2015

Large-scale Geometric Data Decomposition, Processing and Structured Mesh Generation

Wuyi Yu
Louisiana State University and Agricultural and Mechanical College, wuyiyu@lsu.edu

LARGE-SCALE GEOMETRIC DATA DECOMPOSITION, PROCESSING AND STRUCTURED MESH GENERATION

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Electrical and Computer Science
The Division of Computer Science Engineering

by
Wuyi Yu
B.S., Xiamen University, 2005
May 2016

To Sibyl, Alex, Ella and Ekka

ACKNOWLEDGEMENTS

First of all, I would like to gratefully and sincerely thank my Ph.D. supervisor, Professor Xin Li, for his warm encouragement and thoughtful guidance. He is my best role model. I appreciate all his contributions of time, ideas, and funding to my Ph.D. study.

I would like to thank my committee members, Bijaya Karki, Costas Busch, Robert Kooima, Hongchao Zhang and Brian Marx for their patience, discussion and collaborations.

I would also like to thank all members in Geometric and Visual Computing Group at LSU. Many thanks to Shenghua Wan, Huanhuan Xu, Kang Zhang, Shuai Zheng and Celong Liu for delightful collaborations and discussions we had together. Thanks Zhao Yin, Li Wei, Wei Li, Xiao Lin, Tengfei Ye, Wei Yu and so many others for their help during the past years.

Last but not least, I would like to thank my beloved wife Wenting Jing for her understanding and love. Without her support, encouragement and tolerance, all these would not have been possible. Wenting, I love you. I would like to thank my lovely son, Alexander, who brings me great joy. I would like to thank my family for their endless love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Mesh generation is a fundamental and critical problem in geometric data modeling and processing. In most scientific and engineering tasks that involve numerical computations and simulations on 2D/3D regions or on curved geometric objects, discretizing or approximating the geometric data using a polygonal or polyhedral meshes is always the first step of the procedure. The quality of this tessellation often dictates the subsequent computation accuracy, efficiency, and numerical stability.

When compared with unstructured meshes, the structured meshes are favored in many scientific/engineering tasks due to their good properties. However, generating high-quality structured mesh remains challenging, especially for complex or large-scale geometric data. In industrial Computer-aided Design/Engineering (CAD/CAE) pipelines, the geometry processing to create a desirable structural mesh of the complex model is the most costly step. This step is semi-manual, and often takes up to several weeks to finish. Several technical challenges remains unsolved in existing structured mesh generation techniques.

This dissertation studies the effective generation of structural mesh on large and complex geometric data. We study a general geometric computation paradigm to solve this problem via model partitioning and divide-and-conquer. To apply effective divide-and-conquer, we study two key technical components: the *shape decomposition* in the divide stage, and the *structured meshing* in the conquer stage. We test our algorithm on vairous data set, the results demonstrate the efficiency and effectiveness of our framework. The comparisons also show our algorithm outperforms existing partitioning methods in final meshing quality. We also show our pipeline scales up efficiently on HPC environment.

PART I

PROBLEM STATEMENT AND BACKGROUND

1. INTRODUCTION

1.1 BACKGROUND

Mesh generation is a fundamental and critical problem in geometric data modeling and processing. In most scientific and engineering tasks that involve numerical computations (e.g. solving partial differential equations (PDE) using finite difference methods (FDM), finite element methods (FEM), or finite volume methods (FVM)) and simulations on 2D/3D regions or on curved geometric objects, discretizing or approximating the geometric data using a polygonal or polyhedral meshes is always the first step of the procedure. The quality of this tessellation often dictates the subsequent computation accuracy, efficiency, and numerical stability.

To tessellate 2D manifolds (planar regions or curved surfaces) or 3D manifolds (3D regions or solid objects), piecewise mesh elements can be general polygons or polyhedra respectively. But most commonly used tiling elements are triangles (tri) or quadrangles (quad) for 2D manifolds, and tetrahedra (tet) or hexahedra (hex) for 3D manifolds. Figure 1.1 shows these four types of mesh elements generated on the Stanford Bunny model. The tri/tet meshes are the most commonly used meshes in finite element computation and analysis. Effective triangular meshing techniques have been well studied. State-of-the-art triangular meshing software, such as Triangle [6] for surface data, and TetGen [7]/NetGen [8] for volume data, is

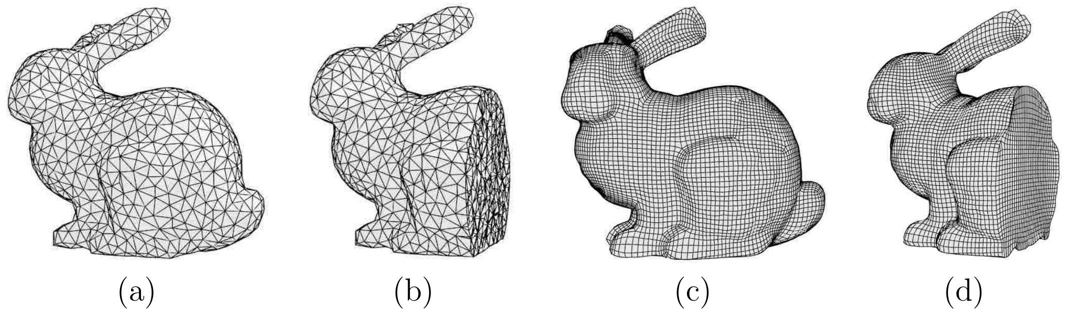


Figure 1.1: Different types of meshes. (a) - (d) Triangular, tetrahedral, quadrilateral, and hexahedral meshes of the Stanford Bunny model

able to efficiently produce millions of high-quality triangle/tetrahedron elements per second for geometric data processing and computation.

On the other hand, compared with tri/tet meshes, the quad/hex meshes are often preferred in many applications due to several reasons.

- **Natural Tensor-product Representation.** The quad/hex meshes directly support the natural representation of the tensor-product high-order bases, which is needed for the industrial standard representations, such as tensor-product B-splines, Non-uniform rational basis spline (NURBS) [9], and Catmull-Clark surfaces [10] in computer-aided design/manufacturing and computer graphics.

- **Higher Computational Accuracy.** In many scientific computing tasks such as elastic analysis, structural analysis, Navier-Strokes computation, etc., quad/hex meshes provide better accuracy than tri/tet meshes for the same computational cost [11].

- **Easier Feature Alignment.** In geometric modeling, quad/hex meshes can better support feature alignment. For example, geometric surfaces has two dominant directions locally, referred to as the principal directions, to which quad/hex meshes can naturally align, for sampling optimization [12].

1.2 CHALLENGING ISSUES

Despite these advantages in engineering designs and simulations, effective quad/hex mesh generation is non-trivial. **In industrial Computer-aided Design/Engineering (CAD/CAE) pipelines, the geometry processing to create a desirable structural mesh of the complex model is semi-manual, and often takes up to several weeks to finish [13]. Therefore, it is often the most time-consuming step within the entire pipeline [11, 13] and can take up to 80% of the total costs in the CAD/CAE procedure [14].** Several technical challenges remains unsolved in existing quad/hex mesh generation techniques.

1. **Generating high-quality elements for complex geometry is difficult.** From the aspect of element quality, an ideal quad element is a square, while a perfect hex element is a cube. However, arbitrarily given geometries cannot be tessellated into meshes with only ideal elements, angle and volume distortions inevitably exist in the elements. The bad-quality (e.g. severely sheared or scaled) elements limit the usability of the mesh. When the geometry is complicated, most existing meshing algorithms can not guarantee to generate a mesh without degenerate (i.e., non-zero edge-length, area, and volume) nor flipped (i.e., self-intersected) elements [15], near which the numerical computation/analysis cannot be performed.

2. **Handling large-scale data is difficult.** When the data size becomes big, the generation of quad/hex mesh with controllable global structure becomes very difficult. On one hand, direct computation is very expensive. On the other hand, when the data have complex topology and geometry, controlling the global layout and singularity

distribution is usually infeasible.

3. **Local topological modification/refinement is difficult.** During mesh generation, local update and refinement is often desirable in generating high-quality adaptive meshes and in improving mesh quality. However, for quad/hex meshes, local topological modifications will propagate across entire region. Many effective refinement techniques in triangle/tetrahedral meshes, such as subdivision, could not be applied [11].

