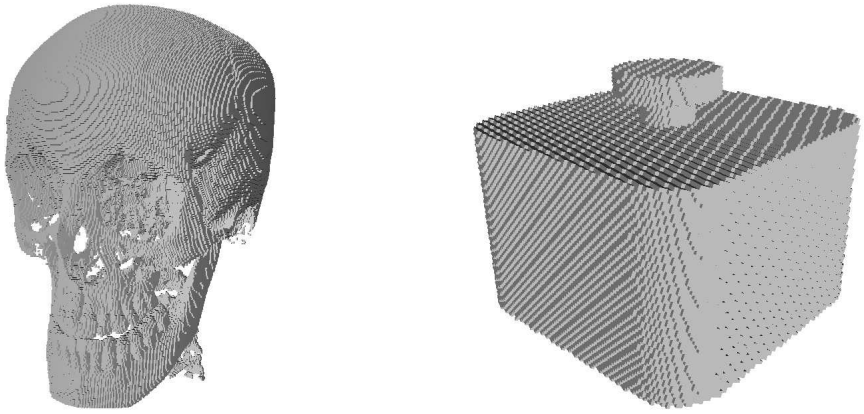
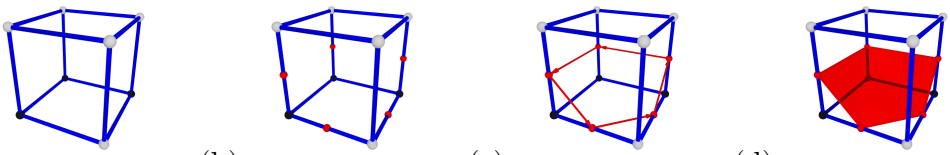


Figure 3.2: Models previously shown on Figure 3.1 rendered using one cube per black voxel.



(a) (b) (c) (d)

Figure 3.3: (a) An example of a cell, where black nodes represent interior samples and white ones, outside samples. This cell is mixed, so a piece of the surface will be extracted from it. (b) Every stick has its corresponding vertex, shown in red in this image. (c) The same cell with its computed border edges. They must be consistently oriented to produce the corresponding sheets. (d) Border edges are grouped into cycles and triangulated to obtain the sheets which intersect the cell.

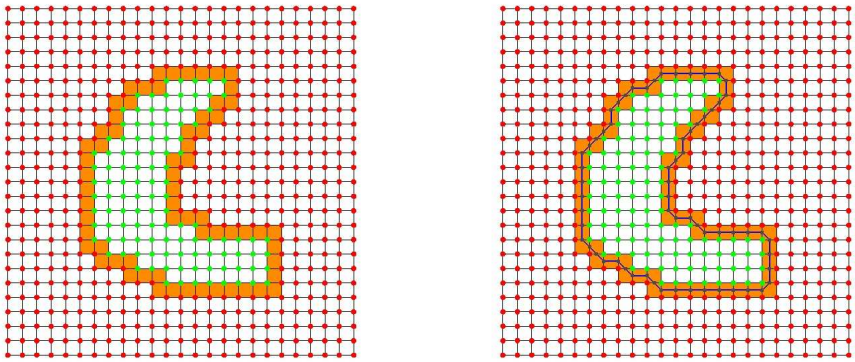


Figure 3.4: A 2D binary grid, where green points represent interior samples and red ones are exterior. The isocurve is in the orange discrete band formed by all the mixed cells.

Isosurface

When dealing with scalar fields, isovalued surfaces are normally extracted. These surfaces are known as isosurfaces. But, as we are working with binary grids, we can only extract one isosurface. That is the surface which separates black and white samples. Moreover, we may assume that such an isosurface is contained in the mixed cells, if the grid's resolution is high enough.

As a consequence, the non-mixed cells are ignored, while the set of mixed cells is called the discrete band (Figure 3.4). The resulting isosurface can be anywhere inside the discrete band, so it provides a boundary for its movement, if we want to respect the original samples.

Sticks

Now that we know where the isosurface will be in terms of cells, we need to ascertain the location of its vertices. One possibility is to consider the edges of the grid that connect a black sample with a white one. These edges, which we will call sticks, have to intersect the resulting isosurface. This fact makes them an excellent choice to put one vertex per stick (as may be seen in Figure 3.3), contraining them to move along their corresponding sticks.

If we had a scalar field, linear interpolation along sticks would be applied to determine exactly where to put those vertices. As we do not have that information or choose to ignore it, vertices are initially located at the middle of its corresponding stick. Further on, we will move those vertices along their sticks to recover as much of

the original shape as possible.

Border edges

We already have an initial estimation of the vertices, but what we really need is to recover the pieces of the surface contained in every mixed cell. After processing the cells' edges, now we must treat their faces. Each of those faces may have four, two or no vertices, depending on the configuration of its four samples. As a result, up to two edges are generated that define the borders of the surface pieces inside the corresponding cell. We call them border edges (Figure 3.3). It is important to note that one face belongs to two cells, so the border edges generated for that face on those cells must be consistent (otherwise a non-manifold will result). This is only a concern when two edges are obtained on a face, as there are two ways of connecting the corresponding four vertices.

Sheets

The border edges may be arranged into cycles which in turn are triangulated to obtain the isosurface's pieces on every cell. We call these separate pieces, sheets. The set of all the extracted sheets from mixed cells will be our triangulation of the isosurface.

Marching Cubes extraction

Marching Cubes can be made more efficient through the use of tables. Each cell has 8 black or white samples, which gives rise to 256 possible combinations. Applying inversion, rotation and symmetries this can be reduced to 15 combinations illustrated in Figure 3.5. But, as we will see later on, we will play with ambiguous cases to optimize some topological properties of the final isosurface. All this is explained in detail in Chapter 4. Anyway, the use of tables for the cell by cell extraction is still possible.

Here we will provide a detailed description of the used extraction algorithm.

Vertex extraction

The first step in the extraction process is to obtain the geometry. We do so by obtaining the vertices. As we will need to move them, we will not represent them by

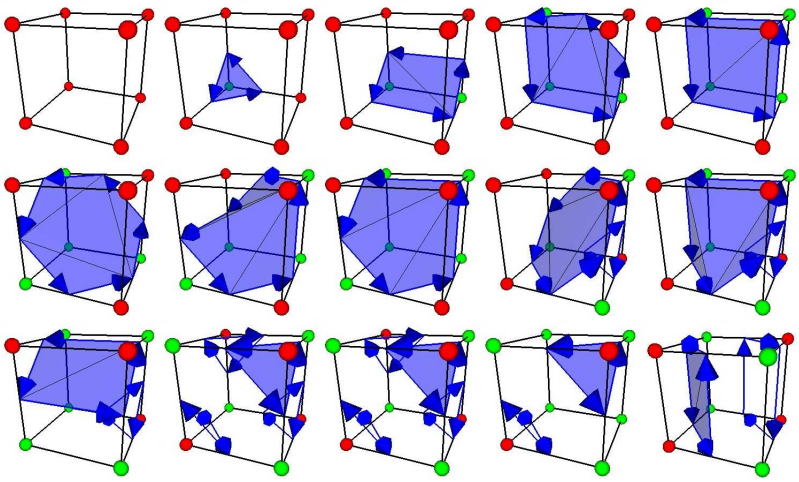


Figure 3.5: The 15 cases used in regular Marching Cubes.

3D positions but by their corresponding sticks. Therefore, we identify all the grid's edges that connect a black sample with a white one.

This identification is divided into three phases. Each of those phases traverses the grid's edges parallel to one of the three axis, adding to the vertices list those which connect mixed samples (see Figure 3.6). This process results in a list of all the sticks of the binary volume, where each of them is stored as the black sample coordinates and a direction that points towards the white sample. There are only six possible directions. Finally, this list is ordered to make it easier to search through it.

Edges

Now that the geometry has been extracted, the topology is next. This step is done cell by cell, considering only mixed cells (those with black and white samples). For each cell, its six faces are analyzed in order to extract the final isosurface's edges. Even though, there are 16 possible combinations of black and white samples, they can be reduced to only five by applying rotation and inversion (which are shown in Figure 3.7).

Furthermore, if we consider the faces by their number of sticks, there are only 3 possibilities: no sticks (like case 3.7a), two sticks (like in cases 3.7b, 3.7c, 3.7e), or four (like in 3.7d). No sticks result in no border edges, while two sticks result in one border edge, and four sticks in 2 border edges. It is important to note that when four sticks are present, an ambiguity arises as to how to connect them. The purpose of Chapter 4 is to explain how to deal with this problem, so for now, we will consider that an uniform decision is applied.

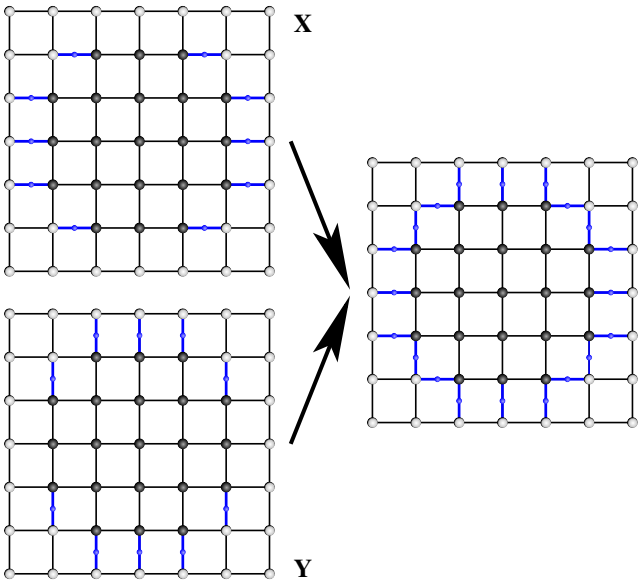


Figure 3.6: In 2D, grid's edges parallel to OX are examined first. Then those parallel to OY are processed and, finally, the results are integrated. Sticks identified in each phase, drawn in blue, show the initial positions of their vertices (also in blue).

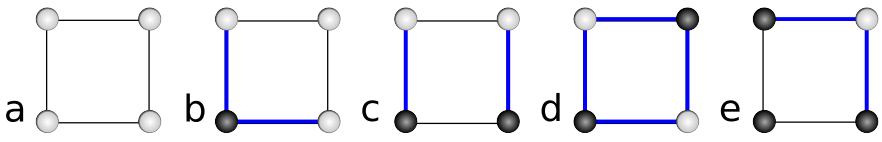


Figure 3.7: Reduced set of possible face combinations.

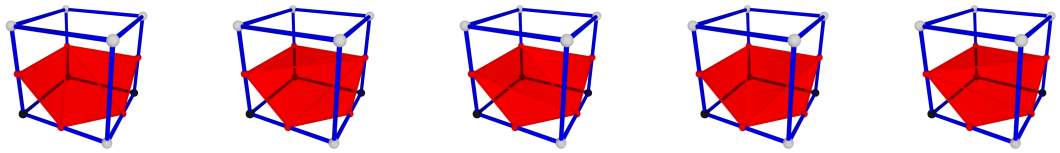


Figure 3.8: Possible triangulations corresponding to the cell on Figure 3.3.

Cycles

Before going on, the edges obtained previously need to be properly oriented. Each of them represents the intersection of a piece of the isosurface with its corresponding face (what we call a sheet), so we want to be able to obtain counter-clockwise cycles as seen from the outside. What we are aiming to obtain is shown in Figure 3.3d.

If a given edge goes from a vertex P to another one Q, its orientation can be tested by means of a simple test. Let us call B_P the black sample of the stick of P, the white one W_P, and the center of the current cell C. If the result of $((Q - P) \times (C - P)) \cdot (W_P - B_P)$ is positive, the edge is correctly oriented; if not, we only need to inverse it.

Once all the edges on a cell have been properly oriented, they can be arranged into cycles.

Triangles

The resulting cycles need to be triangulated to generate the final isosurface, and there are several ways to triangulate each cycle if it has more than three vertices (see Figure 3.8). Consequently, the triangulation has to be chosen so that the extracted isosurface is a manifold. In order to guarantee that property, no triangle can be coplanar to any face of its cell.

In order to be able to traverse the surface, connectivity information is needed. This information will be stored in a corner table as described in [R01].

Topological definitions

In this section we will introduce some math that shows how local topological decisions affect global properties of the extracted mesh, also showing that is impossible to obtain certain global properties using only local decisions. Some global structures are needed.

All the math introduced here depends on the Euler-Poincaré formula for a mesh of triangles, that relates the number of triangles T, with the number of vertices V, the number of handles H, and the number of shells S (number of connected components):

$$T = 2V + 4(H - S) \tag{3.1}$$

The formula gives a relation between some parameters of the extracted surface. As the vertices are fixed, only the other three can vary, the relationship between them being contained in the formula. Another important property is the number of border edges in the mesh, and this one is related with the number of vertices. Every vertex of the mesh is in a stick of the voxelization, thus being in four faces. Given that every face has two border edges, one for each of the two cubes concurring in that face, any vertex is in eight border edges. At the same time, a border edge is delimited by two vertices. These two conditions together enable us to obtain the next formula:

$$B = 4V \tag{3.2}$$

where B is the total number of border edges in the final mesh. As B only depends on the number of vertices, it's constant.

Now, some relation of these parameters is needed at cube level. For obtaining this kind of result we will first look at one cube. Assuming that loops inside this given cube are maintained separately (without connecting them), the number of triangles can be easily determined. For one loop the formula is $t_l = b_l - 2$, where t_l is the number of triangles for the loop, while b_l corresponds to the number of border edges that compose that loop. In one cube, the previous formula can be summed for every loop inside the cube. The result is $t_c = b_c - 2l_c$, where t_c is the number of triangles resulting from the triangulation of all the loops inside the cube, b_c the number of border edges in the cube, and l_c the number of loops inside it. Considering previous formulas and the fact that the number of sheets on a cube is equal to the number of loops (as we don't connect them), a sum for every cube can be computed:

$$T = B - 2L = 4V - 2L = 4V - 2L - 4(s - L) = 4V - 4s + 2L \tag{3.3}$$

s is the total number of sheets on every cube, and L the total number of loops. The previous proof does not take the possibility of connecting loops into account, but only one operation in the presented algorithm makes such kind of change. A cube with two loops can be polygonized so that they become connected. Such operation transforms one sheet into two, but also requires four more triangles, so the previous

formula holds for such cubes.

Chapter 4

Topology disambiguation

4.1 Topology in Marching Cubes

Marching cubes subdivides the space from where a surface is to be extracted in a set of cubes. Using the values of the scalar field on the vertices of the cubes, the surface is built, one cube at a time. The threshold is used to determine if the vertices of one cube are interior or exterior, and the resulting configuration is used to obtain a surface that separates them correctly. But not every possible configuration is unambiguous, some of them having different possible solutions. This choices affect the topology of the resulting surface, many of the variants of the Marching Cubes algorithm solving them in an arbitrary way.

Several alternatives have been proposed to give a solution to this topology ambiguity problem, but many of them rely on local criteria, making them unable to control the topological properties of the final extracted mesh. A global solution is needed, but not at the cost of the Marching Cubes efficiency.

4.2 Degrees of freedom

The surface extraction algorithm presented here does its work cube by cube. First, it tests a given cube for sticks. The points in the middle of sticks are the vertices of the surface inside the cube. Then, for each face of the cube, the border edges are computed. As one face of the cube can have 0, 2, or 4 vertices of the surface, there are a limited number of combinations, some of them shown in Figure 4.1.

If there are no vertices on a face, there won't be any border edges. Two vertices imply one single border edge that connects them through the face (in Figure 4.1,

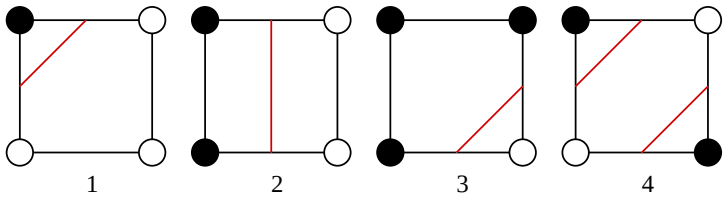


Figure 4.1: Some examples of face configurations and the resulting border edges. The cases marked as 1, 2, and 3 show 2 vertices configurations, while case 4 shows a 4 vertices configuration and one possible way of connecting them using two border edges.

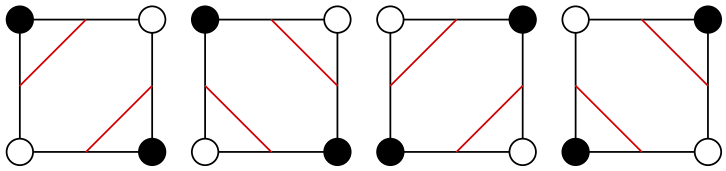


Figure 4.2: X-face configurations and possible slashings.

three configurations can be seen). The ambiguity arises when four vertices are found in a face, as there are two possible ways of connecting them with two border edges. The faces with four sticks will be called X-faces, while the two ways of connecting the four vertices will be the two slashings of the X-face. Figure 4.2 contains all the combinations of X-faces, and their resolution.

Once the border edges of the cube have been obtained, they are grouped in cycles that define the boundary of the different sheets of the surface inside the cube. These cycles are called loops, and their triangulation will yield the sheets or pieces of surface for that cube.

The ambiguity of cubes containing one or more X-faces is controlled dealing with those X-faces, but one more case has a degree of freedom to solve. When more than one loop is present in a cube, they may generate separate sheets or be connected in one sheet. There is only one case (rotations and inversions aside) that has more than one loop and no X-faces to deal with its ambiguity. This configuration has all the nodes of one color (black or white) except two of the other one on two opposite corners of the cube. This generates two loops that can be finally triangulated into two separate sheets or a single cylindrical one that connects them. This configuration, called X-cube, is shown in Figure 4.3.

A third source of ambiguity that it's not dealt with in this contribution is the possible triangulations of those sheets with more than three border edges. Different triangulations of sheets (as in Figure 4.4) can lead to different shapes, so this factor can be used to optimize the area of the resulting surface. As the choice of triangulation

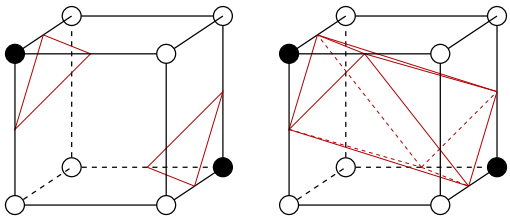


Figure 4.3: X-cube configuration and its possible disambiguations.

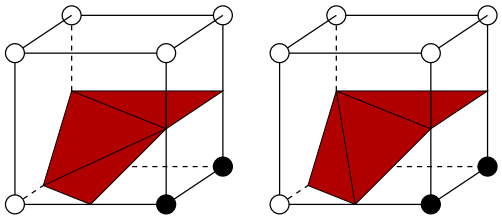


Figure 4.4: Different triangulations of the same sheet.

does not affect the topology of the result, this possibility is not presented here.

Even though we will not use the degree of freedom provided by the different triangulations, a two-manifold resulting surface is wanted. In the introduction this has been deemed as equivalent to obtaining a surface without any triangle on any of the cubes faces. In order to accomplish this condition, the triangulation is done in a special manner. The triangulation is performed in steps, each of them subdividing the sheet into a triangle and a sheet with one less edge. This is done connecting the first and last of three sequential vertices in the cycle that defines the sheet. The problem gets then reduced to choose where to cut, and from the set of possible cuts, all those that are over a face of a cube are immediately discarded. Choosing from the rest guarantees that no triangle is over a face of a cube, thus avoiding artifacts that can cause the surface not to be two-manifold.

This leaves two degrees of freedom to optimize topological properties of the mesh:

The slashing of an X-face determines the border edges on two neighboring cubes, thus modifying the loops on them. This may change the number of loops in the two cubes as well as the number of connected components.

Using two separate sheets or one that joins the two loops of an X-cube modifies the number of connected components, the total count of triangles, and the number of sheets.

Unfortunately, the optimization of a topological property of the surface cannot only be based on local decisions. To solve this, two structures that are able to capture the global topology of the surface are designed.

