NONOBTUSE MESHES WITH GUARANTEED ANGLE BOUNDS

by

John Yung San Li

B.Sc. Honours, Simon Fraser University, 2004

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
Fall 2006

APPROVAL

Name:	John Yung San Li
Degree:	Master of Science
Title of thesis:	Nonobtuse Meshes with Guaranteed Angle Bounds
Examining Committee:	Dr. Gábor Tardos Chair
	Dr. Richard Zhang, Assistant Professor, Computing Science Senior Supervisor
	Dr. Daniel Weiskopf, Assistant Professor, Computing Science Supervisor
	Dr. Torsten Möller, Associate Professor, Computing Science SFU Examiner
Date Approved:	

Abstract

High-quality mesh representations of 3D objects are useful in many applications ranging from computer graphics to mathematical simulation. We present a novel method to generate triangular meshes with a guaranteed face angle bound of $[30°, 90°]$. Our strategy is to first remesh a 2-manifold, open or closed, mesh into a rough approximate mesh that respects the proposed angle bounds. This is achieved by a novel extension to the Marching Cubes algorithm. Next, we perform an iterative constrained optimization, along with constrained Laplacian smoothing, to arrive at a close approximation of the input mesh. A constrained mesh decimation algorithm is then carried out to produce a hierarchy of coarse meshes where the angle bounds are maintained. We demonstrate the quality of our work through several examples.

To my parents

Acknowledgments

I would like to express my gratitude to Dr. Richard Zhang who has been more than a supervisor throughout my studies. He has always provided me with guidance, motivation, and support. Having the chance to work with him was an honour. I would also like to take this opportunity to thank the fellow students at the GrUVi lab for lending their helping hands whenever I desperately needed help. I give special acknowledgement to my parents who were of great influence to me throughout my life. I thank them for their unconditional love and support. Last but not least, I owe special thanks to Jackie for her continuous support and encouragement. Without them, I would not have been able to acheive my goal.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Triangle meshes are the most common surface representation of 3D objects in computer graphics. One of the main reasons triangle meshes are so popular is that graphics hardware devices are optimized for operating on such representations. Moreover, triangle meshes can be used to model objects with arbitrary shapes and topology with ease. In recent years, there have been a wide range of research studies on triangle meshes and many algorithms have been developed specifically for them. For instance, mesh simplification [24] and subdivision [14] are primarily used to generate a hierarchy of meshes to represent geometry at different levels of detail. Mesh compression [3, 31] is designed to reduce the overall size of large meshes. Finite element methods (FEM) are applied to meshes to obtain approximate solutions to boundary value problems in engineering [28, 42]. This is just to name a few in the areas of computer graphics and engineering.

Essentially, a triangle mesh is a set of triangles connected at their common edges. These meshes are often representations of 3D objects that are used in visualization, animation, interactive games, mathematical simulation, and others. Often they are generated by laser scanning devices or constructed from implicit representations. However, the quality of the meshes produced by these algorithms is usually unsatisfactory. Many applications, on the other hand, require meshes with good quality. Thus, much research has been conducted on generating high quality meshes for various purposes. From here on, we refer to meshes as 2-manifold triangle meshes embedded in 3D space.

High quality meshes are often utilized in the field of mathematical simulation, in which they are used to solve complex problems. Moreover, they are also used for other purposes

such as efficient mesh compression [3, 31], geometry processing [30], and interactive free-form modeling [12]. Different quality measures have been devised because of that. Mesh quality is often measured in terms of the aspect ratio of the triangles, the uniformity of the sampling, or the local smoothness of the surface. In a survey paper by Pébay and Baker [40], several triangle quality measures are compared and analyzed including extremal angles, radii-ratio, and edge-ratio. In particular, we are interested in enhancing the angle measure of each triangle on the mesh. Namely, we wish to generate triangle meshes where all the face angles respect the angle bound of $[30°, 90°]$.

We say a triangle is *well-shaped* [16] if and only if all three of its angles are greater than or equal to 30°. If a triangle has all angles less than or equal to 90°, we say the triangle is *nonobtuse*. Then, we define a *well-shaped nonobtuse triangle mesh* to be a triangle mesh composed of triangles that are both well-shaped and nonobtuse. Immediate benefits of having a well-shaped nonobtuse mesh include numerical stability and reliability of computations for FEM simulations, efficient and accurate geodesic distance computations on surfaces, guaranteed validity for planar mesh embedding, and low entropy for mesh compression. We will elaborate on these issues later in this chapter.

To the best of our knowledge, there are no known algorithms that are guaranteed to produce a well-shaped nonobtuse triangle mesh which either interpolates or accurately and smoothly approximates a given point cloud. The same holds for the related remeshing problem, where one seeks a well-shaped nonobtuse triangle mesh which approximates a given mesh. To make effective use of these well-shaped nonobtuse meshes in interactive applications, it is also desirable to be able to construct a model hierarchy where at each level of detail, we have a mesh that satisfies the proposed angle constraints. In this work, we provide a guaranteed solution to the remeshing problem where an input mesh, with or without boundaries, is remeshed into a well-shaped nonobtuse mesh that approximates the input mesh. In addition, we also give a guaranteed solution to perform mesh decimation to produce a sequence of well-shaped nonobtuse meshes at different levels of detail.

1.1 Motivations and Applications

Initially, we intended to develop a method that would generate guaranteed nonobtuse meshes and this was motivated by many works studied in the computer graphics communities. Nonobtusity of a triangle mesh can benefit many existing algorithms. For instance, the

Fast Marching method [30] for computing geodesic distances across a mesh surface requires extra processing at triangles with obtuse angles. Moreover, nonobtusity implies that certain key properties of the discrete Laplacian-Beltrami operator would be analogous to those of the classic Laplacian-Beltrami operator on a surface with Riemannian metric [11]. Also, a nonobtuse triangulation is necessarily a Delaunay triangulation both in the plane [7] and for manifold triangle meshes [11]. Hence, a nonobtuse mesh will have all the desirable properties of a Delaunay mesh.

To improve the usefulness of our work, we aim to achieve an angle lower bound on the meshes as well so as to avoid having small angles. This resulting mesh, with no large angles and no small angles, is beneficial to engineering and mathematical simulations where finite element methods are applied [41]. Furthermore, having a mesh with restricted angle bounds implies that the set of angles in the mesh could have low entropy. This is quite useful in compressing large data sets for mesh processing. We now elaborate on how well-shaped nonobtuse meshes can be used in these applications.

- **Geodesic computation via Fast Marching**: The computation of geodesic distances between pairs of points on a surface is useful in many areas of computer graphics, however, such computations are often quite expensive. In 1998, Kimmel and Sethian proposed a method, called Fast Marching [30], that would compute geodesic distances efficiently. As of yet, many research works are still based on this idea. Although the complexity of their method is manageable, meshes with obtuse angles may introduce significant error in the computation. For these meshes, extra care must be taken in which virtual directional edges are added at obtuse angles. As a consequence, the accuracy of the results degrades. Therefore, using well-shaped nonobtuse meshes is advantageous when accurate and efficient computation of geodesic distance between two points on a surface is needed.

- **Ensuring validity in planar mesh embedding via discrete Harmonic maps**: Often problems are not as complex in the 2D domain when compared to the analogous case in the 3D domain. Moreover, problems in the 2D domain are well studied. Thus, there is interest in parameterizing a triangle mesh from the 3D domain onto a planar domain where it is desirable to preserve the topology and geometry of the original mesh. However, the quality of the parameterization impacts the overall results of the problem being solved. Eck et al. [20] propose a method to parameterize a mesh

in 3D onto the planar domain using discrete Harmonic maps, but Floater [23] has shown that these discrete Harmonic maps may not produce a valid planar embedding due to negative weights. On the other hand, the nonobtusity of a mesh ensures that the weights are always nonnegative. Hence, we can be assured that any well-shaped nonobtuse mesh will have a valid mesh embedding in the planar domain using discrete Harmonic maps.

- **Finite element methods**: Often used in engineering and mathematical simulation, the finite element method (sometimes referred to as finite element analysis) [28, 42] is a computational technique that is used to approximate solutions of boundary value problems. It is often applied in solving engineering problems that involve heat transfer, fluid velocity, elasticity, and structural mechanics, just to name a few. For instance, finite elements can be used to model the stress distribution of a load-carrying device [28]. Instead of attempting to solve for the stress at every point of the device, we can approximate it by discretizing the domain into components called elements. This can be achieved by modeling the load-carrying device as a triangle mesh where the vertices of the mesh carry information about the stress at their locations with respect to an expected range of force applied to the device.

  By discretizing the domain, we can formulate the problem of interest in terms of each element and then assemble the set of local problem formulations into a global one. The critical part of this approach is the discretization. If this step is poorly carried out, the results may become numerically unstable. This can be easily explained by observing the shape of a triangle [41]. Imagine we are flattening a triangle so that one of the angles is approaching π; the gradient along the direction where the flattening occurs will approach infinity. Hence, applications that are interested in the gradient will have large numerical errors in their solutions. Without going into more details, small angles are also troublesome as shown in [41]. Therefore, by generating meshes with the absence of large and small angles in the context of finite elements, one can be certain that the solution is numerically stable and accurate [4, 41].

- **Mesh compression**: Due to advances in model acquisition tools, the number of vertices in the mesh models is growing rapidly to the million and billion mark. One major bottleneck comes down to transferring these large data sets efficiently. Mesh compression techniques are developed specifically for this reason. A mesh is first

analyzed and then encoded using a scheme that requires less number of bits. As suggested in the study of information theory, data with low entropy can be expressed with fewer bits than those with high entropy; thus, meshes with low entropy can be compressed more effectively. Note that a well-shaped nonobtuse mesh has an angle bound of $[30°, 90°]$, which implies that the angle distribution of the mesh tends to have low entropy. Therefore by representing the mesh in terms of the face angles, the overall size of the mesh can be tremendously reduced. Note that this is achievable by applying the Angle-Analyzer compression method [31], where the geometry of the mesh can be recovered by the face angles of the triangles and the dihedral angle between the triangles.

1.2 Contributions

In this work, we present a novel method for generating guaranteed well-shaped nonobtuse remeshings via an extension of the Marching Cubes algorithm, followed by "deform-to-fit" iterative constrained optimization and angle-bounded decimation. Our contributions can be summarized as follows.

- **Well-shaped nonobtuse remeshing**: By utilizing a novel extension of the Marching Cubes algorithm, we are able to remesh an open or closed manifold mesh with any genus into a new mesh with all face angles within $[30°, 90°]$ that roughly approximates the input mesh. This extension allows meshes to be remeshed via Marching Cubes without the need of a 3D scalar field.
- **"Deform-to-fit" optimization**: We devise an iterative constrained optimization that reduces the approximation error of the well-shaped nonobtuse mesh generated from the previous remeshing step. The result is a high-quality approximation of the original input mesh that respects the proposed angle bounds.
- **Extendible framework**: The framework of our optimization allows other simple mesh operations to be applied such that angle constraints are satisfied. In particular, we show how the framework can be extended to perform constrained Laplacian smoothing and angle-bounded mesh decimation.

To the best of our knowledge, the proposed algorithm is the first to produce triangular meshes with a guaranteed angle bound of $[30°, 90°]$. Our work will benefit researchers, engineers, and mathematicians in the various applications described in the previous section.

1.3 Thesis Organization

The rest of the thesis is organized as follows. We begin by discussing some related works in Chapter 2. In Chapter 3, we give a brief overview of the problem we are trying to solve along with the difficulties and attempts in solving it. In addition, we present an overview of the approach that we have taken. Then in Chapter 4, we discuss details of the first stage of our approach in which the input mesh is remeshed into a new well-shaped nonobtuse mesh that roughly approximates the input mesh. In the next chapter, we explain the details of the iterative constrained optimization that we formulated for deforming the mesh generated in the first stage into a good approximation of the input mesh. Following that in Chapter 6, we present how the iterative constrained optimization framework can be extended to perform mesh decimation. We then present experimental results in Chapter 7. Finally in Chapter 8, we conclude and discuss limitations of our work and possible future work.

Chapter 2

Related Works

For the past decade, there have been many interests in generating triangulations with good angle qualities. Acute triangulation of special planar domains, e.g., a triangle or a square, has been a topic of recreational mathematics and studied fairly extensively, e.g., see [13]. An enlightening discussion on acute square triangulation can be found from Eppstein's Geometry Junkyard [21]. A majority of work studied in the past revolves around the planar domain [8, 9, 35] where the goal is to provide a triangulation for the input data set that minimizes the maximum angle or maximizes the minimum angle. The input data sets are often a set of points in 2D or a planar polygonal region. The problem of generating a triangulation in the 3D domain is also well-studied. These research works address the problem of generating a tetrahedralization of a three dimensional polyhedral region, possibly with holes, such that the dihedral angles between two adjacent faces are within a certain bound. In contrast, there is not as much effort invested in studying the problem of generating triangulations on 2D manifold surfaces with guaranteed angle bounds. However, there are some related works where meshes with good angle quality are produced [2, 15, 16]. In this chapter, we give some references of the work in these relevant areas, and also we discuss some related works developed in the past that we have made use of in our work.

2.1 2D Triangulation

Much research with the goal of improving angle qualities is motivated by the need for numerical stability when applying the finite element method. These works aim to provide good angle quality to triangulations, in the planar region, of a given point set or a polygonal

region. Often, *Steiner points*, new vertices that are not in the original input data set, are inserted to ensure the quality of the angles is achieved. The number of Steiner points should not be large as this would degrade the performance of the triangulation. Hence, many past works [9, 10, 34, 35] strive for an algorithm that would provide a triangulation with the best angle bound yet using a minimum number of Steiner points. For instance, Mitchell [35] proposes a technique that would ensure that a planar triangulation of n vertices will have no angles larger than $7\pi/8$ by adding $O(n^2 \log n)$ Steiner points in $O(n^2 \log^2 n)$ time. Following that, Mitchell improves upon his work in [34] where he refines a triangulation, using $O(n)$ Steiner points, in $O(n \log^2 n)$ time where no Steiner points are allowed on any input edge. Improving even further, Bern et al. [10] demonstrate a $O(n \log^2 n)$ method that allows nonobtuse triangulation of a n-vertex polygonal region, possibly with holes, with $O(n)$ number of triangles in the triangulation. On a related topic, Mitchell [36] shows that there exists a tight bound on the number of vertices in the triangulation in terms of local feature size and the smallest angle. It should be noted that Delaunay triangulations also play a key role in providing meshes with good angle quality as they maximize the minimum angle [19]. They are well-studied and many variations have been proposed [1, 16, 15]. For more detail on the topic of quality triangulations in 2D, one may refer to the survey paper by Bern and Eppstein [8].

2.2 3D Tetrahedralization

Although most relevant works on triangulation with good angle quality are studied in the planar domain, efforts have been put to extend these works into d-dimensional Euclidean space. In general, the goal is essentially the same where one tries to triangulate a point set or a polyhedral region in d-dimensional space, possibly with the addition of Steiner points to ensure better quality elements, e.g., so that the interior angles between two facets ($(d-1)$-dimensional faces) are within a certain bound. Instead of using the interior angles as a quality measure, some previous work utilizes the aspect ratio of the simplices for quality assessment. In the 3D domain, the aspect ratio of a tetrahedra, as defined in [37], is the radius of the smallest circumscribing sphere over the radius of the largest inscribed sphere. Having a bounded aspect ratio will ensure the tetrahedra are in good conditions. Bern et al. [9] have extended their work in the 2D domain to d-dimensional space where they are able to triangulate any n points data set in $\mathbb{R}^d$ with no interior angles smaller than a

constant. Their algorithm runs in $O(n \log n+k)$ time where k is the output size. Mitchell and Vavasis [37] propose a similar tetrahedralization solution of a three dimensional polyhedral region with holes. Moreover, an extension to d-dimensional point sets of Bern et al. [6] gives an upper bound of $O(n^{\lceil d/2 \rceil})$ where no obtuse dihedral angles are presented. Other related work includes the paper by Eppstein et al. [22], where they study the problem of tiling a 3D domain using tetrahedra with only acute dihedral angles, and the work from Alliez et al. [1], where they propose a Delaunay-based approach to tetrahedral meshing.

2.3 3D Surface Triangulation

Surface mesh generation and remeshing of triangle meshes are well-studied problems in computer graphics. A great deal of work on remeshing of curved surfaces with various quality criteria is presented in [2]. Despite of that, there is only a limited number of publications on the subject of producing good quality triangulations on 3D surfaces with guaranteed angle bounds. Perhaps one of the most well known works is from Chew's paper on guaranteed-quality mesh generation for curved surfaces [16]. In his work, he proposes a triangulation method that would allow a region of a curved surface to be triangulated where all angles are between 30 and 120 degrees. His approach uses a variation of the Delaunay triangulation (DT) called the *constrained Delaunay triangulation* (CDT) where certain edges are not allowed to be flipped. Based on operations defined for CDT, an initial curved surface CDT is first constructed on the input data set. Then in an iterative fashion, operations such as insert, delete, and edge-split are performed such that the result is still a CDT. The iteration stops when all triangles are within the proposed angle bounds. A similar Delaunay-based method is proposed by Cheng and Shi [15], where they use restricted union of balls to generate an ϵ-sampling of a surface and extract a mesh from the DT of the sampling points. A lower bound on the minimum angle can be as high as 30° with proper choice of parameters. But to approach this bound, ϵ needs to be close to zero, which would result in a high triangle count. Both algorithms fall short of producing guaranteed well-shaped nonobtuse meshes. Generating nonobtuse meshes has been identified as an open problem by Gu and Yau [26], among others, and so far only some simple heuristics [26] have been suggested.

2.4 Other Related Works

In our work, we have utilized several ideas that have been proposed in the past. Briefly, our method utilizes a surface reconstruction technique that is based on Lorensen and Cline's Marching Cubes algorithm [33] and also the quadric error metric of Garland and Heckbert [24] which has been used for surface simplification. In the original Marching Cubes paper, Lorensen and Cline develop a surface reconstruction technique that would generate a triangular mesh for the zero level-set of a given 3D scalar field. Ever since then, various alternations of the original Marching Cubes have been proposed [27, 38]. Particularly in this work, we use a modified version of the Marching Cubes algorithm to generate an initial well-shaped nonobtuse mesh that will be deformed to approximate the input mesh. We have recently discovered that the modified triangulation cases of our work are similar to the ones proposed by Montani et al. [38], where their intention is to reduce the triangle count in the output mesh. With a novel patch-based extension to the Marching Cubes paradigm as an additional support for open meshes, we have developed a reconstruction technique that produces a rough approximation of the input mesh with guaranteed angle bound of $[30°, 90°]$.

This rough approximation of the input mesh is then used as the initial mesh in an iterative constrained optimization where the initial mesh is deformed to approximate the input mesh. We have made use of the quadric error metric as part of the error measure that guides the optimization step. Initially used for surface simplification in [24], the quadric error metric, well known for its success in mesh decimation, measures the sum of squared distances of a point to a set of planes. We take advantage of this concept to measure the distance of every vertex in the deforming mesh to the surface of the input mesh. As shown in our results, this measure performs quite well in our optimization stage. The details of the initial mesh construction and the "deform-to-fit" optimization are discussed in Chapters 4 and 5, respectively.

Chapter 3

Background and Overview

Although there has been some research in nonobtuse triangulations in the planar domain, the problem of generating nonobtuse triangulations on surfaces in three dimensional space has been virtually unexplored. In the next section, we provide several problem instances regarding the generation of nonobtuse meshes that are well-shaped, as well as the difficulties that are behind them. Following that, we discuss some attempts in solving these problems and also their respective drawbacks. Finally, we give an overview of our approach for solving one of the formulated problems in the subsequent section.

3.1 Problem Instances

Several interesting problems involving well-shaped nonobtuse meshing or remeshing can be formulated.

- **Problem 1 (Well-shaped nonobtuse remeshing):** Given an input mesh M, construct a triangle mesh $\hat{M}$ with angle bound $[30°, 90°]$, which smoothly and accurately approximates M using *any connectivity*.
- **Problem 2 (Well-shaped nonobtuse mesh generation):** Same as Problem 1, except the input is now a *point cloud*.
- **Problem 3 (Well-shaped nonobtuse meshing with given connectivity):** Same as Problem 1, except that the connectivity is given.

- **Problem 4 (Well-shaped nonobtuse mesh generation with given connectivity):** Same as Problem 3, except the input is a point cloud.
- **Problem 5 (Well-shaped nonobtuse triangulation):** Given a point cloud S, construct a triangle mesh with angle bound $[30°, 90°]$ that *interpolates* S, possibly with Steiner points added.

In this work, we present a solution to Problem 1. We use a constrained "deform-to-fit" strategy, starting with an approximating mesh, that respects the angle-bound constraints, obtained via a nonobtuse version of the Marching Cubes [33] algorithm. The goal is then to minimize the distance from each vertex of the output mesh to the surface of the input mesh. In contrast, Problem 2, a variation of Problem 1, is more difficult to solve. No explicit surface representation of the input is provided, so the topology and the presence of boundaries are unknown in advance. Hence, it is not easy to approximate, with angle constraints, the underlying surface that the input point cloud is conveying. We discuss this in more detail in Chapter 8.

With the mesh connectivity fixed, as for Problems 3 and 4, local smoothness of a nonobtuse solution may not be ensured if valence-3 vertices are present. After all in flat regions of a mesh, it is not possible for all angles surrounding a valence-3 vertex to be nonobtuse. It is worth noting however that for a closed mesh, both valence-3 and valence-4 vertices can be easily eliminated without modifying mesh geometry. In Figure 3.1, we show two schemes that we have come up with. In general, higher valences appear to be more desirable as far as nonobtuse meshing is concerned. However, high valence vertices located in a flat region of a mesh must have small angles surrounding them. Similar to the case of valence-3 vertex, they cannot be flattened near planar regions of a surface; otherwise it would violate the angle constraints. In any case, even without such undesirable vertex valences, it is still unclear how an obtuse mesh can be deformed into a nonobtuse one, while ensuring adequate final mesh quality.