1.3 RELATED WORK

Our goal is to design effective structural quad/hex mesh generation algorithm for complex and large geometric regions/objects, tackling the above challenges. We propose to develop a **divide-and-conquer** computational pipeline to solve the problem of structured mesh generation for complex and large-scale geometric data. To apply effective divide-and-conquer, we study two key technical components: the **shape decomposition** in the divide stage, and the **structured meshing** in the conquer stage. In the processing of complex or large geometric data, effective data decomposition can partition the data into solvable smaller and geometric simpler subparts, upon which processing can be performed simultaneously in a parallel manner. On each local region, structured meshes should be constructed following boundary consistency constraints along the partitioning boundary, to avoid unnecessary topological modifications after local results are stitched together.

1.3.1 Shape Partitioning

Given a geometric region M, a set of components $\{M_i\}$ is a *decomposition* of M if (1) their union is M, i.e., $\bigcup_i M_i = M$, and (2) all M_i are interior disjoint, namely, $\forall_{i \neq j} M_i^\circ \cap M_j^\circ = \emptyset$, where $M_i^\circ = M_i \backslash \partial M_i$ is the open set of M_i.

Shape decomposition has been widely used in a large number of applications in computational geometry, computer graphics, parallel computing. In the computational geometry and computer graphics applications, the study of shape decomposition focus on how to get desire geometric properties for each sub-region, such as convexity, symemtry, etc. In parallel computing applications, the study focus on the efficiency of parallel computing.

Shape Decomposition in Geometric Processing

In computation geometry, geometric regions have been decomposed into different polygonal/polyhedral primitives, such as Voronoi cells, convex subregions, star subregions, etc. Thorough reviews have been given in surveys [16] and [17]. In computer graphics and visualization, shape decomposition has been studied for different applications such as object recognition, meshing, skeleton extraction. Geometric objects are often decomposed into subpatches with certain desirable geometric criteria such as compactness, flatness, roundness, or along certain feature regions such as concave valleys, sharp ridges, etc. Two thorough surveys in this field have been given in [18] and [19]. In geometric processing, the shape decomposition techniques mainly focus on optimizing geometric properties of the local subparts obtained from the decomposition.

Shape Decomposition in Parallel Computing

Data decomposition is also studied in many parallel computing problems, but existing techniques often pay most attention to the efficiency of parallel computing resulted from this decomposition. Existing partitioning strategies developed in parallel data processing literature can be classified into two categories: *extrinsic space partitioning* and *intrinsic manifold partitioning* methods. The first class method partitions the data by partitioning the data's embedding space using spatial structures such as quad-tree or octree, axis/planes or space-fill curves, etc. In general, data (space) partitioning using space-filling curves or other extrinsic space partitioning methods is very efficient, as demonstrated in several successful applications, such as computational physics, algebraic multigrid and adaptive mesh refinement. However, algorithms based on spatial partitioning are not suitable to handle data that have complex geometry or nonuniform properties. The second category method partitions the data model on its intrinsic tessellation. The data are discretized into a mesh or a graph, where elements or nodes are clustered into subparts directly or recursively. Among this category, a very widely used strategy is the *graph partitioning* which usually produce good-quality partitions with balanced load and reduced communication. These existing graph partitioning algorithms focus on only tackling the load balancing and communication reduction issues. However, only considering these two criteria is insufficient, since flexibly incorporating extra geometric constraints is often critical in geometric and spatial data processing in many scientific applications including our mesh generation task.

1.3.2 Quad/Hex Mesh Generation

In a quad mesh, if the valence of a vertex is 4, then it's a *regular* vertex. Otherwise the vertex is called *irregular*, or a *singularity vertex*. For hex meshes the valence of a regular vertex is 6. If all the interior vertices of a mesh are all regular then we call the mesh is a *structured mesh*, otherwise it's a *unstructured mesh*. Structured meshes can facilitate a number of applications. In animation, structured meshes can act as control meshes for subdivision surfaces. In other applications such texture mapping, structured meshes can be trivially mapped to texture. With careful design, a complicated geometric objects can be represented using a set of structured meshes stitched together, and along the stitching boundary there may be a few singularity vertices.

Generally speaking, there are three types of methods for generating quad/hex meshes: spatial partitioning, plastering, and mapping based methods.

In **spatial partitioning** approaches, the given object or the space it embeds is first decomposed into a set of cells which will then be projected or deformed to conform to the model's boundary geometry. A limitation these approaches is its pose-sensitivity: a small orientation change (e.g. rotation) of the object can lead to different meshing results. Another limitation is that these approaches usually lead to poor quality near the boundary elements. On the other hand, the **plastering** approaches usually start from the structurally meshed boundary, then propagate the construction of new elements toward the object center. The plastering approach can generate high quality elements, however, it often generates large amount of singularities, whose number and distribution cannot be controlled. **Mapping-based** methods parameterize (map) the input object or region M onto a regular-shape domain Ω, namely, compute $\Phi : M \rightarrow \Omega$. On the regular domain Ω, a uniform structural mesh H can be easily constructed using subdivision [20–24] or other standard technique. Then, the inverse of the map $\Psi = \Phi^{-1} : \Omega \rightarrow M$ is used to transfer the regular mesh H back to M. Unlike the first two approaches, the mesh singularities and their distribution can be controlled by the map Φ, however, optimizing the shape of Ω and the map Φ often reduce to solving *nonlinear optimization with nonlinear geometric constrains.* The computation can often be expensive, and therefore, needs to be performed on subregions that have relatively small size, simple geometry, and uniform feature/material structures. A general surface/volume parameterization model for regular mesh generation is to map M onto a canonical 2D/3D domain whose boundary sizes are integers (assuming each cell has a unit size) [25, 26], allowing singularity points on both interior and boundary regions [27, 28]. However, for 3D manifolds, the topological structure of the singularity of the parameterization is much more complicated and its automatic construction/optimization remains unsolved [28].

1.4 OUR APPROACH

In this dissertation, we aim to explore and tackle the aforementioned challenging technical issues in quad/hex mesh generation of complex and large geometric models. We will propose a shape decomposition algorithm based on "visibility" called **Star Decomposition** to partition a given 3D region into a set of star shapes, on which high-quality bijective mapping can be constructed. Then, we will also study effective partitioning efficient parallel processing of large geometric models. Besides load balancing and communication minimization, we also study the incorporation of various geometric properties, to propose a **Geometric-aware Graph Partitioning** computation pipeline for efficient parallel geometric processing.

On the other hand, constructing lowly distorted mapping from the geometric data onto canonical domain dictates the quality such as the stretching, skewness, and uniformity of the constructed meshes. Constructing and optimizing suitable canonical parametric domains is critical for the element quality and singularity distribution control in mapping-based regular mesh construction. We will propose innovative **polycube parameterization** algorithm to map geometric object onto the canonical and regular polycube domain, upon which uniform and perpendicular iso-integral lines can be obtained, and boundary-constrained high-quality mesh can be generated.

The remainder of this dissertation is organized as follows: In part 1.4 we will study two partitioning techniques: visibility-based decomposition technique called *star-decomposition* (Section 2) and a geometric-aware graph partitioning technique (Section 3). A polycube mapping based structured meshing algorithm will be studied in Part 3.5.

Finally is the conclusion and future work(Chapter 7).

PART II

SHAPE PARTITIONING

This chapter is devoted to study the shape decomposition algorithms. In Section 2 we will study the 3D guarding problem and propose a star decomposition algorithm for 3D. In Section 3 we will study a geometric-aware graph partitioning, which is suitable for parallel geometric processing tasks.

2. COMPUTING 3D SHAPE GUARDING AND STAR DECOMPOSITION

This section studies the shape partitioning based on visibility, i.e. *star decomposition*. It segments a 3D volumetric shape to a set of subregions, each of which is visible from a guarding point (such a subregion is called a star shape). It can be shown that a star-shaped subregion has some good properties. For example, harmonic volumetric parameterization can be constructed bijectively upon such domain [29]. In computer graphics and animation, star decomposition can benefit many tasks such as shape matching/retrieval, and morphing. Surface segmentation, generally based on specific local geometric properties of surface patches, has been thoroughly examined (see survey papers [18, 19]). Partitioning 3D objects based on their volumetric properties, such as convexity, symmetry, etc. have also been studied; however, less study has been conducted to the decomposition based on visibility. Star-decomposition is closely related to a well known *art-gallery guarding* problem. The gallery guarding problem has been studied in computational geometry community on 2D planar domains and in 2.5D for terrain guarding. But “Very little is known about gallery guarding in three dimensions” [30], especially for 3D free-form models, due to their much higher complexity.