4.3 Math tools

In this section we will introduce some math that shows how local topological decisions affect global properties of the extracted mesh, also showing that is impossible to obtain certain global properties using only local decisions. Some global structures are needed.

All the math introduced here depends on the Euler-Poincaré formula for a mesh of triangles, that relates the number of triangles T, with the number of vertices V, the number of handles H, and the number of shells (number of connected components):

$$T = 2V + 4(H - S)$$

The formula gives a relation between some parameters of the extracted surface. As the vertices are fixed, only the other three can vary, the relationship between them being contained in the formula. Another important property is the number of border edges in the mesh, and this one is related with the number of vertices. Every vertex of the mesh is in a stick of the voxelization, thus being in four faces. Given that every face has two border edges, one for each of the two cubes concurring in that face, any vertex is in eight border edges. At the same time, a border edge is delimited by two vertices. This two conditions together enables us to obtain the next formula:

$$B = 4V$$

where B is the total number of border edges in the final mesh. As B only depends on the number of vertices, it's constant.

Now, some relation of these parameters is needed at cube level. For obtaining this kind of result we will first look at one cube. Assuming that loops inside this given cube are maintained separately (without connecting them, as it can be done in X-cubes), the number of triangles can be easily determined. For one loop the formula is $t_l = b_l - 2$, where t_l is the number of triangles for the loop, while b_l corresponds to the number of border edges that compose that loop. In one cube, the previous formula can be summed for every loop inside the cube. The result is $t_c = b_c - 2l_c$, where t_c is the number of triangles resulting from the triangulation of all the loops inside the cube, b_c the number of border edges in the cube, and l_c the number of loops inside it. Considering previous formulas and the fact that the number of sheets on a cube is equal to the number of loops (as we don't connect them), a sum for every cube can be computed:

$$T = B - 2L = 4V - 2L = 4V - 2L - 4(s - L) = 4V - 4s + 2L$$

s is the total number of sheets on every cube. The previous proof does not take the possibility of connecting loops into account, but only one operation in the presented algorithm makes such kind of change. An X-cube can be polygonized connecting its two loops. Such operation transforms one sheet into two, but also requires four more triangles, so the previous formula holds for X-cubes disambiguation.

X-face slashing can change two characteristics in the neighboring cubes. One is the number of loops that will be changed in exactly one between different slashings. If one slashing made the two border edges belong to the same loop, the opposite slashing will connect them decreasing the total number of loops by one. The same happens the other way, so if from one cube we decide to connect the loops (decreasing by one the number of them), the neighboring cube cannot increase its number of loops by more than one. Another topological property influenced by X-face slashing is the number of connected components, as the four vertices of an X-face can be connected with the purpose of increasing or decreasing the total number of equivalence classes of vertices (that is equivalent to the number of connected components).

Once the X-face decisions have been made, the number of loops is fixed and only T and s remain available. They are affected by X-cube decisions in opposite ways: as T increases s decreases, and vice versa.

4.4 Global structures

As choices on any given cube influence the results on neighboring cubes, a solution based solely on local decisions is not possible. Some global information is needed and it has to be stored in structures that can be efficiently accessed. In this section two structures will be shown: the X-face graph and the connectivity merge tree.

The X-face graph is basically used to store slashing decisions and how they affect each other. Every cube in the voxelization that has X-faces infers one node of the graph, while X-faces determine the link between two nodes. If two cubes are neighbors through an X-face, their nodes in the graph will be connected through an undirected edge. Thus, slashing information can be easily stored using a boolean for every edge of the graph (in Figure 4.5 an example can be found).

The graph of X-faces can be built using a single pass through the entire voxelization. For every cube that contains one or more X-faces a node is inserted into the graph. Now, we only need to insert edges for the X-faces that go in one of three directions X-, Y-, Z-. Only three of the six possible directions of every cube are considered in order to avoid edge repetition (the graph is undirected).

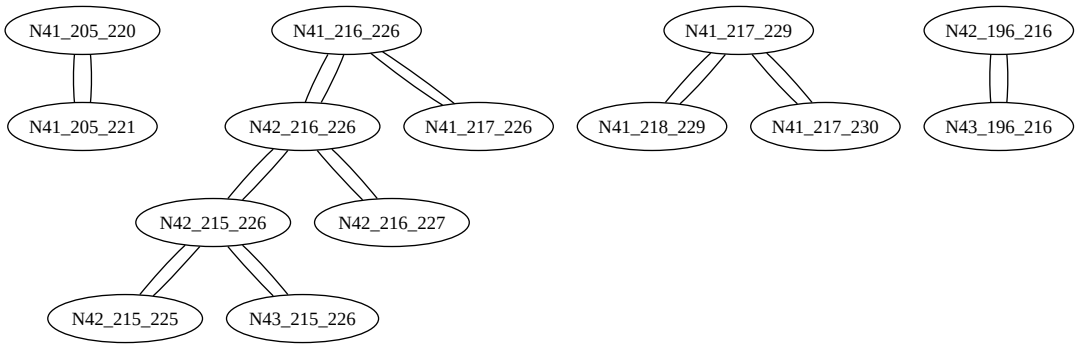


Figure 4.5: Some example components of an X-face graph.

Dataset	Resolution	Non-empty cubes	Number of X-faces				#arcs	#cycles
			1	2	3	6		
Ship	$128 \times 128 \times 128$	35020	54	21	0	0	48	0
Random 1	$8 \times 8 \times 8$	312	49	9	9	2	53	0
Random 2	$8 \times 8 \times 8$	299	34	12	8	2	47	8

Table 4.1: X-face graphs of some datasets. For each one, the resolution, number of non-empty cubes, number of different X-faces configurations, number of arcs, and number of cycles are provided. The Ship model also appears in the results section.

As only cube configurations with 0, 1, 2, 3, 6 X-faces are possible, and 0, 1, 2 X-face combinations account for 93% of the total configurations; the resulting graphs are very sparse, mostly being forests of small sized trees (in Figure 4.5 one of the biggest trees found is shown). Table 4.1 shows the results obtained from some examples. Real datasets present almost no cycles (normally none).

The second tool of the algorithm is the connectivity merge tree that allows to know if any two vertices are connected or not. Answering that is the same like answering whether those two vertices belong to the same connected component or not. The structure may be represented as a set of equivalence classes where two vertices are in the same connected component if they are in the same equivalence class. The most common structure used to represent equivalence classes is the merge tree.

Any decision on X-face slashing changes the number of loops. This also affects the number of shells or the number of handles, but only one of them. As the connectivity merge tree allows us to determine whether the decision has affected S or not, we can easily determine if the number of handles have been affected. It is also useful to determine if an X-cube decision (connecting the loops or not) changes the number of connected components (S).

A merge tree has two main operations: the first joins the classes of two members of

the merge tree (in our case, vertices), while the second allows to know if two members belong to the same class. The connectivity merge tree is initially constructed putting inside every vertex and connecting only those that belong to the same face if it's not an X-face. This considers the connectivity of the mesh, X-face decisions apart.

Every time an X-face slashing decision is made, the four vertices involved are joined properly (following the slashing). The same happens with X-cube decisions. Additionally, this structure allows the computation of the results obtained with every possible decision on X-faces or X-cubes.

Using standard structures for the implementation of the connectivity merge tree, finding the canonical representative of any class has a cost of O(log n) where n is the number of vertices of the extracted mesh. Knowing if two vertices belong to the same class is done by finding their canonical representatives and comparing them, so the cost is bounded also by O(log n). As merging operations can be done in constant time, the initialization of the data structure takes a time bounded by O(nlog n)+m, where n is the number of vertices and m the number of voxels.

4.5 Optimization algorithm

Now that we have the final structures that will be used in the optimization algorithm, its shape becomes more clear. The process consists in two steps: first determine all the X-faces, once we have them the X-cubes are processed. We need some criteria about how to slash X-faces in the first step, and another about whether to connect or not the two loops in an X-cube. The criteria for the first phase is applied to the X-face graph beginning from the leaves, that way the voxel being processed only has one X-face. This will give problems when cycles are found in the X-face graph, as we'll find components of the graph without any leaves to process. In these cases, a cut can be made (solving one of the X-faces of the cycle randomly) but this will give rise to sub-optimal solutions. In any case, real test examples almost never have cycles and the components of the graph are very small, keeping the error to very small regions of the voxelization. The X-cubes are easier to solve as they only interact with other X-cubes or X-faces through connectivity.

The optimization algorithm works as follows:

InitializeXFaceGraph(G)
InititalizeMergeTree(T)
if there are cycles in the G **then**

```
  for each graph cycle C do
    choose a random X-face of the cycle
    solve the X-face slashing randomly
  end
end
while not all X-faces have been fixed do
  ChooseOneTreeLeaf(c, f)
  FixLeaf(f, SlashingCriterion)
  UpdateMergeTree(T)
  PruneLeaf(c)
end
for each X-cube c do
  FixXcube(ConnectingCriterion)
  UpdateMergeTree(T)
end
```

The Slashing Criterion corresponds to the X-faces solving, while the Connecting Criterion is related with solving the X-cubes. After applying the optimization algorithm, the final triangulation within all cubes can be obtained using Marching Cubes.

With the previous algorithm the results only depend on the criteria used to solve the ambiguity of X-faces and X-cubes.

4.6 Slashing criteria and combinations

As we have previously seen every slashing decision of an X-face makes a difference of one loop inside any of the two neighboring cubes. These differences can compatible or not. A decision on an X-face made to minimize the number of loops in one cube can reduce or increase the number of loops on the neighboring cube (through the X-face). As a result the total number of loops (L) can be either left unaffected by this X-face decision or be decreased by two. If slashing is done to maximize the number of loops, the operation may leave L unaffected or increased by two. These two criteria are useful for minimizing or maximizing the number of loops.

Another pair of possible criteria are those that minimize or maximize the number of shells using the connectivity merge tree. The slashing criteria are:

- **Criterion 1**: Take the option that maximizes the number of loops inside the current cube. This clearly tends to maximize the number of loops.

- **Criterion 2**: Take the option that minimizes L inside the current cube.
- **Criterion 3**: Choose the slashing that does not decrease the number of equivalence classes. This tends to maximize the number of shells (S), but it can only be applied if possible.
- **Criterion 4**: Take the option that decreases the number of equivalence classes (also if possible).

The X-cubes decision consists only in deciding if the remaining loops (those of the X-cubes) have to be connected or not. When the loops are connected four new triangles are needed, and due to $T = 2V + 4(H - S)$, an increase in $(H - S)$ is obtained. The criteria used are:

- **Criterion a**: Never connect loops. This option decreases $(H - S)$ while maintaining S low (it never connects equivalence classes). As a result, many disconnected shells are obtained, and the genus is kept low.
- **Criterion b**: Always connect loops. It yields an increase in $(H - S)$ and a decrease in S, resulting in a low S and a high H.
- **Criterion c**: Two loops of an X-cube are only connected if they belong to the same equivalence class. The connection produces an increase in $(H - S)$, but as they belong to the same equivalence class, S is unaffected, while H increases.
- **Criterion d**: Two loops are connected when they do not belong to a same equivalence class. $(H - S)$ is increased as a result of every connection, but as S is decreased by one, H is left constant.

From the combination of the above criteria 16 combinations can be applied to the optimization algorithm, but only four of them have been found useful. In order to show the optimizing properties of these combined criteria special terminology is used. When a given combination is said to obtain the minimum of a certain topological property W it's expressed as min(W). The same is applied when the maximum, a high value or a low one are obtained, using max(W), high(W), and low(W) respectively.

The four combinations are:

- **Case 1-a**: Option 1 maximizes the number of loops and since no loops are connected in X-cubes, we have $s = L$. As we also have $T = 4V + 2L - 4s = 4V - 2L$, a minimization of the number of triangles is obtained. On the other

hand, $T = 2V + 4(H - S)$, which leads to a minimization of $(H - S)$, with a low H and high S (as X-cubes never get connected). The results are $min(T)$, $min(H - S)$, $low(H)$, and $high(S)$.

- **Case 2-b**: L is minimized while $s \geq L$ (X-cubes loops are always connected). This leads to $4(H - S) = T - 2V = 2V + 2L - 4s \leq 2V - 2L$, and $high(H - S)$ with a $high(H)$, and $low(S)$ (X-cubes being connected cannot increase S).
- **Case 3-c**: Both option 3 and c maximize S, while c also increases H. The result is $max(S)$, and $high(H)$.
- **Case 4-d**: This combined criteria does the opposite of what case 3-c does, obtaining $min(S)$, and $low(H)$.

There are cases where some criterion does not determine which of the two options (slashing or not, connecting or not) has to be taken. These situations appear because there are several combinations that are optimum. One possibility in these cases is to choose randomly. The second one is to take two ordered criteria, when the first does not apply the second is used.

4.7 Results

In order to show the different results of the combined criteria, a synthetic example has been used. The results of the algorithm are shown in Figure 4.6, while some numbers are found in Table 4.2. This academic model has a resolution of $16 \times 16 \times 16$ and 4096 voxels. Of them only 240 have X-faces, and there are 40 X-cubes.

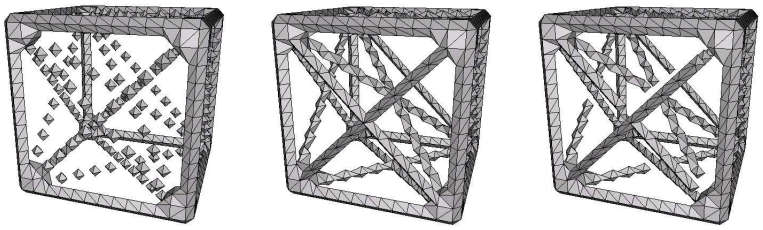


Figure 4.6: Results of the test cube, using combined strategies 1-a, 2-b, and 4-d.

In the table the behaviors of the different combined criteria can be observed. While 1-a and 3-c minimize the number of triangles, 2-b and 4-d minimize the number of shells, but 2-b maximizes the number of handles and 4-d minimizes it.

	#Vertices	#Triangles	#Shells	#Handles
1-a	1410	2296	136	5
2-b	1410	2936	1	30
3-c	1410	2296	136	5
4-d	1410	2836	1	5

Table 4.2: Number of vertices, triangles, shells, and handles of the test cube.

The same criteria applied to a realistic example yields the results shown in Figure 4.7, and the numbers in Table 4.3. The model is the voxelization of a ship's engine at a resolution of $256 \times 256 \times 256$. It's clearly seen that the criteria 3-c maximizes the number of shells, and 2-b minimizes the number of connected components (shells) while maximizing the genus (number of handles).

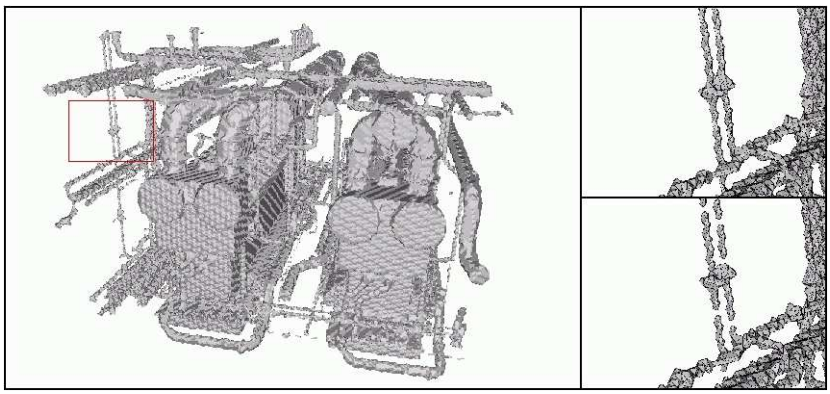


Figure 4.7: The ship's engine model. The right images compare the result of applying strategies 2-b and 3-c.

	#Vertices	#Triangles	#Shells	#Handles
1-a	289806	575776	1182	223
2-b	289806	583932	339	1419
3-c	289806	577232	1185	590
4-d	289806	580076	340	456

Table 4.3: Numbers of the ship's engine model.

The test models were run on a Pentium IV at 1.7 GHz with 256 Mb of RAM. The processing of the ship's model gave an overhead time (above the conventional marching cubes extraction algorithm) of 5.62 seconds. From that 0.98 seconds were spent constructing the X-face graph, 4.23 to build the connectivity merge tree, and

0.41 seconds to solve the graph (with one combined criteria). It can be seen that most of time is spent building the merge tree as it has as much elements as vertices has the extracted mesh.

On the previous section, we have seen four of the possible sixteen criteria combinations. The twelve remaining combinations take conflicting decisions, but its effects need to be further researched. Additionally, while being very rare, cycles do appear in some models (mostly, biomedical ones). Cutting them randomly is not the best option, if we intend to optimize the topological properties of the isosurface. A better system is needed, but not at the cost of examining all the possible cuts and its results.

Chapter 5

Maximal tiles extraction

5.1 Problem statement

Computing the largest planar region (tile) that approximates the surface of a geometric model is useful in a number of applications such as face clustering, impostor-based simplification, and finding object signatures. This chapter covers the results presented in [ABC+04] regarding the retrieval of maximal tiles and its direct application to appearance-preserving simplification.

This problem has been addressed in several contributions, like the solution proposed in Decoret et al [DDSD03] and the tile building algorithm from Andújar et al [ABE+03]. Optimality guarantees, shorter running times, and independence from the number of geometric primitives of the model are the primary goals of the contribution presented in this report.

Although one of the possible applications of finding maximal tiles is face clustering, the solution proposed here is different from those which grow and merge faces like Garland et al [GWH01] and Kalvin and Taylor in [KT96]. The main difference with them is that tiles may approximate many separate parts of the surface.

In order to obtain the n largest tiles approximating the surface of a given 3D model, an efficient algorithm that computes the largest tile is needed. Applying this algorithm several times yields the required tiles. Given a voxelization of the 3D object, a grid of black - exterior - and white - interior - points is defined. The edges connecting a black point with a white one (called sticks) will be used in a voting-based approach to select the largest tile. In fact, the problem gets transformed to finding the plane that intersects the maximum number of sticks. The desired algorithm not only has to be efficient, but it has to provide the maximal tile given a sufficiently dense discretization of the space of candidate planes.

5.2 Plane space parameterization properties

The efficiency of such an algorithm is clearly dependent on the choice of the parameterization of the planes' space, and thus some properties are required. As the method has been reduced to vote for every region of nodes, and this region infers a set of planes, one can impose over the region of stabbing planes the restriction of being a convex shape. This makes the voting operation cheaper (as it only uses point-plane tests), this restriction being condensed in the following definition:

> **Definition 1 (Linear parameterization of planes)** *A parameterization of planes is called linear if the constraint that a point is inside or outside the half-space defined by a plane is represented by a linear constraint in the parameterization space.*

Another important property is that the resolution of the discretization of the plane space is enough to capture any stabbing plane in the input voxelization. In fact, any plane crossing the universe (the region containing all the white and black nodes), has to intersect 3, 4, 5, or 6 edges of the universe cube. All these possible cases can be reduced to six, as shown in Figure 5.1, with 4.1 being the same as 5.1, and 4.2 the same as 5.2. This condition is described in:

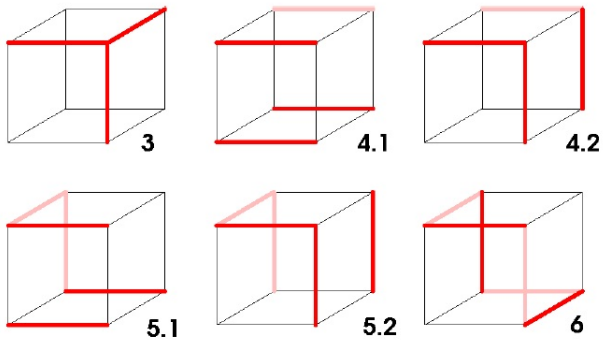


Figure 5.1: Possible intersections with a plane. The edges used to define faithfulness are marked in red.