Interpolatory triangulation of a point cloud (Problem 5) that upholds the angle bound constraints appears to be the hardest. A solution may not exist for any given point cloud or region boundary, unless Steiner points are allowed to be added. The process of adding Steiner points is also unclear. As Steiner points are added, valences of connected vertices increase. As discussed previously, high valence vertices may not be desirable when the angles are lower-bounded. In certain cases, we may need to add Steiner points to avoid introducing

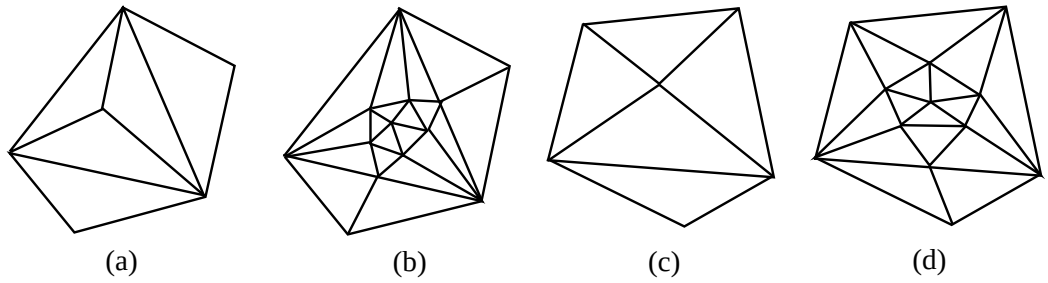


Figure 3.1: **Elimination of valence-3 and valence-4 vertices.** (a) A valence-3 vertex and its neighborhood. (b) Seven new vertices are inserted and the resulting configuration removes the valence-3 vertex without creating any new valence-3 or valence-4 vertices. In addition, none of the existing vertices have their degrees decreased. (c) A valence-4 vertex and its neighbourhood. (d) Configuration after removal of the valence-4 vertex. Note that one can easily have appropriate edges aligned so that (b), respectively, (d), does not differ from (a), respectively, (c), geometrically.

obtuse angles in the output mesh. Moreover, the placement of these Steiner points is also important in triangulating the point cloud that ensures the angle constraints are satisfied. It is quite conceivable that a solution would be hard to find, judging from the difficulty of producing guaranteed nonobtuse meshes in the much simpler domain of 2D polygons [10] or meshing curved surfaces under weaker angle conditions [15, 16]. Minimizing the number of Steiner points used is likely a hard problem as well.

3.2 Obvious Attempts and Difficulties

2D nonobtuse triangulation + mesh parameterization: For simplicity, let us consider the problem of generating nonobtuse meshes. One may take advantage of the well known solutions in the planar domain [10] to achieve nonobtuse triangulation in 3D space. An obvious attempt is to segment the input mesh into patches where they can be each mapped onto a planar polygon. The polygon is then triangulated, with the addition of Steiner points, to meet the angle bound requirements. Then the patches are mapped back to 3D via an angle-preserving mesh parameterization. One of the major drawback of this approach is that to ensure nonobtusity is preserved, the parameterization technique that

is utilized must have an angle distortion ratio λ, such that $\lambda\alpha \leq \pi/2$ where α is the maximum angle in the planar mesh. As of yet, we are not aware of any approach that can achieve this. Moreover, the addition of Steiner points introduces consistency issues along the boundaries of the patches. It is unclear as to how consistencies can be achieved in order to avoid T-vertices. In addition, generating well-shaped nonobtuse meshes is even more difficult as it requires a 2D nonobtuse triangulation technique, that is well-shaped, and a parameterization algorithm that can preserve the necessary angle bounds.

Advancing fronts: Another attempt to generating well-shaped nonobtuse meshes is to use an advancing front approach. Similar to the SwingWrapper approach [3], one can try to tile the mesh using well-shaped nonobtuse triangles across the surface. Obviously, difficulties will arise as the front is close to meeting with itself and no well-shaped nonobtuse triangle can fill the gaps. Devising either a back-tracking or a look-ahead strategy which would provide a guarantee is difficult.

Local mesh manipulations: Gu and Yau [26] proposed a heuristic to remove nonobtuse triangles by performing subdivision followed by a sequence of constrained edge collapses. One may also perform other simple mesh manipulation techniques such as edge flipping, vertex removal, and constrain movements of vertices. However, none of these attempts guarantees the resulting mesh is well-shaped and nonobtuse. Furthermore, it is quite likely such a process may "get stuck" where a large number of bad angles still exist.

3.3 Our Approach

As discussed in the previous section, many heuristics can be considered for generating nonobtuse meshes; however, none of these heuristics guarantees that the resulting mesh is nonobtuse or well-shaped. In contrast, our approach ensures that all face angles of the output mesh are within the angle bounds. We do so by ensuring at each step of the algorithm, we do not violate the desired angle constraints. The method we used in generating well-shaped nonobtuse meshes is a remeshing technique, where the input is an orientable manifold mesh. The overall procedure can be divided into three main stages. First, the input mesh is remeshed into an initial mesh that respects the angle bounds. Next, we iteratively move the vertices, in a greedy fashion that does not violate the angle constraints,

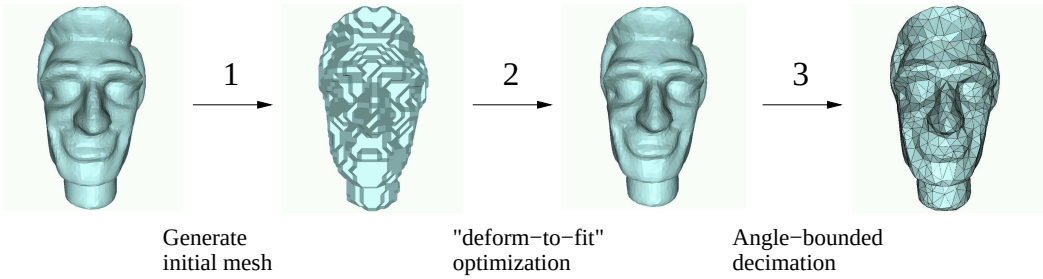


Figure 3.2: **Three main stages of our algorithm.**

with the goal of minimizing the error between the input mesh and the output mesh. At the final stage, we perform an angle-bounded decimation to transform the output mesh into various sizes. The overall steps of our algorithm are illustrated in Figure 3.2.

In the first stage, we attempt to generate an initial mesh that roughly approximates the input mesh while satisfying the angle constraints. This is achieved through a nonobtuse version of the Marching Cubes [33] algorithm. The original Marching Cubes algorithm requires the signs of a 3D scalar field as the input. However, this is undefined for meshes with boundaries. Therefore, we take a different approach where the signs are determined by the intersection of the input mesh with the sampling cartesian grid. Simply stated, the initial mesh is generated through several steps. The input mesh is first intersected, in a per triangle basis, with a cartesian grid in 3D space. For each intersected cube in this grid, we group the intersecting triangles of the mesh into connected patches. Based on the orientation of the input mesh, we then assign a sign to each corner of the cube which determines its relative location of the input surface. With the signs labeled to the cube corners, we triangulate each cube independently. The sign assignment and triangulation step are performed for each patch. Once all the intersected cubes are triangulated, a stitching procedure based on patch connectivity is carried out as a post-processing step to avoid topological changes and non-manifold cases. The resulting mesh is a well-shaped nonobtuse mesh that roughly approximates the input mesh and is used as the input for the optimization stage of our algorithm.

To approximate the original input mesh, we deform the initial mesh by relocating the vertices one at a time. Given a valence-k mesh vertex v, we refer to the *region of well-shapeness and nonobtusity* $\mathcal{WN}(v)$ for v as the set of points which would form k well-shaped

nonobtuse triangles (or $k-1$ if v is on the boundary) with the one-ring vertices of v. Note that $\mathcal{WN}(v)$ can be empty in an arbitrary mesh, but for our initial mesh, $\mathcal{WN}(v) \neq \emptyset$ for all v. We constrain the movement of each vertex to be within its region of well-shapeness and nonobtusity and wish to find a final mesh that is the closest, in terms of a least-square error measure, to the original mesh. This is a constrained global optimization problem, with a complex search domain that is the set of all well-shaped nonobtuse meshes having a particular connectivity. We solve this in a greedy manner while examining only a subset of the search domain. To ensure local smoothness of the mesh, we apply constrained Laplacian smoothing after the mesh optimization step. This allows vertices to be moved tangentially along the surface. As a consequence, rough surfaces are smoothed out. We interleave mesh optimization and constrained Laplacian smoothing with the former being the first and last steps of our "deform-to-fit" procedure.

A computational obstacle in the iterative optimization is that the angle constraints are nonconvex and nonlinear with respect to vertex positions. To make the optimization problem more feasible to solve, we conservatively linearize and convexify the constraints and transform the general nonlinear optimization problem into a *quadratic program* with a convex search domain. Note that our linearization and convexification step can reduce the size of the allowable search region (not to the empty set however). But by being conservative, the constraints still ensure that we always obtain a well-shaped nonobtuse mesh at each step.

Once a high-quality output mesh, in terms of the approximation error and smoothness, is arrived at, we can perform an angle-bounded decimation via a constrained greedy edge collapse based on a quadric error metric. This procedure resembles the quadric-based mesh decimation algorithm of Garland and Heckbert [24]. As we will describe in Chapter 6, the decimation framework is quite similar to the iterative constrained optimization setup that is carried out during the remeshing stage.

Chapter 4

Initial Mesh Construction

Generating well-shaped nonobtuse meshes is a difficult problem as discussed in Chapter 3. It is unclear how the angle constraints can be enforced as the mesh is manipulated through traditional surface reconstruction algorithms. In our approach, we focus on satisfying the angle constraints. Instead of attempting to generate a close approximation of the original mesh M from the start and slowly converting M to satisfy the angle constraints, we start off by generating an initial mesh M_0 that is guaranteed to have face angles within the proposed angle bounds. Although M_0 may only provide a rough approximation of the original mesh M, it serves as an initialization for our constrained "deform-to-fit" optimization, which we will discuss in Chapter 5.

To construct the initial mesh M_0, we utilize the Marching Cubes (MC) algorithm [33] of Lorensen and Cline. The MC algorithm produces a triangle mesh by computing the iso-surface of a given 3D scalar field. The scalar field is discretized at the vertices of a cubical cartesian grid. Depending on the signs at the cube vertices, a set of triangles is generated with the triangle vertices placed along the cube edges. The location of the triangle vertices are determined by the scalar values via linear interpolation. By considering rotations and symmetries, Lorensen and Cline reduce all 256 possible configurations down to 15 distinguishable cases. Subsequently, Nielson and Hamann [39] introduce several new cases to resolve ambiguities in the original MC algorithm that may generate holes in the output mesh. A more in-depth analysis of all the Marching Cubes cases can be found in [5].

To ensure the initial mesh M_0 satisfies the angle constraints, we need to make modifications to the original MC cases. In our work, several MC cases are changed topologically and geometrically. The nonobtusity of the triangles produced is ensured when triangle vertices

are placed on the midpoints of the cube edges or at the cube centers. Moreover, an angle lower bound of 30° is also guaranteed. We have recently discovered that these modifications resemble closely the Discretized Marching Cubes (DMC) [38] scheme. Proposed by Montani et al. [38], the DMC algorithm is designed to reduce the triangle count in the output mesh.

Since the original MC algorithm uses a 3D scalar field as an input, meshes with boundaries cannot be constructed as it is ambiguous to determine whether a point in 3D space is inside or outside the mesh. We overcome this problem with a patch-based approach where each cube is triangulated based on the intersection of the cube edges and patches from the input mesh. Triangle edges are then merged along the cube boundaries to ensure holes and boundaries that are not in the input mesh will not be generated.

We discuss the modifications made to the original MC cases in Section 4.1 followed by a detail explanation of the patched-based surface reconstruction process in Section 4.2. Finally, we show experimental results of the algorithm in Section 4.3.

4.1 Nonobtuse Marching Cubes

We modify the original Marching Cubes (MC) cases to ensure that all face angles of the triangles produced are within the proposed angle bounds as illustrated in Figure 4.1. In contrast to the original MC algorithm, triangle vertices that are placed on a cube edge are always located at the midpoint of the edge. Moreover, vertices are also placed in the center of the cube as in case 5, 9, 11, 12, and 14. To resolve possible ambiguities caused by the original MC cases, we include six extra cases [32] into consideration as shown in Figure 4.2. For cases 5, 9, 11, 12, 14, 3c, 6c, 7c, and 12c, we place a vertex at the center of the cube and triangulate appropriately with the vertices in the original cases. Take case 11 as an example, there is a 120° angle in the original configuration. We modify it by placing a vertex at the center of the cube and form a one-ring patch with the vertices in the original case. This results in a triangulation composed of 4 equilateral triangles and 2 isosceles right triangles.

We have recently discovered that the modified configurations that we proposed are similar to the cases given in DMC. The only difference between our configurations and the cases in DMC is that no valence-4 vertices are introduced in our scheme. In general, valence-4 vertices are undesirable in flat regions due to the nonobtusity constraints.

It can be shown, in a case by case manner, that our modified MC configurations do not

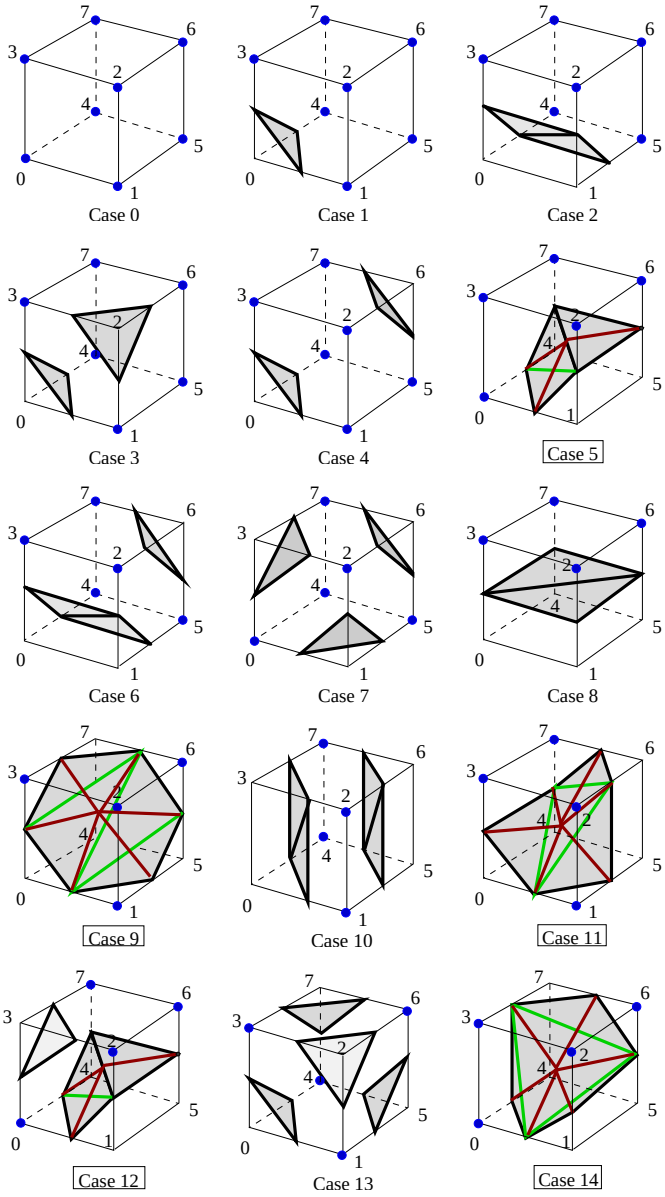


Figure 4.1: **Nonobtuse Marching Cubes: regular cases.** We show the 15 regular cases modified from the original Marching Cubes paper [33], where triangle vertices at a cube edge are located at the midpoints of the edge. Cases 5, 9, 11, 12, and 14 have obtuse angles in the original MC algorithm. Simple modifications can be made to the local triangulations to rectify that. In the figures, dark green line segments represent edges from the original triangulations [33] and brown line segments replace them. Edges shared by the original and new triangulations are colored in dark black. All the configurations shown are identical to the cases proposed in the DMC algorithm [38], except for cases 2, 6, 8, and 10, where we removed valence-4 vertices with a different triangulation.

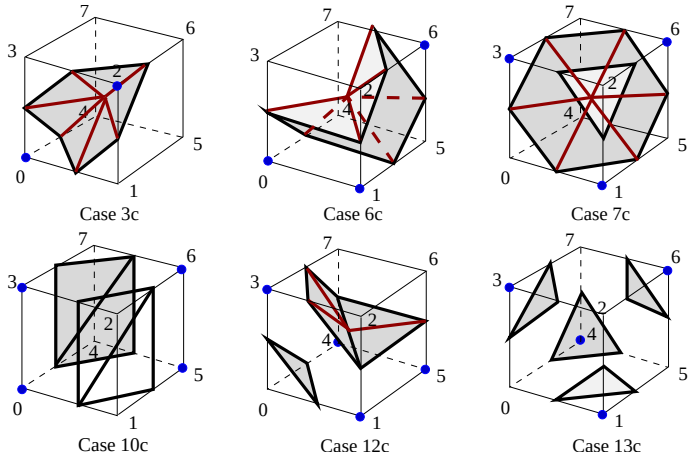


Figure 4.2: **Nonobtuse Marching Cubes: extra cases.** Six extra cases are modified to ensure that no obtuse angles are introduced. Brown segments are new and dark black segments come from the original Marching Cubes triangulation. All six configurations are identical to the cases from [38].

generate any triangles with obtuse angles. Furthermore, an angle lower bound of approximately 35° is also achieved. This can be seen in case 2, 6, and 10, where there is a right triangle with side lengths of $\frac{\sqrt{2}}{2}$, 1, and $\frac{\sqrt{6}}{2}$, assuming that all the cubes have unit edge lengths. The guaranteed angle bounds enable us to generate an initial mesh M_0 that satisfies the angle constraints, which can then be used in a constrained optimization algorithm for approximating the original input mesh M.

For meshes with boundaries, a precise 3D scalar field cannot be computed since it is ambiguous to determine whether a point in 3D space is inside or outside the mesh. For this reason, we devise a patch-based approach that would allow open and closed manifold meshes of any genus to be remeshed using the nonobtuse MC cases. We discuss the details of our approach in the next section.

4.2 Initial Mesh Construction via Patch-based MC

For the nonobtuse Marching Cubes cases that we propose, the triangle vertices are placed either at the midpoint of cube edges or in the center of the cube (i.e. no linear interpolation is used). This implies that the nonobtuse marching cube cases only require the assignment of signs to the cube corners. The signs of the cube corners dictate whether a point located at a cube corner is inside or outside the mesh. This is not always well-defined for meshes

with boundaries. However, since our input is an orientable manifold mesh, we can simply intersect the input mesh with the cubes and assign signs to the cube corners based on the intersections and orientation of the original mesh triangles. Once signs are consistently labeled, we can triangulate the cube using the nonobtuse MC cases. To avoid introducing holes and boundaries that are not in the original input mesh, we "stitch" up the triangle edges along the common faces between two cubes. To summarize, the surface reconstruction process is divided into three phases:

- **Phase 1: Mesh-Cube Intersections**
  First, we discretize the 3D space with a regular cubical grid. Then given an input mesh M, we determine the set of cubes in the cubical grid that intersect with M. During this process, we also keep track of the set of triangles that intersect each cube.

- **Phase 2: Patch-based MC triangulation**
  Once all the intersected cubes are identified, we group the set of triangles from M that intersect a cube into connected patches. Then for each patch, we assign signs to the corners of the intersected cube and triangulate the cube using the nonobtuse marching cube scheme as long as the sign assignment can be made consistent.

- **Phase 3: Stitching**
  After all intersected cubes have been triangulated, we attempt to "stitch" the resulting triangles into a manifold mesh by examining the connectivity of the patches along the common face of two adjacent cubes.

4.2.1 Mesh-Cube Intersections

The goal in the first phase of our surface reconstruction process is to identify the set of cubes in the cubical cartesian grid that intersect with the input mesh M. At the same time, we wish to assign to each cube the set of triangles in M that intersect it. The intersected cubes are processed in phases two and three of the algorithm to produce a remeshing of M. Instead of checking every cube in the grid in a brute-force manner, we devise a slice-based approach that determines the set of cubes intersecting a particular triangle. The use of a triangle voxelization algorithm [29] is not sufficient in this phase. This is simply because we need the set of cubes intersecting the mesh rather than a discretization of the mesh in the cubical cartesian grid. Thus to ensure the correct set of cubes is identified, a post-processing

after voxelization must be carried out to examine the neighbours of each cube to add any missing ones that are intersected.

In our approach, we process triangles in the input mesh M one at a time through an axis-aligned scan on each of the three coordinate axes. For simplicity, let us consider the z axis only. Given a triangle T, we compute the tightest grid interval $[z_1, z_2]$ on the z axis that bound T where z_1 and z_2 are z-coordinates of certain cube vertices. Then we consider *slices* at grid points $z_1 + 1, \ldots, z_2 - 1$, which are planes passing through the corresponding grid points and perpendicular to the z axis. We intersect each slice S_i (located at grid point $z_1 + i$) with the triangle T, which results in a line segment l_i as shown in Figure 4.3(a).

We can then use a simple incremental algorithm to determine the pixels in slice S_i that intersect l_i. We achieve this in output sensitive time by examining the adjacent pixel along the direction of l_i starting from the pixel that contains one of the endpoints of l_i. An illustration is given in Figure 4.3(b). For each intersected pixel centered at $(x, y, z_1 + i)$, we mark the two cubes with the center located at $(x, y, z_1 + i - \frac{1}{2})$ and $(x, y, z_1 + i + \frac{1}{2})$ as intersected. For all the triangles in the input mesh M, we repeat this process on each of the three coordinate axes. There is one special case where this process may fail to identify an intersected cube. This is when a triangle is entirely inside a cube. To resolve this, we simply examine the coordinates of the triangle vertices to determine if it is inside a cube.

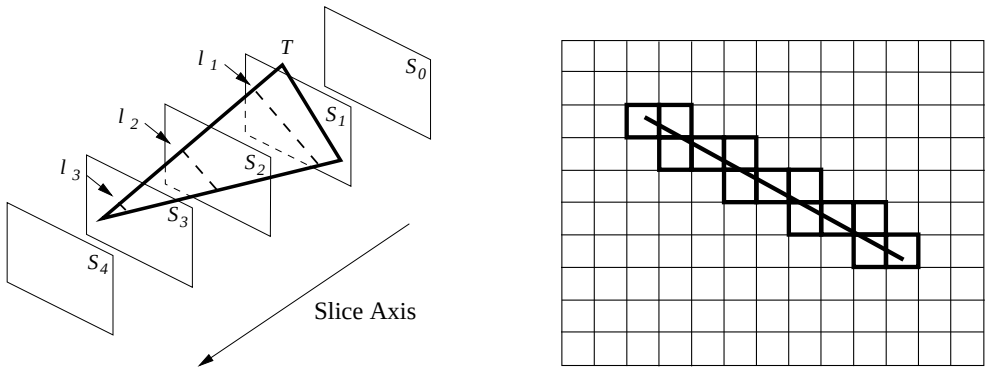


Figure 4.3: **Marking intersected cubes.** (a) Triangle T is intersected with slicing planes S_i along the slice axis. The intersection results in a line segment l_i projected onto plane S_i. (b) For each slicing plane S_i, the projected line segment l_i intersects a set of pixels, marked in bold. These pixels determine the set of cubes that intersects T at plane S_i.