The main contributions include:

- We develop an effective progressive integer linear programming (PILP) optimization paradigm to compute approximate optimal guarding of complex 3D free-form domains;

- We present a region-growing algorithm to compute the star-decomposition of a given 3D model, seeded from guarding points computed in the PILP;
- We explore two direct applications of our proposed guarding/decomposition framework: shape morphing and shape matching.

2.1 RELATED WORK

2.1.1 Shape Guarding

We consider the following shape guarding problem in 3D. A point $g \in M$ is *visible* to another point $p \in M$ if the line segment $\overline{gp}$ entirely locates inside M. A region (shape) M is called a *star region* (star shape) if there exists a point $g \in M$ that any point $p \in M$ is visible to g; we call such a point g a *guarding point*. All convex shapes are star shapes; but more complicated shapes usually can not be visually covered by a single guard so these regions are not star shapes. Given a general solid shape M whose boundary ∂M is a closed surface, usually tessellated by a triangle mesh, we want to find a smallest set of points $G = \{g_i\}$ inside M that every point $p \in \partial M$ is *visible* to at least one point in G.

Various versions of this problem are generally called *art gallery* problems, which are known to be a famous difficult problem. Finding minimal guards has been shown to be NP-hard for 2D polygons with holes [31], 2D simple polygons [32], and even 2D simple orthogonal polygons [33], using either vertex or point guards. Approximation algorithms have been studied in 1.5 ([30]) and 2D ([34]) to get an close-to-optimal result in polynomial time complexity. Lien [35] computes guarding for 3D point cloud data, approximating visibility using ϵ-view. The algorithm is based on a randomized greedy approach.

2.1.2 Shape Decomposition

Given a solid model M, represented by a tetrahedral mesh $\{T, V\}$, where T is the set of tetrahedra and V is the set of all the vertices, a decomposition is a partitioning of T into a set of subregions $M_i = \{T_i, V_i\}$, so that (1) $T_i \subset T$, (2) $\bigcup_i T_i = T$, and (3) $T_i \bigcap T_j = \emptyset, i \neq j$. If each subregion T_i is a star shape, we call this partitioning a star decomposition.

Shape decomposition has been widely studied in computation geometry and graphics. In computation geometry, different decomposition methods (e.g. Voronoi decomposition, convex decomposition, etc.) have been proposed; while a thorough review on other types of decompositions is beyond the scope of this work, we refer readers to surveys [16] and [17]. In computer graphics and visualization, surface segmentation has been studied for different applications such as object recognition, meshing, skeleton extraction. Two thorough surface segmentation surveys were given in [18] and [19], in both of which, segmentation techniques are classified as *surface-based* methods (segmentation guided by surface properties of sub-surface-patches) and *part-based* methods (segmentation guided by volumetric properties of sub-solid-regions).

General approaches for decomposition can also be classified into two categories: *top-down* methods, by iteratively segmenting sub-parts to finer components; and *bottom-up* methods, by iteratively gluing small elements/components to larger parts. For example, the Approximate Convex Decomposition [36] is a top-down approach. It iteratively measures the convexity of each (sub-)region M; if it fails to satisfy the convexity criterion, we shall further *cut* it into two sub parts M_1 and M_2. The algorithm continues until all sub-regions are convex enough. A difficult issue in *top-down* methods is to find the suitable *cut* so that shapes of smaller regions become nice in a few steps. On the other hand, popular surface segmentation techniques such as region growing ([37], [38]), watershed ([39]), or clustering ([40], [41]) algorithms are bottom-up approaches. These approaches start from a set of seeds, then expand to include neighboring primitives (vertices, faces, tetrahedra) until their unions cover the entire region.

2.2 3D SHAPE GUARDING

Given a solid shape M, whose boundary is discretized by a triangle mesh $\partial M = \{V, F\}$, where $V = \{v_1, v_2, \ldots, v_{N_V}\}$ are vertices and $F = \{f_1, f_2 \ldots, f_{N_F}\}$ are triangle facets. Seeking fewest necessary guarding points is challenging. Our algorithm is based on the following two intuitions. (1) As demonstrated in medical visualization, medial axes (skeletons) usually have desirable visibility to the shape (referred as the "reliability" of skeletons). The skeleton can guide camera navigation and ensure full examination of the organ. (2) Hierarchical skeletons can be effectively computed, progressively reducing the size of the optimization problem and improving computation's numerical efficiency and stability against boundary perturbations or noise.

Many effective skeletonization algorithms (see a great survey [42]) have been developed for 3D shapes. We use the algorithm/software of [43] since it efficiently generates skeletons on medial-axis surfaces of the 3D shapes. For the boundary triangle mesh ∂M with N_F triangles and the extracted skeleton with N_K nodes, the guarding problem can be converted to finding a minimal-size point set G from this N_K points, such that all N_F boundary faces are visible to G. Note that here we require each boundary triangle face is visible to G. We define the visibility of a face as follows:

- Vertex Visibility: A point $p \in M$ is *visible* to another point $q \in \partial M$ if the line segment $\overline{pq}$ connecting p and q is inside M, namely, it only intersects ∂M on q: $\overline{pq} \bigcap \partial M = \{q\}$.
- Face Visibility: A triangle face $f \in F$ is *visible* to a point p if all its three vertices are visible to p.

The shape guarding problem can be approximated as finding G to guard all the boundary vertices, in which we will only need the concept of vertex visibility. For the subsequent star decomposition purpose, to make each triangle of the boundary surface on the sub-region fully visible, we shall use the face visibility. Guarding all the faces of a region is stronger (i.e. can require more guards) than guarding all the vertices. When the triangle mesh is dense enough, face visibility well approximates the visibility in the continuous case.

2.2.1 Visibility Detection

A basic operation is to detect the visible region of a (skeleton) point p. Specifically, on the boundary surface $\partial M = \{F, V\}$, we define the *visibility region* $V(p)$ of an interior point p to be the collection of all visible boundary triangles: $V(p) = \{f | f \in F, f \text{ is visible to } p\}$. To compute $V(p)$, one should check intersection between each line segment $\overline{pv_i}$ and ∂M, where $v_i \in V$ is a vertex. If intersection is detected on a point $q \in \partial M$ other than v_i and the Euclidean distance $|\overline{pq}| < |\overline{pv_i}|$, then v_i is not visible from p. Simply enumerating every $\overline{pv_i}$ then detecting its intersections with every triangle $f \in F$ is time consuming: for a single skeleton point p, it costs $O(N_V \cdot N_F) = O(N_V^2)$ time to check its visibility on N_V vertices. We develop the following *sweep algorithm* to improve the efficiency. We create a spherical coordinate system originated at p. Each vertex $v_i \in V$ is represented as $\overline{pv_i} = (r(v_i), \theta(v_i), \varphi(v_i))$, where $r(v_i) \geq 0, -\pi/2 < \theta(v_i) \leq \pi/2, -\pi < \varphi(v_i) \leq \pi$. For every triangle $f_i = (v_{i,1}, v_{i,2}, v_{i,3}) \in F, 1 \leq i \leq N_F$, its max $\theta(f_i)$ can be defined as $\theta_{max}(f_i) = \max\{\theta(v_{i,j})\}, 1 \leq j \leq 3$, the $\theta_{min}(f_i), \varphi_{max}(f_i), \varphi_{min}(f_i)$ can be defined similarly. The segment $\overline{ov_k}$ cannot intersect with a triangle f unless

$$\begin{cases} \theta_{min}(f) \leq \theta(v_k) \leq \theta_{max}(f) \\ \varphi_{min}(f) \leq \varphi(v_k) \leq \varphi_{max}(f), \end{cases} \tag{2.1}$$

therefore we ignore triangles outside this range and only check ones that satisfy this condition (denoted as active triangles).

The angle functions θ and φ are not continuously defined on a sphere. When a triangle f spans $\theta = \pi$, we duplicate it to ensure that each θ of the original f is between $[\theta_{min}(f), \theta_{min}(f) + 2\pi)$ and θ of its duplicate is between $[\theta_{max}(f), \theta_{max}(f) + 2\pi)$, by adding or subtracting θ by 2π. For each triangle f spans $\varphi = \pi$, we duplicate it in the same way. Using $\theta(v_i)$ as the primary key and $\varphi(v_i)$ as the secondary key, we then sort all line segments $\overline{pv_i}$. Then we sweep all segments following the angle functions one by one in an ascending order and check intersection between the sweep line and all active triangles satisfying condition (2.1).