> **Definition 2 (Faithful discrete parameterization)** *A discrete parameterization of planes is faithful if given any three different voxels on any of the possible combinations of three sides in red in Figure 5.1, there is a plane representable in the discrete parameterization that intersects all three voxels.*

But faithfulness as defined is not enough, since any parameterization can accomplish it using a sufficiently dense discretization. Some condition has to be used to obtain uniformness of the parameterization, and the following definition shows how:

> **Definition 3 (c-Concise faithful parameterization)** *A faithful discrete parameterization of planes is said to be c-Concise for* $c \epsilon \mathbb{N}$ *if the total number of planes representable in that parameterization divided by the total number of triplets of voxels used to test faithfulness is smaller or equal to c.*

Due to deficient uniformness of the Hough parameterization, it needs a very dense discretization to be faithful, being unable to accomplish the c-Concise condition for a reasonable c. The Hough parameterization is used in [DDSD03].

5.3 Stick neighborhoods

It has been previously pointed that the entire voxelization has to be divided into stick regions. Those stick regions have an associated set of stabbing planes that will be used in the voting algorithm to obtain the largest approximating plane. In order to achieve a voting algorithm that works traversing the sticks of the model, issuing one vote per stick, every stick region has to be centered around one stick.

The stick region, which we will call stick neighborhood, has the shape of a box centered around a stick. Smaller boxes lead to more precise results, but at the cost of larger sets of stabbing planes for a given stick neighborhood, and thus longer times to compute the vote for that stick. Greater boxes are more efficient when voting, but it is more difficult to find sticks whose neighborhood has a non empty set of stabbing planes, leading to lower accuracy.

In a compromise between efficiency and accuracy a box of size $5 \times 5 \times 4$ has been used. This decision causes that not every stick gets to vote, as those that are in non planar regions of the surface will not have any plane that separates black and white nodes. As a result, sticks will be considered planar or not depending on if the set of stabbing planes is non empty or not, respectively.

An example of a planar neighborhood and its region of stabbing planes can be seen in Figure 5.2.

In the previous outline of the algorithm, for every stick, its neighborhood is extracted and the set of stabbing planes computed. Using a $5 \times 5 \times 4$ box limits the number of possible planar combinations, so a look-up table can be assembled from

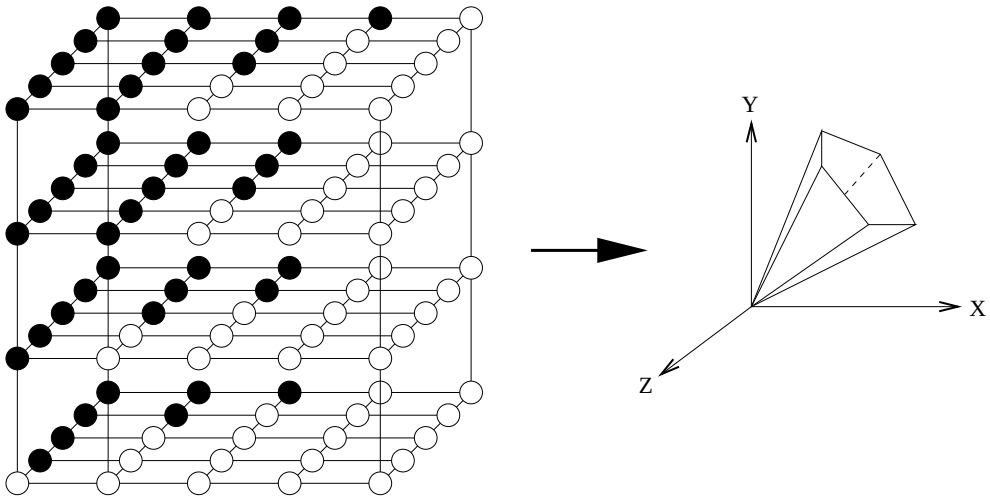


Figure 5.2: The neighborhood of sticks, and the corresponding 4D open cone.

those. This table or dictionary of planar combinations is be used to speed the voting process.

The convex regions of stabbing planes can also be stored with the corresponding planar stick neighborhoods. Those regions need to be independent of the stick neighborhood location in the voxelization, therefore a local plane parameterization is needed to encode them. Besides, this gives rise to the necessity of translating the regions of stabbing planes from the local plane parameterization to a global one. It is in this global parameterization that the vote process will take place.

Efficiency concerns have lead to the differentiation of two plane parameterizations, a local one to the corresponding stick neighborhood, and a global one needed to unify the representation of stick votes. The global parameterization has to observe certain properties, and the operation that transforms from the local one to the global one has to be efficient.

5.4 Local parameterization

The local parameterization is chosen to give compact representations for the regions of stabbing planes. A plane is represented by four weights on the vertices of an orthogonal tetrahedron (as seen in Figure 5.3). These weights can be interpreted as the signed distances to the represented plane, or as four intensities in a 3D image where the resulting zero isosurface is the plane.

The weights of the tetrahedron four points determine a 4D space where every

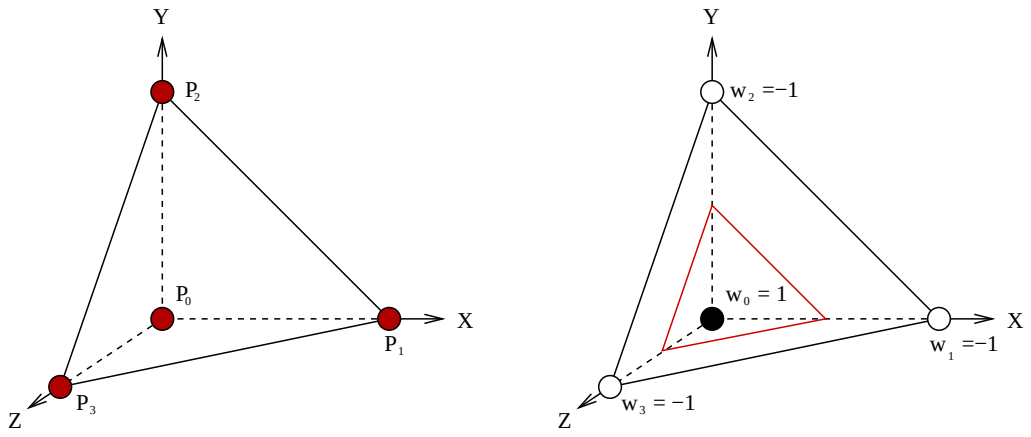


Figure 5.3: Tetrahedron used for the local parameterization of planes and plane represented by four weights (the plane is drawn in red).

point is a plane. The plane defines a signed distance scalar field and the weights are the values of the field at the tetrahedron points. For any point P, the value of the scalar field at P can be expressed as $\alpha_0 \cdot w_0 + \alpha_1 \cdot w_1 + \alpha_2 \cdot w_2 + \alpha_3 \cdot w_3$ where $\alpha_0, \alpha_1, \alpha_2, \alpha_3$ are the barycentric coordinates of the point P. From this it becomes easy to transform any interior or exterior point into a linear restriction. If P is an exterior point, the value of the scalar field has to be greater or equal to zero ($\alpha_0 \cdot w_0 + \alpha_1 \cdot w_1 + \alpha_2 \cdot w_2 + \alpha_3 \cdot w_3 \geq 0$), while if P is an interior point the field value has to be smaller than zero ($\alpha_0 \cdot w_0 + \alpha_1 \cdot w_1 + \alpha_2 \cdot w_2 + \alpha_3 \cdot w_3 < 0$).

As we have seen, any 3D plane may be represented using a tuple of four weights (w_0, w_1, w_2, w_3), but due to how this parameterization works, $(\lambda \cdot w_0, \lambda \cdot w_1, \lambda \cdot w_2, \lambda \cdot w_3)$ represents the same plane for any real value of λ. This means that 3D planes are represented in our 4D representation as rays that pass through the origin, as illustrated in Figure 5.4.

Therefore, 3D planes are represented as 4D points that result from the intersection of the 4D line of points defined by the plane and a 4D hypercube centered at the origin. The line is intersected with the faces of the hypercube, which are 8 3D cubes that correspond to the minimum and maximum of each of the four weights. Without loss of generality, we may define a hypercube that has $w_k \leq 1$, such that any oriented plane has an unambiguous representation as a point in the its surface.

Additionally, every point restriction is equivalent to a halfspace in the 4D plane space, and the 4D hyperplane that delimits that halfspace always passes through the origin. As a consequence, the resulting shape for a set of restrictions is a 4D hypercone that can be computed using only two tools, an initial cone and a method

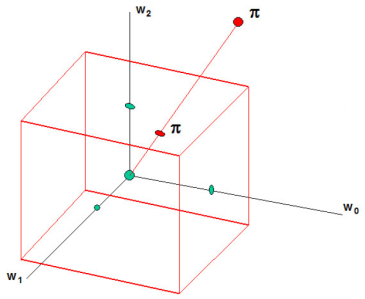


Figure 5.4: Any point on the line through the origin and point π defines the same plane. Thus, planes may be represented by the intersection of the line and a cube centered at the origin.

for adding a new hyperplane to the actual cone.

The initial cone can be computed from the points on the vertices of the tetrahedron, as four restrictions are the minimum needed to obtain a 4D cone, and these four points result in orthogonal hyperplanes. In fact, the initial cone will be one of the 16 4D hyper-octants on the 4D space of planes.

Adding an hyperplane to the cone implies some steps. First, a test for finding if the cone is completely interior or exterior to that hyperplane has to be performed. If the cone is completely interior, the restriction is irrelevant and can be thus ignored, but if it's completely exterior, a null cone is obtained (no plane separates the black and white points). Otherwise the cone has to be updated, dividing its hyperfaces between completely interior, completely exterior, and intersecting (as the 3D example on Figure 5.5). Interior hyperfaces are left untouched, while exterior ones are discarded (their planes are considered irrelevant). The intersecting hyperfaces are then updated in the same manner (because a 4D hyperface is a 3D pyramid). There's only one step left, and that is closing the result with a new hyperface obtained from the intersections of the added hyperplane with the initial hyperfaces.

5.5 Precomputing the dictionary

Now that we have a way to compute if a given stick neighborhood configuration is planar or not, and the resulting 4D cone of stabbing planes can be obtained, a look-up table (dictionary) of planes can be precomputed. A first idea would be to use a brute force test of every possible configuration. In the case of a $3 \times 3 \times 2$ neighborhood this method works quite fast because there are only $2^{3*3*2} = 2^{18} \simeq 256K$ configurations.

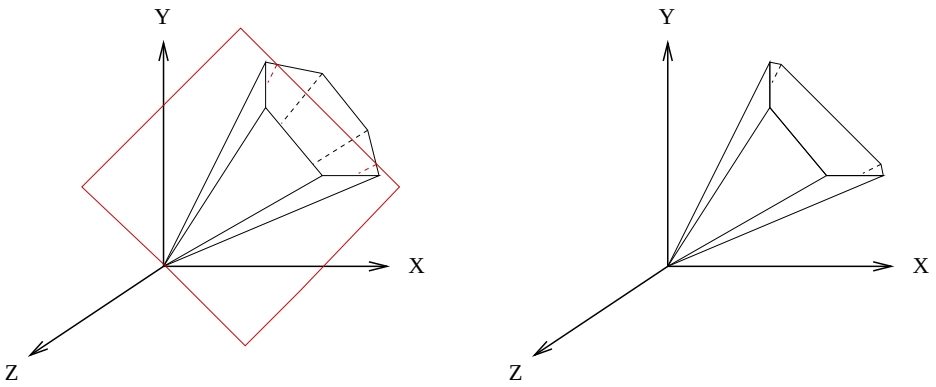


Figure 5.5: 3D example of the addition of a plane to a pyramid.

But it's clearly impossible to apply it to $5 \times 5 \times 4$ neighborhoods since there are $2^{5*5*4} = 2^{100} \simeq 10^{30}$ configurations to test. A different algorithm which reduces the number of configurations is needed in order to obtain the dictionary.

The main idea is to see the $5 \times 5 \times 4$ neighborhood as the combination of four 5x5 slices of white and black points. Since we're only interested in separable configurations, the tested configurations can be reduced to those that are combinations of 4 5x5 separable slices (with the exception of the all-white and all-black slices).

The first step is to compute all valid (separable) slices. For this, two methods can be used, but both rely on a method to test if a 5x5 black and white configuration is separable or not. This problem is solved considering the points as 2D, and looking if the set of lines which separate the black points from the white ones is void or not (the same as with planes, as shown in Figure 5.6).

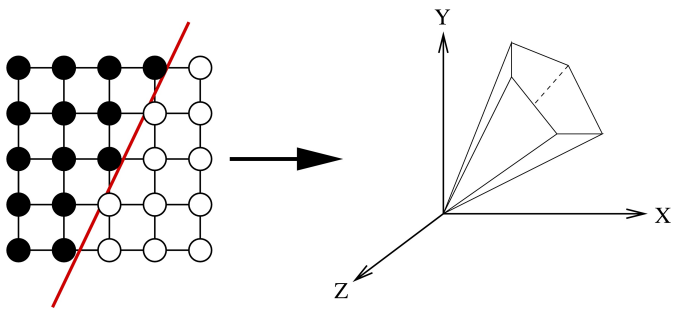


Figure 5.6: 3D region of 2D separating lines inferred by a set of black and white points.

As a result around 500 separable slices are obtained. Now these are combined to give rise to $5 \times 5 \times 4$ planar neighborhood candidates which are tested for separability.

But one more check can be applied before trying to build the 4D cone of valid planes. When the first slice is chosen, it determines the direction of the central stick (this can be seen in Figure 5.7). This fixes the type (black or white) of the central point of the other three slices, as well as restricts the sticks that follow the same axis as the central stick to follow its same direction (as seen in Figure 5.8).

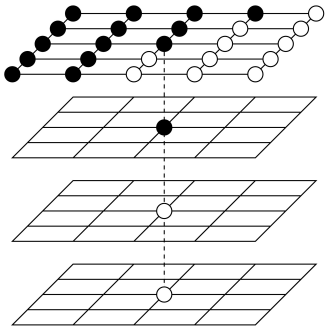


Figure 5.7: The first slice determines the central stick of the $5 \times 5 \times 4$ neighborhood.

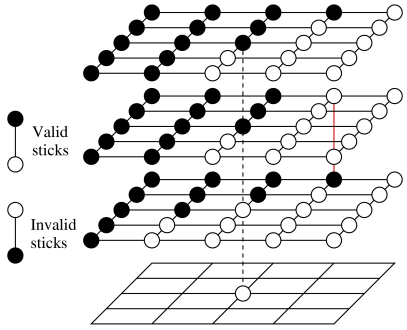


Figure 5.8: Once the central stick is fixed, it restricts the direction of the sticks that follow the same axis. The second slice is compatible with the first one, but that's not true for the third (the invalid stick is shown in red).

With these methods many cases has been discarded quickly. Now the 4D cone is computed to the resulting configurations and only those that lead to non-null regions of stabbing planes are stored.

The dictionary is stored as a hash table of the valid cases, with the relevant hyperplanes for each entry. The important properties of the obtained dictionary are:

- Only three types of cones are obtained, with a total of 4, 6, or 8 relevant hyperplanes. Th rest of the constraints are irrelevant.
- The components of 1D directed edges of the 4D cones are integers with values in the range $[-36, 36]$ and can be thus stored as signed bytes.

- From the relevant hyperplanes of a 4D cone, the exact geometry can be reconstructed in an average time of 0.39 milliseconds in a Pentium IV running at 1.7 GHz.
- There are 32,986 entries in the dictionary that represent every separable $5 \times 5 \times 4$ configuration, and 442 $3 \times 3 \times 2$ planar combinations. These results are considering only one of three possible stick directions.
- The dictionary occupies a total of 264 KB.
- The result is a local and tolerance-independent look-up table.

5.6 Global parameterization

With the local parameterization described above, the region of planes assigned to every stick neighborhood can be easily extracted. This region is used to cast the vote of that stick, but a common parameterization is needed in order to do so. The goal is to obtain a plane that it's inside the maximum number of those regions, and this will be achieved using the voting algorithm outlined in the next section. As the global plane parameterization needs to be discretized in order to vote, the chosen parameterization needs to accomplish certain properties, like being faithful, linear, and easy to obtain from the local parameterization.

A plane representation consisting of an integer identifying a choice of three edges of the Universe cube, the stabbing points on these edges and a bit for the orientation of the plane is (obviously) faithful but it is not linear. Also the naive parameterization of planes by their axis intercepts is non linear: if we describe a plane by the triplet of numbers (x, y, z) such that in a fixed reference system the points $(x, 0, 0)$, $(0, y, 0)$ and $(0, 0, z)$ lie on the plane, then the condition that a certain point $P = (\overline{x}, \overline{y}, \overline{z})$ is on the plane is $\overline{x}yz + \overline{y}xz + \overline{z}xy - \overline{xyz} = 0$, which is obviously non linear in the parameters x, y and z (although it is linear on each of them separately).

The global parameterization we use is given by a set of four weights (w_0, w_1, w_2, w_3) on the four points of the tetrahedron shown in Figure 5.9.

Given a point $P = (x, y, z)$ with barycentric coordinates $\alpha_0, \alpha_1, \alpha_2, \alpha_3$, the value on P of the scalar field defined by a plane is: $\alpha_0 \cdot w_0 + \alpha_1 \cdot w_1 + \alpha_2 \cdot w_2 + \alpha_3 \cdot w_3$. Therefore, it is easy to define linear restrictions from interior and exterior points, and the resulting regions are also 4D hypercones. Besides, translating one given restriction from the local space to the global one can be done obtaining the barycentric coordinates of the corresponding point with respect to the global tetrahedron.

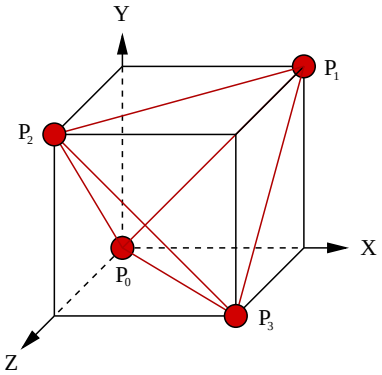


Figure 5.9: Tetrahedron used for the global parameterization of planes.

Another important property is linearity, which ensures that regions of stabbing planes are convex. In fact, it's clear that the intersections of the 4D cones with the faces of the hypercube are convex polyhedra, but it can be proved showing that the intersection of a 4D hyperplane (one of the restrictions) with one face of the hypercube is a plane. Every restriction has the form:

$$k_0 \cdot w_0 + k_1 \cdot w_1 + k_2 \cdot w_2 + k_3 \cdot w_3 \geq 0 \tag{5.1}$$

Obtaining the intersection with face $w_0 = 1$, as the other 7 cases are symmetric:

$$k_0 + k_1 \cdot w_1 + k_2 \cdot w_2 + k_3 \cdot w_3 \geq 0 \Rightarrow k_1 \cdot w_1 + k_2 \cdot w_2 + k_3 \cdot w_3 + k_0 \geq 0 \tag{5.2}$$

results in the equation of a plane with coefficients $a = k_1, b = k_2, c = k_3, d = k_0$.

In order to compare faithfulness between the Hough parameterization and the Connected Cubes parameterization proposed here, some method has to be devised. Such a method has to evaluate the faithfulness of a given discretization of each of the plane parameterizations.

The system proposed here consists in, for each triplet of voxels, searching among the planes from the discretization one that intersects those three voxels. The percentage of triplets that are captured by the discretization will be used as a measure of its faithfulness.

Table 5.1 shows faithfulness test results of the Hough distance transform for different model resolutions, being divided into four sub-tables, one for each resolution (shown as N in the first row). Each sub-table has two columns. The first one makes reference to the subdivision factor of the discretization of the plane parameterization

space, with respect to the model resolution. As an example, if the model has a resolution of N=8 and the chosen subdivision factor is 3, the total size of the discretization (of the plane space) is $24 \times 24 \times 24$. The second column presents the percentage of captured triplets of voxels.

N=8		N=16		N=32		N=64	
Subdivision factor	**Percentage captured**	**Subdivision factor**	**Percentage captured**	**Subdivision factor**	**Percentage captured**	**Subdivision factor**	**Percentage captured**
1	26.56%	1	21.97%	1	22.07%	1	21.13%
2	86.13%	2	81.78%	2	78.63%	2	77.75%
3	99.80%	3	99.03%	3	98.35%	3	98.32%
4	100%	4	100%	4	100%	4	100%

Table 5.1: Faithfulness results for Hough parameterization. A subdivision factor of 3 is enough.