Since each cube identified as intersected is considered twice during each axis-aligned

scan, then each intersected cube is examined at most six times for each triangle in the input mesh. Assuming the maximum valence of a vertex in the input mesh M is a small constant, then the cubes in the grid intersect only a constant number of triangles. Hence, the computational complexity of this algorithm is roughly proportional to the number of intersected cubes. The correctness of the algorithm can be shown in the following way.

Claim 1: Each cube marked by the algorithm intersects some triangle T in mesh M.
Proof: There are two cases in which a cube C is marked. First case is when C contains some triangle T. This is trivial because the algorithm checks if T is inside a cube explicitly. Let us consider the second case. A cube C is marked only when one of its faces has a pixel p that is marked. Moreover, pixel p is marked only if it intersects some line segment l_i. Since l_i is the intersection of some slice S_i and some triangle T in mesh M, p must intersect T. Hence, C must intersect T.

Claim 2: Each cube that intersects some triangle T in mesh M is marked by the algorithm.
Proof: Let C be a cube that intersects triangle T in mesh M. There are two cases to be considered.

- **Case 1: The triangle T is inside the cube C.** This is trivial since we explicitly check the vertex locations of T to determine if it is inside cube C.
- **Case 2: Without loss of generality, assume triangle T intersects the face of cube C that is parallel to the xy-plane.** Since T intersects one of the faces, say face f, that is parallel to the xy-plane, then there must exist some slice S_i on the z axis that passes through f and intersects T. The intersection of slice S_i and T is a line segment l_i that passes through face f. Since l_i passes though f, the two cubes that share face f are marked. Hence, C is marked as intersected.

4.2.2 Patch Construction and Sign Assignment

After phase one is completed, each intersected cube will be associated with a set of triangles in M intersecting it. In this phase, we attempt to triangulate each cube independently based on the nonobtuse Marching Cubes scheme discussed in Section 4.1. For each intersected cube C, we first group the set of triangles in M intersecting C into connected patches. Then,

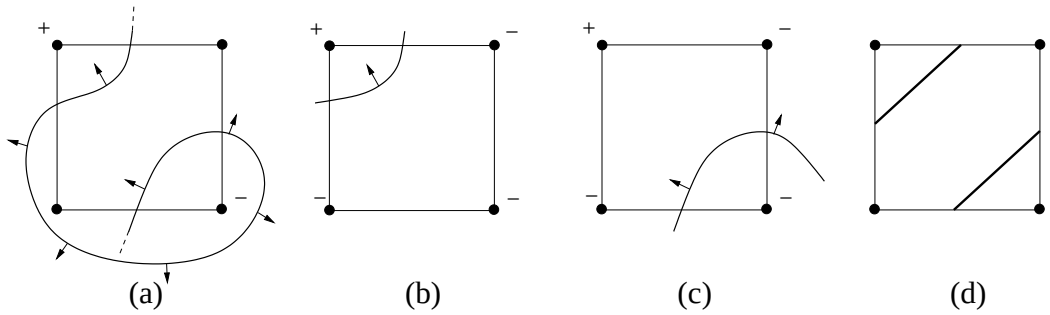


Figure 4.4: **The necessity of grouping into connected patches.** +, − denote outside and inside respectively. (a) Only the top-left and bottom-right corner of the illustrated square can be assigned with a sign consistently based on the curve orientation and the intersection of the curve and the edges of the square. To avoid this ambiguity that occurs for opened and self-intersected meshes, we group the intersected triangles into a set of connected patches and triangulate the cube based on the individual patches separately as shown in (b) and (c). The resulting triangulation of the square is shown in (d).

we process each patch independently. For each patch, we assign signs to the corners of C by examining the orientation of the patch and the intersection of the patch with the edges of C. Once a consistent sign assignment is arrived at, we triangulate C using the scheme illustrated in Figure 4.1 and 4.2.

To understand the reason for a patch-based approach, consider a 2D example illustrated in Figure 4.4. We would like to reconstruct part of the contour that is within the cube. By examining the orientation at the intersection of the contour and the square, we can be certain about the signs of only two corners of the square. However, if we group the intersecting line segments (triangles in 3D) into connected segments (patches in 3D), we can arrive at a consistent sign assignment for all the corners of the square as shown in Figure 4.4(b) and Figure 4.4(c). The conflict in assigning signs in the first case comes from the fact that the original contour is either open or has self-intersection rather because of low resolution in the sampling grid.

To triangulate an intersected cube C, we first group the set of triangles intersecting C into connected patches. Then for each patch P, we compute all intersections between P and edges of C. For each cube vertex v with incident cube edges e_1, e_2, e_3, we attempt to assign a sign of "+" (denoting outside) or "−" (denoting inside) to v. We achieve this by examining the intersection of P and e_1, e_2, e_3. Each edge e_k, $k \in \{1, 2, 3\}$, casts a vote of "+", "−", or "◯" based on the orientation of the triangle in P that intersects e_k closest to

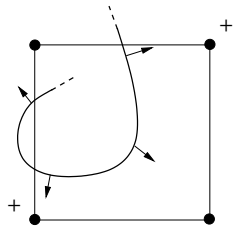


Figure 4.5: **Inconsistent signs, example 1.** Consider the top-left corner of the square. The horizontal incident edge casts a vote of "−" whereas the vertical incident edge casts a vote of "+".

v. If P does not intersect e_k, then e_k casts a vote of "◯". If all three incident edges voted "◯", then v is *unassigned*. If both "+" and "−" are voted, then v has a *sign conflict* and patch P is simply ignored. Otherwise, a consistent sign assignment for v is implied. A 2D example illustrating sign conflicts is shown in Figure 4.5.

After each cube vertex has been either assigned with a sign or marked as unassigned, we perform a *sign flooding* process, depicted in Figure 4.6, to resolve the unassigned vertices. The flooding process starts off at a vertex v with consistent sign assignment. Then the sign of v is propagated to its adjacent unassigned vertices. This process stops when all cube vertices are assigned with a sign or if a sign conflict arises, in which case patch P is ignored. An illustration of the sign conflict during the flooding process is illustrated in Figure 4.7.

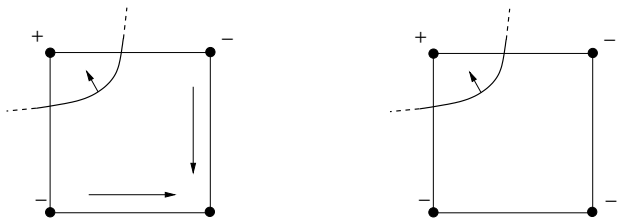


Figure 4.6: **Flooding.** (a) The bottom-right corner of the square is unassigned whereas the other corners have consistent sign assignment. We then proprogate the signs to adjacent unassigned corners. (b) Configuration after flooding.

Once a consistent sign assignment for all cube vertices is arrived at, we can then triangulate the cube using the nonobtuse Marching Cubes scheme. We do this for each patch intersecting the cube. For patches that gives inconsistent sign assignments, we simply ignore the patch. This occurs along open boundaries and areas with self-intersection. By ignoring these patches, the output meshes may introduce trimmed open boundaries and

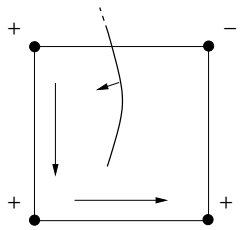


Figure 4.7: **Inconsistent signs, example 2.** The top-left corner is assigned with + and the top-right corner is assigned with −. As flooding proceeds, the bottom-left corner is assigned with "+" and that propagates to the bottom-right corner. However, the top-right corner propagates a sign of "−" to the bottom-right corner.

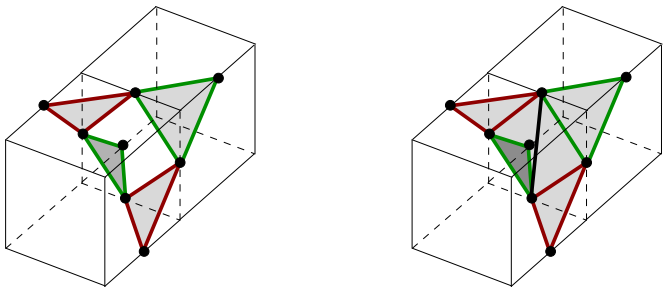
holes near areas with self-intersection. In the following section, we describe a method to connect together the edges of the triangles produced in each cube such that a manifold mesh is formed.

4.2.3 Stitching

In contrast to the original MC algorithm, the triangles produced in each cube in our patch-based approach are not implicitly connected with triangles generated in the adjacent cubes. The reason for this is that each cube is triangulated with respect to a set of patches. This may result in placing more than one vertex on a cube edge. In the original MC algorithm, this will never happen, therefore, triangulation of each cube can always reuse vertices that have already been introduced on a cube edge and the resulting set of triangles are connected to form a manifold mesh. In our approach, we need to perform a post-processing step to ensure proper connection of triangles in order to produce a manifold mesh. We call this the *stitching* step. It is a heuristic based on the connectivity of the patches associated with the cubes. An illustration is given in Figure 4.8 to show a nontrivial case where a stitching is necessary.

To understand when stitching should be performed, consider the 2D example illustrated in Figure 4.9. After the second phase of the surface reconstruction algorithm, the upper and lower square will generate edges e_1 and e_2, respectively, with an endpoint located along the common edge between the two squares. We determine whether e_1 and e_2 should be stitched by considering the two patches P_1 and P_2 which are responsible for generating e_1 and e_2. Since P_1 and P_2 are connected, we should connect e_1 and e_2 together.

Let us consider another example shown in Figure 4.10. After triangulating the squares,



(a) (b)

Figure 4.8: **Stitching example.** Red triangle edges are generated inside the front cube and green triangle edges are generated inside the back cube. (a) When a saddle-shaped surface intersects the common face of two cubes, the resulting triangulation of the two cubes will generate a hole on the common face. (b) We attempt to stitch up the edges on the common face by examining the connectivity between the patches responsible for generating the edges. As a result, two additional triangles are generated. This process is known as the stitching step.

two edges e_1 and e_2 are generated in the lower square with an endpoint located in the common edge between the upper and lower square. We would like to stitch up e_1 and e_2 if the patches P_1 and P_2 are connected. As illustrated in the figure, however they are not directly connected. Instead, P_1 and P_2 are connected via a common patch P_3 in the adjacent square which does not generate any edges. We define a *black patch* to be a patch that generates no triangles within the cube it is associated with. Patch P_3 in Figure 4.10 is an example of a black patch.

After phase two of the surface reconstruction algorithm, a set of triangles is produced within each cube. These triangles are not connected with triangles in the adjacent cubes to start with. We define an *open edge* to be a triangle edge that is shared by only one triangle. So after the triangulations of the cubes, all triangle edges lying on a cube face are open edges. We attempt to connect the set of triangles together by stitching the open edges on each cube face to form a manifold mesh using a heuristic.

Our stitching heuristic is as follow. Let e_i and e_j be two open edges lying on the same face f in the cubical grid, and P_i and P_j be two patches responsible for generating e_i and e_j, respectively. Note that e_i and e_j can be generated in the same cube or in different cubes that share face f. We stitch e_i and e_j if and only if $e_i \neq e_j$ and P_i and P_j are connected via a sequence of zero of more *black patches* $P_{b_1}, \ldots, P_{b_m}$ where each P_{b_k} intersects a cube that shares f. If $e_i \neq e_j$ and $P_i = P_j$, then we simply do not consider stitching e_i and e_j, since

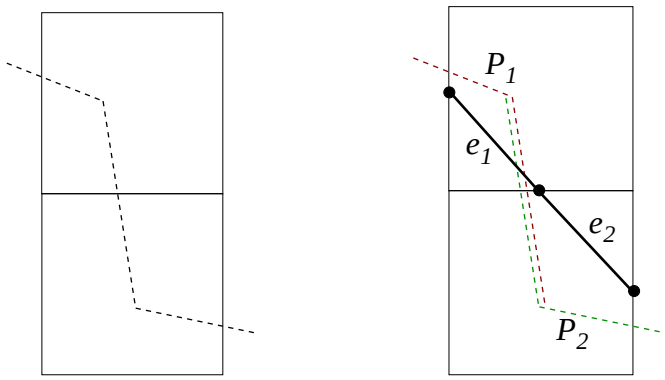


Figure 4.9: **2D stitching example 1.** (a) The original contour is depicted by dashed line segments. (b) Since patches P_1 and P_2, coloured as red and green respectively, are connected, we connect e_1 and e_2 at the common edge of the squares.

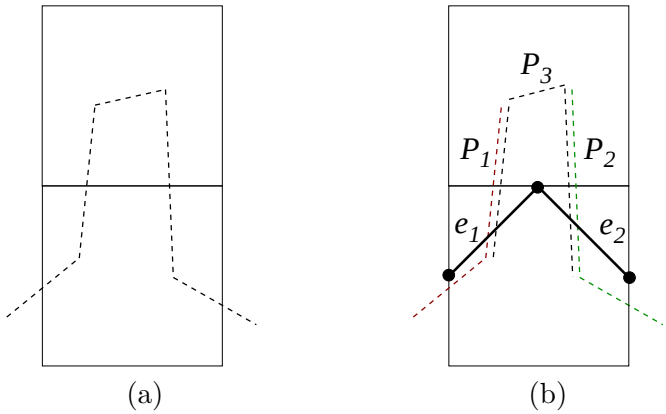


Figure 4.10: **2D stitching example 2.** (a) The original contour is depicted by dashed line segments. (b) Since patches P_1 and P_2, coloured as red and green respectively, are connected to a black patch P_3, coloured as black, we connect e_1 and e_2 together at the common edge of the two squares.

having P_i and P_j be the same patch implies that e_i and e_j will always be stitched based on the stitching rule. In this case, we simply ignore the edge pair and allow other pairs of open edges on face f to stitch up any holes that may be introduced on face f. An example of this can be generated from triangulating case 3 in Figure 4.1.

To perform the stitching procedure, we process each face one at a time with no particular ordering. In our work, a scan-line ordering along the three coordinate axes is chosen. When there are two or more open edges lying on a cube face, we repeatedly attempt to stitch up pairs of edges based on the stitching rule. After two edges are stitched, they become closed edges and are no longer in consideration. We continue to pair up open edges until no more stitching can be performed. As examples to show our stitching heuristic at work, refer to Figure 4.8, 4.9, 4.10, and 4.11.

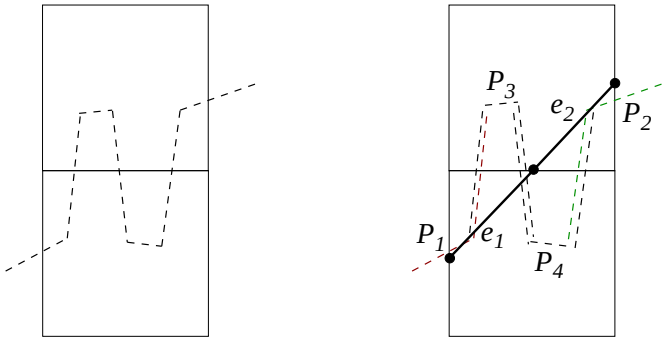


Figure 4.11: **2D stitching example 3.** (a) The original contour is depicted by dashed line segments. (b) Since patches P_1 and P_2, coloured as red and green respectively, are connected via a sequence of black patches, namely P_3 and P_4, we connect e_1 and e_2 together at the common edge of the two squares.

In Figure 4.9 and 4.10, patch P_1 is connected to P_2 via zero and one black patch respectively, thus according to the stitching rule, we connect the two edges e_1 and e_2. Let us revisit Figure 4.8. According to the stitching rule, the two red edges lying on the common face will not be considered to be stitched if they are generated from the same patch, similarly, the same situation applies for the two green edges. However, stitching occurs when considering pairing up a red edge with a green edge. As a result, the hole on the common face is filled up. In Figure 4.11, we show a 2D example of where two patches P_1 and P_2 are connected via a sequence of two black patches associated with the squares sharing the common edge.

Considering rotations and symmetries, we can list six basic stitching configurations that

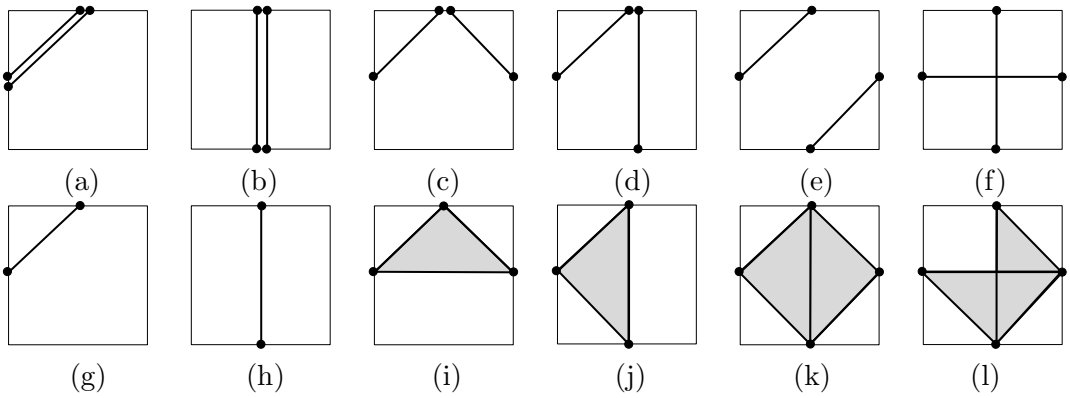


Figure 4.12: **Stitching cases.** Each square represents a face of a cube where the edges inside the square are open edges lying on the face. Top row: Six possible stitching cases. Bottom row: Corresponding manipulation of the stitching cases. Note that case (f) only occurs on self-intersecting meshes.

may occur on a face of a cube. The six cases and the corresponding manipulation of the cases are depicted in Figure 4.12. Note that the displacement of the vertices and edges in the figure are for illustration purposes only. In reality, the vertices lie on the midpoint of the cube edges. When more than three open edges are needed to be stitched, these basic cases will be recursively applied.

By making use of the connectivity between patches, we are able to determine when edges on a face of a cube should be stitched up. The result is a manifold mesh M_0 that satisfies the angle constraints and roughly approximates the original input mesh M. Although we do not provide a correctness proof for the surface reconstruction algorithm at this point, experimental results show that the algorithm works quite well in practice even in a coarse sampling grid, where nonmanifold edges and vertices have not been found.

4.3 Experimental Results

We demonstrate the effectiveness of our initial mesh construction algorithm through experiments. In Table 4.1, we show various statistics about the initial well-shaped nonobtuse meshes that are generated. All input mesh models used in the experiment are manifold meshes with no self-intersections and are remeshed using grid size of 100^3, 70^3, 40^3, and 10^3. The remeshed models are shown in Figure 4.13, 4.14, 4.15, 4.16, respectively. Experiments are conducted using a Sun Ultra 40 workstation that is equipped with two dual core

processors running at 2.4GHz, 8GB of RAM, and an ATI X1900 XTX video card.

Consider column B_{out} in Table 4.1, no new boundaries are created for five of the mesh models. However, different tests on the bunny model reported different number of boundaries compared with the input mesh. This is because the sampling of the cubical grid is not sufficiently fine in certain areas near the boundaries. Hence, disconnected components with boundaries are created during the mesh construction. This problem can be overcome with fine resolution of the cubical grids. In most situations, disconnected components occur near thin boundaries of a surface. In the case of the bunny model, this is shown in Figure 4.17. The mesh is also disconnected for the horse and the knot model remeshed at a grid size of 10^3. In both meshes, two triangles with normals pointing at opposite directions are stitched together. This introduces a valence-2 vertex in the meshes as shown under column VR in Table 4.1. All other valence-2 vertices reported are located on the open boundaries of their respective output meshes. It is also worth noting that the initial mesh models generated have no vertices with valence ≥ 10; see column VR in Table 4.1.

In general, topology preservation for a surface reconstruction algorithm is difficult to achieve. For example, an input mesh model with thin structures may be remeshed into several disconnected components if the cubical grid is not sampled sufficiently fine. Similar to the original MC algorithm, we do not guarantee topology. This can be easily seen in the remeshed models at low resolution as shown in Figure 4.16. Since the later stages of our algorithm do not change the topology of the output mesh, it is important that the initial mesh produced in this stage preserve the topology of the input mesh. If the resolution of the sampling grid is sufficiently fine, we would expect our algorithm to produce results with the same topology as the input mesh.

We have not address the issue of choosing a proper grid size during initial mesh construction. Currently, the grid size is chosen based on the number of triangles produced in the initial mesh. The cubes in a coarse sampled grid intersect more triangles than cubes in finer grids. Therefore, more patches intersect the cubes. This implies that more stitching needs to be performed on each cube face. In contrast, finer grids generate more triangles in the initial mesh but with fewer stitching needs to be performed. With that said, a better approach is to choose a grid size, where each cube in the grid has at most one patch intersecting it. This will lead to simpler stitching procedure on the cube faces as there are less patches to be considered. However, this may increase the number of triangles produced. An extension to use an adaptive grid would be desirable, in which fine resolution of the grid

can be sampled at areas of interest. This will help reduce the triangle count of the initial mesh.

For all the six mesh models (open and closed) that we have tested in our experiments, nonmanifold edges and vertices were not introduced for all four grid sizes as shown under column NM of Table 4.1. This confirms that our stitching rule works quite well in practice. As pointed out in Section 4.2.2, patches that give sign conflicts can only come from meshes with boundaries or with self-intersection. Our results again confirm this as shown under column PI in Table 4.1.