Specifically, we define a counter c_i on every triangle f_i. Initially, $c_i = 0$; when the segment $\overline{pv}, v \in f_i$ is being processed, $c_i \leftarrow c_i + 1$. The following two cases indicate that the sweep has not reached the neighborhood of the triangle f_i, and we do not need to check its intersection with line segment $\overline{pv}$:

$$c_i = 0 \rightarrow \quad \theta_{min}(f_i) > \theta(\overline{ov}), \text{ or } \varphi_{min}(f_i) > \varphi(\overline{ov});$$

$$c_i > 3 \rightarrow \quad \theta_{max}(f_i) < \theta(\overline{ov}), \text{ or } \varphi_{max}(f_i) < \varphi(\overline{ov}).$$

Therefore we maintain a list L of active triangles $\{f_i\}$ whose counters have $1 \leq c_i \leq 3$. When the sweep segment hits a new triangle t_j, we have $c_j = 1$ and add t_j into L; when a counter $c_j = 3$, we remove t_j from L after processing the current segment.

Given a skeleton point p, for a boundary triangle mesh with N_V vertices it takes $O(N_V \log N_V)$ to compute and sort angles of all segments. For each segment, if the size of the active triangle list L is m, it takes $O(m)$ intersection-detecting operations. Therefore, the total complexity is $O(\log N_V + mN_V)$. The incident triangles around a vertex v_i is usually very small: $m < \log N_V$. Therefore the algorithm finishes visibility detection of p in $O(N_V \log N_V)$ time. On a skeleton containing N_K nodes, it takes $O(N_K N_V \log N_V)$ pre-computation time to know the visibility region for all nodes.

2.2.2 Greedy and Optimal Guarding

Once visibility regions for all skeletal nodes are computed, the guarding problem can be converted into a set-covering problem. Consider a set in which each element corresponds to a face on the boundary; a skeleton node can see many faces so it covers a subset of elements. We want to pick several skeleton nodes so that the union of their covered subsets is the entire set. The set-covering problem, shown to be NP-complete [44], can be formally defined as follows: given the universe $V = \{v_i\}$ and a family S of subsets $S_j = \{s_{j,k}\}, s_{j,k} \in V$, a cover is a subfamily $C \subset S$ of sets whose union is V. We want to find a covering C that uses the fewest subsets in S. C indicates an optimal subset of skeletal nodes that can guard the entire region. Skeletons generated using medial-axis based methods with dense enough nodes usually ensure S itself is a covering. This holds in all our experiments. If a coarsely sampled skeleton can not cover the entire V, we further include all those invisible vertices into the skeleton point set. A **greedy** strategy for the set covering is as follows: iteratively pick the skeletal nodes that can cover the most faces in V, then remove all guarded faces from V (and update S accordingly since the universe becomes smaller), until $V = \emptyset$. Such a greedy strategy is quite effective and it yields $O(\log n)$ approximation [45] to the set covering problem.

An **optimal** selection can be computed by $0-1$ programming, also called Integer Linear Programming (ILP). We assign a variable x_i on each skeleton node p_i: $x_i = 1$ if p_i is chosen, and $x_i = 0$ otherwise. The *objective function* to minimize is then $\sum_{i=1}^{m} x_i$.

Every element should be visible, for $\forall f_i \in F$ visible to some skeletal nodes $P_i = \{p_{(i,1)}, \ldots, p_{(i,k)}\}$, at least one node in P_i should be chosen to ensure f_i guarded. Thus we solve:

$$\min \sum_{i=1}^{m} x_i, \text{ subject to} \tag{2.2}$$

$$x_i = 0, 1, \text{ and } \sum_{j \in J(i)} x_j \geq 1, \forall i \in \{1, \ldots, n\}, \tag{2.3}$$

where $J(i)$ is the index set of nodes p_j visible to f_i.

The above optimization can be solved using branch-and-bound algorithms. When the dimension is small (e.g. a few hundreds to a few thousands), we can use the TomLab Optimization package [46] to solve it efficiently.

2.2.3 Progressive Guarding

Directly solving optimal guarding is expensive, or prohibitive for big models. In contrast, greedy algorithm generates many unnecessary guards and is sensitive to local boundary geometric perturbations.

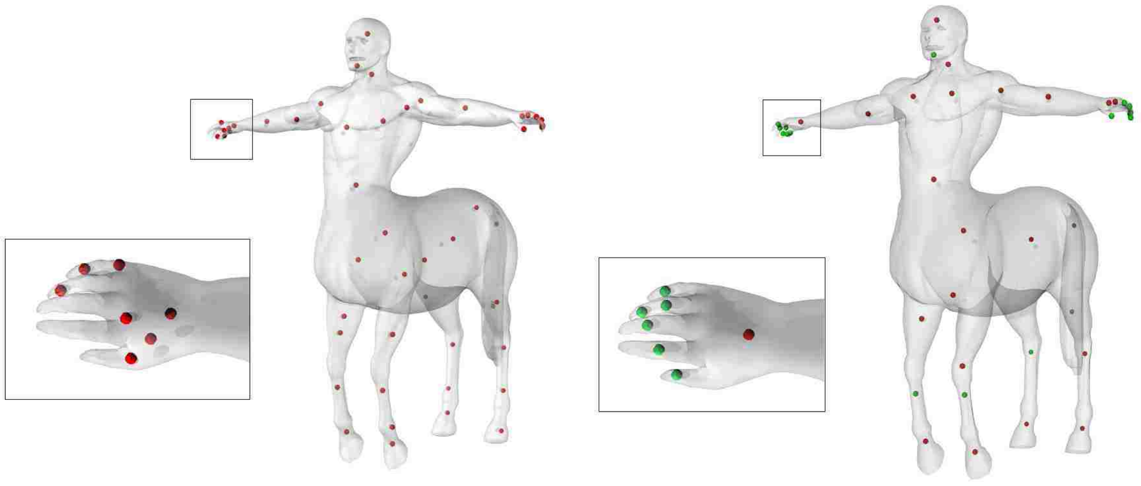


Figure 2.1: Greedy (left) and PILP (right) Guarding of Centaur.

To solve this problem, we propose a progressive integer linear programming (PILP) framework using progressive mesh simplification and refinement [47], adaptively combining ILP and multiresolutional refinement.

We progressively simplify the boundary mesh ∂M into several resolutions $\partial M^i = \{T^i, V^i\}, i = 0, \ldots, m$. On each level we keep the problem size within the scale that ILP can solve. The finest skeleton is re-sampled with joints preserved and used for computation. In the coarsest level $i = 0$, we solve the optimal guarding using ILP. Then we progressively move to finer levels with more details. On each finer level i, we greedily remove regions covered by existing guards computed in level $i-1$, then again use ILP to find necessary new guards. With we move toward finer level with increased details, new guards are added until the finest resolution ∂M^m is covered. On every new level, we also resolve a few least significant existing guards (whose visibility region covers a small area $Area(V(p)) < \epsilon Area(M^i)$). We do not directly insert them into the current level's guarding set G^i and do not remove their covered boundary faces from the universe.

On each level, we also conduct four **reduction** operations before ILP computation. These reductions significantly reduce the optimization problem size. Suppose we store the visibility information in an incidence matrix A: $a_{ij} = 1$ if the skeletal node p_i can see the face f_j, and $a_{ij} = 0$ otherwise. Originally the A is $|N_K| \times |N_F|$, and we apply the following **four rules** to reduce the size of A:

1. If column j has only one non-zero element at row i, we must pick p_i in order to see v_j. Therefore, add p_i into G, remove column j. Also, for all non-zero element a_{ik}, remove column k (since all vertices visible to p_i are now covered, and can be removed).

2. If row i_1 has all its non-zero elements non-zero in row i_2, i.e. $a_{i_1,j} = 1 \rightarrow a_{i_2,j} = 1$, then p_{i_2} sees all vertices that p_{i_1} can see, and we can remove the entire row i_1.

3. If column j_1 has all its non-zero elements non-zero in column j_2, i.e. $a_{i,j_1} = 1 \rightarrow a_{i,j_2} = 1$, then guarding v_{j_1} guarantees the guarding of v_{j_2}, and we can remove the entire column j_2.