Table 5.2 shows the result for the Connected Cubes parameterization. It can be noticed that Connected Cubes has a maximum faithfulness for a given resolution (it cannot achieve 100% capture). This effect is produced because the parameterization cannot capture the planes that are almost coplanar to universe cube faces. But every model has an exterior cover of outside nodes, making these planes unnecessary. The final difference between the parameterizations may seem small, but the subdivision factor applies to the three dimensions of the model resolution. This means that a subdivision factor of 2 makes a discretization that contains $2 \times 2 \times 2 = 8$ times the cells of the model, while a factor of 3 makes one with $3 \times 3 \times 3 = 27$ times the cells of the model. This results in a difference of more than three times.

N=8		N=16		N=32		N=64	
Subdivision factor	**Percentage captured**	**Subdivision factor**	**Percentage captured**	**Subdivision factor**	**Percentage captured**	**Subdivision factor**	**Percentage captured**
1	94.48%	1	90.87%	1	88.10%	1	85.30%
2	98.53%	2	99.62%	2	99.90%	2	99.98%
3	98.53%	3	99.62%	3	99.90%	3	99.98%

Table 5.2: Results for Connected Cubes. The maximum faithfulness is achieved with a subdivision factor of two.

The conclusion is that a discretization of $2N \times 2N \times 2N$ in the eight parameterization cubes is necessary in order to ensure that the convex polyhedral regions contain at least one represented plane. This guarantees the faithful property.

From the last property, we know that a discretization of $2N \times 2N \times 2N$ in the eight parameterization cubes is necessary in order to ensure the faithful property. This gives a total of $8(2N)^3 = 8 \times 8(N^3)$ planes that can be represented.

In contrast, the well-known Hough representation (spherical coordinates), must be discretized in a finer way to be faithful: the required discretization is $3N \times 3N \times$

$3N$. The Hough parameterization is 3-concise, with a total of $8(3N)^3 = 8 \times 27(N^3)$ represented planes.

According to Figure 5.1, any plane may intersect the universe cube in 3, 4, 5 or 6 edges, and the intersection cases can be reduced to only four configurations, as the cases 4.1 and 5.1 (resp. 4.2 and 5.2) can be represented with the same sides of the cube. Let us count the number of planes represented by each of these configurations:

- The configuration 3 represents N^3 planes (combinations of intersecting voxels in the edges), and this configuration appears in eight possible orientations in the Universe, around each of its corners. The total of represented planes is therefore $8 \times N^3$.

- The configurations 4.1 and 5.1 represent N^3 planes each, and this configuration appears in twelve possible orientations in the Universe: Three possible orthogonal orientations, and four possible locations of the non-red edge in each case. The total of represented planes is therefore $12 \times N^3$.

- The configurations 4.2 and 5.2 represent N^3 planes (combinations of intersecting voxels in the edges), and this configuration appears in twenty-four possible orientations in the Universe: Twelve edges, two configurations for each of them (only rotations must be taken into account, as symmetries switch to a different configuration). The total of represented planes is therefore $24 \times N^3$.

- The configuration 6 represents N^3 planes, and this configuration appears in eight possible orientations in the Universe: Four pairs of diagonally opposed vertices, two sets of alternating edges in the unique path that does not contain these two vertices in each case. The total of represented planes is therefore $8 \times N^3$.

In short, there are $(8 + 12 + 24 + 8) \times (N^3) = 52(N^3)$ planes represented by the configurations in Figure 5.1. As we are in fact representing $8(3N)^3$ planes, the Connected-Cubes plane representation is $8 \times 27/52$ concise.

One important fact for the discretization of the plane space is that, as we have already seen for the local parameterization, points of the form $(k \cdot w_0,\ k \cdot w_1,\ k \cdot w_2,\ k \cdot w_3)$ encode the same plane for any real $k \neq 0$. As a result, any given plane is represented by a line that passes through $(0, 0, 0, 0)$ and $(w_0,\ w_1,\ w_2,\ w_3)$ and we will only consider the intersection of the 4D hypercones with the boundary of the hypercube centered at the origin. Using this, the redundancy is removed, and the

vote space is reduced to the 8 faces of the hypercube, that being 3D cubes will be discretized to contain the votes.

Every cube has a singular vertex, and they correspond to the two intersections of the line $w_0 = w_1 = w_2 = w_3$ with the boundary of the hypercube. Those points represent a constant scalar field without any zero isosurface.

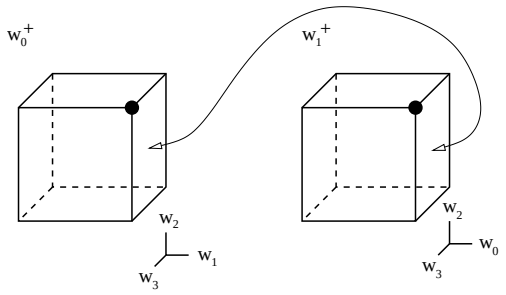


Figure 5.10: Two neighbor cubes of the global parameterization with their singular points shown.

The boundary of the hypercube is a 3D closed variety (this means that every cube has a neighbor through each face -see Figure 5.10-), composed of 8 cubes, 24 faces, 32 edges, and 16 vertices. The vertices correspond to all the possible combinations of $w_k = \pm 1$ ($(-1,-1,-1,-1)$, ..., $(1,1,1,1)$), while the 24 faces correspond the combinations of $(w_k = 1, w_l = 1)$, $(w_k = 1, w_l = -1)$ and $(w_k = -1, w_l = -1)$ when $k \neq l$. In fact, this representation maps the 4D Gauss sphere (see Figure 5.11).

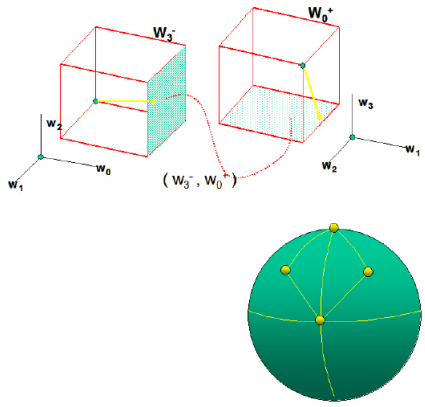


Figure 5.11: Two of the eight cubes and their joint significant face. This significant face (w_3^-, w_0^+) corresponds to the rombic region in the Gauss sphere represented below.

5.7 Voting algorithm

Given the previously described tools, the voting algorithm can be outlined as follows:

```
for all stick s in the model do
  if FoundInDictionary(s) then
    Compute R as Stab(Neighborhood(s))
    for each cell C of the parameterization of planes
     such that NonNullIntersection(C, R) do
      Add a vote from s to the vote counter of C.
    end for
  end if
end for
```

This outline does not count with the used multigrid approach, as it makes the algorithm more complex. Given a $N \times N \times N$ voxelization, a discretization of $2N \times 2N \times 2N$ needs to be applied to the plane space for getting a faithful parameterization. This discretization needs to be subdivided in levels in order to adapt to the multigrid approach.

As multigrid can be applied using different strategies, we compared them in terms of efficiency. First, the behavior of the algorithm using a multigrid scheme was tested, and then new strategies were derived from the results. Our initial multigrid approach subdivided plane space into two levels of equal size. Tables 5.3 and 5.4 contain running times for this method.

Models	**First phase**	**Second phase**	**Third phase**	**Total**
TAXI-64	16	86	233	336
TAXI-128	80	403	2645	3128
HELICOPTER-128	11	53	466	530
BUDDA-64	1	7	39	48
BUDDA-128	5	48	428	481

Table 5.3: Mean running times (measured in milliseconds) per plane for a multigrid solution of two levels of equal size, using $5 \times 5 \times 4$ neighborhoods.

As can be noted in these two tables, the running time spent in the second and third phases of the algorithm (those that depend on the size of the second level of the multigrid approach) is greater than the time spent in the first phase. In order to achieve better performance, the first level can be made larger than the second one.

Models	First phase	Second phase	Third phase	Total
TAXI-64	35	111	453	600
TAXI-128	179	623	6515	7317
HELICOPTER-128	190	330	4832	5353
BUDDA-64	21	64	1558	1645
BUDDA-128	114	351	19961	20427

Table 5.4: Mean running times (in milliseconds) per plane for a multigrid solution of two levels of equal size, using $3 \times 3 \times 2$ neighborhoods.

As this operation makes the second and third phases shorter to compute, and the first one longer, a subdivision has to be found that approaches the optimum total time per plane. Tables 5.5 and 5.6 contain the resulting times for this strategy.

Models	First phase	Second phase	Third phase	Total
TAXI-64	58	22	56	136
TAXI-128	264	128	1048	1441
HELICOPTER-128	40	14	152	207
BUDDA-64	4	1	8	14
BUDDA-128	17	14	170	202

Table 5.5: Mean resulting times (in milliseconds) per plane for a multigrid solution of two levels, the first one being bigger than the second. The neighborhood's size is 5*5*4.

Models	First phase	Second phase	Third phase	Total
TAXI-64	176	31	112	319
TAXI-128	1020	256	2722	3999
HELICOPTER-128	731	68	1155	1955
BUDDA-64	240	32	915	1188
BUDDA-128	614	117	10254	10986

Table 5.6: Mean running times (in milliseconds) per plane for a multigrid solution of two levels, the first one being bigger than the second. The neighborhood's size is 3*3*2.

Now the first and third phases spend most of the time. In some cases, the third phase turns to be much larger than the rest of the algorithm. This effect is produced because there are many cells that can have more votes than the first candidate computed by the first and second phases. All these cells have to be refined and this takes much time. In order to decrease the time taken by each of these cells, a subdivision in cells of size $2 \times 2 \times 2$ is applied to the second level of the multigrid solution. As

before, the size of the first level has to be adjusted to approach the optimum of the total time per plane. Tables 5.7 and 5.8 show the results for a multigrid solution consisting of a first level of size $16 \times 16 \times 16$, and as much $2 \times 2 \times 2$ sized levels as necessary (according to the model's resolution).

Models	**First phase**	**Second phase**	**Third phase**	**Total**
TAXI-64	10	10	20	40
TAXI-128	28	53	250	331
HELICOPTER-128	5	5	57	67
BUDDA-64	2	2	20	24
BUDDA-128	3	8	367	378

Table 5.7: Mean times per plane resulting from a multigrid solution of a $16 \times 16 \times 16$ first level and as much as necessary $2 \times 2 \times 2$ levels. The size of the neighborhood is $5 \times 5 \times 4$.

Models	**First phase**	**Second phase**	**Third phase**	**Total**
TAXI-64	28	18	125	171
TAXI-128	78	81	666	825
HELICOPTER-128	65	17	402	484
BUDDA-64	13	10	718	741
BUDDA-128	43	50	6520	6613

Table 5.8: Mean resulting times per plane from a multigrid solution consisting of a first level of size $16 \times 16 \times 16$, and multiple $2 \times 2 \times 2$ sized levels. Used neighborhood's size is $3 \times 3 \times 2$.

All timings of the different variants of the algorithm have been measured on a computer with a Pentium 4 at 1.7 GHz and 512 MB of memory.

Therefore, the first level consists of eight grids of $16 \times 16 \times 16$, with all the other levels using a $2 \times 2 \times 2$ subdivision (like an octree) for going deeper until a $2N \times 2N \times 2N$ discretization is achieved.

With this division into different levels the voting algorithm gets also divided into three phases. In the first phase, all the voxelization sticks are traversed, voting into the first level of the multigrid subdivision. From its results the most voted cell is obtained and used in the next phase.

Once this first phase is completed a second one is used to subdivide recursively the resulting cell until a first candidate plane is obtained. From the final number of votes, the number of stabbing sticks can be induced, but this result can be smaller than the number of votes obtained by another cell in the first level. This implies that a search

to test for better planes is needed, delimited by the quality of the first candidate (consequently, its cost is lower than a search through all the space of planes).

The third phase only has to look for cells in the first phase subdivision with more votes than the candidate plane. The complete search of the discretization of planes is bounded and, thus, accelerated by the results of the first two phases. In order to get more acceleration, the candidate refinement process of the third phase can be parallelized.

Applying the complete voting algorithm recursively yields a set of approximating planes. At every step, the sticks stabbed by a certain plane are removed from the voting process, as they already have a representative plane.

5.8 Appearance-preserving application

Now that we have a set of tiles that approximate the surface of a given object, and they are the largest ones, we can use them to compute a impostor extreme simplification of the model. This use of approximating planes applied to appearance-preserving simplification is also described in [DDSD03]. The voting algorithm has provided a set of tiles that may be used to compute a group of quadrilaterals with a texture. These textures will also need an alpha component in order to have transparency.

Since what it's needed for every impostor is the near surface of the model, an orthographic projection of the object on the tile plane is enough. This operation can be done using OpenGL, but it needs bounds for the near and the far plane. Clearly, the distance between these two planes have to be strongly related with the maximum distance that tiles use for capturing sticks. This capture happens after every voting step, removing from the voting process those sticks that have voted for the selected plane, so it is limited by the length of a stick (or the resolution of the voxelization).

Using the above idea is very simple to obtain the impostors for a given set of planes, using the graphics hardware. For every tile, the camera has to be aligned with the normal vector of the current plane, using an orthographic projection of the appropriate size (for example, the 2D bounding box of the projection of every model's vertex on the tile plane). Before doing any rendering the frame buffer is cleared using a transparent color, thus making invisible the parts of the impostor where no geometry gets projected. With this preparation complete, the render of the model can be done centered on the plane within a tolerance determined by the length of the stick. Finally, the bounding box of the opaque zone is computed, unprojected, and used as a quadrilateral where the texture will be mapped. The texture is extracted

from the frame buffer, after being cut by the bounding box of the non-transparent zone.

After the process, a simplified version of the initial model is returned. It consists in a set of quadrilaterals with their corresponding textures. The impostors can be rendered in any order, as the alpha test takes place before the depth test in the OpenGL pipeline. A variation that can increase the quality of the result is the computation of a normal map besides the texture. This allows re-lighting the simplified model, but it comes at a performance cost.

5.9 Results

Figure 5.12 shows some examples of the results of the extraction of the first tile, while Figure 5.13 shows the first 10 tiles for the same models.

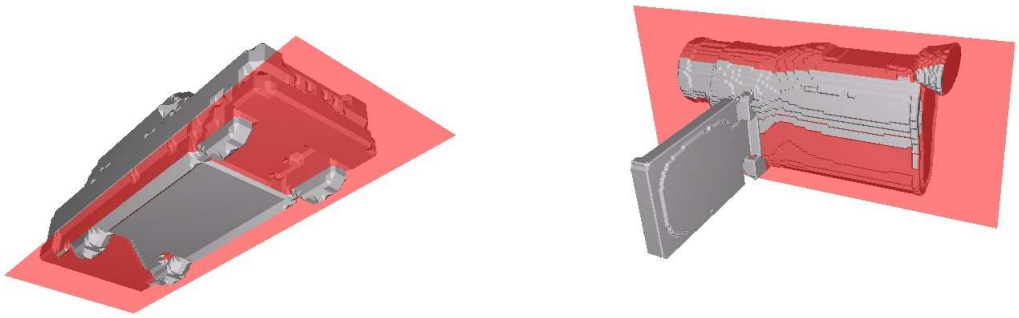


Figure 5.12: The maximal tile obtained using the voting algorithm. The marching cubes result of the voxelization used is shown as reference. On the left the taxi model with a resolution of 64^3, and on the right the handycam model with 128^3.

These tiles may be used to render objects as a set of textured quadrilaterals as shown in Figures 5.14, 5.15 and 5.16.

Running times as well as other statistics related to the models are compiled in two tables 5.9 and 5.10.

As it can be easily deduced from the the voting algorithm, the running times does not depend on the number of mesh elements that form the model. This may be observed comparing the total running times for the Buddha and the Handycam models. Despite the Buddha has almost six times the number of polygons that are

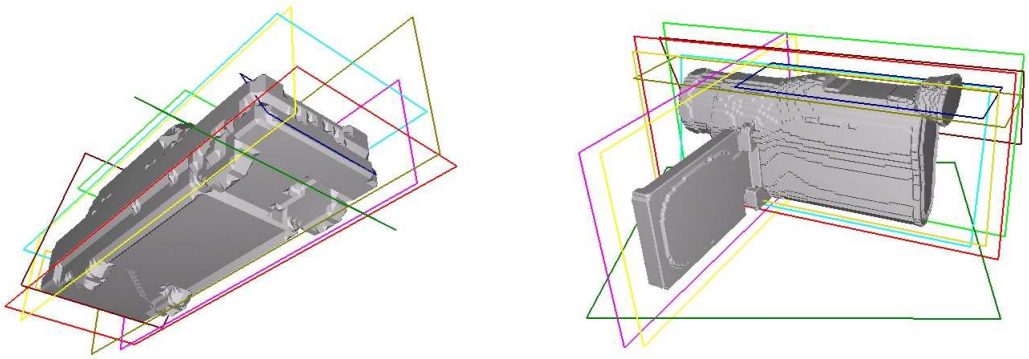


Figure 5.13: The first 10 tiles of the same two models as in Figure 5.12.

Figure 5.14: The original taxi model rendered as a triangular mesh.

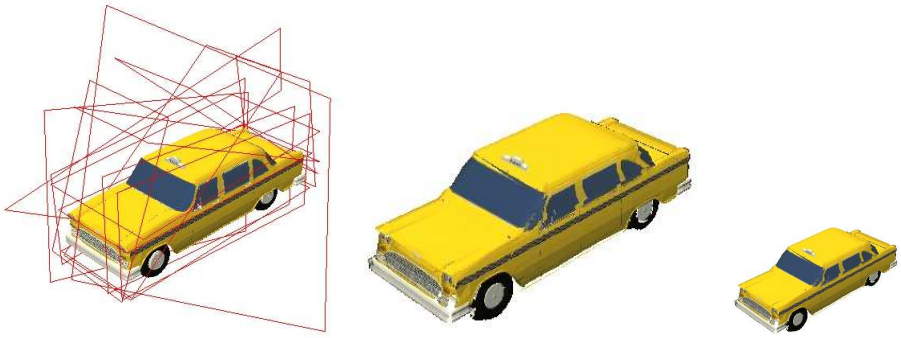


Figure 5.15: The 14 quadrilaterals that will be used for orthographic projection, and the resulting impostor model as seen at different sizes (from the $64 \times 64 \times 64$ voxelization).

Figure 5.16: The Handycam model and its 78 impostors simplification at different sizes.

Model	#Triangles	Resolution	Error %	**#Tiles**
Taxi	37380	$64 \times 64 \times 64$	1.6	14
		$128 \times 128 \times 128$	0.8	44
Handycam	52044	$128 \times 128 \times 128$	0.8	78
Buddha	293232	$64 \times 64 \times 64$	1.6	25
		$128 \times 128 \times 128$	0.8	67

Table 5.9: Test models and resulting tiles.

		Average time per tile (ms)				
Model	Resolution	Phase 1	Phase 2	Phase 3	Total	**Total (s)**
Taxi	$64 \times 64 \times 64$	9.85	12.42	17.85	40.12	0.5
	$128 \times 128 \times 128$	17.18	27.02	227.34	271.54	11.9
Handycam	$128 \times 128 \times 128$	11.35	65.03	441.38	517.76	40.3
Buddha	$64 \times 64 \times 64$	1.72	1.88	19.20	22.80	0.5
	$128 \times 128 \times 128$	2.81	7.40	362.92	373.13	24.9

Table 5.10: Running times for the test models.

present on the Handycam, the voting scheme runs faster when computing tiles for the Buddha model.