In the current implementation, speed has not been optimized. As shown in Table 4.1, the remeshing with grid size of 10^3 requires significantly longer time to compute. This is mainly due to the increased number of triangles in a patch, thus, it takes more computation to check patch connectivities between two open edges lying on the common face. As the grid size increases, the number of patches within a cube decreases, so the number of patch connectivity checks within each face of a cube are reduced. However, the increase in grid size increases the number of cube faces. This results in an increased number of stitching, thus, the time required to construct the initial mesh is increased as well.

Finally, we confirm that the initial mesh models generated from our experiments satisfy the angle constraints that we have proposed.

Input ($\triangle_{in}$, $\mathbf{B}_{in}$)	$\triangle_{out}$	$\mathbf{B}_{out}$	**NM**	**MP**	**PI**	**SF**	**VR**	**AR**	**t**
Arm (50K, 0)	12940	0	0	25	0	10756	[4, 8]	[35, 90]	31s
Arm (50K, 0)	7368	0	0	19	0	6020	[4, 8]	[35, 90]	18s
Arm (50K, 0)	2098	0	0	17	0	1694	[4, 9]	[35, 90]	8s
Arm (50K, 0)	78	0	0	6	0	69	[4, 8]	[35, 90]	49s
Bunny (70K, 5)	50100	6	0	50	224	38632	[2, 9]	[35, 90]	57s
Bunny (70K, 5)	22348	6	0	48	153	17118	[2, 8]	[35, 90]	25s
Bunny (70K, 5)	8083	6	0	58	93	6072	[2, 8]	[35, 90]	17s
Bunny (70K, 5)	474	1	0	23	22	346	[2, 9]	[35, 90]	31s
Horse (40K, 0)	31400	0	0	44	0	24669	[3, 8]	[35, 90]	33s
Horse (40K, 0)	16088	0	0	53	0	12546	[3, 8]	[35, 90]	21s
Horse (40K, 0)	5312	0	0	40	0	4114	[3, 8]	[35, 90]	10s
Horse (40K, 0)	232	0	0	13	0	117	[2, 8]	[35, 90]	33s
Knot (10K, 0)	101852	0	0	64	0	76519	[4, 8]	[35, 90]	79s
Knot (10K, 0)	41486	0	0	66	0	31006	[4, 8]	[35, 90]	22s
Knot (10K, 0)	16488	0	0	72	0	12036	[4, 9]	[35, 90]	5s
Knot (10K, 0)	488	0	0	44	0	461	[2, 8]	[35, 90]	0s
Roll (2.8K, 1)	34776	1	0	0	852	34352	[2, 6]	[35, 90]	35s
Roll (2.8K, 1)	19176	1	0	0	636	18860	[2, 6]	[35, 90]	17s
Roll (2.8K, 1)	6480	1	0	94	462	6298	[2, 6]	[35, 90]	4s
Roll (2.8K, 1)	280	1	0	0	1	45	[2, 6]	[35, 90]	1s
Max P. (398K, 1)	52006	1	0	33	51	40592	[2, 8]	[35, 90]	114s
Max P. (398K, 1)	21116	1	0	49	68	16410	[2, 8]	[35, 90]	94s
Max P. (398K, 1)	6944	1	0	26	43	5378	[2, 8]	[35, 90]	88s
Max P. (398K, 1)	384	1	0	10	7	284	[4, 8]	[35, 90]	2251s

Table 4.1: **Initial mesh statistics.** For each model, we test for four grid sizes: 100^3, 70^3, 40^3, and 10^3 in that order using a Sun Ultra 40 workstation that is equipped with two dual core processors running at 2.4GHz, 8GB of RAM, and an ATI X1900 XTX video card. $\triangle_{in}$: number of triangles in the input mesh; $\mathbf{B}_{in}$: number of boundaries in the input mesh; $\triangle_{out}$: number of triangles in the output mesh; $\mathbf{B}_{out}$: number of boundaries in the output mesh; **NM**: number of nonmanifold edges and vertices; **MP**: number of cubes containing multiple patches; **SF**: number of cube faces needed stitching; **PI**: number of patches ignored due to conflicting sign assignments; **VR**: range of vertex valences; **AR**: range of face angles; **t**: time of the algorithm in seconds.

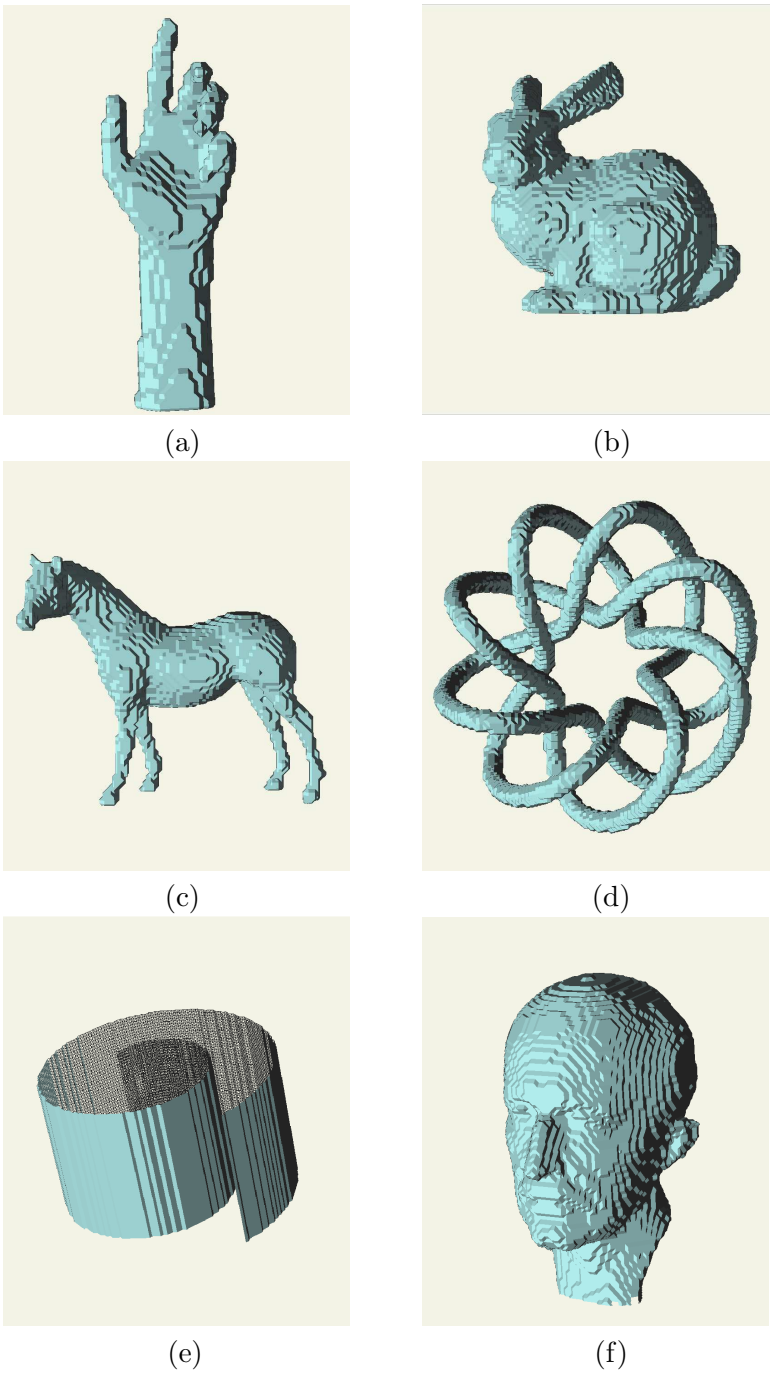
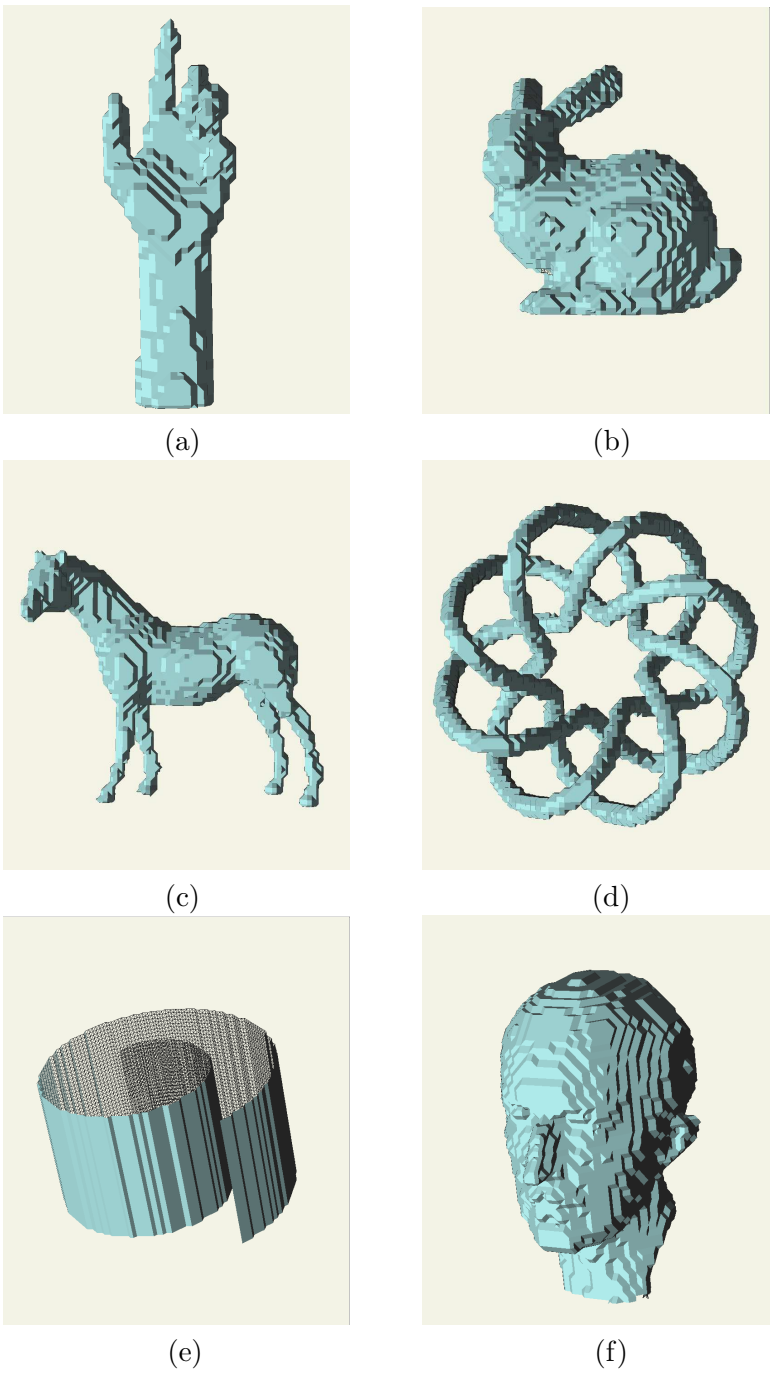


Figure 4.13: **Initial mesh construction results 1.** All meshes shown here are constructed with a grid size of 100^3. (a) Arm model. (b) Bunny model. (c) Horse model. (d) Knot model. (e) Roll model. (f) Max Planck model.



(a) (b) (c) (d) (e) (f)

Figure 4.14: **Initial mesh construction results 2.** All meshes shown here are constructed with a grid size of 70^3. (a) Arm model. (b) Bunny model. (c) Horse model. (d) Knot model. (e) Roll model. (f) Max Planck model.

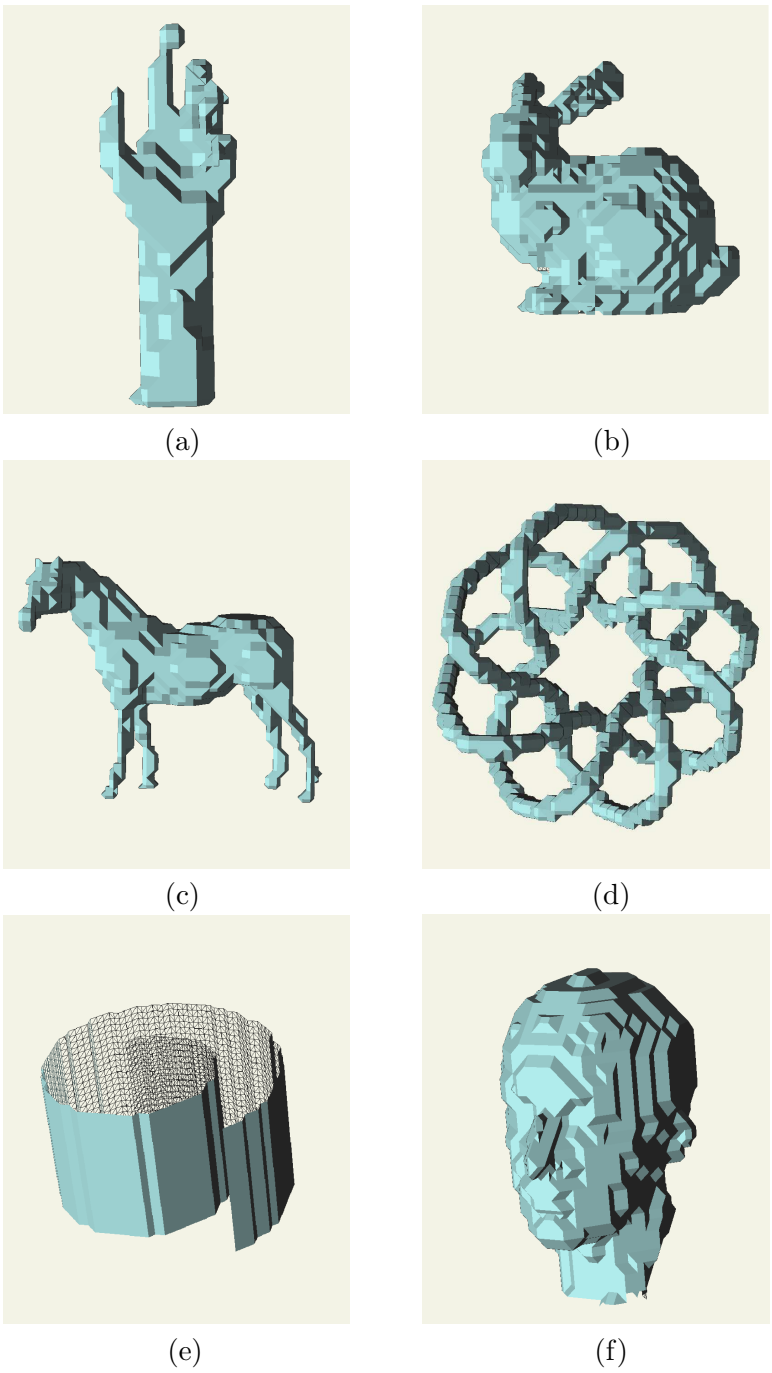


Figure 4.15: **Initial mesh construction results 3.** All meshes shown here are constructed with a grid size of 40^3. (a) Arm model. (b) Bunny model. (c) Horse model. (d) Knot model. (e) Roll model. (f) Max Planck model.

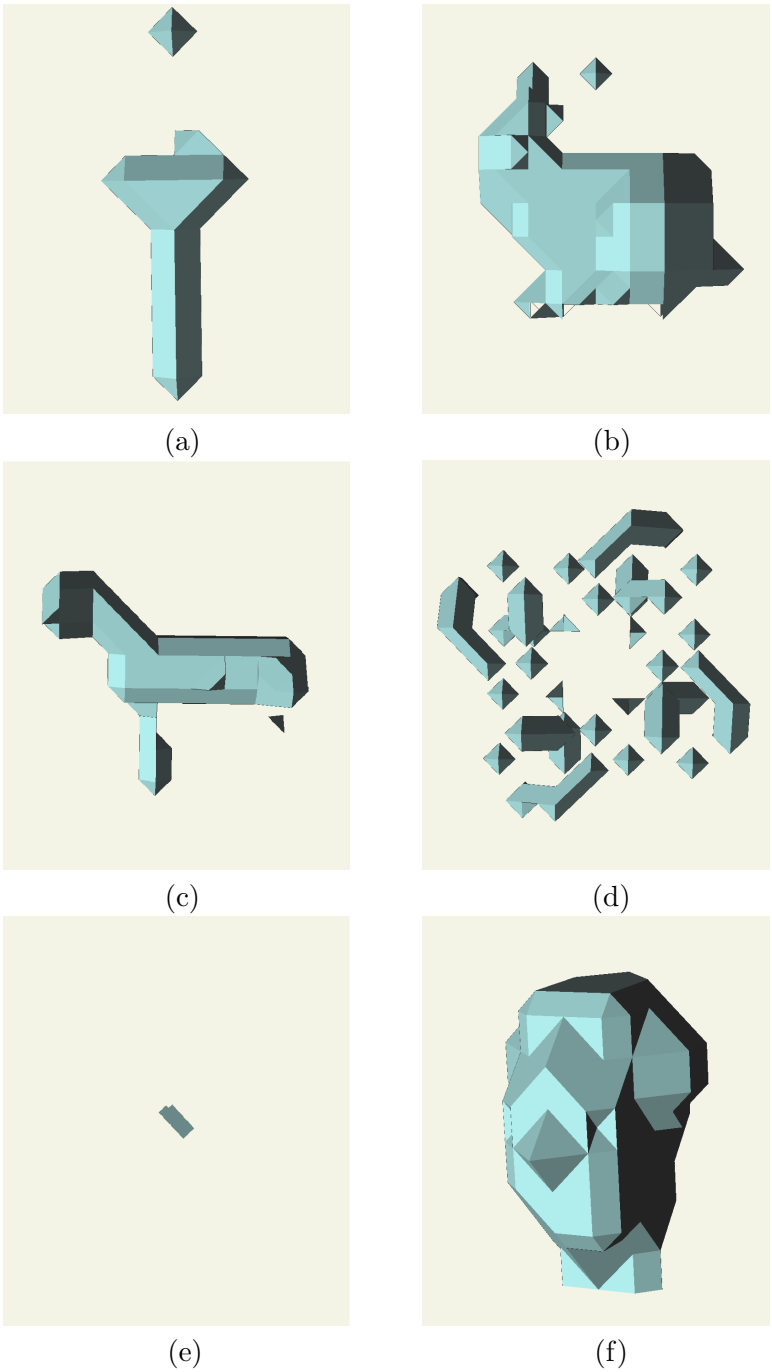
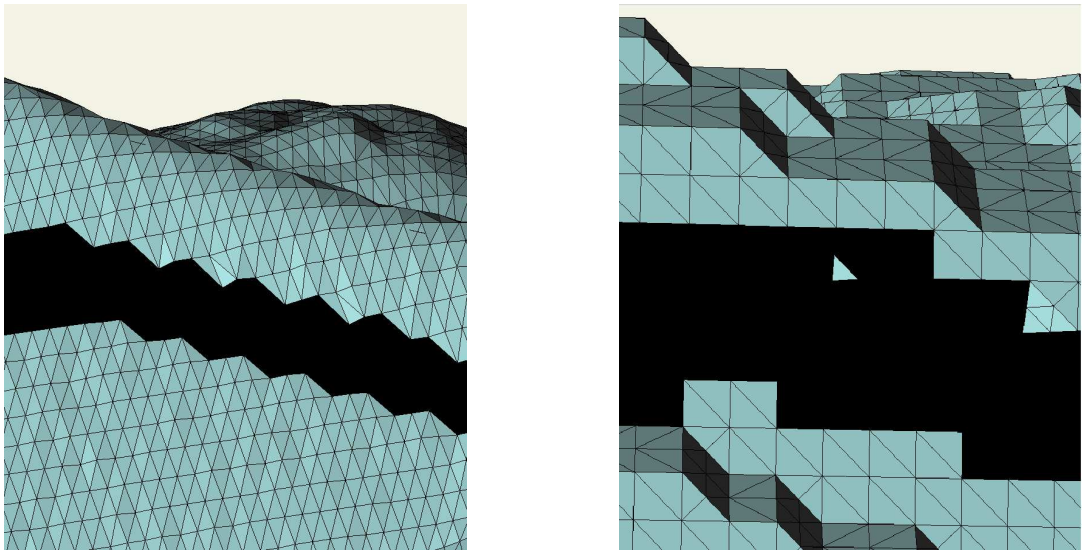


Figure 4.16: **Initial mesh construction results 4.** All meshes shown here are constructed with a grid size of 10^3. At such coarse resolution, topology may not be preserved and this may result in generating disconnected components in the output mesh. (a) Arm model. (b) Bunny model. (c) Horse model. (d) Knot model. (e) Roll model. (f) Max Planck model.



(a) (b)

Figure 4.17: **Disconnected components.** (a) The original bunny model. (b) The remeshed bunny model at grid size of 100^3. When the cubical grid is not sampled sufficiently fine, the output mesh may have disconnected components. This is shown here where a triangle is disconnected from the bunny. Hence, Table 4.1 reports 6 boundaries in the output mesh. The 5 boundaries of the original mesh have all been correctly recovered.

Chapter 5

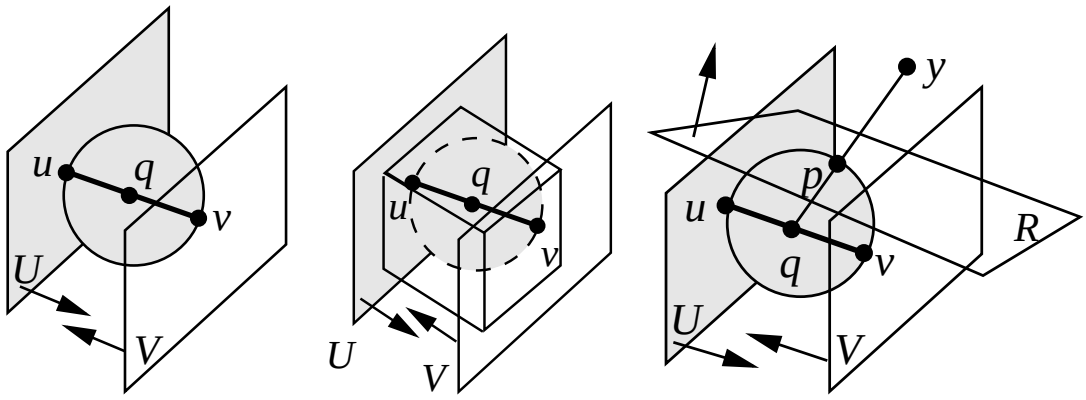
Optimization with Angle Bounds

The initial mesh M_0, produced by the modified MC algorithm in the last section, provides a rough approximation of the input mesh M. This initial mesh is guaranteed to be well-shaped and nonobtuse, hence, it gives us an initialization for our optimization as we deform M_0 to approximate M.