4. If the matrix A is composed of several blocks, we partition A to several small matrixes $\{A_k\}$.

In step 4, since we remove faces that have been covered by existing guards, remaining boundary faces could be partitioned to several connected-components far away from each other. And these sub-components may be optimized separately, which significantly reduces optimization size.

This PILP scheme can efficiently compute the guarding for large size 3D volumetric regions and generate a hierarchical guarding graph. The pipeline is fully automatic, and furthermore,

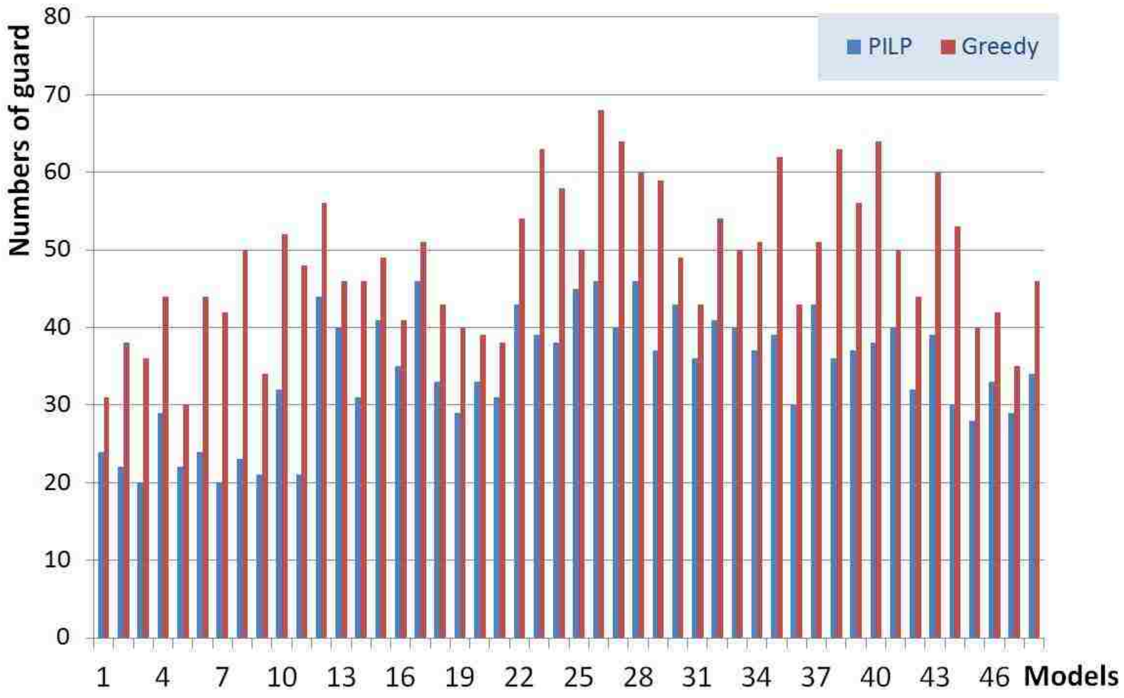


Figure 2.2: Greedy vs PILP on 48 models in TOSCA dataset [1]. The x-axis lists the 48 models, the y axis indicates the necessary guards computed. The blue bar indicates the PILP result and red is the greedy result. PILP has similar computational performance with the greedy approach, but generate better guarding (on average, using 20% less guards).

it has the following important **advantages** over both the pure greedy strategy and global ILP optimization (more statistical results are shown in Table 2.1 and Fig. 2.2). Note that the t_P in Table 2.1 does not include the progressive mesh computation time. However, progressive mesh can be computed efficiently. Simplifying a $10k$ mesh takes roughly 10 seconds. From our experiments, we can see that

- PILP is **much faster** than global ILP. Its computational efficiency is improved for several orders of magnitude over ILP on large-size geometric models, and can therefore handle massive data.

- With comparable speed to the greedy approach, PILP usually provides **much better** guarding solutions. Firstly, the PILP guards number is smaller than the greedy

Table 2.1: Guarding Statistics. N_V is the boundary surface vertex number. N_I, N_G, N_P indicate the number of necessary guards computed by ILP, Greedy, and PILP approaches, respectively. t shows the computational time in seconds. Guarding of big models cannot be solved directly using ILP, so their statistics are not applicable.

Models (N_V)	N_I	N_G	N_P	t_I	t_G	t_P
Greek (9,994)	15	22	18	4,122.4	290.2	293.1
David (9,996)	16	22	17	107,391.2	233.9	235.2
Female (10,002)	13	18	14	2,046.2	264.2	281.1
Male (10,002)	14	16	15	3,074.3	298.6	310.2
Cat (10,004)	14	19	15	3173	375.4	393.1
Wolf (10,005)	13	18	15	8044	328.1	349.9
Dog (15,002)	–	39	27	–	412.3	433.2
Victoria (15,000)	–	35	27	–	408.7	421.2
Horse (20,002)	–	38	29	–	376.1	384.2
Michael (20,002)	–	46	31	–	321.0	332.9
Gorilla (30,004)	–	60	46	–	462.4	490.1
Centaur (30,002)	–	52	32	–	488.1	514.5

approach; secondly, the PILP guarding is hierarchical and therefore is **robust** against geometric noise (Figure 2.1 shows an example. In PILP, global structure from coarser levels is stable under local refinement to new details).

In our experiments, we simplify the boundary mesh to the coarsest level with 5k vertices for the first round ILP optimization. On each iteration, we refine to next level with additional 10k vertices. When the size of constraints is around 5k, and the size of variables (skeletal nodes) is around 1k, the optimization takes 10-30 seconds to solve. We set the significance threshold parameter to be $\epsilon = 10\%$.

Figures 2.1 shows an example of guarding the Centaur model, where we can see the PILP guarding provides a stable hierarchial guarding. The guards added in the finest level are colorized in green while the one computed on coarser level are rendered in red. We perform extensive experiments on our new algorithm. And it demonstrates great effectiveness. More guarding results are visualized in Figure 2.3. Statistical comparison is shown in Table 2.1; a