Although it's true that the complexity is independent of the number of triangles, there are other factors that influence it. The first one is grid resolution, which can be found comparing the times for Taxi and Buddha in their different sizes: $64 \times 64 \times 64$ and $128 \times 128 \times 128$. Another factor that modifies running times is the number of planar sticks for a given model (those found in the dictionary). Only the sticks that have an entry in the stick neighborhood dictionary cast a vote, so if a model has few planar regions casts less votes and has to go faster. This can be noted comparing the first two phases of Taxi and Buddha. The third phase, instead, does not seem to be affected by this, and the reason for it can be found in the last factor that influences time complexity. The first two phases are designed to prune many planes from the search for the maximal tile. As the third phase makes the complete search helped by this pruning, its running time depends on how good this initial candidate is. In the Taxi model, there are large zones that can be approximated by a single plane. In the other two, many curved regions can be found, causing many cells to have a very similar number of votes. This makes more difficult to extract the maximal tile, making the running times for the third phase of Handycam and Buddha larger than the Taxi model.

The whole algorithm can be further improved if the third phase is discarded. The results are sub-optimal but many of the initially discarded maximal tiles will appear later. This simplification only works when we want to extract the N largest tiles, not only the maximal one.

Nevertheless, the scheme is able to obtain tiles from the most expensive model, the Handycam with 50K triangles, at a rate of 2 per second, while the approach by Decoret et al [DDSD03] only achieves 0.32 tiles/s with similar models.

Chapter 6

Pressing

6.1 Retrieving shape from binary grids

As we have seen in Chapter 5, shape may be retrieved from binary grids, even though the information we have is very limited. We have already recovered planar regions, without worrying about the rest of the isosurface. In order to complete the shape extraction process, an algorithm called *Pressing* will be introduced here.

Pressing begins by identifying groups of clusters that are separable by planes. Then, it slides the remaining vertices along their sticks to smooth the remaining regions. The last step recovers previously detected sharp features, adding new vertices. Figure 6.1 shows a 2D example.

This whole process guarantees that the final surface is contained into the discrete band of the binary volume, thus the reconstruction error is bounded. As a side effect, we have a model segmented into planar and smooth regions sometimes separated by sharp features, which may prove useful in shape identification and manufacturing.

6.2 Detection of flats and junctions

To recover the original shape as presented in the previous section, the planar regions are extracted first. This step uses the same algorithm introduced in Chapter 5. Once the maximal tiles have been obtained, each vertex that belongs to one of them is *frozen*, meaning that their position remains fixed at the intersection between their stick and its corresponding tile.

Before applying any smoothing, we need to determine where sharp features will be, and where there will be smooth transitions between regions. In order to do this,

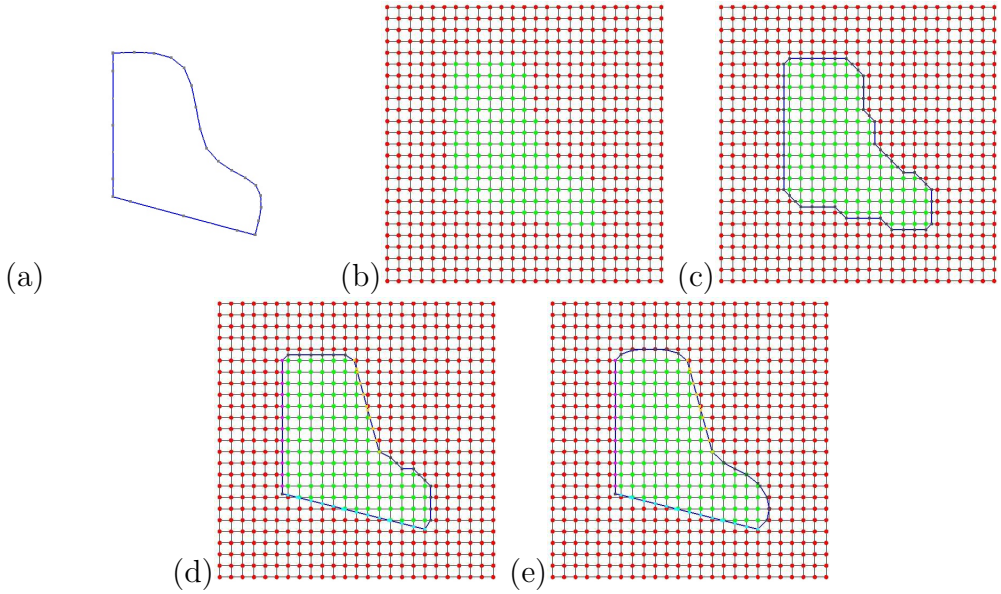


Figure 6.1: (a) The original curve. (b) Its binary discretization. (c) Resulting isosurface if each vertex is placed at the middle point of its stick. (d) Planar regions are identified and the vertices on their sticks are positioned correspondingly. (e) Finally, the remaining regions are smoothed.

edges that connect planar regions with smooth ones are examined. Those which are found to correspond to a sharp feature are called *junction points* (they can be seen in Figure 6.2).

Consider an edge that connects a vertex from a planar region V_P with one on a smooth region V_S, and their associated normals N_P and N_S respectively. V_S is classified as a junction point if the angle formed by the normals N_P and N_S is greater than a certain threshold angle. N_P is determined by the plane assigned to V_P, but N_S is computed by adjusting a plane using a least squares fit to the 2-ring neighborhood

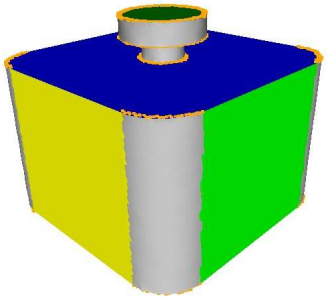


Figure 6.2: This model of a pump shows its junction points (those that are next to a sharp feature) using orange points.

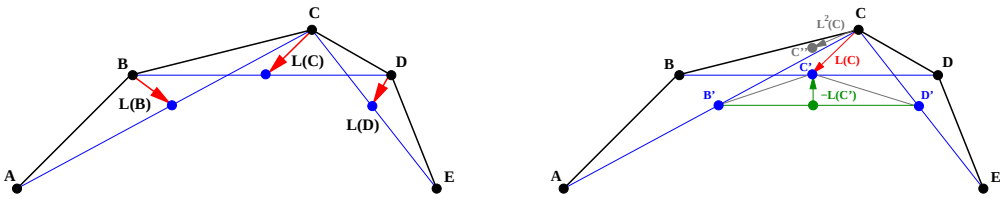


Figure 6.3: $L^2(C)$ is equivalent to two Laplacian steps. In the left picture, C are moved forward to C' by its laplacian $L(C)$. Vertices B and D are also moved forward, because they will be needed to compute the backwards movement. In the right picture, C''s Laplacian $L(C')$ moves it to the average of its neighbors B' and D'. Nevertheless, C'' is obtained from $C' - L(C')$.

of V_S. The threshold angle used is 30 degrees, because it achieves the best compromise detecting sharp features without overshooting.

6.3 Smoothing

Now, the remaining vertices may be smoothed by means of the following algorithm, which needs to keep them on their corresponding sticks. To this end, the smoothing process will be an iterative application of two steps: smoothing and snapping. While the smoothing step moves the vertices regardless of their constraints, the snapping step reprojects them to their corresponding stick.

For now, let us consider that we want to smooth a 2D curve. Further on, we will see that 3D smoothing may be achieved through the application of two 2D smoothing displacements. Consider 5 vertices A, B, C, D, E of a polygonal curve, the smoothing displacement associated to vertex C applying the bilaplacian is:

$$L^2(C) = \frac{-A + 4B - 6C + 4D - E}{4} \tag{6.1}$$

The smoothing is applied by precomputing the vertices' displacement ($L^2(C)$), and then moving each vertex to $C'' = C + sL^2(C)$. We use a value of $s = 0.85$, to accelerate its convergence while maintaining stability. In fact, this is equivalent to the application of two Laplacian steps, one forward, the other one backwards, as seen in Figure 6.3.

Even though this smoothing algorithm works well in regular models, the sticks' restrictions as represented by the snapping step, affect it adversely. This effect arises because the snapping step preserves irregular sampling. We found that, rather than approaching zero, some displacement vectors were orthogonal to the sticks, so that projecting a displaced vertex simply returned it to its previous position (see Fi-

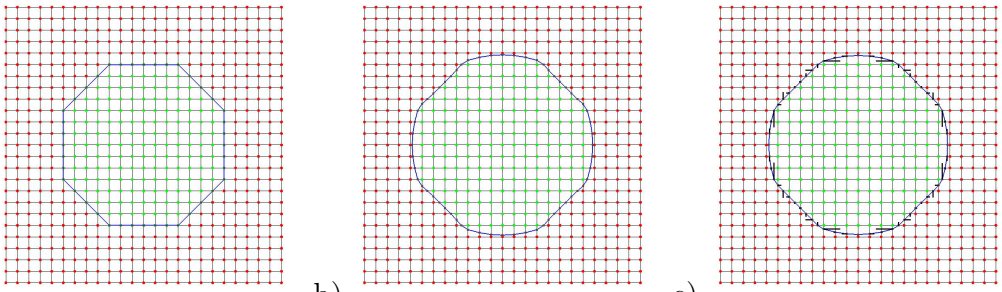


Figure 6.4: (a) A circle rasterized to obtain a 2D binary grid. (b) Smoothing and applying a projection to maintain every vertex on its stick does not yield the expected result. (c) Some of the displacements computed using the bilaplacian filter turn orthogonal to their sticks, thus being canceled by the projection.

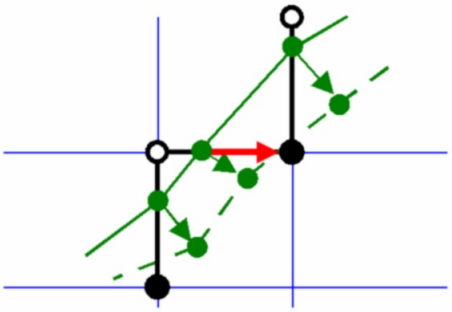


Figure 6.5: The red arrow represents the movement of one vertex using the front tracking technique. The original isocurve is drawn in solid green, the green arrows correspond to the displacements computed by the bilaplacian filter, while dashed green lines are used to show where the isocurve would move if not for the sticks' constraints.

gure 6.4).

To address this problem we explored replacing the simple projection step by a front-tracking algorithm. Instead of trying to project the new vertices to their sticks, we track the intersections of the evolving curve with the rays of the discretization as the vertices travel from their current to their displaced positions. Figure 6.5 illustrates the motion of one vertex under this scheme.

The original curve and the movements of its vertices define a set of quads, whose edges are shown in green. The boundary of each quad is delimited by one edge joining adjacent vertices the original curve, another edge joining the vertices after they are displaced, and the two edges connecting each original vertex to its displaced position. In some sense, if the curve is considered to be at some time $t = 0$ when its vertices are at their original positions and at some time $t = 1$ when they are at their desired

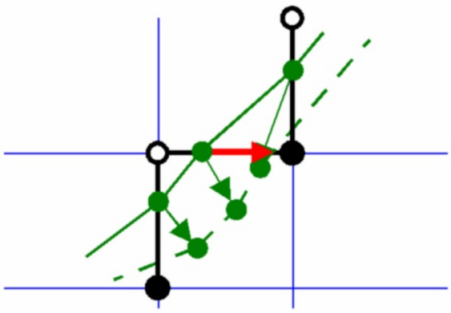


Figure 6.6: The ray that contains the vertex of one stick may intersect multiple quadrilaterals which need to be inspected to track the front's movement.

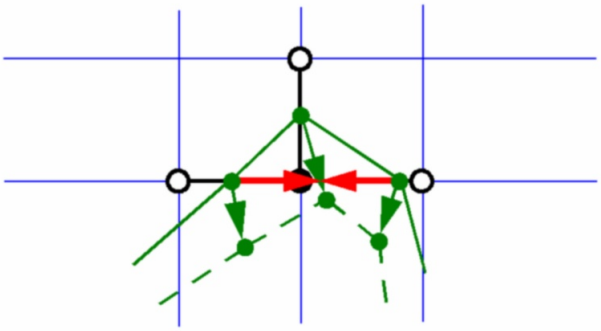


Figure 6.7: The bilaplacian filter may move the isocurve so that it does no longer intersect a given ray. This cannot be allowed because each vertex has to remain on its corresponding stick.

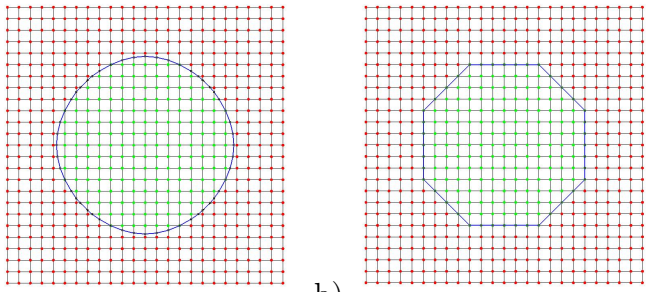
destinations, the quads delimit the evolution of the ray intersections.

Thus, one solution to determine the motion of a given intersection point along its stick would be to find out which of the two neighboring quads intersects the stick ray. If the ray intersects the corresponding edge of the final curve, that intersection point will be the final position. One such case is shown in Figure 6.5.

However, if the ray leaves the quad through one of the edges traced by the moving vertices, then the motion of the intersection will be defined by a sequence of neighboring quads. Figure 6.6 shows an example.

We can then compute the vertex position along its ray by traversing the quads until an intersection with the final curve is obtained. However, it is also possible for the curve to leave a ray altogether (as shown in Figure 6.7).

In that instance, traversing the quads will lead us to an intersection with the original curve, when we would like to find the point where the curve leaves the ray. To handle this case we return to the idea of defining a motion between the original curve, at time $t = 0$, and the final curve, at time $t = 1$. Using that fact, a time can



a) b)

Figure 6.8: The front tracking method solves the problem of displacements becoming orthogonal to the sticks, obtaining the desired result before converging (picture a). While the isocurve approaches this state, some vertices on adjacent sticks converge onto the same position, adversely affecting the computation of the bilaplacian. As a result, the smoothing process does not stop and begins to form kinks around those corners (picture b).

be associated to all intersections between the quads and the rays. Consequently, if while traversing the quads searching for an intersection with the final curve, we find that the associated times go backwards, we know the curve is leaving the ray. The target vertex position is then the intersection point associated with the latest time. We then map target positions on the ray that are outside the vertex's stick to the closest endpoint of its stick.

This approach to snapping eliminates the problem of the curve appearing to get stuck in a partially smoothed state. However, as the curve continues to evolve it shrinks and develops corners where it approaches the constraints. These corners are always associated with short edges created when vertices approach a cell corner (see Figure 6.8), which led us to reconsider our formulation of the bilaplacian displacement.

As this effect mainly derives from the fact that some of the edges are seeing their length reduced to zero progressively, we might have thought of using other smoothing algorithms which can deal with anisotropic distributions of vertices. Unfortunately, although the use of such algorithms delays the creation of kinks, it fails to prevent their formation.

We then considered using two virtual neighbors CL and CR on the polygonal curve, at a certain arc-length distance of C on each side (like in Figure 6.9), instead of using vertices B and D to compute each Laplacian step. We have selected this distance to be 75% of the cell's size. This arc-length resampling prevented the formation of corners and yielded satisfactory results when used with either projection or front tracking, so we decided to combine smoothing and projection at each step due to its cheaper computational cost.

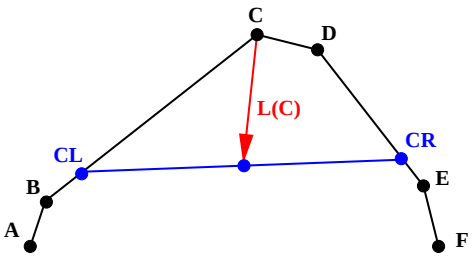


Figure 6.9: CR and CL are used to compute the Laplacian of C ($L(C)$), instead of B and D.

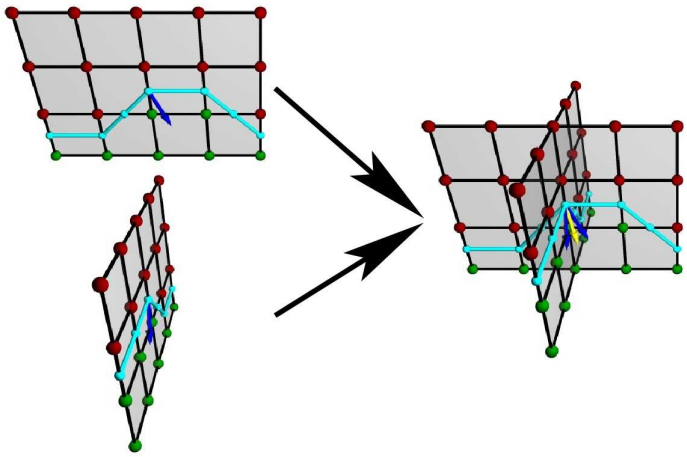


Figure 6.10: The isocurves (in cyan) in two slices, are used to compute two smoothing displacements (in blue). Combined, they result in the 3D displacement (in yellow).

We further investigated another approach. The idea is to use as a vertex's neighbors the centers of mass of the portions of the curve extending out to a fixed arc length distance on either side of it. Because one of these portions of the curve is a set of line segments, its center of mass is given by the average of the line segment midpoints, each weighted by its corresponding line segment length. This however yielded no benefit over the simpler approach of using projection with arc-length resampling, while resulting in a slower algorithm because the displacements were smaller.

As shown in Figure 6.10, the 3D displacement vector used in the smoothing step results from the combination of the displacement vectors obtained from the stick's two slices.

One last point in the smoothing algorithm requires our attention. Some vertices fall too close to a boundary (marked with *junction points*) to compute their arc-length neighbors. Regular bilaplacian smoothing is not applied to these vertices. Instead, they are aligned with the smooth side, as shown in Figure 6.11.

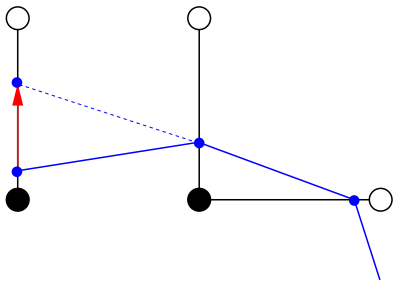


Figure 6.11: Vertices along the boundaries of "smooth" regions align themselves with the neighbors in their "smooth" side.

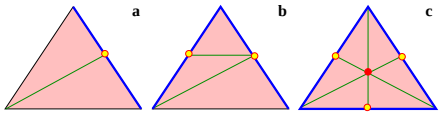


Figure 6.12: Subdivision of a chamfer triangle with (a) one, (b) two, (c) or three chamfer edges. Chamfer edges have their vertices on two different tiles.

6.4 Recovering sharp features

After this process the isosurface has been divided into smooth and planar regions, with all the vertices positioned to recover those shapes. However, there is an inherent problem with the use of isosurfaces obtained through Marching Cubes extraction. They are not feature sensitive, an issue that cannot be solved without adding new vertices.

There are several approaches to solve it, like those presented in [KBSS01] and [AFRS04], of which we choose the second one, because [KBSS01] needs three axis-oriented distances per grid point that we do not have. [AFRS04] detects those triangles that need to be subdivided, which are called *chamfer triangles*, and moves the additional points to capture the extracted features.

Adapting the Edge Sharpener algorithm to our output, we can remove the step used to identify chamfer triangles. Each vertex belongs to one tile or a smooth region. We will consider all vertices in smooth regions to be in one additional virtual tile. Triangles that connect vertices of different tiles need to be subdivided, and thus, they are considered chamfer triangles.

Now, new vertices are added subdividing all chamfer triangles. One vertex is added per edge with its two vertices on two different tiles, and one additional vertex is generated for each triangle on three different tiles. The resulting possible subdivisions are displayed in Figure 6.12.