We iteratively deform M_0 so as to obtain progressively better approximation to M. This is accomplished through a heuristic search, within the space of well-shaped nonobtuse meshes having the same connectivity as M_0, for solving a global optimization problem. In order to ensure our deformed mesh $\hat{M}$ is both well-shaped and nonobtuse, we define a set of linear constraints so that at each iteration of the optimization, our deformed mesh $\hat{M}$ stays well-shaped and nonobtuse. The objective function is a point-wise least-square measure, combining approximation error and mesh smoothness quality. In the following sections, we describe these in details in addition to several heuristics for improving the overall performance of the optimization and the overall quality of the deformed mesh $\hat{M}$.

5.1 Region of Well-shapedness and Nonobtusity (RWSNO)

Region of nonobtusity for a mesh edge: Given an edge $e = (u, v)$, we define its *region of nonobtusity* (RNO) $\mathcal{N}(e)$ as the set of points p which would make the triangle $\triangle puv$ nonobtuse. Let U and V be the planes, with normals facing each other, orthogonal to edge e and intersecting vertices u and v respectively. And let $sphere(e)$ be the sphere of diameter



(a) (b) (c)

Figure 5.1: **Region of nonobtusity (RNO) and its linearization and convexification.** (a) Region of nonobtusity (RNO) for edge (u, v). (b) Linearized RNO for edge (u, v); the sphere is replaced by a cube. (c) Feasible RNO for edge (u, v), with respect to the center vertex y, is a convex region delimited by planes U, V, and R.

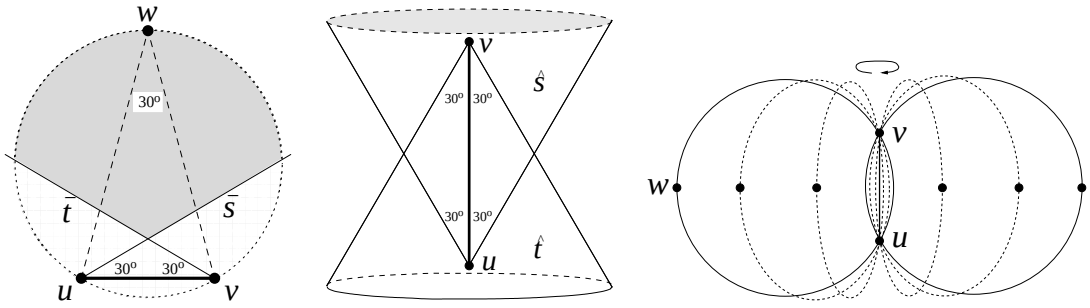
$|e|$ centered at the midpoint q of e. Then it is not hard to see that

$$\mathcal{N}(e) = sandw(e) - sphere(e) \tag{5.1}$$

where $sandw(e)$ is the intersection of the front half-spaces of planes U and V. This is illustrated in Figure 5.1(a).

Region of well-shapedness (RWS) for a mesh edge: Given an edge $e = (u, v)$, we define its *region of well-shapedness* (RWS) $\mathcal{W}(e)$ as the set of points p which would make the triangle $\triangle puv$ well-shaped. For simplicity, let us start off considering the planar case. Refer to Figure 5.2(a), we define $\bar{s}$ and $\bar{t}$ to be the lines that intersect vertices u and v respectively. Moreover, the lines $\bar{s}$ and $\bar{t}$ span a 30° angle with edge e at its respective intersection. Let w be a point such that $\angle uwv = 30°$, then we define $circle(w, u, v)$ to be the circumcircle that intersects points w, u, and v. For all points p in the interior of $circle(w, u, v)$ and on the opposite side of e along $\bar{s}$ and $\bar{t}$, the triangle $\triangle puv$ is well-shaped. One side of the RWS is depicted in Figure 5.2(a).

With this idea in mind, we can then generalize the well-shaped region for a mesh edge in 3D space. Lines $\bar{s}$ and $\bar{t}$ are generalized as 3D lines revolving around edge e resulting in two cone-shaped regions $\hat{s}$ and $\hat{t}$ as illustrated in Figure 5.2(b). Similarly, $circle(w, u, v)$ is



(a) (b) (c)

Figure 5.2: **Region of well-shapedness (RWS).** (a) One side of the RWS for an edge (u, v) on a plane. (b) Cones $\hat{s}$ and $\hat{t}$ are generalization of $\bar{s}$ and $\bar{t}$ in 3D. (c) $plum(w, u, v)$ region is the generalization of w in 3D.

generalized as a plum-shaped region $plum(w, u, v)$ constructed by revolving $circle(w, u, v)$ along edge e. This is illustrated in Figure 5.2(c). Therefore, the RWS $\mathcal{W}(e)$ of edge e in 3D is defined as

$$\mathcal{W}(e) = plum(w, u, v) - \hat{s} - \hat{t} \tag{5.2}$$

Region of well-shapedness and nonobtusity (RWSNO) for an interior mesh vertex: Given an interior vertex v of a mesh and its one-ring vertices $v_0, v_1, \ldots, v_{h-1}$ in order, we define the *region of well-shapedness and nonobtusity* (RWSNO) $\mathcal{WN}(v)$ of v to be the set of points p such that triangles $\triangle pv_iv_{i+1 \ mod \ h}$, $\forall i = 0, \ldots, h-1$, are well-shaped and nonobtuse. Geometrically, this is the intersection of all RWS and RNO of edges $(v_i, v_{i+1 \ mod \ h})$, $\forall i = 0, \ldots, h-1$. Hence,

$$\mathcal{WN}(v) = (\bigcap_{i=0}^{h-1} \mathcal{N}[(v_i, v_{i+1 \ mod \ h})]) \cap (\bigcap_{i=0}^{h-1} \mathcal{W}[(v_i, v_{i+1 \ mod \ h})]) \tag{5.3}$$

Region of well-shapedness and nonobtusity (RWSNO) for a boundary mesh vertex: Given a boundary vertex v of a mesh and its one-ring vertices $v_0, v_1, \ldots, v_{h-1}$ in order, the *region of well-shapedness and nonobtusity* (RWSNO) $\mathcal{WN}(v)$ of v is the set of points p such that triangles $\triangle pv_iv_{i+1}$, $\forall i = 0, \ldots, h-2$, are well-shaped and nonobtuse. Hence,

$$\mathcal{WN}(v) = (\bigcap_{i=0}^{h-2} \mathcal{N}[(v_i, v_{i+1})]) \cap (\bigcap_{i=0}^{h-2} \mathcal{W}[(v_i, v_{i+1})]) \tag{5.4}$$

It should be obvious that in general the RWSNO of a vertex is nonlinear and nonconvex. To make the constrained optimization problem feasible to solve, we linearize and convexify the search space, which should then result in a search space that is a subset of $\mathcal{WN}(v)$ per vertex. The linearization is the process of representing the search space using a set of linear constraints and convexification is the process of approximating the search space using a subset of the search space that is convex. A simple way to linearize the constraints is to model the search space with a set of planes. For convexification, we will need to consider the region of interest.

5.2 Linearization and Convexification of RWSNO

Linearizing and convexifying RNO: Consider the RNO $\mathcal{N}(e)$ for edge $e = (u, v)$. The linearization can be achieved by replacing $sphere(e)$ with a set of four planes that are tangent to $sphere(e)$. This is illustrated in Figure 5.1(b). However, the search space is still nonconvex, therefore, we conservatively replace the set of four planes with just one plane R. Specifically, let y be a vertex that is incident to both u and v. And let q be the midpoint of edge e and p be the intersection between the line segment $\overline{yq}$ and $sphere(e)$. Then, plane R is defined as the plane through p and tangent to $sphere(e)$. Hence, we have linearized and convexified $\mathcal{N}(e)$, with respect to y, to $\mathcal{N}_y^*(e)$, the set of points intersected by $sandw(e)$ and the front half-space of R. Refer to Figure 5.1(c) for an illustration. We call $\mathcal{N}_y^*(e)$ the feasible RNO of e with respect to y.

Linearizing and convexifying RWS: In order to linearize and convexify the RWS $\mathcal{W}(e)$ for edge $e = (u, v)$, let us consider linearizing and convexifying the constrained region depicted by the cones $\hat{s}$ and $\hat{t}$ first. Similar to the approach taken for the RNO case, we replace cones $\hat{s}$ and $\hat{t}$ with planes S and T. In detail, let Y_0 be the plane that intersects u, v, and y where y is incident to u and v. Then we define S and T to be the planes orthogonal to Y_0 and tangent to the two cones $\hat{s}$ and $\hat{t}$ respectively.

To linearize and convexify the constrained region $plum(w, u, v)$, let us consider the region $drum(w_0, u, v)$, the intersection of $plum(w_0, u, v)$ and $sandw(e)$, as illustrated in Figure 5.3. Note that w_0 is the point lying on plane Y_0 such that $\angle uw_0v = 30°$. The presence of constraints U, V, and R enables us to focus only on a subset of $plum(w_0, u, v)$, namely, the

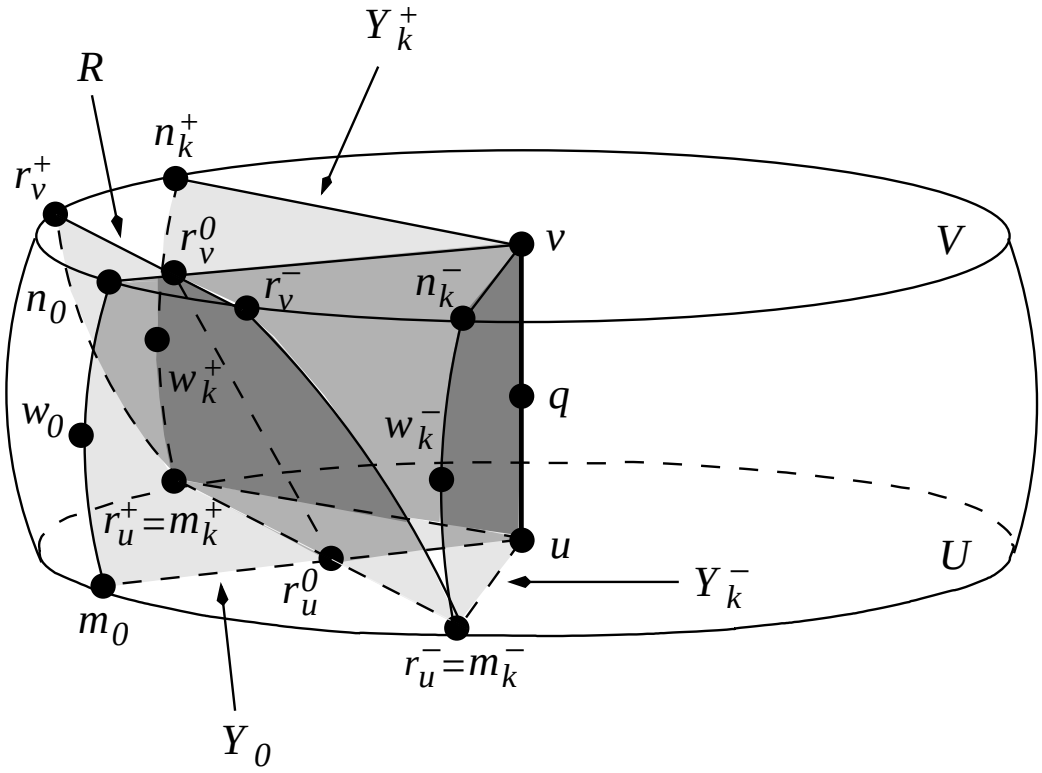


Figure 5.3: **The drum region.** Intersection of U, V, and $plum(w_0, u, v)$. The search space is reduced significantly by examining the region intersected by U, V, and R.

intersection of $drum(w_0, u, v)$ and the front half-space of R. Let r_u^+ and r_u^- be the intersecting point of R, U, and the boundary of $plum(w_0, u, v)$, and let r_u^0 be the midpoint between them. Similarly, we define r_v^+, r_v^-, and r_v^0 in the same way but with the occurrence of U replaced with V. Let r_c^0 denote the point from the set $\{r_u^0, r_v^0\}$ that is closest to edge e. Then let Y_k^+ and Y_k^- be the planes through u, v that intersect r_c^+ and r_c^- respectively. Now consider the region spanned by rotating Y_0 along e in both, counterclockwise and clockwise, directions until it reaches Y_k^+ and Y_k^-. Essentially, our goal is to use a set of planes to cover this spanned region. We do so by using planes M_i^+, N_i^+, M_i^-, N_i^-, $\forall i = 1, \ldots, k$. We define the planes as follows. Let us suppose we slice the two regions spanned from Y_0 to Y_k^+ and from Y_0 to Y_k^- into k equal sections. We name the $k-1$ slices as planes Y_i^+ and Y_i^-, $\forall i = 1, \ldots, k-1$, respectively. Then, we define m_i^+ to be the intersecting point of U, the boundary of $plum(w_0, u, v)$, and Y_i^+, $\forall i = 0, \ldots, k$. Equivalently, we define m_i^-, n_i^+ and n_i^- in the same way but with the occurrence of U replaced with V and $+$ with $-$ accordingly. Also, $\forall i = 0, \ldots, k$, let w_i^+ and w_i^- be the points on planes Y_i^+ and Y_i^- such that it is lying half way along the arc from m_i^+ to n_i^+ and from m_i^- to n_i^- respectively. Then, M_i^+ is the plane that intersects m_{i-1}^+, m_i^+, and w_i^+. N_i^+, M_i^-, and N_i^- are defined analogously with the occurrence of m replaced with n and $+$ with $-$ accordingly. This is illustrated in Figure 5.4.

Linearizing and convexifying RWSNO: Hence, we can then summarize the linearized and convexified version of the RWSNO, $\mathcal{WN}_y^*(e)$, of an edge e with respect to a vertex y as follows. For simplicity, we will refer to $\mathcal{WN}_y^*(e)$ as the *feasible region* $\mathcal{F}_y(e)$ of e with respect to y from now on.

$$\mathcal{F}_y(e) = \mathcal{WN}_y^*(e) = \mathcal{N}_y^*(e) \cap S \cap T \cap (\bigcap_{i=1}^{k-1} M_i^+ \cap N_i^+ \cap M_i^- \cap N_i^-) \tag{5.5}$$

Feasible region for a vertex: Given an interior mesh vertex v and its one-ring vertices $v_0, v_1, \ldots, v_{h-1}$ in order, the *feasible region* $\mathcal{F}(v)$ is defined as

$$\mathcal{F}(v) = \bigcap_{i=0}^{h-1} \mathcal{F}_v[(v_i, v_{i+1 \ mod \ h})] \tag{5.6}$$

Similarly for a boundary mesh vertex v,

$$\mathcal{F}(v) = \bigcap_{i=0}^{h-2} \mathcal{F}_v[(v_i, v_{i+1})] \tag{5.7}$$

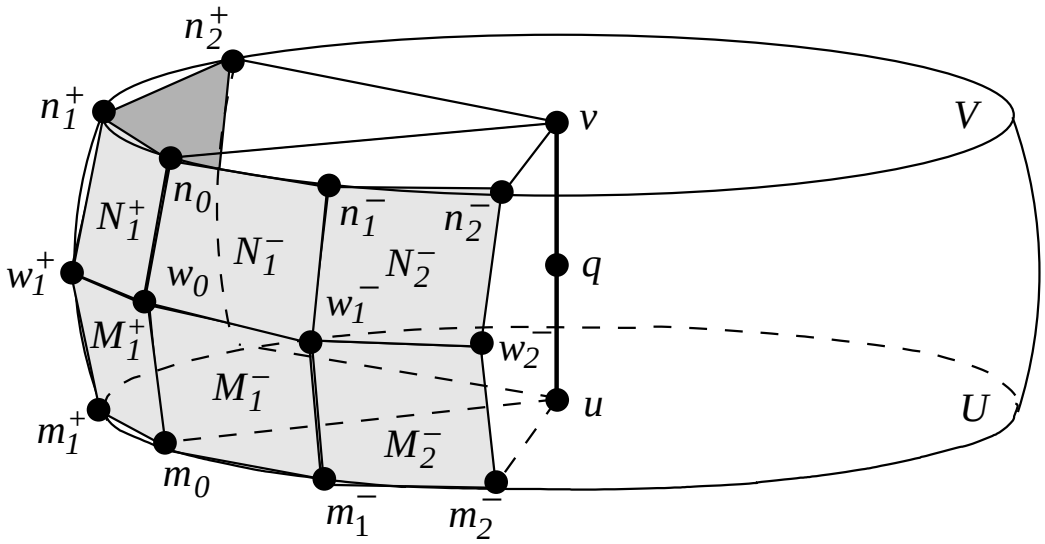


Figure 5.4: **Linearizing and convexifying** $plum(w_0, u, v)$**.** The planes M_i^+, N_i^+, M_i^-, and N_i^-, $\forall i = 1, \ldots, k$, are used to approximate the constraint depicted by $plum(w_0, u, v)$, U, and V. In our experiments, we have chosen $k = 2$.

5.3 Objective Function

The global optimization problem for well-shaped nonobtuse remeshing seeks to find a mesh $\hat{M}$, within the space $\mathcal{WN}(M_0)$ of well-shaped nonobtuse meshes having the same connectivity as the initial mesh M_0, which minimizes a point-wise least-square measure. Formally, we wish to

$$minimize \sum_{v \in V(\hat{M})} \mathcal{L}(v, M), \ subject\ to\ \hat{M} \in \mathcal{WN}(M_0), \tag{5.8}$$

where $V(\hat{M})$ is the set of vertices in $\hat{M}$ and

$$\mathcal{L}(v, M) = \alpha \cdot \mathcal{D}(v, M) + (1 - \alpha) \cdot \mathcal{S}(v), \tag{5.9}$$

where $\mathcal{D}(v, M)$ is a distance measure for approximating the error of v with respect to M, $\mathcal{S}(v)$ is a smoothness (regularization) term, and $0 \leq \alpha \leq 1$ is a user-defined parameter that controls the trade-off between error reduction and smoothness.

Quadric error: We associate a quadric Q_v with each interior vertex v in $\hat{M}$. Introduced by Garland and Heckbert [24] in 1997, the quadric $Q_v(x)$ captures information about a set

of planes associated with vertex v and any point x in 3D space. Specifically, $Q_v(x)$ is the sum of squared distances from point x to the set of k planes associated with v. Written in quadratic form,

$$\mathcal{Q}_v(x) = \sum_{i=0}^{k-1} (n_i^T x + d_i)^2 = \sum_{i=0}^{k-1} x^T (n_i n_i^T) x + 2 d_i n_i^T x + d_i^2 \tag{5.10}$$

where n_i is the unit length normal of plane i and d is the distance of plane i from the origin. We define

$$\mathcal{D}(v, M) = Q_v(v)/k, \ for\ v \in Int(\hat{M}) \tag{5.11}$$

where $Int(\hat{M})$ is the set of interior vertices of $\hat{M}$. $\mathcal{D}(v, M)$ gives an *average* squared distances from the interior vertex v to a set of k supporting planes corresponding to the triangle T of M that is closest to v, as well as to the r-ring neighbour triangles of T in M. Initially, we locate the triangle T efficiently utilizing the patches used during the construction of the initial mesh M_0. The parameter r is user-defined and in our experiments, we set $r = 1$. The success of quadric error metric in the past has led us to believe it is a suitable metric for measuring the point-to-surface distances. Not only did it approximate errors well in mesh decimation, the additive property of quadrics provides a fast and simple way to allow for angle-bounded decimation as we describe in Chapter 6.

The distance measure $\mathcal{D}(v, M)$ for a boundary vertex v is defined differently. The objective is to ensure the boundary is preserved as close as possible during the optimization. We achieve this by minimizing the point-to-boundary distances for all boundary vertices v in $\hat{M}$. Thus, we define

$$\mathcal{D}(v, M) = ||v - b||^2, \ for\ v \notin Int(\hat{M}) \tag{5.12}$$

where b is the closest point to v that is on a boundary of M.

Smoothness term: At an interior vertex v, the smoothness term

$$\mathcal{S}(v) = ||v - \mathcal{C}(v)||^2, \ for\ v \in Int(\hat{M}) \tag{5.13}$$

measures the squared distance between v and the centroid $\mathcal{C}(v)$ of its one-ring neighbours. In the case of boundary vertices,

$$\mathcal{S}(v) = ||v - \hat{\mathcal{C}}(v)||^2, \ for\ v \notin Int(\hat{M}) \tag{5.14}$$

where $\hat{\mathcal{C}}(v)$ is the midpoint of the two boundary neighbour vertices of v. Incorporating the smoothness term tends to produce a better distribution of points across the meshes.

5.4 Optimization via Heuristic "Deform-to-Fit"

The solution to problem (5.8) is likely intractable, we thus resort to heuristics. Our strategy is an iterative approach in which vertices are moved in a constrained greedy fashion that optimizes $\mathcal{L}(v, M)$. As mesh vertices are moved to better approximate the original mesh M, their associate distance measure $\mathcal{D}(v, M)$ is updated accordingly. Relying on a priority queue, the iterative process enhances $\hat{M}$ to a local minimum. When the optimization ends, the deformed mesh $\hat{M}$ may have rough surfaces. This is due to the fact that the optimization is highly localized. We resolve this issue by applying constrained Laplacian smoothing as we discuss in Section 5.5.

Originally, the vertices on the initial mesh M_0 are within their respective, linear and convex, feasible regions. Hence, we are assured that $\mathcal{F}(v)$, for all $v \in V(\hat{M})$, is nonempty. The idea is then to move vertices on $\hat{M}$ such that their new locations are still within their respective feasible region. Under this constraint, an optimal location v^*

$$v^* = \text{argmin}_{v \in \mathcal{F}(v)}[\mathcal{L}(v, M)] \tag{5.15}$$

is computed for every vertex $v \in V(\hat{M})$ such that it minimizes $\mathcal{L}(v, M)$. Since our objective function $\mathcal{L}(v, M)$ is defined in quadratic form, we can formulate the computation of v^* as a *quadratic programming* problem. In our experiments, we have utilized the OOQP solver of Gertz and Wright [25].