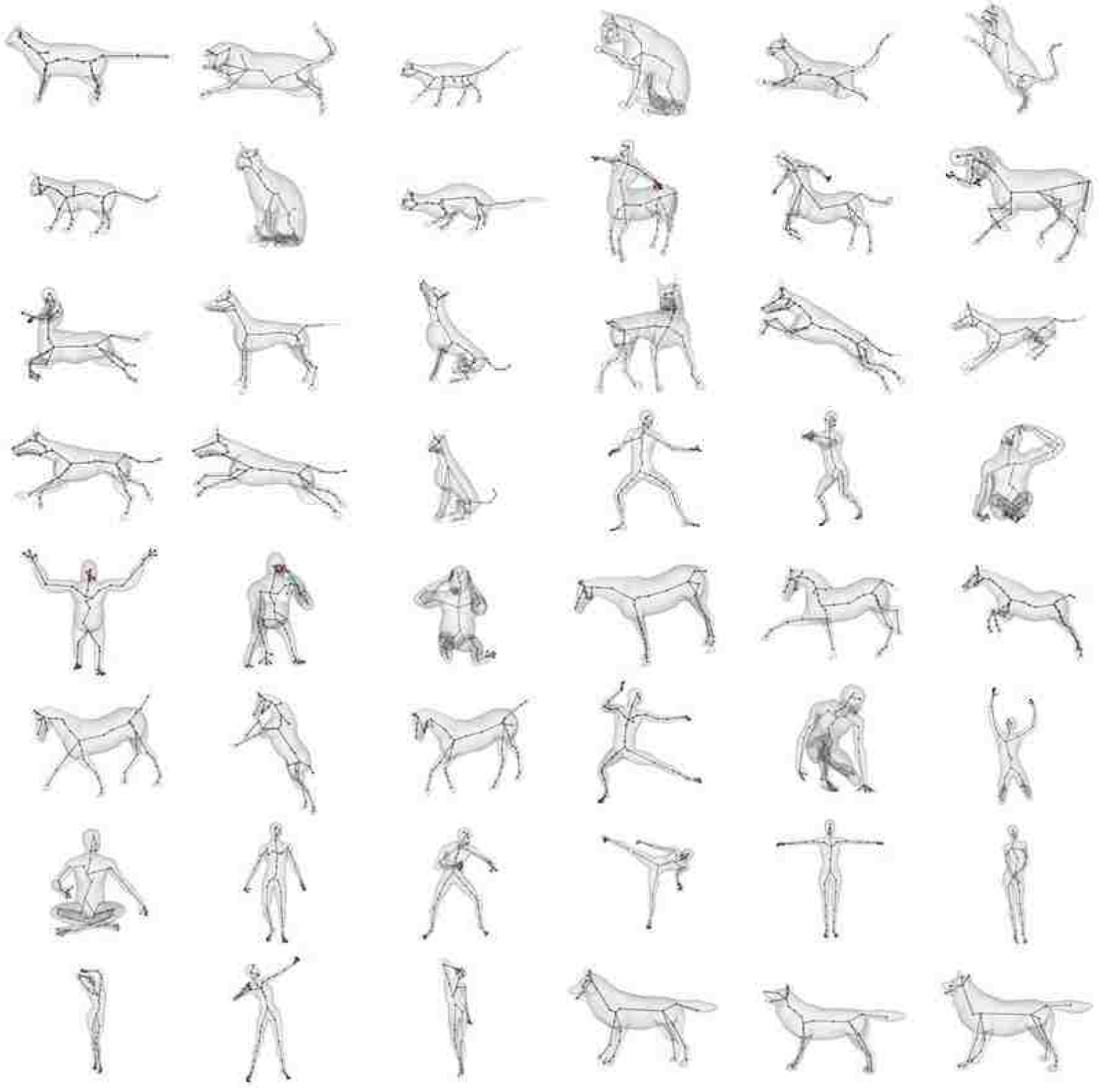


Figure 2.3: Visualizing Guarding of Models in TOSCA dataset.

more thorough comparison chart between greedy and PILP approaches on 48 models from the TOSCA dataset is depicted in Figure 2.2. As we can see in this table and the chart, PILP has similar optimality as the ILP solution, but is much faster; while compared with greedy approach, PILP has similar efficiency, but provides the guarding 20% better than that of greedy method on average. Considering that the greedy approach is generally a nice approximation for this problem, the guarding generated by PILP is very nice.

Table 2.2: Skeletal-Nodes Guarding versus Tetrahedral-Vertices Guarding. $|V_{\partial M}|$ and $|V_M|$ are numbers of boundary vertices and tetrahedral vertices, respectively; $|S|$ is the number of nodes on extracted skeletons; $|G_{V_M}|$ and $|G_S|$ are the sizes of computed guarding sets (solved by ILP) when using all tetrahedral vertices as candidates and using only skeletal nodes as candidates, respectively.

Models	$\lvert V_{\partial M}\rvert$	$\lvert V_M\rvert$	$\lvert S\rvert$	$\lvert G_{V_M}\rvert$	$\lvert G_S\rvert$
Kitten	400	1,682	122	3	3
Beethoven	502	2,895	88	2	2
Bimba	752	5,115	139	2	2
Buddha	502	3,002	155	2	2
Bunny	998	8,320	270	5	5

Discussion. It should be noted that theoretically our algorithm solves an approximate optimal solution. In our current computational framework, two aspects need to be considered on the approximation of the optimal guarding problem. (1) We enforce the *face visibility* of the guarding. This can be considered as an approximation to guarding all points on the boundary. However, when the boundary surface mesh is dense enough with respect to the geometry of the model, we usually can assume this guarding is accurate enough because it is unlikely to have a branch that blocks the interior region of a face while leaving its all three vertices visible. (2) We compute the guarding points from the curve skeleton. Because of this, our simplified problem setting is not guaranteed to get the optimal solution. We perform experiments to evaluate whether using guarding candidates from the skeleton leads to larger guarding point set. Table 2.2 shows results of these experiments: guarding using skeletal candidates produces the same optimality compared with guarding computed using all tetrahedral vertices. Therefore, our intuition of choosing guards from skeleton is experimentally justified; our approximation is close to the optimal solution.

2.3 STAR DECOMPOSITION

Guarding points are natural seeds to start region growing for the star decomposition. The sweep algorithm (Section 2.2.1) can be generalized to tetrahedral mesh vertices so that visibility among vertices and guards can be efficiently pre-computed. We start region growing from all guards while simultaneously preserving star-property on all subregions.

We start from the guarding points $G = \{g_i\}, i = 1, \ldots, K_g$, and assign a specific color-value c_i on each guard g_i. The growing procedure can then be illustrated as assigning a unique color c_i to each tetrahedron, so that at the end, the connected component in color c_i is a star shape guarded by g_i. We can grow sub-regions on the dual graph of the tetrahedral mesh using the following notations and operations.

Similar to the face visibility, we say that a tetrahedron is visible from a point g if all its four vertices are visible from g. Given a guarding point g and a tetrahedron t, we define the *visibly dependent* tetrahedral set $T(g, t) = \{t_i\}$ such that it contains all the tetrahedra on the way of the four ray segments $\overline{gv_j}, v_j \in t$ pass through. In other words, t is visible to g if $\forall t \in T(g, t)$ is visible to g.

We then denote the *one-ring faces* surrounding a vertex v as $F_n(v)$: $F_n(v) = \{f | \forall t, v \subset t, f \subset t, v \not\subset f\}$; and denote the neighboring tetrahedra of t as $T_n(t) = \{\cup_{v_j \in t} t' | F_n(v_j) \subset t'\}$; then we can define $\widetilde{T}(g, t) = T(g, t) \cap T_n(t)$. Intuitively, including tetrahedra in $\widetilde{T}(g, t)$ into a sub-region guarded by g prevents triangles in $F_n(v_j)$ from becoming the boundary of this sub-region (which could block the visibility of t from g). It is not difficult to further show that

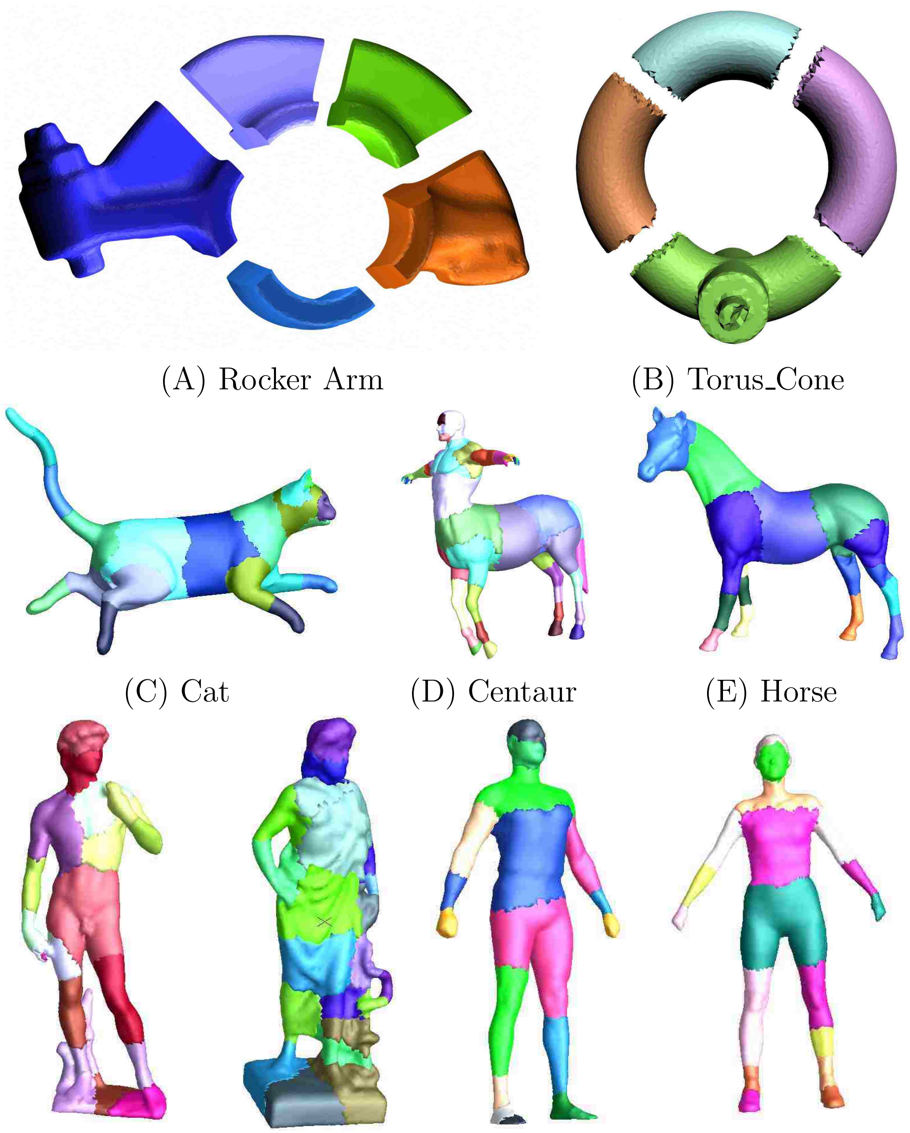
a) t is visible if $\forall t' \in \widetilde{T}(g,t)$ are visible.

b) t can be safely added into a sub-region M_g seeded in g without violating its star-property, if all $t' \in \widetilde{T}(g,t)$ are in M_g.