These new vertices need to be positioned. If all the tiles on a chamfer triangle are

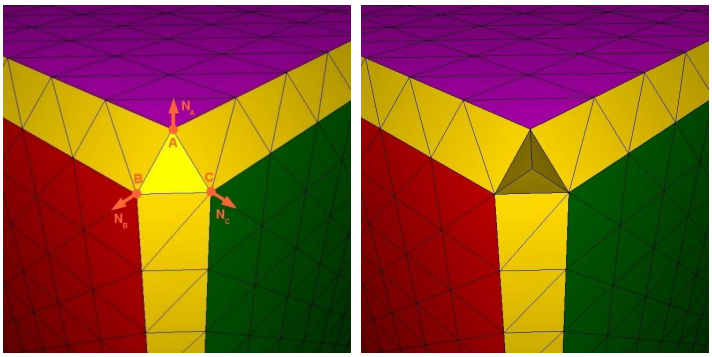


Figure 6.13: Vertex generated from a chamfer triangle on three different tiles.

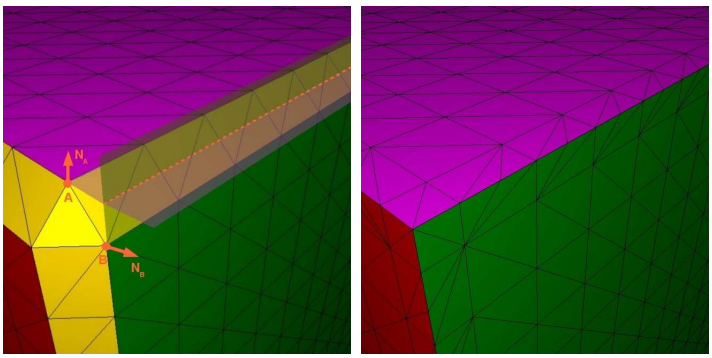


Figure 6.14: Vertices added from subdividing chamfer edges.

planar, we have two possibilities:

1. Vertices produced by chamfer triangles with three different tiles are positioned at the intersection of their three planes.

2. Vertices generated by chamfer edges are positioned at the nearest point in the line resulting from the intersection of the corresponding planes.

This is illustrated in Figures 6.13 and 6.14.

When a vertex is on a smooth tile, its plane is computed using its position and a normal. This normal is obtained, like in the computation of the junction points, as the normal of the minimum square fit of the "smooth" neighbors.

Finally, recognized features are arranged into polygonal curves which are smoothed using the bilaplacian algorithm previously described.

6.5 Results

Figure 6.15 displays the good behavior of the presented smoothing algorithm, but as can be seen in Figure 6.17, smoothing only is not enough for mechanical models.

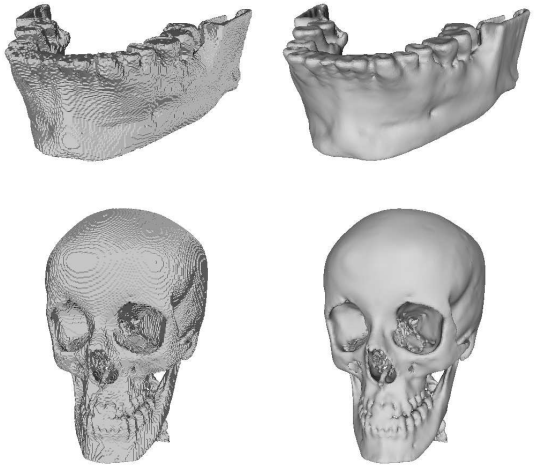


Figure 6.15: Results of our smoothing algorithm applied to biomedical models like a jaw and a skull.

Stats (times in seconds)	MECHANICAL PART	FANDISK
TIME (MaxTiles)	15.99	58.89
TIME (Detect Sharp Features)	0.175	0.272
TIME (Smoothing)	5.314	6.036
TIME (Edge Sharpening)	0.248	0.244
# Tiles	8	11
# Iterations	68	65

Table 6.1: Running time, number of reconstructed flat regions and number of iterations for each of the presented models.

Figures 6.16 and 6.17 show the results of "Pressing" isosurfaces, recovering smooth regions, planar ones and extracting sharp features if present.

Table 6.1 presents the running times of all steps necessary to "press" the two models shown in Figures 6.16 and 6.17. The computer used to produce this timings was a Pentium 4 at 3.4 GHz with 1 GB of RAM. The most expensive step is the tile extraction which was presented in the previous chapter, followed by the smoothing step. All the other steps are negligible when compared to the maximal tiles identification.

Table 6.2 contains the evolution of reconstruction errors through all the steps of the algorithm. This error is measured as the average of the unsigned distances between a vertex of the isosurface and the intersection of its stick with the original model (where a cell has size $1.0 \times 1.0 \times 1.0$). Notice that error consistently decreases due to both the tile identification process and smoothing algorithm.

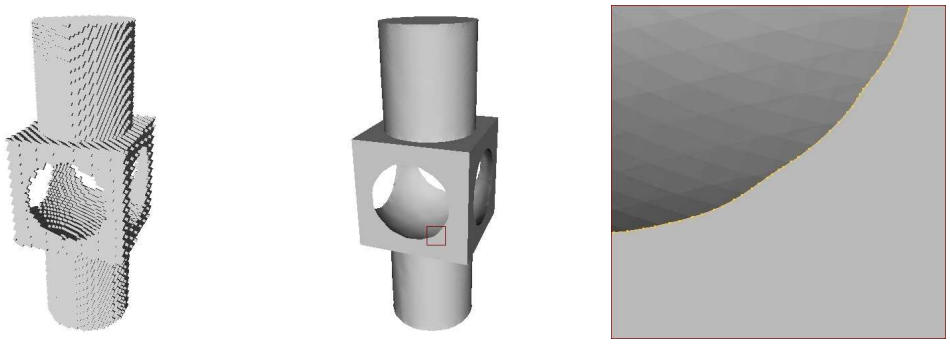


Figure 6.16: A volumetric discretization of a mechanical part at a resolution of $128 \times 128 \times 128$. The left image shows a cuberille visualization, while the middle image shows the result of "pressing" it. A close-up of the extracted features may be seen on the right picture.

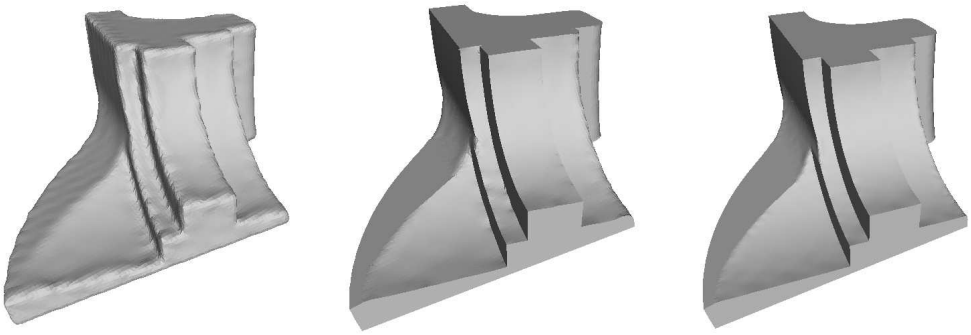


Figure 6.17: A voxelization of the fandisk model. Left, applying smoothing only does not preserve sharp features, while pressing recovers them (middle image) on a $128 \times 128 \times 128$ discretization. Some artifacts may be observed, but their size depends on cell's size. In the right image, a resolution of $256 \times 256 \times 256$ gives much better results.

Error	MECHANICAL PART	FANDISK
Midpoint	0,2837	0,2882
Midpoint + Tiles	0,2267	0,1835
MidPoint + Tiles + Smoothing (100)	0,0856	0,0892
MidPoint + Tiles + Smoothing (1000)	0,0850	0,0857

Table 6.2: Square root of the average square error after the different steps of the Pressing algorithm for each of the presented models.

Chapter 7

Visibility based feature extraction

7.1 Visibility inside the discrete band

In Chapter 6, we have seen that it is possible to extract shape from binary grids. This extraction is based on the subdivision of the isosurface into regions through the identification of tiles. Such tiles correspond to planar regions, which will be preserved, while the remaining regions are smoothed. After this process is completed, features will be extracted along the boundaries of the initial tiles. Thus, tiles define where features may be identified.

In order to identify all features, we might choose to use curvature. The problem is that a binary model has very little information left to work with. If we used the mid-point isosurface to estimate curvature, the resulting value would be too noisy. We could smooth the isosurface before curvature calculation, but smoothing would destroy the features we are trying to find. A different solution is needed.

The previously presented approaches define an isosurface first, then they use it to identify features. But the isosurface may be anywhere within the discrete band. The solution needs to take this fact into account. In order to do just that, let us imagine a creature crawling on the discrete band. This creature will only be able to see a limited part of the discrete band, which will be determined by the local geometry. On a planar region it will see the whole area but no further. If it walks towards an edge, the area visible to the creature will increase as a new region enters into view. This effect is even greater when a vertex of the isosurface is reached, and thus, it shows that there is a relation between variations in visibility on a local cell's neighborhood and the chance of a feature being present there.

As we work on a discretized setting, we propose to use cell to cell visibility to determine where features are located. Considering that visibility segments may only

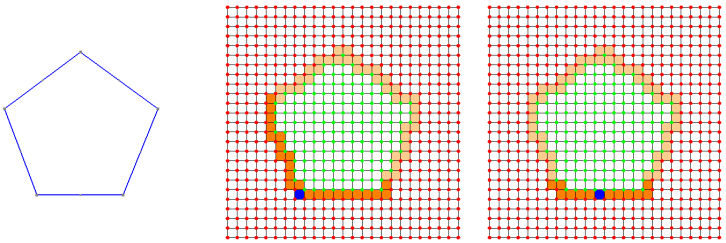


Figure 7.1: The discretization of a pentagon defines a set of cells which can used to compute cell to cell visibility. Visibility from a cell containing a vertex results in a greater set of visible cells.

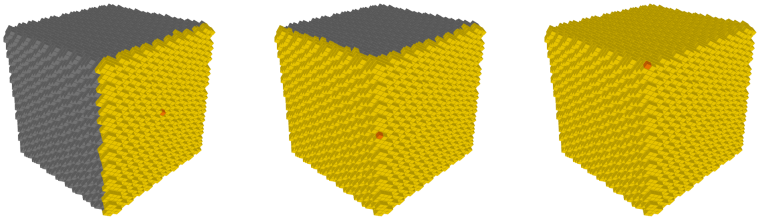


Figure 7.2: Visibility from cells on a face, edge, and vertex of the discretization of a cube. The source cell is in orange, while its visible set is in yellow.

go through mixed cells, a given pair of cells are mutually visible if there are two points, one on each cell, that are connected through a visibility segment. Therefore, one cell's visibility is the set of cells which are visible from it. In 2D, this visibility set is greater for cells that contain vertices, as they see two full edges, while it is smaller for cells that contain edges. Figure 7.1 shows why this visibility approach makes sense in the 2D case.

As shown in Figure 7.2, these arguments also apply to 3D discretizations. Consider a binary grid obtained from the voxelization of a polyhedron. A cell in the middle of a face will only see the cells that intersect that face, while a cell on an edge will see all the cells that intersect its two adjacent faces. Finally, a cell containing a vertex will have a visible set comprised of all the cells that intersect its adjacent faces (which at least have to be three).

As we can see, features seem identifiable by selecting cells with a high number of visible cells. But just choosing the cells with the largest visibility count is not enough. A cell on a large face may have a higher count than a cell on a vertex where small faces meet, which would result in problems identifying them correctly. Our first approach to solve this effect, an adaptive thresholding technique, compares the visibility count of a cell with that of cells in its local neighborhood.

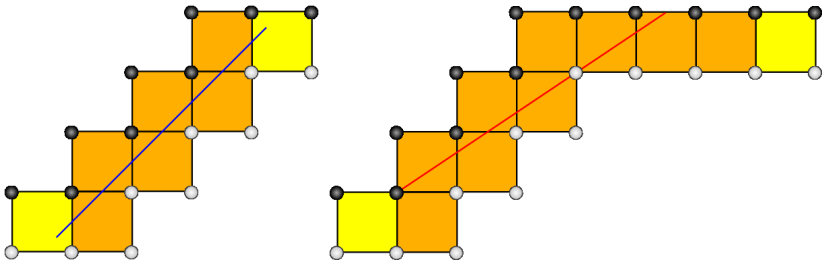


Figure 7.3: 2D examples of cell to cell visibility. In the left picture there are two points, one per cell, connected by a segment which only intersects mixed cells. In the right picture, there is no visibility.

Just like these extracted features may be used to enhance the results of the Pressing algorithm, they may be interpreted as signatures. Databases of 3D models can be searched using such signatures.

7.2 Unconstrained visibility map

Let us define an integer function $uVis(C)$, which we call the *unconstrained visibility map*, over the set of mixed cells. The value $uVis(C)$ is the number of mixed cells visible from the cell C. In order to compute this visibility map, mixed cells are considered to be empty while homogeneous cells (those with eight black nodes or eight white) are considered to be filled, Therefore, visibility rays have to travel inside the discrete band, intersecting mixed cells and mixed faces. A 2D example is shown in Figure 7.3.

Extending this visibility approach to 3D introduces an efficiency problem. The cost of computing the unconstrained visibility map grows with the square of the number of cells. A drastic reduction of the number of computations is needed.

The first thing that may be applied to achieve this reduction is the knowledge that visibility is bidirectional. This means that the visibility table is a triangle not a square, and that we have half the visibility queries.

Another property that proves useful is that if two mixed cells are visible through a visibility ray, all the mixed cells stabbed by the ray are mutually visible. As a consequence, the visible set of a given source cell may be computed by propagating a wave from that source cell. We already know that the source cell's neighbors are visible to it. After adding them to the visible set, we may test the source cell's second ring neighbors and so on. By means of this method, we don't test all cells against the source cell. Instead, we end testing only those that are visible and some non-visible

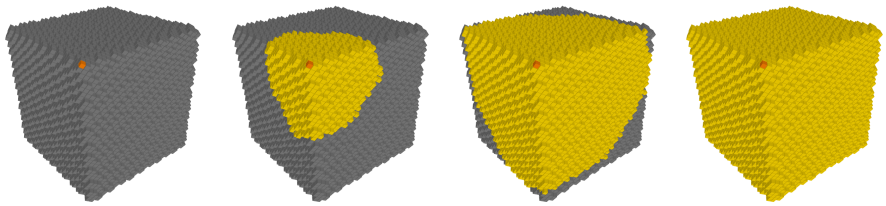


Figure 7.4: Visibility is computed from a source cell, propagating it like a wave.

cells that are on the outer ring of the visible set. This process may be seen in action in the pictures of Figure 7.4.

Another effect that can be harnessed for efficiency is similarity. The visible sets of two neighboring cells are very different only if the curvature is high enough. Even in those cases the two sets intersect each other. Thus, to compute the full unconstrained visibility map, we can compute the visible set of one cell as previously described, then use it to compute all the rest. Once the first visible set has been computed, the rest are obtained by taking the visible set of one neighbor (one already computed) and stretching the outer ring of visible cells until the new visible set results.

Now we need to determine how to know if two cells are visible or not. If exact visibility between a pair of mixed cells is computed, even small models will take too much time to process. The strategy that is used here is to apply work on erosion based visibility by Decoret et al [DDS03], and adapt it to our setting.

Consider an object O, in our case all of space except the discrete band, and a convex set of vectors C that represent the visibility cell:

$$C = \{(x, y, z), |x| \leq \frac{1}{2} \wedge |y| \leq \frac{1}{2} \wedge |z| \leq \frac{1}{2}\} \tag{7.1}$$

If a segment PQ intersects $O \ominus C$, then any segment $P'Q'$ intersects O with $P' \in \{P\} \oplus C$ and $Q' \in \{Q\} \oplus C$. $O \oplus C$ stands for the Minkowski sum of both sets, while $O \ominus C$ defines the erosion of O by C. This property is illustrated in Figure 7.5, and it allows to reduce the cost of computing visibility between two cells to finding if the segment between their two centers intersects $O \ominus C$. It comes at the price of obtaining the erosion of an object, though.

Normally, computing exact erosion of general objects is a very complex problem, but our object is all of the voxelization space except the discrete band (which results in an object which may be thought as a union of cubes). Thus, it is a lot simpler to shrink. If we subdivide each cell into 8 equal subcells, $O \ominus C$ is all of space except those subcells obtained from mixed cells and those that are adjacent (through a vertex, an

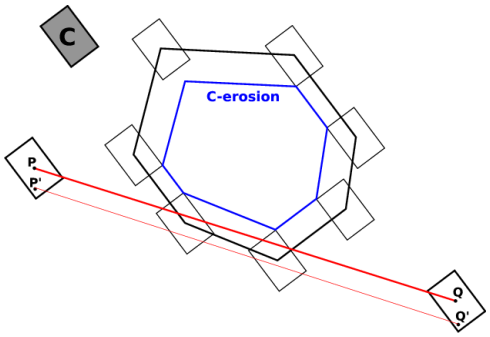


Figure 7.5: If there is a segment from a point P' in one cell to a point Q' on another cell which does not intersect an object, then the segment between P and Q, centers of their corresponding cells, does not intersect the C-erosion of the object. As a consequence, there is visibility between the two cells.

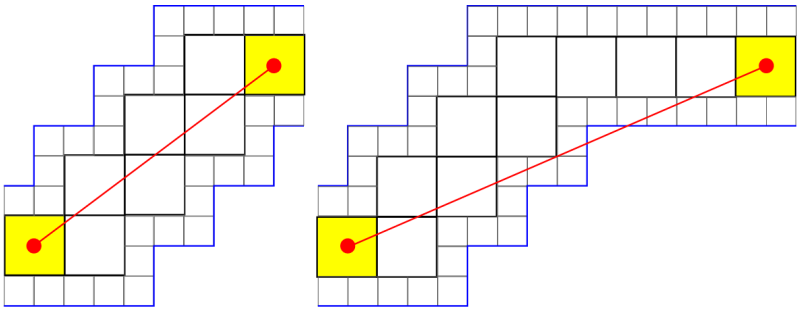


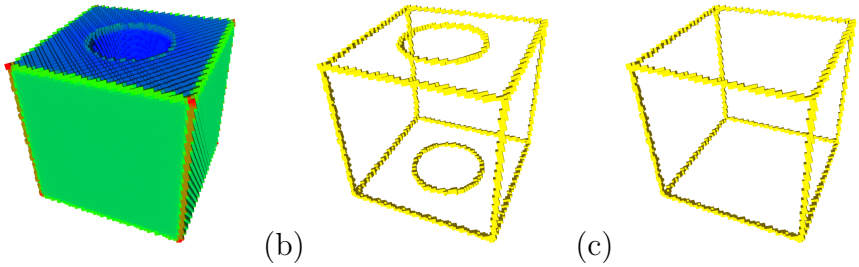
Figure 7.6: Visibility computation for the examples shown in Figure 7.3. Testing a segment that connects two cell's centers for intersection with the eroded object (shown in blue) is enough using the erosion based technique.

edge or a face) to a mixed cell. A 2D example of this is illustrated in Figure 7.6.

Applying all these techniques we can efficiently obtain the cell to cell visibility for any pair of mixed cells. To extract features we are going to use the visibility count (how many mixed cells are visible from a given one). Cells with higher counts than their neighbors should be on a feature. An unconstrained visibility count map of a cube is shown in Figure 7.8a.

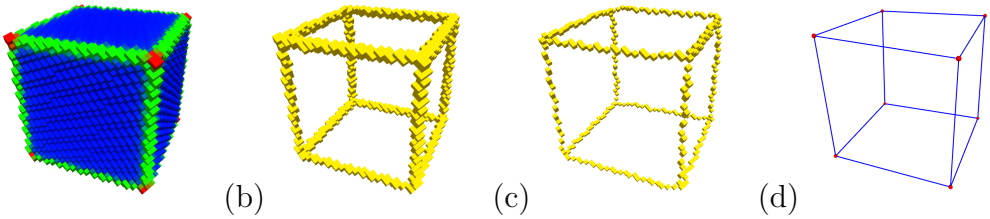
Cells on features are recognized by applying an adaptive thresholding technique taken from the computer vision field. For each cell with a visibility count of V_i, the mean visibility count of its 3-neighborhood is computed M_i. If $V_i > M_i + C \cdot r^2$, where r is the resolution of the volumetric model, then the cell is marked as a feature. This method results in a set of cells like those shown in Figure 7.8b. C controls which features are recognized, as may be seen in Figure 7.7.