The priority $\mathcal{H}(v)$ for vertex v is given by the *improvement* made by moving v to v^*:

$$\mathcal{H}(v) = \mathcal{L}(v, M) - \mathcal{L}(v^*, M) \tag{5.16}$$

At each iteration, the vertex with the highest priority is moved to its optimal location. Afterwards, all vertices influenced by the move will be re-inserted back into the priority queue, since their priorities and associated optimal locations need to be updated due to changes in their distance measure $\mathcal{D}(v, M)$, smoothness term $\mathcal{S}(v)$, or feasible region $\mathcal{F}(v)$.

In order to update the distance measure $\mathcal{D}(v, M)$ for an interior vertex v, the associated quadric Q_v will need to be recomputed. To recompute the quadric Q_v for an interior vertex v, we need to find a triangle T' in M that is closest to v. We simply execute a local search starting from the previous closest triangle T. This heuristic has worked quite well in practice, with a local search confined to within a 3-ring neighbourhood of T. Updating the distance measure for boundary vertices are carried out analogously. We perform a similar

local search to find the closest boundary point b' starting from the previous closest boundary point b and along the boundary edges of M. After updating the closest boundary point b', we can then compute $\mathcal{D}(v, M)$ for the boundary vertex as defined in equation 5.12.

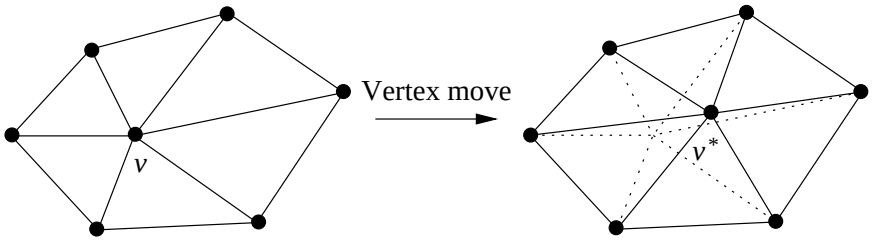


Figure 5.5: **Updating the optimal location after each vertex move.** Following the vertex move from v to v^* at each iteration, the feasible regions for the one-ring neighbours of v have changed, thus, their optimal locations need to be recomputed as well.

Following each vertex movement in the optimization, only the vertex v that has just moved requires an update to $\mathcal{D}(v, M)$. However, since v has moved to a new location, then for all one-ring vertices $v_0, v_1, \ldots, v_{k-1}$ of v, the smoothness term $\mathcal{S}(v_i), \forall i = 0, \ldots, k-1$, will need to be updated as well. But more importantly, the feasible region $\mathcal{F}(v_i), \forall i = 0, \ldots, k-1$, has changed after the movement of v, hence, to ensure the constraints are not violated, the optimal locations of each v_i, along with v, will need to be recomputed.

We adopt a crude stopping criterion for the iterative constrained optimization. The optimization stops when no vertex can be moved, within its feasible region, to reduce the objective function $\mathcal{L}$ further; $\mathcal{H}(v) \leq 0, \forall v \in V(\hat{M})$. To speed up the optimization, we have implemented lazy evaluation and early smoothing in our work. The details are discussed in Section 5.6.

5.5 Constrained Laplacian Smoothing

Experimentally, the output meshes produced by the optimization procedure described so far are already quite satisfactory in terms of approximation error, but they may have rough surfaces. This may be attributed to the fact that our optimization scheme is highly localized. A vertex v may get into a bad situation in which its one-ring vertices have been moved in such a way that leaves no room for v to improve upon. To this end, we suggest

to follow the mesh optimization step with a Laplacian smoothing step, which encourages *tangential* movement of the vertices to smooth out the roughness. This enables vertices near the rough region of the surface to be set "free" so that further optimization allows these vertices to have a larger feasible region to act upon. Note that such tangential movements may increase the approximation error. Thus we can interleave mesh optimization with constrained Laplacian smoothing, with the former being the first and the last steps of our "deform-to-fit" procedure. Constrained Laplacian smoothing is performed in similar fashion as the mesh optimization procedure described in the last section. We also process vertices one at a time, but without any particular order. Each vertex v processed is moved to its optimal location v^* within v's feasible region; the objective function is simply defined as

$$\mathcal{L}(v, M) = \mathcal{S}(v) \tag{5.17}$$

5.6 Performance Speed-Up

In order to improve upon the performance of the optimization step, we have implemented *lazy evaluation* [18] and *early smoothing* in our experiments.

Lazy evaluation: During each iteration of the constrained optimization, a vertex v is chosen to be relocated at an optimal location that improves the approximation error of the mesh. Vertex v is chosen such that $\mathcal{H}(v)$ is the largest among all other vertices. We utilize a priority queue in aid of this. When a vertex is moved, the cost $\mathcal{H}$ for the one-ring vertices and itself will need to be recomputed. To speed up the overall optimization procedure, we apply lazy evaluation in which only the vertex that has been moved will have its cost updated. All other affected vertices will be marked as dirty. When a dirty vertex is at the top of the priority queue, then its cost is updated and the vertex is reinserted into the priority queue. If no vertices can be moved and there are dirty vertices in the priority queue, then a full update to the cost of all dirty vertices is taken. The overall "deform-to-fit" optimization ends when the optimization stops with no dirty vertices in the priority queue. Since we need to solve a quadratic program for each cost update, the overall speed of the algorithm can be much improved due to the less frequent cost updates offered by the lazy evaluation paradigm. The vertex movement at each iteration may not have the highest improvement

value $\mathcal{H}(v)$, hence, the optimization has a slower improvement rate. However, this is compensated by the increase in the overall speed of the "deform-to-fit" optimization.

Early smoothing: Recall that we interleave the mesh optimization step with the constrained Laplacian smoothing step. Since lazy evaluation is applied in our implementation, full updates of the cost of all dirty vertices may need to be carried out several times. To speed this up, we perform constrained Laplacian smoothing early on when a full update to the cost of all dirty vertices is first needed. The reason is that the cost of all vertices will need to be recomputed after the smoothing step, hence, we carry out smoothing early on to reduce the number of full updates in the overall "deform-to-fit" procedure. Each required full update is an indication that less number of vertices are being moved in the optimization. This implies that the improvement gained at each iteration is decreasing. By performing smoothing step at the point when the first full update is required, the difference of error reduction between performing smoothing step earlier and later will not be significantly large.

5.7 Removal of Bad Valence Vertices

To further improve upon the local smoothness of the mesh $\hat{M}$, we attempt to remove all bad valence vertices before the optimization begins. We define bad valence vertices as those that disrupt local smoothness on a planar surface due to the constraint that all vertices must lie within their RWSNO. In the case of interior vertices, we declare all interior vertices v as bad if the valence of v is < 5 or > 11. This is because in flat regions of a mesh, it is impossible for all angles surrounding such vertices to be well-shaped and nonobtuse. Vertices with valence 4 or 12 may have all its surrounding angles satisfying the angle constraints, but the allowable movement of the vertices are limited. In addition, all boundary vertices are bad if their valence is < 4.

It can be shown that the initial mesh M_0 generated by the procedures discussed in Chapter 4 has no vertices of valence > 11. In fact, the maximum valence of a vertex in the initial mesh is 10. This can be seen by examining all the cases illustrated in Figure 4.1 and Figure 4.2 that the highest valence of a vertex introduced on an edge of a cube is 3 and the highest valence of a vertex introduced inside the cube is 7. Since the edges of a cube are shared by 4 cubes, then by taking into account of double counting the edges that are shared by two adjacent cubes, the highest valence of a vertex that can be produced by the initial

mesh construction algorithm is 8. Note that the stitching step discussed in Section 4.2.3 increases the valence of a vertex by at most 2. Therefore, the maximum valence of a vertex in M_0 is 10.

We have derived simple heuristics, based on local mesh manipulation techniques, to remove bad valence vertices. These heuristics do not provide a guaranteed removal of all bad valence vertices. However in practice, the removal of such bad valence vertices helps improve the local smoothness of a mesh. We first attempt to remove all boundary vertices that are of valence 2 and all interior vertices that are of valence 3. It should be obvious that valence-3 interior vertices will give a rough surface near planar regions. Similarly, boundary vertices of valence 2 diminish the local smoothness of a boundary. Upon the removal of such vertices, we proceed with the removal of valence-4 interior vertices, followed by, the removal of valence-3 boundary vertices. Refer to Figure 5.6 for illustration of the removal process.

Removal of valence-2 boundary vertices: To remove valence-2 boundary vertex v as shown in Figure 5.6(a), we simply discard v and its adjacent triangle. To do so, we have to first ensure the removal of such vertex v will not introduce any new valence-2 boundary vertices. This can be achieved by checking whether the valence of the two adjacent vertices of v is ≥ 4. This verification step is needed in order to avoid continuous shrinking of the boundary as illustrated in Figure 5.7.

Removal of valence-3 interior vertices: Similar to the procedure described above, we discard valence-3 interior vertex v and its adjacent triangles. After such removal, we triangulate the one-ring of v by adding a new triangle. Such removal can only be carried out if the valence of the one-ring vertices v_i, $\forall i = 0, \ldots, k-1$, of v is ≥ 6 (or ≥ 4 if $v_i \notin Int(\hat{M})$). An illustration is given in Figure 5.6(b). Notice that we may introduce valence-3 boundary vertices. The preference of valence-3 boundary vertices over valence-3 interior vertices is because valence-3 interior vertices introduce a larger error on near planar region than valence-3 boundary vertices do.

Removal of valence-4 interior vertices: We attempt to remove a valence-4 interior vertex v in two ways. First, we try to discard v and retriangulate the underlying quadrilateral as shown in Figure 5.6(c). Vertex v can be discarded only if there exist two adjacent vertices v_i, v_j, with valence ≥ 6 (or ≥ 5 if $v_i, v_j \notin Int(\hat{M})$), such that they lie on diagonally

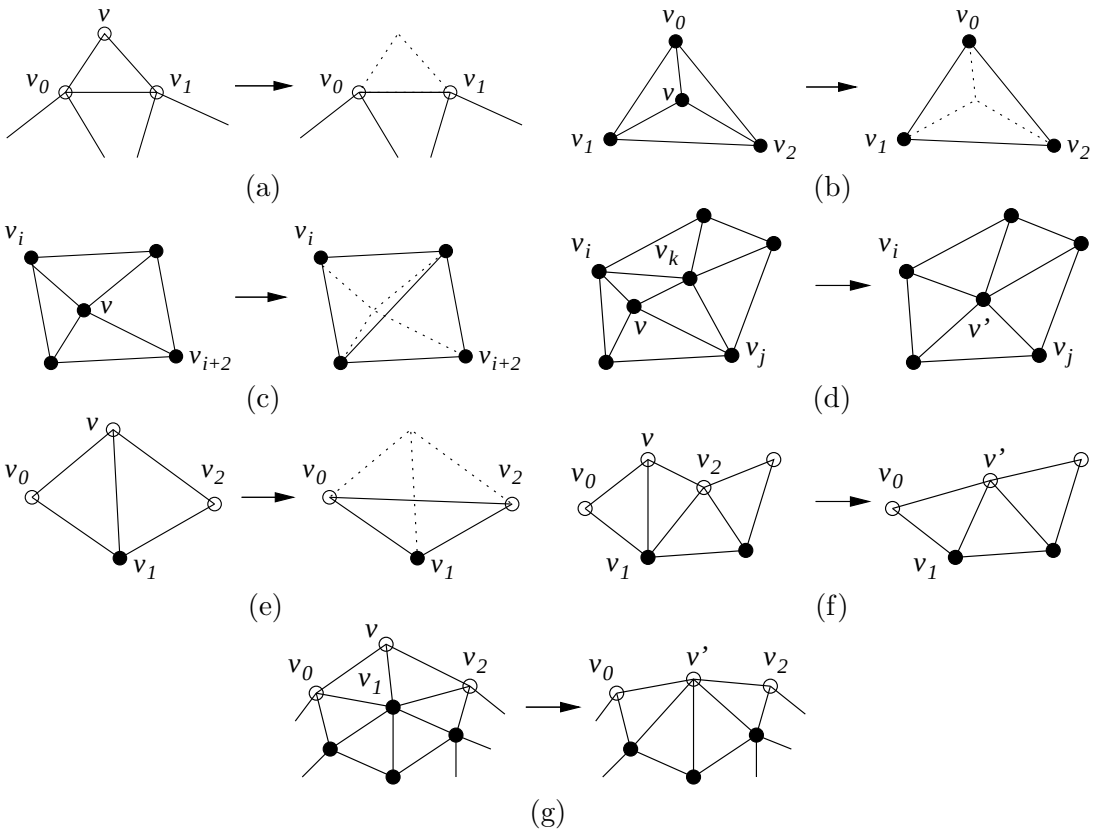


Figure 5.6: **Removing bad valence vertices.** Black vertices denote interior vertices, white vertices denote boundary vertices, and $Val(v)$ denote the valence of vertex v. We list the valence requirements of nearby vertices here.
(a) Removal of v if $Val(v_0) \geq 4$, $Val(v_1) \geq 4$
(b) Removal of v and triangulate if $Val(v_0) \geq 6$, $Val(v_1) \geq 6$, $Val(v_2) \geq 6$
(c) Removal of v and triangulate if $Val(v_i) \geq 6$, $Val(v_{i+2}) \geq 6$
(d) Removal of v via edge collapsing (v, v_k) if $Val(v_i) \geq 6$, $Val(v_j) \geq 6$, $Val(v_k) \geq 5$
(e) Removal of v and triangulate if $Val(v_1) \geq 6$
(f) Removal of v via edge collapsing (v, v_2) if $Val(v_1) \geq 6$, $Val(v_2) \geq 3$
(g) Removal of v via edge collapsing (v, v_1) if $Val(v_0) \geq 5$, $Val(v_1) \geq 6$, $Val(v_2) \geq 5$

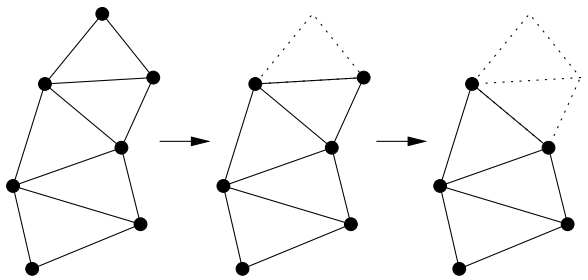


Figure 5.7: **Continuous shrinking of the boundary.** To avoid continuous shrinking of the boundary, removal of valence-2 boundary vertices only occur when the adjacent vertices are of valence ≥ 4.

opposite of each other in the underlying quadrilateral. The underlying quadrilateral can be triangulated only if a split along the diagonal of the quadrilateral will not violate any angle constraints. Notice that this is not always achievable. Otherwise, we attempt to perform edge collapse on one of the adjacent edges of v. We collapse edge $e = (v, v_i)$ to an unified vertex v' only if the valence of v_i is ≥ 5 (or ≥ 4 if $v_i \notin Int(\hat{M})$), and the valence of the two vertices v_j, v_k, that are adjacent to both v and v_i, are ≥ 6 (or ≥ 5 if $v_j, v_k \notin Int(\hat{M})$). The location of the unified vertex v' can be computed by solving the quadratic programming problem discussed in Section 5.4 with the constraints constructed by the one-ring vertices of edge e and

$$\mathcal{L}(v, M) = ||v - q||^2 \tag{5.18}$$

where q is the midpoint of e. An example of the edge collapse is illustrated in Figure 5.6(d)

Removal of valence-3 boundary vertices: The removal of valence-3 boundary vertices is similar to the method taken for the removal of valence-4 interior vertices. We first try the discard-and-retriangulate method, then follow by edge collapse if the former was unsuccessful as shown in Figure 5.6(e) and Figure 5.6(f) respectively. Let v_0, v_1, v_2 be the three adjacent vertices of v listed in order where v_0, v_2 are on the boundary. Vertex v can be discarded only if v_1 has valence ≥ 6 (or ≥ 5 if $v \notin Int(\hat{M})$). And v_0, v_1, v_2 can be triangulated only if $\triangle v_0 v_1 v_2$ is well-shaped and nonobtuse. If all else fails, we attempt to perform edge collapse on one of the adjacent edges of v. We collapse edge (v, v_0) to an unified vertex v' only if the valence of v_0 is ≥ 3 and the valence of v_1 is ≥ 6 (or ≥ 5 if $v \notin Int(\hat{M})$). The same rules

apply to edge (v, v_2). As for edge (v, v_1), we collapse it only if $v_1 \in Int(\hat{M})$, the valence of v_0, v_2 are ≥ 5, and the valence of v_1 is ≥ 6. This is illustrated in Figure 5.6(g).

Chapter 6

Decimation with Angle Bounds

Once a high-resolution well-shaped nonobtuse mesh $\hat{M}$ is obtained, we can proceed with angle-bounded decimation to reduce the size of $\hat{M}$ to produce a hierarchy of well-shaped nonobtuse meshes. Our decimation process follows the work done by Garland and Herbert [24] on quadric error metric in which a cost, computed based on quadric error of the adjacent vertices, is assigned to each edge of the mesh. The ordering of edges to collapse is based on the cost of the edges. In the original work of Garland and Herbert [24], no constraints are present. In contrast, our focus is to ensure our decimated mesh is both well-shaped and nonobtuse. Hence, we formulate our decimation process analogous to the remeshing case in which a constrained local optimization problem is solved in an iterative greedy fashion. We iteratively perform edge collapses, starting from a full-resolution well-shaped nonobtuse mesh, where the position of the unified vertex can be computed similarly with a quadratic program as in the case of remeshing.

6.1 Angle Constraints

When an edge $e = (u, v)$ is collapsed, the position of the unified vertex w needs to be computed. Essentially, the computation of the location for vertex w is seen similarly as in the case for computing the location of vertex v^* as shown in Section 5.4. Therefore, let $w_0, w_1, \ldots, w_{h-1}$ be the one-ring vertices, listed in order, surrounding edge $e = (u, v)$ where $u, v \in Int(\hat{M})$ and $Int(\hat{M})$ denotes the set of interior vertices of mesh M, we define the constraints for collapsing edge e to be the feasible region $\mathcal{F}(u, v)$ determined similarly as in

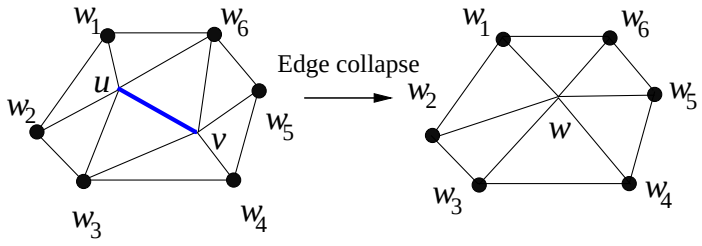


Figure 6.1: **Angle-bounded edge collapse.** As we collapse edge (u, v) into a new vertex w, neighboring vertices of (u, v), marked by dark dots, form a one-ring which define the feasible region when computing an optimal position of w.

the vertex case by the neighboring vertices of e,

$$\mathcal{F}(u,v) = \bigcap_{i=0}^{h-1} \mathcal{F}_q[(w_i, w_{i+1 \ mod \ h})], \tag{6.1}$$

where q, the center point used to define the feasible regions, is taken to be the midpoint of edge e. Similarly, if either $u \notin Int(\hat{M})$ or $v \notin Int(\hat{M})$, then

$$\mathcal{F}(u,v) = \bigcap_{i=0}^{h-2} \mathcal{F}_q[(w_i, w_{i+1})]. \tag{6.2}$$

6.2 Objective Function

As in the quadric-based mesh decimation algorithm of Garland and Heckbert [24], we use the sum of squared distances from a point to a set of planes as an error measure for our decimation algorithm. We associate a quadric Q_w with each mesh vertex w prior to the decimation. For all vertices $w \in Int(\hat{M})$, Q_w is defined as in equation 5.10 in which the set of planes associated with w is the set of supporting planes for the triangles incident to w. And for all vertices $w \notin Int(\hat{M})$, the quadric for w is simply defined as the squared L_2 distance to itself,

$$Q_w(x) = ||x - w||^2 \tag{6.3}$$

Note that other choices for the quadric at a boundary vertex are possible, but we have found the above to work quite well in practice.

6.3 Iterative Edge Collapse

With the constraints and objective function defined, the edge collapse operation can be formulated as follows. For edge $e = (u, v)$ we wish to collapse, we compute the optimal position w^* for the unified vertex w. The position w^* should minimize the quadric error associated with edge e. Specifically,

$$w^* = \text{argmin}_{w \in \mathcal{F}(u,v)}[Q_u(w) + Q_v(w)] \tag{6.4}$$

Therefore to perform greedy angle-bounded decimation over mesh $\hat{M}$, we place all candidate edges to collapse in a priority queue and compute the optimal position w^* for each edge. We then associate a cost to each edge which will then be used as a sorting key for the priority queue. The cost of an edge reflects the amount of error introduced when the edge is collapsed. Thus, the cost $Cost(e)$ for collapsing edge $e = (u, v)$ is defined as:

$$Cost(e) = Q_u(w^*) + Q_v(w^*) \tag{6.5}$$

If $\mathcal{F}(u, v)$ is empty, we set the cost to be ∞. A candidate edge at the top of the priority queue, having the *lowest* cost, is collapsed first and the resulting new edges are re-inserted into the queue. Upon collapsing edge (u, v) to w, we assign the new vertex w with the quadric $Q_w = Q_u + Q_v$.

6.4 Lazy Evaluation

Lazy evaluation has worked quite successfully in our mesh optimization stage and in principle, it is applicable to mesh decimation as well. In fact, it was the use of the lazy queuing algorithm of [18] for mesh decimation that has inspired our adoption of this speed-up scheme. However, as we argue below and also confirmed by our experiments, lazy evaluation does not work so effectively for our angle-bounded mesh decimation.

For mesh optimization, the existence of dirty vertices can cause the deformed mesh to improve upon its approximation to the original mesh at a slower rate, as the vertex being deformed at each iteration may not be the vertex that has the highest improvement value $\mathcal{H}(v)$. However, due to the much fewer updates carried out, lazy evaluation more than compensates this slower improvement rate with a speed-up in processing times. The overall optimization process is significantly faster without noticeable degradation in approximation quality.