The dual graph D of the given tetrahedral mesh is defined in the following way: a **node** $n_i \in D$ is defined for each tetrahedron t_i. For a node n_i visible from g_k (with the color c_k), we create a **directed edge in color** c_k to n_i from another node n_j if $t_j \in \widetilde{T}(g_k, t_i)$; and we call n_j is a **color-c_k predecessor** of n_i. Since recursively, $T(g_k, t_i) = \widetilde{T}(g_k, t_i) \cup T(g_k, t_j)$, we only need to store each node's visibility dependency relationship. $\tilde{T}(g_k, t_i)$ can be computed in $O(1)$ time by checking the intersections of $\overline{g_k, v}$ and $F_n(v)$ for each $v \in t_i$.

For each guard g_k we generate a virtual node in D and connect it to nodes corresponding to all its one-ring tetrahedra. Then each guard g_k and its visible region defines a direct acyclic graph R_k. The entire 3D region guarded by K_g points $\{g_i\}$ corresponds to a connected graph with K_g sources. Each source g_i has an individual color c_i, the region growing assigns each node a unique color. A node n_j can be assigned by a color c only when all its color-c predecessors (on which n_j is visible dependent) are already assigned by color-c.

The region growing on the dual graph can be applied using the **node-merging**. When there is a color-c edge from a color-c node n_i to an uncolored node n_j, and all edges *entering* n_j are leaving from color-c nodes, then n_j can be safely colored by c. Therefore we can merge them together to one node in color-c, preserving all distinct outgoing edges. The region



(A) Rocker Arm (B) Torus_Cone

(C) Cat (D) Centaur (E) Horse

(F) David (G) Greek (H) Male (I) Female

Figure 2.4: Star Decomposition of Solid Models. Different subparts are rendered in different colors.

growing procedure is converted to iteratively merging each uncolored node to one of the K_g "growing" colored nodes. Therefore, the region growing algorithm can be summarized as follows.

Cost Function. For each node n_i, we can compute how many nodes are directly or indirectly visually dependent on n_i with respect to g_k. This can be pre-computed in linear time: after the dependency graph R_k is created from the source g_k to leafs, inversely the dependency cost can be accumulated and stored as $f(g_k, n_i)$.

Nodes Merging. We merge uncolored nodes with colored region based on the cost function $f(g, n)$. A node shall merge into a growing region that can see it and has biggest corresponding cost. This repeats until all nodes are colorized or no node can be further merged. If a node cannot be given the color of its any entering edges, it is left unclassified after the region growing. We collect each uncolored connected components, and respectively compute their guarding and re-apply the region growing until all tetrahedra are colored.

Computational Complexity. For each guard, the preprocessing step computes the visibility dependency in $\mathrm{O}(m_V \log m_V)$, where m_V is the number of tetrahedral vertices. Computing the visibility dependency of one tetrahedral vertex v takes $\mathrm{O}(K_n)$, where K_n is v's one-ring tetrahedra. Since $K_n < \log m_V$, the preprocessing time for each guard is $\mathrm{O}(m_V \log m_V)$. The region growing can be finished in $O(m_T)$, where m_T is the number of tetrahedra. So the total decomposition complexity is $\mathrm{O}(K_g m_V \log m_V + m_T)$, where K_g is the number of guards. For example, it takes about 550 seconds to perform the star decomposition on solid David (175,079 tetrahedra, 17 guards), including 400 seconds in guarding computation. Figure 2.4 illustrate our decomposition results on many solid models.

The visibility dependency relationship is the sufficient condition to guarantee the tetrahedron visibility. A tetrahedral sub-region that grows following this dependency relationship is guaranteed to be star-shaped. However, this constraint is stronger than necessary, especially when tetrahedral mesh is sparse, and some tetrahedra near inner partitioning boundary may not be considered acceptable during the region growing. In practice, in order to include these tetrahedra, we release this constraint by accepting a tetrahedron if two to three of its vertices are visible. Finally, we perform a Laplacian smoothing step on the inner-border of subregions after the region growing. This further moves these interior tetrahedral vertices, and the smoothed boundary improves the decomposition result: more than 99% tetrahedral vertices are visible from their corresponding guards.

2.4 APPLICATIONS

Our proposed guarding and decomposition framework can benefit many geometric processing tasks. In this section, we demonstrate two direct applications in computer graphics: shape matching and shape morphing [48].

2.4.1 Shape Matching and Retrieval

We define a descriptor for a shape based on its guarding. The descriptor has two parts: the guarding skeleton (or guarding graph) $\mathcal{G}$ and histograms $\mathcal{H}$ defined on nodes.

The guarding graph $\mathcal{G}$ is a graph extracted following the skeletal graph, whose nodes are the guarding point set $G = \{g_i\}$. At each guard g_i, we compute a histogram $\mathcal{H}(g_i)$ storing the distances from g_i to the object boundary surface towards a set of sampling directions. Specifically, the histogram is constructed as follows. From each guarding point p, we shoot

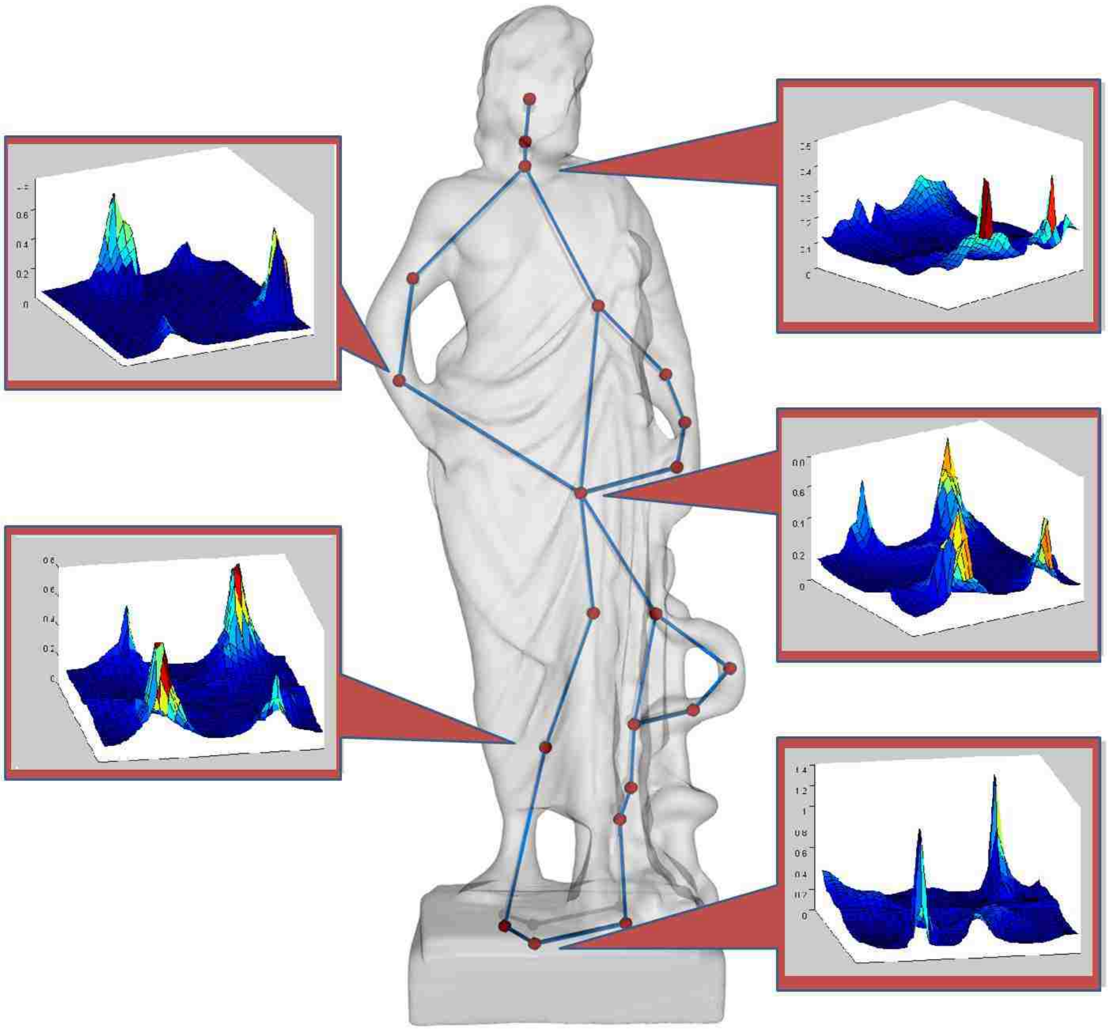


Figure 2.5: Shape Descriptor of Greek Sculpture. Each histogram stores distances from each guarding point to the boundary surface along sampling directions.