In order to improve these results morphology operators are used [SOS00]. First, thinning is applied to the set of feature cells, to reduce its number while preserving its



(a) (b) (c)

Figure 7.7: (a) The unconstrained visibility count map of a $128 \times 128 \times 128$ discretization of the difference of a cube and a cylinder. The result of applying different C values is shown in (b) with $C = 0.03$ and (c) with $C = 0.1$.



(a) (b) (c) (d)

Figure 7.8: (a) Unconstrained visibility map of a $64 \times 64 \times 64$ discretization of a cube, where blue cells have the lowest visibility count and the red ones have the highest. (b) Yellow cells are recognized as features. (c) The results are improved applying a thinning algorithm. (d) Finally, a graph is obtained.

connectivity. This process generates a skeleton of the features, but thinning also tends to leave undesirable short spurs that are caused by irregularities in the boundary of the initial volume. These spurs may be removed by a process called pruning which, in fact, is another form of thinning. The skeleton of a cube obtained through this technique is shown in Figure 7.8c.

Now, with the resulting set of cells, an skeleton may be built. Considering a 26-adjacency of these cells, the valence of each of them is the number of adjacent cells. Cells with a valence of 3 or more will be considered to be vertices in the final skeleton graph. The remaining cells connecting them through chains, will be added to the graph as edges. If there is visibility along all of the new edge, it will be introduced into the graph as a straight edge. If that is not the case, it will be added as a curved edge.

This distinction is useful in order to render the graph; straight edges may be drawn as segments, while curved edges may be rendered smoothing its corresponding chain of cells. The skeleton graph for the example cube is shown in Figure 7.8d.

Some unconstrained visibility count maps are shown here, along with their resulting skeletons and graphs. Figures 7.9 and 7.10 present the visibility count maps of the

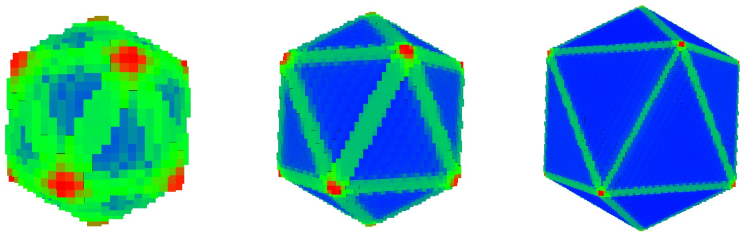


Figure 7.9: Unconstrained visibility count maps of three discretizations of an icosahedron (at $32 \times 32 \times 32$, $64 \times 64 \times 64$, and $128 \times 128 \times 128$). As expected, features are clearer on higher resolution maps, which makes it easier to identify them.

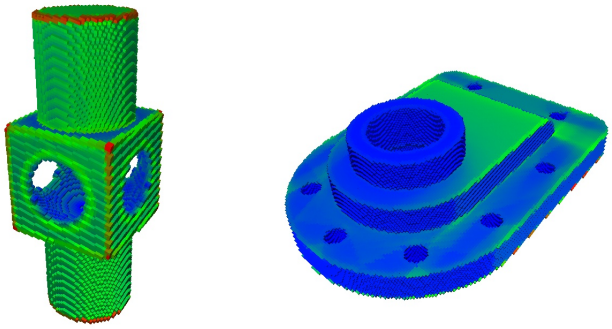


Figure 7.10: Unconstrained visibility count maps of two mechanical parts. Shadows appear next to through-holes making it more difficult to distinguish features.

discretizations of an icosahedron and two mechanical pieces. As may be seen, features stand out in these pictures, but a sufficiently high resolution model is needed. Figure 7.9 stresses this fact. As resolution grows, features are more easily distinguished.

Figure 7.10 also shows false edges on its right picture. This artifact is caused by through holes, which shadow visibility from a cell on a given face. This effect is illustrated in Figure 7.11. Notice that in this case, no false edges will be inserted, due to the symmetry of the pierced faces.

Table 7.1 presents the running times of the visibility computation algorithm, running on a Pentium 4 at 3.4 GHZ with 1 GB of RAM. Obtaining the visibility from any mixed cell to any other mixed cell is a very expensive operation. Applying the methods exhibited above, running times are reduced, so that large enough models might be processed.

Visibility count maps are translated into skeletons through identification of features and morphology operators. Finally, feature graphs are extracted from these skeletons. Figures 7.12 and 7.13 show sample results of this process. Red spheres indicate vertices, while blue segments and green curves represent straight and curved

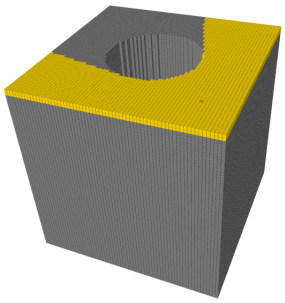


Figure 7.11: Holes induce visibility shadows that may cause wrong edge identification, like in the second mechanical part of Figure 7.10.

MODEL	RESOLUTION	TIME
ICOSA-64	64^3	23.96 seconds
ICOSA-128	128^3	302.58 seconds
MECHANICAL PART 1	128^3	51.5 seconds
MECHANICAL PART 2	256^3	1556.37 seconds

Table 7.1: Running times of visibility map computation for the models seen in Figures 7.9 (ICOSA-64 and ICOSA-128) and 7.10 (left picture is MECHANICAL PART 1 and right picture is MECHANICAL PART 2).

features, respectively.

As expected, features appear clearly in the unconstrained visibility maps, but they have very different visibility counts due to the diverse sizes of the model's faces. This is easily compensated by applying adaptive thresholding techniques, borrowed from the digital image processing field. It may also be observed that some noise appears in the maps. This noise is due to the the visibility being computed in a discrete setting, but it is dealt with by the used thresholding method. The main drawback of the technique is the fact that shadows caused by holes induce the appearance of false edges (which may be a severe problem as shown in Figure 7.10). Additionally, vertices

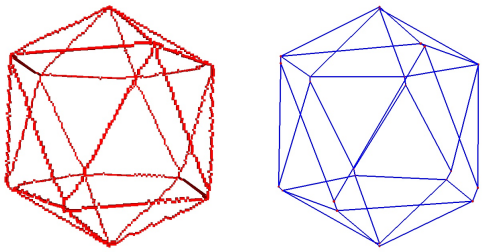


Figure 7.12: Skeleton and extracted graph of the $128 \times 128 \times 128$ icosahedron.

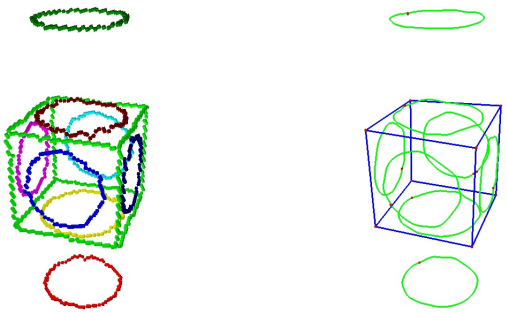


Figure 7.13: Skeleton and graph of one of the mechanical parts from Figure 7.10.

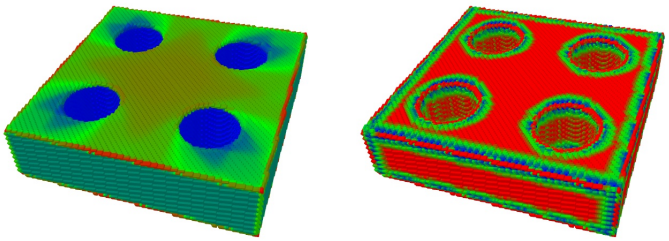


Figure 7.14: On the left, holes lead to "shadow" artifacts in the unconstrained visibility map. The picture on the right, which presents a constrained map, does not present this drawback.

are identified as the intersection of several edges which may be adversely affected by noise.

7.3 Constrained visibility map

In order to avoid the shadow artifacts caused by holes which occlude visibility, we choose to use a different value to subdivide the mixed cells into components. We define the *constrained visibility map* $Vis(C)$ which measures the number of complete rings of mixed cells centered at C that are visible from this cell C. The first ring of a mixed cell C is defined as usual as the set of mixed cells contiguous to C, and the k-ring is the set of cells that are contiguous to the (k-1)-ring and do not belong to any previous ring. Given one mixed cell, we first obtain the mixed cells in its 6-neighborhood, then the set of mixed cells on the 6-neighborhood of the previous set and so on. This process is stopped when a single cell in one of these rings is not visible from the source, and the number of rings traversed during the process is what

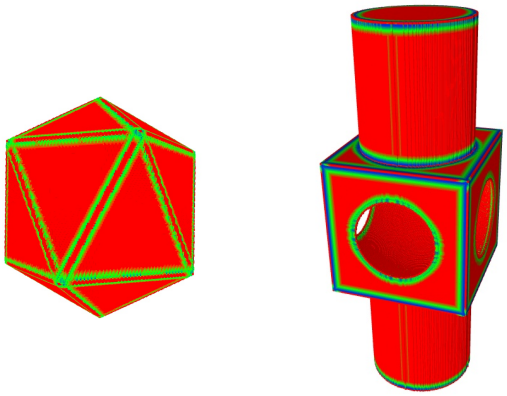


Figure 7.15: Unconstrained visibility maps of an icosahedron and a mechanical part.

we call the *visibility radius* of the cell.

This kind of map is open to further improvements as the model's components are separated by relatively small visibility radius values. This is used by limiting the number of rings visited during the visibility radius computation to a certain maximum. This maximum radius is determined by the minimum radius we will use to identify which cells are candidates to be in a component in the next step. The difference between the unconstrained and constrained versions of the visibility map may be observed in Figure 7.14. Figure 7.15 presents the constrained visibility map of two simple models which clearly show vertex, edge and face components.

7.4 Feature extraction

Now that we are able to compute a visibility map with information about the components of the model, we need to extract, classify and connect them. Cells are selected as candidates that belong to components if their visibility radius is greater than a certain minimum. A compromise between the need of not missing any component and to clearly separate them has to guide to choosing of this minimum radius. All the figures from this point forward use a value of 11, which has yielded good results. This is caused by the fact that the way visibility radius values fall off as one approaches features is independent of the resolution of the model.

The candidate cells generated by the previous selection method need to be grouped into connected components and this is easily done by means of a flooding technique. Figure 7.16 shows the result of extracting components from the models in Figure 7.15.

With all the candidate cells grouped into components, these need to be classified as vertices edges and faces. Smaller components (components composed of 25 or less

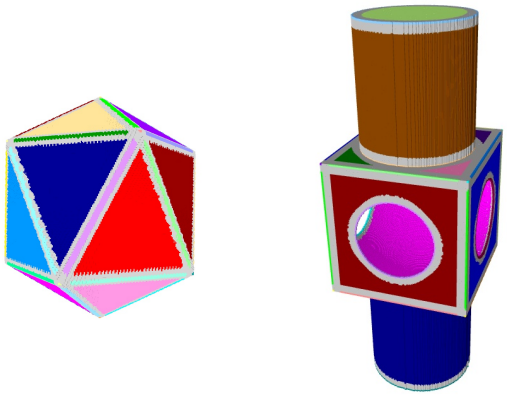


Figure 7.16: Color-coded components extracted from an icosahedron and a mechanical part. Grey is reserved for cells which were not selected to be part of a component.

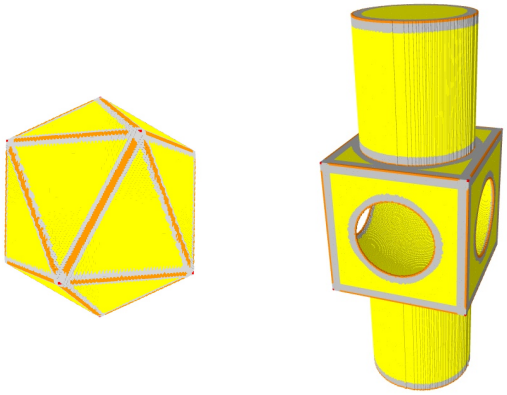


Figure 7.17: Components of an icosahedron and a mechanical part after being classified. Red, orange and yellow correspond to vertices, edges and faces, respectively.

cells) are identified as vertices. For the rest, we use a quantity which measures the "linearity" of a component. We define the *dimensionality* of a component as the mean of the dimensionalities of its constituent cells. The dimensionality of a given cell is the number of 6-neighbors which also happen to be in the same component. Dimensionality values of less than 3.5 correspond to edges, while values greater than 3.5 are associated to faces.

The components shown previously in Figure 7.16 are classified to obtain Figure 7.17.

The analysis of the components automatically yields part of the feature lines we wanted to recover. They are in the form of vertices and edges, which may be curved, but they still remain unconnected. To end the process of feature extraction, we define a structure, the *connectivity graph*, which has vertex and edge components as nodes. Only the arcs are needed to complete it and we will only establish vertex-edge

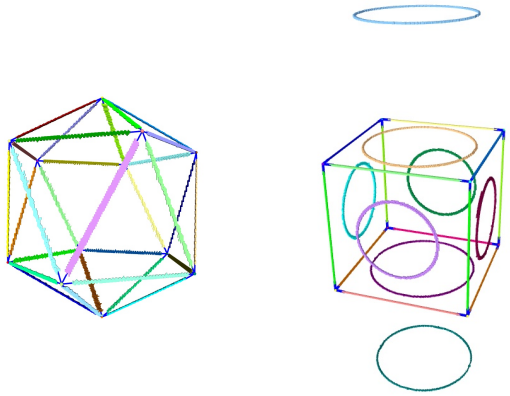


Figure 7.18: Resulting connectivity graphs of the icosahedron and the mechanical part from Figures 7.15, 7.16 and 7.17. Vertex components are represented here as spheres.

connections. The detection of such connectivities is performed by looking for cells from an edge component in the neighborhood of a vertex component. Vertices are grown using a dilation operator and the result is tested for intersections with edge components. If there is a non-empty intersection, an arc is added to the connectivity graph. Thus, feature lines are obtained (as seen in Figure 7.18).

7.5 Results

In order to analyze the results of the algorithm based on the constrained version of the visibility map, three more complex models are introduced in Figure 7.19. These models have been randomly rotated around an arbitrary axis to ensure that the technique is tested in a sufficiently general case.

Notice that the method is able to recover the sharp features of the presented objects accurately, even though a conservative estimate of the visibility was used. Only two issues may be seen in the pictures. First, the FANDISK model lacks two features in its left part. This effect is caused by these two particular features vanishing into neighboring faces. As a result, the features cannot be completely separated from the adjacent faces, and the algorithm fails to identify them. The second problem makes it difficult to recover a complete face region which is poorly voxelized with respect to its curvature (see highlight in Figure 7.19 and Figure 7.20). This issue is always solvable using larger resolutions.

Running times of the different parts of the algorithm are presented in Table 7.2. These were measured in a Pentium 4 at 3.4 GHz with 1 GB of RAM. The resolution

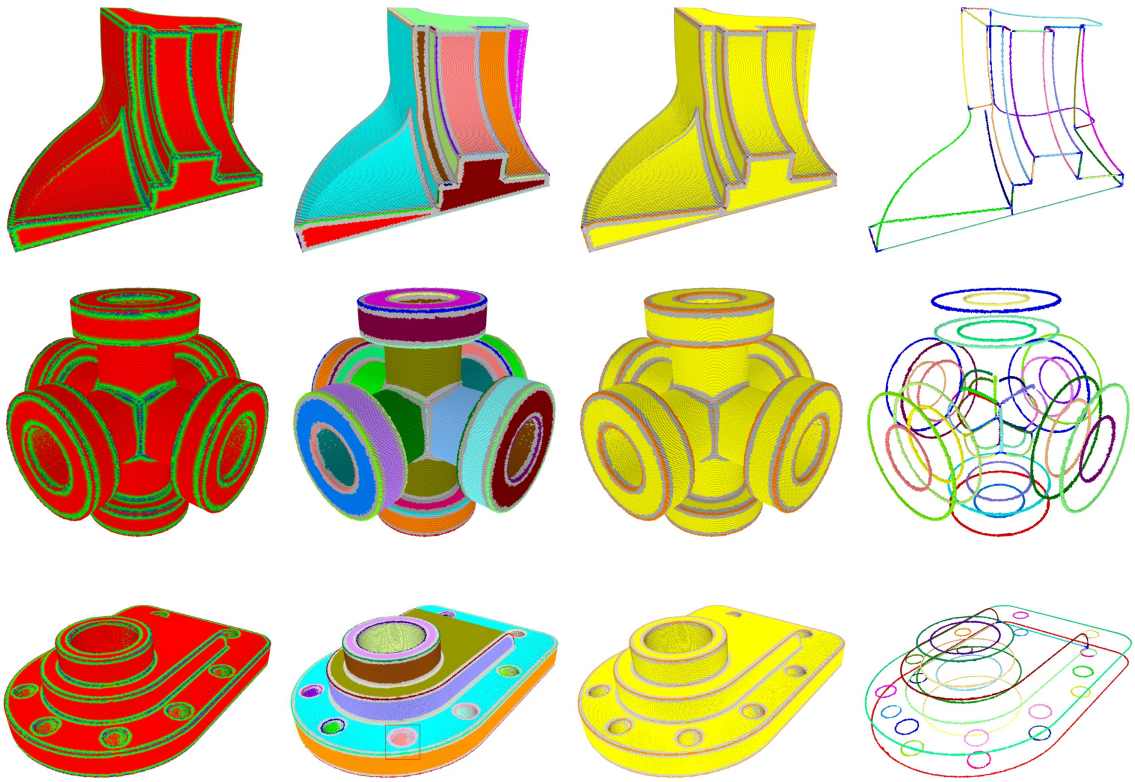


Figure 7.19: From left to right: visibility maps, identified components, its classification and the resulting connectivity graphs. From top to bottom: the models of the FANDISK, a CROSS pipe and another mechanical part (COVER).

and the number of mixed cells of all models are also listed (running times are related to the number of mixed cells, not the resolution). It may be seen that most of the weight of the process is in the visibility computation step.

In conclusion, this method is able to identify, classify and connect the components of a mechanical model given as a binary voxelization (which may have been obtained from any other representation). If the model has thin shells, we may expect them not to be recognized by the technique. This issue is easily solved choosing a discretization scale below the size of the features we want to recover.

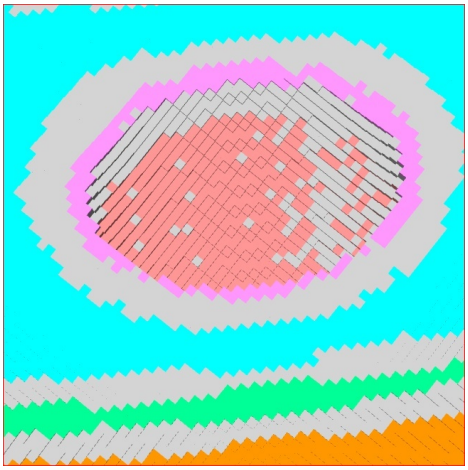


Figure 7.20: Zoom of the highlight of the COVER mechanical part in Figure 7.19.

	Resolution	Mixed cells	Visibility(seconds)	Components(ms)	Classification(ms)	Graph(ms)
ICOSA	256^3	220786	134	327	93	1960
PART	512^3	400600	254	285	125	1131
FANDISK	512^3	497513	253	434	193	5967
CROSS	512^3	615218	616	500	181	6365
COVER	768^3	970741	567	859	280	1024

Table 7.2: Resolution, number of mixed cells and running times of the four steps of the algorithm for the models shown in Figures 7.15 and 7.19.

Chapter 8

Conclusions & future work

Multiple tools that help extract shape signatures have been presented in this document. They all take binary voxelizations as input. In the case of scalar field discretizations, these techniques use only segmentation results, thus reducing the influence of any noise of the original scalar field (but temporary losing any additional information, even if it may be used after their application).

Four possible algorithms have been presented to optimize the topological properties of the isosurfaces resulting from the Marching Cubes extraction of binary grids. Changes of the topological properties of the isosurfaces caused by the decisions on these degrees of freedom are tracked through the use of two structures: the X-face graph and the vertex merge tree. Furthermore, strategies based on atomic decisions which are applied to the traversal of the X-face graph have been developed and the four previously mentioned algorithms are derived from a combination of those. As a result the number of shells, genus, total triangle count of the final triangular mesh may be easily optimized.