Now consider applying lazy evaluation to our angle-bounded decimation. Initially, the edges are sorted according to their associated costs. As edges are collapsed, a number of edges are marked as dirty. Thus, the collapsed edge at each iteration may not be the edge that has the actual lowest cost. Consequently, at each iteration, we introduce a larger error than it needed to be. The more dirty edges there are, the faster the mesh quality is expected to degrade with each edge collapse, compared to the case where no lazy evaluation is applied. At the same time, having more dirty edges increases the chance of having them appear at the top of the priority queue, which would call for updates to be done immediately; this tends to increase the number of updates carried out.

It turns out that the number of dirty edges for which the associate quadrics or feasible regions have changed is much greater than the number of dirty vertices introduced by a vertex movement in the optimization stage. Consider Figure 6.2, it should be obvious that the quadrics for the set of new edges, introduced during the edge collapse, need to be updated. However, since the collapsed edge is unified into a vertex w, the feasible region of any edge whose set of one-ring vertices contains w has changed. All the edges that need to be marked dirty with the edge collapse $(u, v) \rightarrow w$ are marked by dark lines on the right in Figure 6.2; there are 29 of them. In contrast, the number of dirty vertices introduced by moving w is only 6. Finally, in our case, an edge collapse can be expected to introduce significant changes to the feasible regions of the relevant edges, which can lead to large discrepancies between the actual and un-updated cost at a dirty edge. In the case of unconstrained mesh decimation, e.g., when the quadric error is used, the corresponding discrepancy is expected to be smaller.

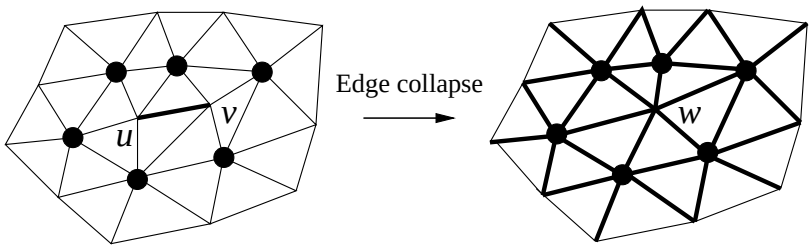


Figure 6.2: **Dirty edges after an edge collapse.** After edge (u, v) is collapsed to a unified vertex w, all edges, marked by dark lines, that are incident to w or to the neighbors of w, are marked as dirty in lazy evaluation.

6.5 Early Termination

Unlike the original quadric-based decimation paper [24], the decimation that we carried out is angle-bounded. Hence, the location of the unified vertex w may not be the position that minimizes the error. As the mesh is decimated, the degradation of the mesh quality becomes apparent. Therefore to avoid the decimated mesh from introducing any large errors, we stop the iterative edge collapse process when an error is above a certain user-specified threshold. The threshold will be set relative to the size of the input mesh. As is typically done, we choose the length of the bounding box diagonal of the input mesh as a reference and define the threshold to be a percentage of that length.

Chapter 7

Experimental Results

In this section, we use several experiments to demonstrate the effectiveness and different characteristics of our optimization framework and the quality of the well-shaped nonobtuse meshes produced. All experiments have been conducted using a Sun Ultra 40 workstation that is equipped with two dual core processors running at 2.4GHz, 8GB of RAM, and an ATI X1900 XTX video card. Unless otherwise specified, all the results given in this section are generated with $\alpha = 0.5$, where we recall that α is the user-defined parameter that controls the trade-off between error reduction and smoothness of the mesh produced.

Analysis of the linearization and convexification of RWSNO: During each iteration of the constrained optimization, a vertex in the mesh $\hat{M}$ is relocated to an optimal location that is within its RWSNO. The RWSNO is linearized and convexified into a feasible region $\mathcal{F}(v)$. We model this using a set of linear equations. This set of linear equations are constructed based on the geometry of the vertex one-ring. For each one-ring edge, there are $5 + 4k$ number of linear constraints where k is the number of subdivisions used to convexify the RWSNO as discussed in Section 5.2. Given that our initial mesh M_0 has a maximum valence of 10, the maximum number of linear constraints used to model $\mathcal{F}(v)$ is at most $10(5 + 4k) = 50 + 40k$. We have chosen $k = 2$ in our work as it appears to work quite well throughout our experiments. We have been conservative in modeling the RWSNO. It is possible for the RWSNO to be modelled with exact precision using a set of quadratic constraints. However, solving a quadratic programming problem with quadratic constraints in each iteration would be too expensive. The convexification of the RWSNO plays an important role in the optimization as well. With nonconvex constraints, the optimization

(a) (b) (c)

Figure 7.1: **Arm.** (a) Remeshed arm model with 9287 vertices: $\epsilon = 0.9864\%$. (b) After 50% of vertices decimated: $\epsilon = 0.0816\%$. (c) After 90% of vertices decimated: $\epsilon = 0.5895\%$.

for each vertex will need to be carried out more than once in order to find the optimal location within the nonconvex region. Hence, we have chosen to use at most $50 + 40k$ linear constraints per vertex.

Remeshing and decimation of well-shaped nonobtuse meshes: In Figure 7.1−7.6, we show the six well-shaped nonobtuse meshes that have been generated in our experiments. The approximation error, ϵ, which is the Hausdorff distance returned by the Metro tool [17], is given as a percentage of the bounding box diagonal. This error measure is commonly used to evaluate the difference between two aligned triangular meshes, e.g., for mesh decimation and compression [3, 24, 31]. The errors are measured against the original mesh for remeshing and against the full-resolution well-shaped nonobtuse mesh in the case of decimation. As can be observed both via visual examination and numerical results reported, our algorithm is capable of producing high-quality well-shaped nonobtuse meshes in terms of approximation error. A close-up screenshot of the triangulation in the interior and on the boundary of two meshes are given in Figure 7.7. As shown in the figure, local smoothness across the mesh surface and boundaries is maintained. Moreover, the mesh vertices are distributed evenly.

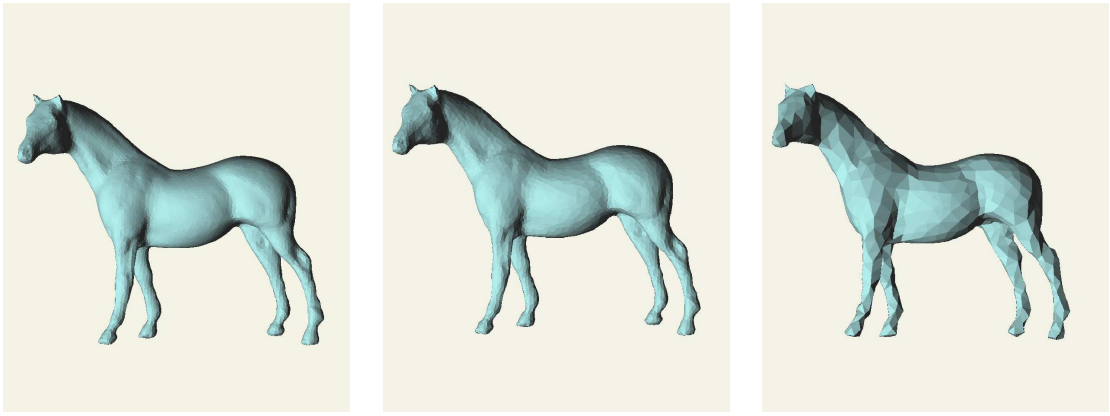
Correctness: Although we do not ensure that our mesh optimization step will not produce

(a) (b) (c)

Figure 7.2: **Bunny.** (a) Remeshed bunny model with 16154 vertices: $\epsilon = 1.1186\%$. (b) After 50% of vertices decimated: $\epsilon = 0.2115\%$. (c) After 90% of vertices decimated: $\epsilon = 0.7776\%$.

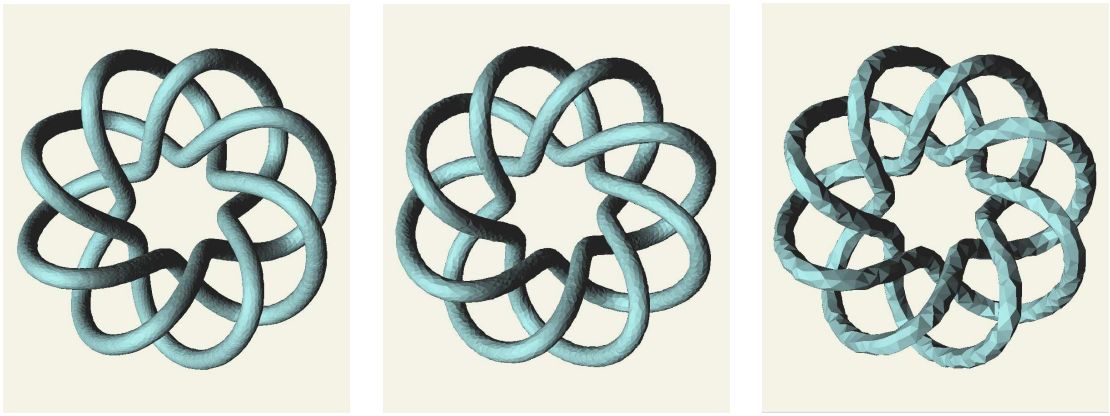
self-intersections and other unwanted artifacts on the output mesh surface, our experimental results show that the output meshes that our algorithm produced do not generate any high curvature peaks. In Figure 7.8, we show the mean curvature plot of the well-shaped nonobtuse meshes that we produced in our experiments. As it can be seen that there are no undesired high curvature values across the mesh surface. This confirms that no undesired artifacts occur in all our results. Furthermore, the curvature plots closely resemble the original mesh. Notice that the meshes our algorithm produces are generally smoother than the original mesh. This is due to both the smoothness term in the objective function and the constrained Laplacian smoothing step.

Importance of the smoothing step: Consider the meshes shown in Figure 7.9. Clearly, the local smoothness of the surface for both meshes are significantly different. This example shows the importance of the constrained Laplacian smoothing step in the overall "deform-to-fit" optimization. The iterative constrained optimization moves vertices in a greedy fashion that optimizes the distance from the moved vertex to the original surface. Since this is performed in a local manner, vertices may reach a local minimum where its feasible region limit the amount of movement the vertices can make in the future. The role of the smoothing step is then to encourage tangential movement of the vertices to smooth out roughness on the surface and at the same time allow vertices that are "stuck" to be moved away. These experiments show that the constrained Laplacian smoothing steps can



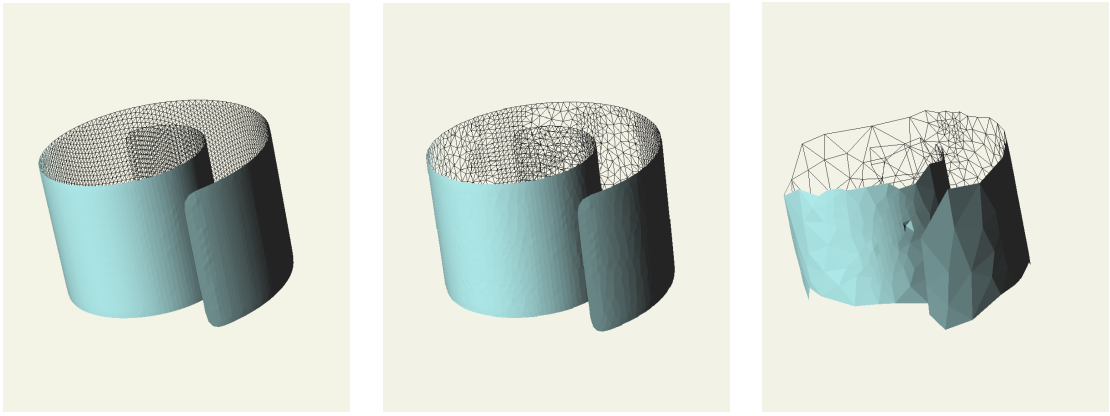
(a) (b) (c)

Figure 7.3: **Horse.** (a) Remeshed horse model with 9941 vertices: $\epsilon = 1.1480\%$. (b) After 50% of vertices decimated: $\epsilon = 0.1365\%$. (c) After 90% of vertices decimated: $\epsilon = 2.2533\%$.



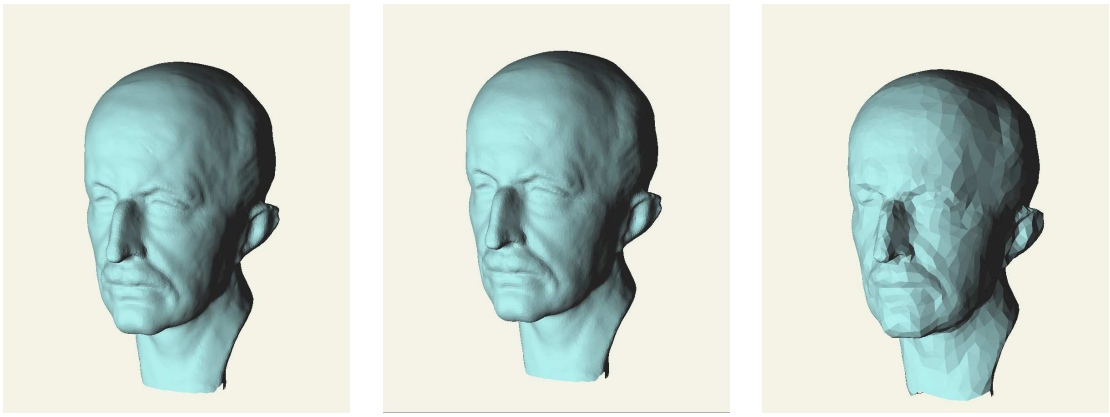
(a) (b) (c)

Figure 7.4: **Knot.** (a) Remeshed knot model with 20819 vertices: $\epsilon = 0.2357\%$. (b) After 50% of vertices decimated: $\epsilon = 0.1362\%$. (c) After 90% of vertices decimated: $\epsilon = 0.7358\%$.



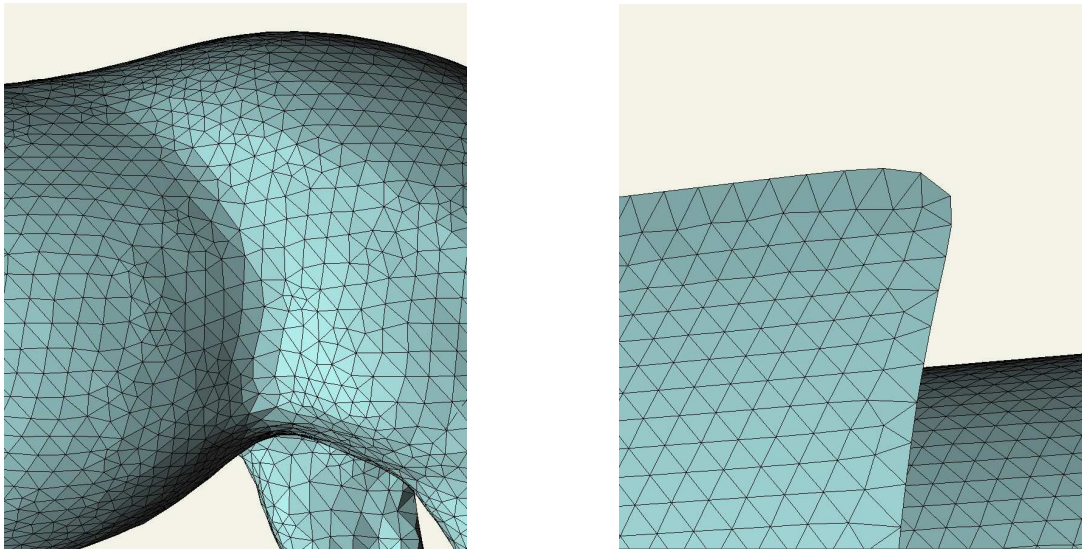
(a) (b) (c)

Figure 7.5: **Roll.** (a) Remeshed roll model with 6446 vertices: $\epsilon = 1.6992\%$. (b) After 50% of vertices decimated: $\epsilon = 0.1419\%$. (c) After 90% of vertices decimated: $\epsilon = 7.9848\%$. Notice that there is a dent on the surface of the mesh in (c). This is caused by a valence-4 vertex introduced during the edge-collapse. The user can specify a valence threshold so that the decimation would not produce any vertices with valence lower than the threshold. During the decimation, an edge will not be collapsed if it will introduce a vertex with valence lower than the parameter specified by the user. All results produced in this chapter are generated with valence-2 vertices avoided.



(a) (b) (c)

Figure 7.6: **Max Planck.** (a) Remeshed Max Planck model with 35588 vertices: $\epsilon = 0.7440\%$. (b) After 50% of vertices decimated: $\epsilon = 0.0273\%$. (c) After 90% of vertices decimated: $\epsilon = 1.5127\%$.



(a) (b)

Figure 7.7: **Result close-up.** (a) Interior surface of the horse model shown in Figure 7.3(a). (b) Boundary of the roll model shown in Figure 7.5(a).

be quite important in producing high-quality nonobtuse remeshing.

Effectiveness of the removal of bad valence vertices: In Table 7.1, we show statistics regarding the removal of bad valence vertices prior to the optimization. Notice that our simple heuristic works quite well in practice in removing a large portion of the bad valence vertices. The reason that not all bad valence vertices were removed is because these vertices are connected to a low valence vertex. In this case, removing the bad valence vertex will introduce a new bad valence vertex. Consider the horse model as an example. There is a valence-3 vertex connected to two valence-4 vertex. Based on the heuristic that we proposed, we cannot remove them by simple removal or edge collapses. Perhaps a better approach is to perform vertex split so that valences of these bad valence vertices will increase. All other bad valence vertices that remain in the mesh lie on the boundary. This may create a boundary that is not smooth, especially when a valence-2 vertex is on the boundary. In this case, vertex split will probably help resolve this as well.

Significance of lazy evaluation and early smoothing: We show the difference in performance of applying lazy evaluation and early smoothing in Table 7.2. As indicated in the

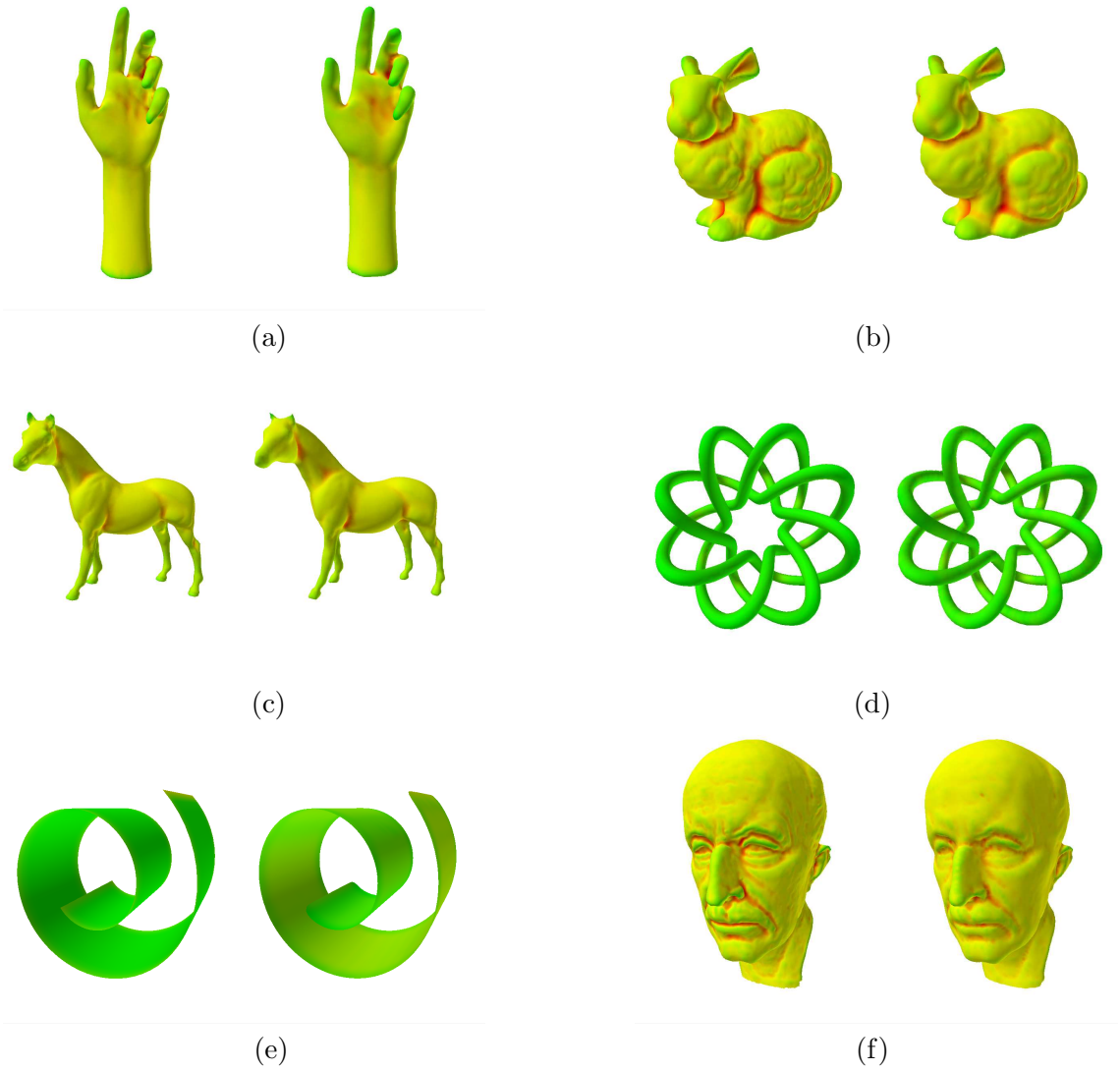
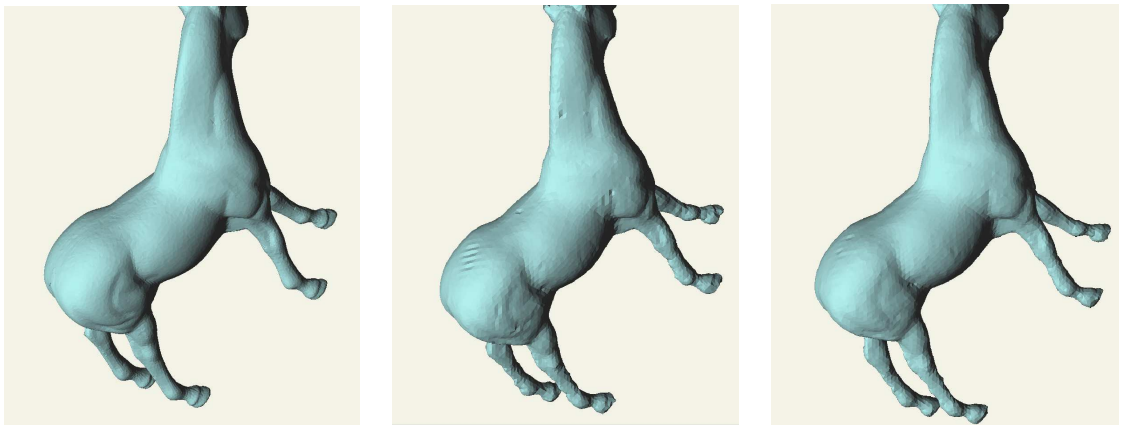


Figure 7.8: **Curvature plots.** The mesh on the left is the original model and the mesh on the right is the well-shaped nonobtuse mesh generated from our algorithm using a sampled grid size of 100^3. Green shading denotes negative mean curvature and red shading denotes positive mean curvature. (a) Arm model. (b) Bunny model. (c) Horse model. (d) Knot model. (e) Roll model. (f) Max Planck model.



(a) (b) (c)

Figure 7.9: **Importance of the smoothing step.** (a) Original horse model. (b) Horse model remeshed without the smoothing step. Notice there are roughness on the mesh surface. (c) Horse model remeshed with the smoothing step. Local smoothness is maintained throughout the surface.

Input	V_{in}	V_{out}	**v2**	**v3**	**int-v4**	**v2***	**v3***	**int-v4***
Arm	9305	9287	0	0	18	0	0	0
Bunny	16218	16154	23	24	25	1	7	0
Horse	9966	9941	0	1	29	0	1	4
Knot	20891	20819	0	0	72	0	0	0
Roll	6580	6446	2	134	0	0	2	0
Max P.	35650	35588	9	24	33	0	4	0

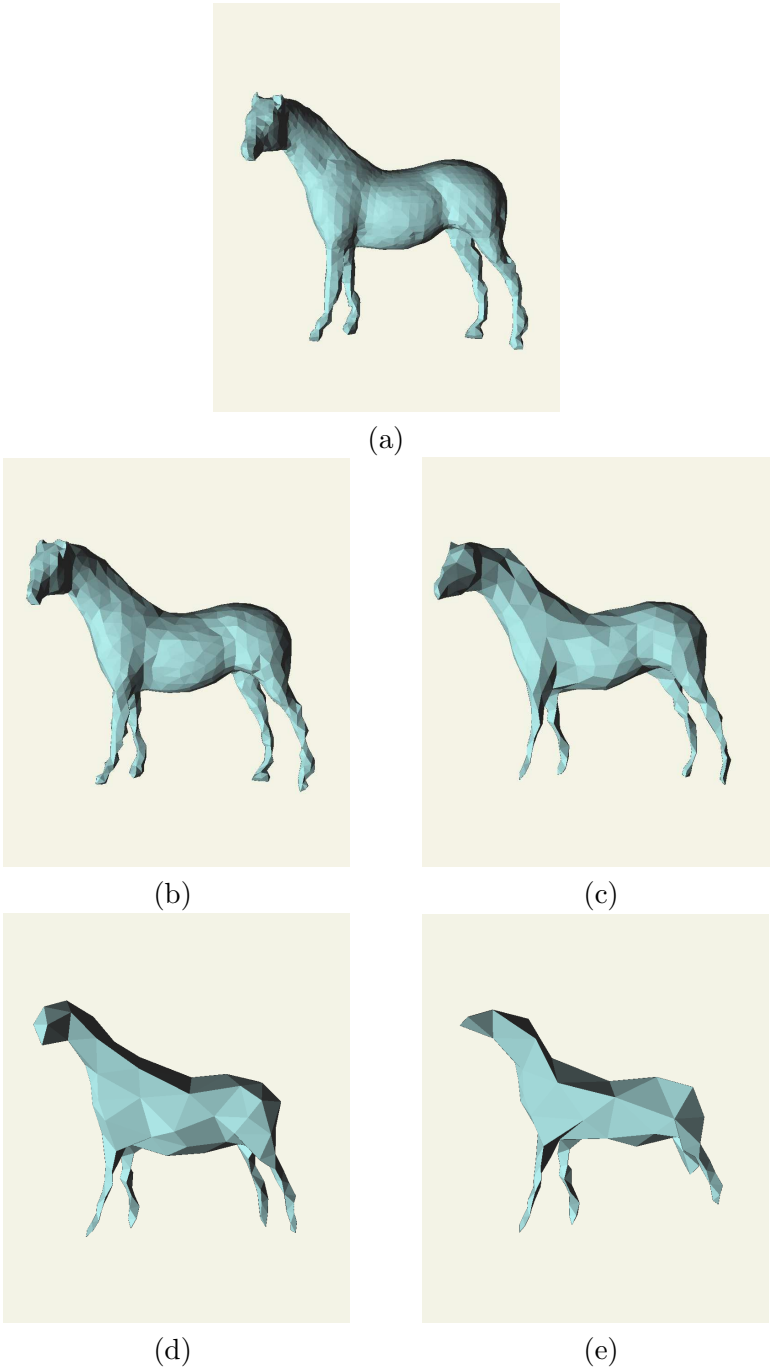
Table 7.1: **Removal of bad valence vertices.** V_{in}: number of vertices in the input mesh before the removal of bad valence vertices. V_{out}: number of vertices in the output mesh after the removal of bad valence vertices. **v2**: number of valence-2 vertices in the input mesh before the removal of bad valence vertices. **v3**: number of valence-3 vertices in the input mesh before the removal of bad valence vertices. **int-v4**: number of interior valence-4 vertices in the input mesh before the removal of bad valence vertices. **v2***: number of valence-2 vertices in the output mesh after the removal of bad valence vertices. **v3***: number of valence-3 vertices in the output mesh after the removal of bad valence vertices. **int-v4***: number of interior valence-4 vertices in the output mesh after the removal of bad valence vertices.

table, significant increase in performance is arrived at when lazy evaluation is performed. Moreover, the speed can be further improved upon by carrying out early smoothing. Notice that there is nearly no change in the approximation error produced by applying lazy evaluation and early smoothing. This confirms that even though at each iteration, the vertex chosen to be moved may not produce the highest improvement value, the performance speed-up added by lazy evaluation and early smoothing compensates for this slower improvement rate of the overall mesh optimization. Our main focus of this work is to provide a solution that provides guaranteed well-shaped nonobtuse meshes, hence, we did not investigate thoroughly on the speed issue. As the results show, our optimization can be quite time-consuming on large models. We may be too conservative on the stopping criteria. Nevertheless, we believe this is an important issue and we discuss this in Chapter 8.

Early termination during angle-bounded decimation: Finally, we show in Figure 7.10 that by using early termination during the decimation stage, we can ensure the output mesh would not introduce an error greater than the user-specified threshold. By terminating the decimation at an earlier stage, we can avoid introducing large errors that would change the overall shape of the mesh significantly as shown in Figure 7.10(d).

Input ($\triangle_{in}$)	$\triangle_{out}$	V_{out}	**G**	**LE**	**LE+ES**	ϵ_G	ϵ_{LE}	ϵ_{LE+ES}
Arm (50K)	18570	9287	4791s	1943s	1161s	1.0290	1.0035	0.9864
Bunny (34.7K)	32167	16154	6164s	2038s	1493s	1.0235	1.0417	1.1186
Horse (15.8K)	19878	9941	2421s	1826s	922s	1.1160	1.1205	1.1480
Knot (40K)	41638	20819	10399s	4326s	3792s	0.2372	0.2328	0.2357
Roll (29K)	12519	6446	3633s	1545s	1091s	1.6781	1.6731	1.6992
Max-P. (32.5K)	71082	35588	17524s	5851s	5005s	0.7122	0.7440	0.7740

Table 7.2: **Performance speed-up statistics.** We compare the timing required to remesh the six models using greedy optimization, optimization with lazy evaluation, and optimization with lazy evaluation and early smoothing. All meshes are generated with two smoothing step and the number of vertex movement in each optimization step is linear in terms of the number of vertices in the mesh. $\triangle_{in}$: number of triangles in the input mesh. $\triangle_{out}$: number of triangles in the output mesh. V_{out}: number of vertices in the output mesh. **G**: time required to remesh the models using greedy approach where no lazy evaluation and early smoothing are applied. **LE**: time required to remesh the models with lazy evaluation. **LE+ES**: time required to remesh the models with lazy evaluation and early smoothing. ϵ_G: metro error of the mesh generated with the greedy approach where no lazy evaluation and early smoothing are applied. ϵ_{LE}: metro error of the mesh generated with lazy evaluation. ϵ_{LE+ES}: metro error of the mesh generated with lazy evaluation and early smoothing.



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

Figure 7.10: **Early termination.** Decimation of the horse model can be stopped early to prevent introducing an error greater than some user-specified threshold. Without early termination, the mesh model is decimated until the number of vertices desired is reached. (a) Original well-shaped nonobtuse mesh prior to the decimation. (b) Horse decimated with threshold = 0.0001; $\epsilon = 1.0465\%$. (c) Horse decimated with threshold = 0.001; $\epsilon = 2.4694\%$. (d) Horse decimated with threshold = 0.05; $\epsilon = 8.4535\%$. (e) Horse decimated with threshold = 0.5; $\epsilon = 9.9976\%$.

Chapter 8

Conclusion and Future Work

In this thesis, we have presented a solution to the open problem of generating triangular meshes with a guaranteed angle bound $[30°, 90°]$. Motivated mainly by the need of high-quality triangle meshes in various fields of computer graphics, engineering, and mathematical simulation, we are able to produce well-shaped nonobtuse meshes via initial mesh construction using a modified Marching Cubes algorithm, iterative constrained "deform-to-fit" optimization, and angle-bounded decimation. Specifically, given an input mesh M, we first remesh it into an initial mesh M_0 that respects the proposed angle bounds. Next, we deform M_0 in a greedy iterative fashion to approximate the original mesh M. Finally, we perform angle-bounded decimation to generate a hierarchy of well-shaped nonobtuse meshes with different levels of detail.

We have utilized the Marching Cubes (MC) algorithm during the initial mesh construction stage. With the goal of supporting both open and closed manifold meshes, we have devised a technique that utilizes a modified MC algorithm without the need of a 3D scalar field being explicitly specified. The process of generating the initial mesh is divided into three phases. First, we identify all the cubes in a cubical cartesian grid that intersect the input mesh M and record for each intersected cube all triangles in M that intersect it. Then for each intersected cube, we group the triangles that intersect it into connected patches where these patches are then processed independently to assign signs to the cube corners. When a consistent assignment of signs is reached, we triangulate the cube based on the modified MC cases that we have devised. Once all cubes are triangulated, we perform a post-processing stitching step in the last phase to connect the set of generated triangles into a manifold mesh. The result is a well-shaped nonobtuse mesh M_0 that roughly approximates

the original input mesh M.

Once an initial well-shaped nonobtuse mesh M_0 is arrived at, we perform iterative constrained optimization by moving vertices of M_0 to approximate the original surface of M. This is achieved by formulating each vertex movement as a quadratic program. The objective function minimizes the distance from the vertex to the original surface and the set of constraints restrict the new location of the vertex to be within a region that does not violate the angle constraints. To ensure the local smoothness of the output surface, we interleave mesh optimization and constrained Laplacian smoothing with the former being the first and the last step of the "deform-to-fit" procedure. We have implemented lazy evaluation and early smoothing to speed up this whole process. The local smoothness of the output mesh can further be enhanced when bad valence vertices are removed prior to the optimization. We have suggested several heuristics for this purpose.

The optimization framework that we have developed is quite extendible. We show how it can be extended to perform angle-bounded decimation. At each iteration of the optimization, we perform edge collapse with the location of the unified vertex computed using the optimization framework. As a result, we are able to produce guaranteed well-shaped nonobtuse meshes in different levels of detail.

The problem of generating triangle meshes with guaranteed angle-bounds is an interesting one. Many areas of our work can still be improved upon. We discuss below possible future works and the difficulties in solving them.

- **Adaptive marching cubes:** In our current work, the resolution of the initial well-shaped nonobtuse mesh depends on the uniform sampling using the cubical cartesian grid. To reduce the triangle count of the initial mesh, a better approach is to apply adaptive marching cubes so that more samples are taken at areas of interest. In order to use an adaptive approach, one must ensure that no T-vertices and cracks are produced in the output mesh. On top of that, we must ensure the triangles produced are well-shaped and nonobtuse.

- **Robustness in construction of initial mesh:** We do not have a correctness proof for our initial mesh construction algorithm. Although nonmanifold edges and vertices were not produced in our experimental results, it would be best to have a theoretical proof of their complete absence for our approach. Furthermore, an initial mesh generated from a coarse resolution grid may introduce disconnected components.

In such cases, extra boundaries may be generated.

- **Better technique in removing bad valence vertices:** We have suggested several heuristics for removing bad valence vertices. As discussed in Chapter 7, the removal of such vertices improves the local smoothness of the output mesh. However, the current approach does not guarantee that all bad valence vertices will be removed. In the future, we would like to implement vertex split in aid of this. The main advantage of vertex split is that no vertices will have their respective valence decreased after the operation. On the other hand, the difficulty is determining the location of two vertices simultaneously; these two vertices result from a split. We must ensure that the two splitted vertices are located within their respective feasible region. This involves quadratic constraints, thus, the computation for the vertex locations is expensive.

- **Performance speed-up:** As shown in Chapter 7, the speed of generating well-shaped nonobtuse meshes in our method is not that pleasing. The algorithm will be appreciated more if the speed can be significantly improved. Currently, our optimization performs a vertex movement (or an edge collapse in the case of decimation) in a greedy fashion. There may exist a different processing order of the vertices such that it reduces the number of vertex movements required and at the same time reduces the overall approximation error in a faster rate. However, it is unclear what contributes to a good ordering of the vertices.

- **Feature preservation:** Our current approach does not recover or preserve sharp features. It may be desirable in certain applications that these features remain in the generated well-shaped nonobtuse mesh. A possible approach is to identify feature edges prior to the optimization either by thresholding or user intervention. Then vertices on these feature edges are optimized using the same scheme proposed for boundary edges. This way the optimization will attempt to preserve the feature edges as much as possible.

- **Meshing from point cloud:** Often meshes are obtained by first collecting a set of sampled points via a laser scanning device. A triangular mesh is then produced from the point cloud using surface reconstruction algorithm. It would be desirable to extend our approach to handle point cloud as input. The difficulty is that our initial mesh construction algorithm requires the intersection of a 3D mesh and a sampled cubical

grid for open meshes. Moreover, the "deform-to-fit" stage requires a point-to-surface distance measure to guide the optimization. A possible solution is to use a well-shaped nonobtuse sphere as the initial mesh. Then we move vertices of this sphere along its normal direction such that the new location is within the feasible region of the vertex. We repeatedly do so until the sphere fits the input point cloud. A problem with this suggestion is that local subdivision may need to be performed in order for the sphere to expand to thin regions of the underlying surface conveyed by the point cloud.

- **Well-shaped and nonobtuse triangulation:** Perhaps the most difficult problem is to interpolate a point cloud such that the resulting mesh is composed of well-shaped nonobtuse triangles. The main difficulty is that the geometry of the original input points are fixed. A feasible solution may not exist if no Steiner points are allowed to be inserted. On top of that, it is unclear how Steiner points should be added that would allow for a well-shaped nonobtuse triangulation of the point set. Minimizing the number of Steiner points used is also a difficult problem.

Bibliography

[1] P. Alliez, D. Cohen-Steiner, M. Yvinec, and M. Desbrun. Variational tetrahedral meshing. *ACM Trans. on Graphics*, 24(3):617–625, 2005.

[2] P. Alliez, G. Ucelli, C. Gotsman, and M. Attene. Recent advances in remeshing of surfaces, 2005. Part of the state-of-the-art report of the AIM@SHAPE EU network.

[3] M. Attene, B. Falcidieno, M. Spagnuolo, and J. Rossignac. Swingwrapper: Retiling triangle meshes for better edgebreaker compression. *ACM Trans. on Graphics*, 22(4), 2003.

[4] B. S. Baker, E. Grosse, and C. S. Rafferty. Nonobtuse triangulation of polygons. *Discrete and Computational Geometry*, 3(2):147–168, 1988.

[5] David C. Banks and Stephen Linton. Counting cases in marching cubes: Toward a generic algorithm for producing substitopes. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 8, 2003.

[6] M. Bern, L. P. Chew, D. Eppstein, and J. Ruppert. Dihedral bounds for mesh generation in high dimensions. In *ACM-SIAM Symp. on Discrete Algorithms*, pages 89–196, 1995.

[7] M. Bern and D. Eppstein. Polynomial-size nonobtuse triangulation of polygons. In *SCG '91: Proceedings of the seventh annual Symposium on Computational Geometry*, pages 342–350, 1991.

[8] M. Bern and D. Eppstein. Mesh generation and optimal triangulation. In *Computing in Euclidean Geometry, Lecture Notes Series on Computing*, volume 1, 1992.

[9] M. Bern, D. Eppstein, and J. Gilbert. Proably good mesh generation. In *Proc. of 31st IEEE Symp. Foundations of Comp. Sci.*, pages 231–241, 1990.

[10] M. Bern, S. Mitchell, and J. Ruppert. Linear-size nonobtuse triangulation of polygons. In *ACM Symp. on Computational Geometry*, pages 221–230, 1994.

[11] A. I. Bobenko and B. A. Springborn. A discrete lapplacian-beltrami operator for simplicial surfaces, February 2006. preprint: arXiv:math.DG/0503219.

[12] M. Botsch and L. Kobbelt. A remeshing approach to multiresolution modeling. In *SGP '04: Proceedings of the 2004 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, pages 185–192, 2004.

[13] C. Cassidy and G. Lord. A square acutely triangulated. *J. Rec. Math.*, 13(4):263–268, 1980.

[14] E. Catmull and J. Clark. Recursively generated b-spline surfaces on arbitrary topological meshes. pages 183–188, 1998.

[15] H.-L. Cheng and X. Shi. Quality mesh generation for molecular skin surfaces using restricted union of balls. In *IEEE Visualization*, pages 399–405, 2005.

[16] P. Chew. Guaranteed-quality mesh generation for curved surfaces. In *ACM Symp. on Computational Geometry*, pages 274–280, 1993.

[17] P. Cignoni, C. Rocchini, and R. Scopigno. Metro: Measuring error on simplified surfaces. *Computer Graphics Forum*, 17(2):167–174, 1998.

[18] J. Cohen, D. Manocha, and M. Olano. Simplifying polygonal models using successive mappings. In *IEEE Visualization*, pages 395–402, 1997.

[19] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications, Second Edition*. Springer, 2000.

[20] M. Eck, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, and W. Stuetzle. Multiresolution analysis of arbitrary meshes. In *SIGGRAPH '95*, pages 173–182, 1995.

[21] D. Eppstein. Acute square triangulation, 1997. The Geometry Junkyard: http://www.ics.uci.edu/ eppstein/junkyard/acute-square/.

[22] D. Eppstein, J. M. Sullivan, and A. Ün̈gör. Tiling space and slabs with acute tetrahedra. *Computational Geometry: Theory and Applications*, 27(3):237–255, 2004.

[23] M. Floater. Parametric tiling and scattered data approximation. *Int. J. on Shape Modeling*, 4:165–182, 1998.

[24] M. Garland and P. S. Heckbert. Surface simplification using quadric error metrics. In *SIGGRAPH '97*, pages 209–216, 1997.

[25] E. M. Gertz and S. J. Wright. Object-oriented software for quadratic programming. *ACM Trans. on Math. Software*, 29:58–81, 2003. http://www.cs.wisc.edu/∼swright/ooqp/.

[26] X. Gu and S.-T. Yau. Global conformal surface parameterization. In *Proc. of Symp. on Geometry processing*, pages 127–137, 2003.

[27] C.-C. Ho, F.-C. Wu, B.-Y. Chen, Y.-Y. Chuang, and M. Ouhyoung. Cubical marching squares: Adaptive feature preserving surface extraction from volume data. *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2005)*, 24(3):537–545, 2005.

[28] D. V. Hutton. *Fundamentals of Finite Element Analysis*. McGraw-Hill, 2004.

[29] A. Kaufman. An algorithm for 3d scan-conversion of polygons. In *Eurographics Conference Proceedings 1987*, pages 197–208, 1987.

[30] R. Kimmel and J. A. Sethian. Computing geodesic paths on manifolds. *Proc. of National Academy of Science USA*, 95:8431–8435, 1998.

[31] H. Lee, P. Alliez, and M. Desbrun. Angle-analyzer: A triangle-quad mesh codec. *Computer Graphics Forum*, 21(3):383–392, 2002.

[32] D. Lingrand, A. Charnoz, R. Gervaise, and K. Richard. The marching cubes, 2002. http://www.essi.fr/∼lingrand/MarchingCubes/algo.html.

[33] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *SIGGRAPH '87*, pages 163–169, 1987.

[34] S. A. Mitchell. Finding a covering triangulation whose maximum angle is provably small. In *Proc. of 17th Annual Computer Science Conference*.

[35] S. A. Mitchell. Refining a triangulation of a planar straight-line graph to eliminate large angles. In *IEEE Symposium on Foundations of Computer Science*, pages 583–591, 1993.

[36] S. A. Mitchell. Cardinality bounds for triangulations with bounded minimum angle. In *Proc. of Sixth Canadian Conference on Computational Geometry*, pages 326–331, 1994.

[37] S. A. Mitchell and S. A. Vavasis. Quality mesh generation in three dimensions. In *Symposium on Computational Geometry*, pages 212–221, 1992.

[38] C. Montani, R. Scateni, and R. Scopigno. Discretized marching cubes. In *Proc. of the conference on IEEE Visualization '94*, pages 281–287, 1994.

[39] G. M. Nielson and B. Hamann. The asymptotic decider: Resolving the ambiguity in marching cubes. In *IEEE Visualization*, pages 83–91, 1991.

[40] P. P. Pébay and T. J. Baker. A comparison of triangle quality measures. In *Proc. of the 10th International Meshing Roundtable*, pages 327–340, 2001.

[41] J. R. Shewchuk. What is a good linear finite element? interpolation, conditioning, and quality measures. In *Proc. of the 11th International Meshing Roundtable*, pages 115–126, 2002.

[42] G. Strang and G. J. Fix. *An analysis of the finite element method*. Prentice Hall, 1973.