Additionally, a method to identify large planar regions of models in binary grid representation is also introduced. The presented approach reformulates the problem as finding the plane which stabs the maximum number of sticks (segments that connect IN and OUT nodes, and thus intersect the isosurface). This plane is obtained using a voting algorithm on a discretization of a suitable plane parameterization. The process is repeated to obtain a set of planar tiles which may be used in appearance-preserving simplification or recognition applications. One such impostor-based method is also shown.

Furthermore, a smoothing algorithm which works on isosurfaces is also presented. The smoothing operator used can work in the presence of equality and inequality constraints, so that the isosurface's topology is preserved. The constraints are enforced

applying a snapping step after each bilaplacian smoothing step. Used in combination with the maximal tiles detection, planar and non-planar regions of the isosurface may be recovered, while sharp features are extracted using the Edge Sharpener algorithm [AFRS04]. The result has potential applications in the shape recognition, simplification, compression, reverse engineering and manufacturing fields.

As a final contribution, a visibility-based feature extraction is demonstrated. The approach is able to identify, classify and connect in a topological graph, vertex, edge and face components. This is an improvement from the feature detection used in Pressing, which was based into the detection of large angles.

This work has resulted in the following publications:

- Carlos Andújar, Pere Brunet, Antoni Chica, Isabel Navazo, Jarek Rossignac, Àlvar Vinacua. Optimal Iso-Surfaces. *CAD Conference 2004 Proceedings*, pp. 503-512.

- Carlos Andújar, Pere Brunet, Antoni Chica, Isabel Navazo, Jarek Rossignac, Àlvar Vinacua. Computing Maximal Tiles and Application to Impostor-Based Simplification. *Computer Graphics Forum*, 2004, Vol. 23(3), pp. 401-410.

- Carlos Andújar, Pere Brunet, Antoni Chica, Isabel Navazo, Jarek Rossignac, Àlvar Vinacua. Optimizing the Topological and Combinatorial Complexity of Isosurfaces. *Computer-Aided Design*, 2005, Vol. 37(8), pp. 847-857.

- Antoni Chica, Jason Williams, Carlos Andújar, Pere Brunet, Isabel Navazo, Jarek Rossignac, Àlvar Vinacua. Pressing: Smooth Isosurfaces with Flats from Binary Grids. *Accepted for publication in Computer Graphics Forum.*

- Antoni Chica. Visibility-Based Feature Extraction From Discrete Models. *Submitted to Graphical Models.*

Future work will focus on improving current algorithms, develop them to be able to take advantage of the additional information contained in scalar fields, and apply them to a diverse set of fields.

There are two points which need improvement in the topology optimization algorithm. One is how to cut cycles in the X-face graph while still optimizing topological properties properly. Another is the problem of handling ties when the slashing of a single X-face is being decided upon.

Adapting the maximal tiles computation to work with thin models and non-orientable meshes. Furthermore, independently detect unconnected planar regions

of the model would provide a better set of tiles. Right now, the algorithm extracts one tile which may be capturing several planar regions that happen to be on the same plane.

As we have seen previously, the Pressing algorithm has a diverse set of potential applications. We have already tested it for reverse engineering mechanical models. It would be interesting to apply it to the rest of these fields in order to view how it behaves. Also, the smoothing operator could be adapted to use scalar field information by restricting the vertices movement to a certain distance ε of the initial estimation (not the whole stick). The initial estimation of each vertex would be computed using the bilinear interpolation of the two scalar samples from its corresponding stick, and the distance ε would correspond to the size of the noise we want to remove.

Improvements of the visibility-based feature extraction technique described here include: reducing the execution time of the visibility computation and adapt to method to be used on noisy models. The visibility step could benefit from adaptive multiresolution. Higher resolutions would be applied to high frequency details. Additionally, hardware accelerated visibility algorithms exist that might further improve efficiency. Finally, as an strategy to handle noise, the discrete band could be expanded before computing visibility.

Bibliography

[ABC*02] Alexa, M., Behr, J., Cohen-Or, D., Fleishman, S., Levin, D., Silva, C. Computing and rendering point set surfaces, 2002.

[AL99] Aliaga, D. G., Lastra, A. Automatic image placement to provide guaranteed frame rate. *Proceedings of the Conference on Computer Graphics (Siggraph99)*, pp. 307–316, 1999. AL99.pdf.

[ABK98] Amenta, N., Bern, M., Kamvysselis, M. A new Voronoi-based surface reconstruction algorithm. *Computer Graphics*, volume 32(Annual Conference Series):pp. 415–421, 1998.

[ACK01] Amenta, N., Choi, S., Kolluri, R. K. The power crust, unions of balls, and the medial axis transform. *Computational Geometry*, volume 19(2-3):pp. 127–153, 2001.

[ABA02] Andújar, C., Brunet, P., Ayala, D. Topology-reducing surface simplification using a discrete solid representation. *ACM Transactions on Graphics*, volume 21(2):pp. 88–105, 2002. ABA02.pdf.

[ABC+04] Andújar, C., Brunet, P., Chica, A., Navazo, I., Rossignac, J., Vinacua, À. Computing maximal tiles and application to impostor-based simplification. *Computer Graphics Forum*, volume 23(3):pp. 401–410, 2004. ABC+04.pdf.

[ABE+03] Andújar, C., Brunet, P., Esteve, J., Monclús, E., Navazo, I., Vinacua, À. Robust face recovery for hybrid surface visualization. *Proceedings of the VMV'03 Fall Workshop*, 2003. ABE+03.pdf.

[AFRS04] Attene, M., Falcidieno, B., Rossignac, J., Spagnuolo, M. Edge-sharpener: Recovering sharp features in triangulations of non-adaptively re-meshed surfaces. *Proc. of EG/ACM SIGGRAPH Symposium on Geometry Processing 2004*, pp. 62–69, 2004. AFRS04.pdf.

[AFSR05] Attene, M., Falcidieno, B., Spagnuolo, M., Rossignac, J. Sharpen&bend: Recovering curved edges in triangle meshes produced by feature- insensitive sampling. *IEEE Transactions on Visualization and Computer Graphics*, volume 11(2):pp. 83–91, 2005. AFSR05.pdf.

[BK02] Bischoff, S., Kobbelt, L. P. Isosurface reconstruction with topology control. *Proceedings of Pacific Graphics 2002*, pp. 246–255, 2002. BK02.pdf.

[B94] Bloomenthal, J. *An implicit surface polygonizer*, pp. 324–349. Academic Press, 1994. B94.pdf.

[CBC*01] Carr, J. C., Beatson, R. K., Cherrie, J. B., Mitchell, T. J., Fright, W. R., McCallum, B. C., Evans, T. R. Reconstruction and representation of 3D objects with radial basis functions. In E. Fiume, editor, *SIGGRAPH 2001, Computer Graphics Proceedings*, pp. 67–76. ACM Press / ACM SIGGRAPH, 2001.

[CGMS00] Cignoni, P., Ganovelli, F., Montani, C., Scopigno, R. Reconstruction of topologically correct and adaptive trilinear isosurfaces. *Computer and Graphics*, volume 24(3):pp. 399–418, 2000. CGMS00.pdf.

[CMRS98] Cignoni, P., Montani, C., Rocchini, C., Scopigno, R. A general method for preserving attribute values on simplified meshes. *IEEE Visualization '98*, pp. 59–66, 1998. CMRS98.pdf.

[COM98] Cohen, J., Olano, M., Manocha, D. Appearance-preserving simplification. *Computer Graphics*, volume 32:pp. 115–122, 1998. COM98.pdf.

[CCS03] Cohen-Or, D., Chrysanthou, Y. L., Silva, C. T., Durand, F. A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics*, volume 9(3):pp. 412–431, 2003.

[CAD04] Cohen-Steiner, D., Alliez, P., Desbrun, M. Variational shape approximation. *Proceedings of the 2004 SIGGRAPH Conference*, volume 23(3):pp. 905 – 914, 2004. CAD04.pdf.

[DDS03] Décoret, X., Debunne, G., Sillion, F. Erosion based visibility preprocessing. In *Eurographics Symposium on Rendering*, pp. 281–288. 2003.

[DDSD03] Décoret, X., Durand, F., Sillion, F., Dorsey, J. Billboard clouds for extreme model simplification. *ACM Transactions on Graphics*, volume 22(3):pp. 689–696, 2003. DDS03.pdf.

[DSSD99] Décoret, X., Sillion, F., Schaufler, G., Dorsey, J. Multi-layered impostors for accelerated rendering. *Computer Graphics Forum*, volume 18(3), 1999. DSSD99.pdf.

[DMSB99] Desbrun, M., Meyer, M., Schröder, P., Barr, A. H. Implicit fairing of irregular meshes using diffusion and curvature flow. *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pp. 317–324, 1999. DMSB99.pdf.

[DTS01] Dinh, H. Q., Turk, G., Slabaugh, G. Reconstructing surfaces using anisotropic basis functions. In *ICCV '01*, pp. 606–613. 2001.

[DDP02] Durand, F., Drettakis, G., Puech, C. The 3d visibility complex. *ACM Transactions on Graphics*, volume 21(2), April 2002.

[D88] Dürst, M. J. Letters: Additional reference to marching cubes. *Computer Graphics*, volume 22(2):pp. 72–73, 1988.

[EV99] El-Sana, J., Varshney, A. Generalized view-dependent simplification. *Computer Graphics Forum*, volume 18(3):pp. 83–94, 1999. EV99.pdf.

[EBV05] Esteve, J., Brunet, P., Vinacua, À. Piecewise algebraic surface computation and fairing from a discrete model. Technical report, LSI Research Report, LSI-05-44-R, 2005.

[FH86] Faugeras, O. D., Herbert, M. The representation, recognition, and locating of 3d objects. *The international journal of robotics research*, volume 5(3):pp. 27–49, 1986.

[FCO*05] Fleishman, S., Cohen-Or, D., Silva, C. T. Robust moving least-squares fitting with sharp features. *ACM Transactions on Graphics*, volume 24(3):pp. 544–552.

[GH97] Garland, M., Heckbert, P. S. Surface simplification using quadric error metrics. *Proceedings of SIGGRAPH '97*, pp. 209–215, 1997. GH97.pdf.

[GWH01] Garland, M., Willmott, A., Heckbert, P. S. Hierarchical face clustering on polygonal surfaces. *Proceedings of ACM Symposium on Interactive 3D Graphics*, 2001. GWH01.pdf.

[GG04] Gelfand, N., Guibas, L. J. Shape segmentation using local slippage analysis. *Eurographics Symposium on Geometry Processing*, pp. 214 – 223, 2004. GG04.pdf.

[G98] Gibson, S. F. F. Constrained elastic surface nets: generating smooth surfaces from binary segmented data. *MICCAI?98, Medical Image Computation and Computer Assisted Surgery*, 1998. G98.pdf.

[GWM01] Gumhold, S., Wang, X., McLeod, R. Feature extraction from point clouds, 2001.

[GW01] Guskov, I., Wood, Z. J. Topological noise removal. *Proceedings of Graphics Interface 2001*, pp. 19–26, 2001. GW01.pdf.

[HR95] Hill, S., Roberts, J. C. *Surface models and the resolution of n-dimensional cell ambiguity*, pp. 98–106. Academic Press, 1995. HR95.ps.

[HWCC05] Ho, C.-C., Wu, F.-C., Chen, B.-Y., Chuang, Y.-Y., Ouhyoung, M. Cubical marching squares: Adaptive feature preserving surface extraction from volume data. *Computer Graphics Forum (Eurographics 2005)*, volume 24(3):pp. 537–546, 2005. HWCC05.pdf.

[HDD*92] Hoppe, H., DeRose, T., Duchamp, T., McDonald, J., Stuetzle, W. Surface reconstruction from unorganized points. *Computer Graphics*, volume 26(2):pp. 71–78, 1992.

[HMM+03] Huang, R., Ma, K.-L., McCormick, P., Ward, W. Visualizing industrial ct volume data for nondestructive testing applications. In *Proceedings of IEEE Visualization 2003 Conference*. IEEE, 2003.

[JWB+06] Jenke, P., Wand, M., Bokeloh, M., Schilling, A., Strasser, W. Bayesian point cloud reconstruction. In *Proc. Eurographics 2006*. 2006.

[JLSW02] Ju, T., Losasso, F., Schaefer, S., Warren, J. Dual contouring of hermite data. *ACM Transactions on Graphics*, volume 21(3):pp. 339–346, 2002. JLSW02.pdf.

[KT96] Kalvin, A. D., Taylor, R. H. Superfaces: Polyhedral approximation with bounded error. *Medical Imaging: Image Capture, Formatting, and Display*, pp. 2–13, 1994. KT96.pdf.

[KR04] Klette, R., Rosenfeld, A. *Digital Geometry*. Morgan Kaufmann, 2004.

[K97] Kobbelt, L. P. Discrete fairing. *Proceedings of the Seventh IMA Conference on the Mathematics of Surfaces (1997)*, pp. 101–131, 1997. K97.pdf.

[KBSS01] Kobbelt, L. P., Botsch, M., Schwanecke, U., Seidel, H.-P. Feature sensitive surface extraction from volume data. *ACM Computer Graphics (Siggraph 2001)*, pp. 57–66, 2001. KBSS01.pdf.

[L96] Lachaud, J.-O. Topologically defined iso-surfaces. *Proceedings of 6th Discrete Geometry for Computer Imagery (DGCI '96)*, pp. 245–256, 1996. L96.pdf.

[LB03] Lopes, A., Brodlie, K. Improving the robustness and accuracy of the marching cubes algorithm for isosurfacing. *IEEE Transactions on Visualization and Computer Graphics*, volume 9(1):pp. 16–29, 2003. LB03.pdf.

[LC87] Lorensen, W. E., Cline, H. E. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics*, volume 21(4):pp. 163–169, 1987. LC87.pdf.

[LT97] Low, K.-L., Tan, T.-S. Model simplification using vertex-clustering. *Proceedings of the Symposium on Interactive 3D Graphics '97*, pp. 75–82, 1997. LT97.pdf.

[LE97] Luebke, D., Erikson, C. View-dependent simplification of arbitrary polygonal objects. *Proceedings of SIGGRAPH '97*, pp. 199–208, 1997. LE97.pdf.

[MAV*05] Mederos, B., Amenta, N., Velho, L., Figueiredo, L. H. Surface reconstruction from noisy point clouds. In *Symposium of Geometry Processing '05*, pp. 53–62. 2005.

[MSS*94] Montani, C., Scateni, R., Scopigno, R. Discretized marching cubes. *Visualization '94 Proceedings*, pp. 281–287, 1994. MSS94.pdf.

[MSS94] Montani, C., Scateni, R., Scopigno, R. A modified look-up table for implicit disambiguation of marching cubes. *The Visual Computer*, volume 10(6):pp. 353–355, 1994.

[MSS00] Montani, C., Scateni, R., Scopigno, R. Decreasing isosurface complexity via discrete fitting. *Computer Aided Geometric Design*, volume 17:pp. 207–232, 2000. MSS00.pdf.

[N03] Nielson, G. M. On marching cubes. *IEEE Transactions on Visualization and Computer Graphics*, volume 9(3):pp. 283–297, 2003. N03.pdf.

[N04] Nielson, G. M. Dual marching cubes. *IEEE Visualization 2004*, pp. 489–496, 2004. N04.pdf.

[NFHL91] Nielson, G. M., Foley, T. A., Hamann, B., Lane, D. Visualizing and modeling scattered multivariate data. *IEEE Computer Graphics and Applications*, volume 11(3):pp. 47–55, 1991. NFHL91.pdf.

[NGH+03] Nielson, G. M., Graf, G., Holmes, R., Huang, A., Phielipp, M. Shrouds: Optimal separating surfaces for enumerated volumes. *EG-IEEE TCVG Symposium on Visualization 2003*, pp. 75–84, 2003. NGH+03.pdf.

[NH91] Nielson, G. M., Hamann, B. The asymptotic decider: Resolving the ambiguity in marching cubes. *Proceedings of IEEE Visualization 91*, pp. 83–91, 1991. NH91.pdf.

[NB93] Ning, P., Bloomenthal, J. An evaluation of implicit surface tilers. *IEEE Computer Graphics and Applications*, volume 13(6):pp. 33–41, 1993. NB93.pdf.

[NBG02] Nirenstein, S., Blake, E., Gain, J. Exact from-region visibility culling, 2002.

[NT03] Nooruddin, F. S., Turk, G. Simplification and repair of polygonal models using volumetric techniques. volume 9(2):pp. 191–205, April 2003.

[R01] Rossignac, J. 3d compression made simple: Edgebreaker with zip&wrap on a corner-table. In *Proceedings of the International Conference on Shape Modeling & Applications*, pp. 278–283. 2001.

[RB93] Rossignac, J., Borrel, P. Multi-resolution 3d approximation for rendering complex scenes. *Second Conference on Geometric Modelling in Computer Graphics*, pp. 453–465, 1993. RB93.pdf.

[SOS00] Seul, M., O'Gorman, L., Sammon, M. J. *Practical Algorithms for Image Analysis*. Cambridge University Press, 2000.

[S01] Sheffer, A. Model simplification for meshing using face clustering. *Computer Aided design*, volume 33(13):pp. 925–934, 2001. S01.pdf.

[SDB97] Sillion, F., Drettakis, G., Bodelet, B. Efficient impostor manipulation for real-time visualization of urban scenery. *Computer Graphics Forum*, volume 16(3):pp. 207–218, 1997. SDB97.pdf.

[SH97] Stander, B. T., Hart, J. C. Guaranteeing the topology of an implicit surface polygonization for interactive modeling. *Computer Graphics*, volume 31(1):pp. 279–286, 1997. SH97.pdf.

[T95] Taubin, G. A signal processing approach to fair surface design. *SIGGRAPH 95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pp. 351–358, 1995. T95.pdf.

[TO99] Turk, G., O'Brien, J. F. Shape transformation using variational implicit functions. *Computer Graphics*, volume 33(Annual Conference Series):pp. 335–342, 1999.

[VKSM04] Varadhan, G., Krishnan, S., Sriram, T., Manocha, D. Topology preserving surface extraction using adaptive subdivision. *Proc. of EG/ACM SIGGRAPH Symposium on Geometry Processing 2004*, pp. 241–250, 2004. VKSM04.pdf.

[VMC97] Várady, T., Martin, R. R., Cox, J. Reverse engineering of geometric models - an introduction. *Computer Aided Design*, volume 29(4), 1997. VMC97.pdf.

[W91] Wallin, A. Constructing isosurfaces from ct data. *IEEE Computer Graphics and Applications*, volume 11(6):pp. 28–33, 1991. W91.pdf.

[W00] Whitaker, R. T. Reducing aliasing artifacts in iso-surfaces of binary volumes. *VVS ?00: Proceedings of the 2000 IEEE symposium on Volume visualization*, pp. 23–32, 2000. W00.pdf.

[WG90] Wilhelms, J., Gelder, A. V. Topological considerations in isosurface generation. *Computer Graphics*, volume 24(5):pp. 79–86, 1990. WG90.pdf.

[W02] Wilson, A. T. *Spatially Encoded Image-Space Simplifications for Interactive Walkthrough*. Ph.D. thesis, University of North Carolina, 2002. W02.pdf.

[WWS00] Wonka, P., Wimmer, M., Schmalstieg, D. Visibility preprocessing with occluder fusion for urban walkthroughs. In B. Péroche, H. Rushmeier, editors, *Rendering Techniques 2000 (Proceedings of the Eurographics Workshop on Rendering 2000)*, pp. 71–82. Eurographics, Springer-Verlag Wien New York, 2000.

[WMW86] Wyvill, B., McPheeters, C., Wyvill, G. Data structures for soft objects. *The Visual Computer*, volume 2(4):pp. 227–234, 1986.

[Z92] Zahlten, C. Piecewise linear approximation of isovalued surfaces. *Advances in Scientific Visualization*, pp. 105–118, 1992.