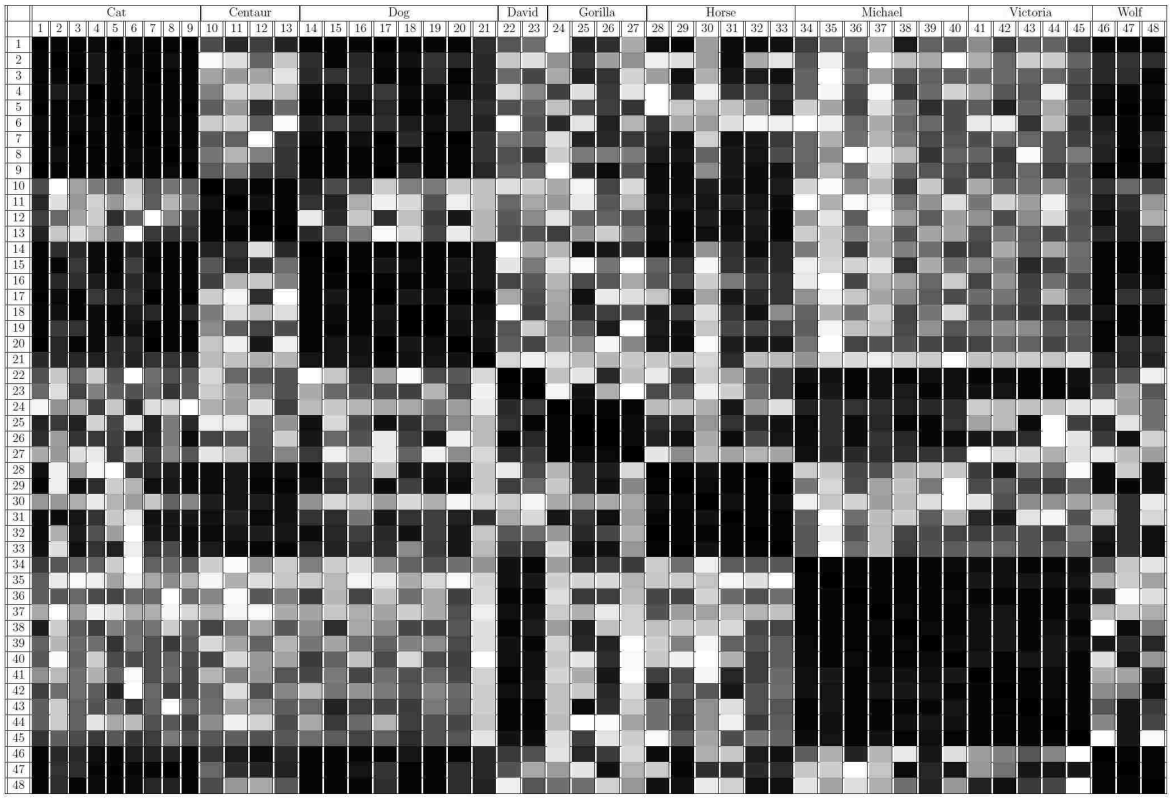


Figure 2.6: Shape retrieval experiment conducted in the TOSCA dataset of 48 models. Black indicates better similarity. Those blocks of black regions indicate the following groups are more similar: cats−dogs−wolves, David−Michael−Victoria, horses−centaurs, and ect.

rays $\{r_i\}$ towards all spatial directions defined on a unit sphere. Each ray r_i intersects with ∂M on a point q_i. The length of the line segment $\overline{pq_i}$ is stored in the histogram H_p. This histogram captures the geometry near this point. Fig. 2.5 illustrates the descriptor of the Greek sculpture.

The proposed descriptor has two good properties:

- *Completeness.* The geometry of the original shape can be completely reconstructed from its descriptor. Distance-to-boundary distributions nicely capture geometry characteristic of the solid shape and are suitable for the matching purpose. The guarding graph can visibly covers the entire region, so the shape descriptor is complete.
- *Conciseness.* The graph structure $\mathcal{G}$ has fewest necessary nodes because the guarding is optimized. Therefore, descriptor matching is efficient.

To compare two 3D models M and M', we match their guarding graphs $\mathcal{G} = \{G, E_{\mathcal{G}}\}$ and $\mathcal{G}' = \{G', E_{\mathcal{G}'}\}$, where vertex sets G, G' are guarding point set and edge sets $E_{\mathcal{G}}, E'_{\mathcal{G}}$ following the adjacency of the decomposition. We match them by solving a deformation of one skeleton to fit the other. The deformation is guided by a weighted energy $E_{\mathcal{G}}(\mathcal{G}, \mathcal{G}')$ composed of three terms: (1) the matching error E_M on each node, (2) the smoothness error E_S on the deviation of the transformations of two adjacent nodes, and (3) the length-preserving error E_L on each edge. Formally, suppose we define the affine transformation ϕ_i on each node g_i of the guarding graph $\mathcal{G}$, then these three terms are:

$$\begin{aligned} E_M &= \textstyle\sum_{g_i \in G} D(\phi_i(g_i), G')^2, \\ E_S &= \textstyle\sum_{[g_i,g_j] \in E_{\mathcal{G}}} ||\phi_i - \phi_j||_2, \\ E_L &= \textstyle\sum_{[g_i,g_j] \in E_{\mathcal{G}}} (||\phi_i(g_i) - \phi_j(g_j)|| - ||g_i - g_j||)^2, \end{aligned} \tag{2.4}$$

where $D(\phi_i(g_i), G')$ denotes the distance from the transformed point $\phi_i(g_i)$ to the guarding skeleton. In practice, we integrate two costs: (1) geometric distance, approximated using the distance from $\phi_i(g_i)$ to its closest point in G', and (2) topology distance, represented by the valence information of g_i. Therefore each node is represented as a 4-dimensional vector and the distance between two nodes is computed using the L_2 norm. During the optimization, this shortest distance can be efficiently recomputed using a k-d tree data structure. In E_S, the $||\phi_i - \phi_j||_2$ is the L_2 norm of the transformation matrix $\phi_i - \phi_j$. In E_L, the $||g_i - g_j||, ||\phi_i(g_i) - \phi_j(g_j)||$ denote the distance between adjacent points before and after the transformation.

The final objective function $E_{\mathcal{G}}(\mathcal{G}, \mathcal{G}')$ is a quadratic weighted sum of these cost functions:

$$E_{\mathcal{G}}(\mathcal{G}, \mathcal{G}') = \alpha_1 E_M + \alpha_2 E_S + \alpha_3 E_L, \tag{2.5}$$

where in our experiments we set $\alpha_1 = 0.1, \alpha_2 = 1, \alpha_3 = 1$. We compute transformations defined on all the nodes by minimizing $E_{\mathcal{G}}(\mathcal{G}, \mathcal{G}')$; the solution is a non-rigid mapping between $\mathcal{G}$ and $\mathcal{G}'$. The quadratic optimization can be solved efficiently.

After the transformation is computed, we add in the histogram matching error

$$E_H = \sum_{g_i \in G} ||\mathcal{H}(g_i) - \mathcal{H}(g'_j)||_2,$$

where g'_j denotes the closest point in G' under the previous matching. The difference between two histograms is again measured using the L_2 norm. The shape matching energy is

$$E(M, M') = E_{\mathcal{G}} + \alpha_4 E_H, \tag{2.6}$$

where we set $\alpha_4 = 0.5$. Finally, we use the symmetric energy $(E(M, M') + E(M', M))/2$ as the shape distance between M and M'.

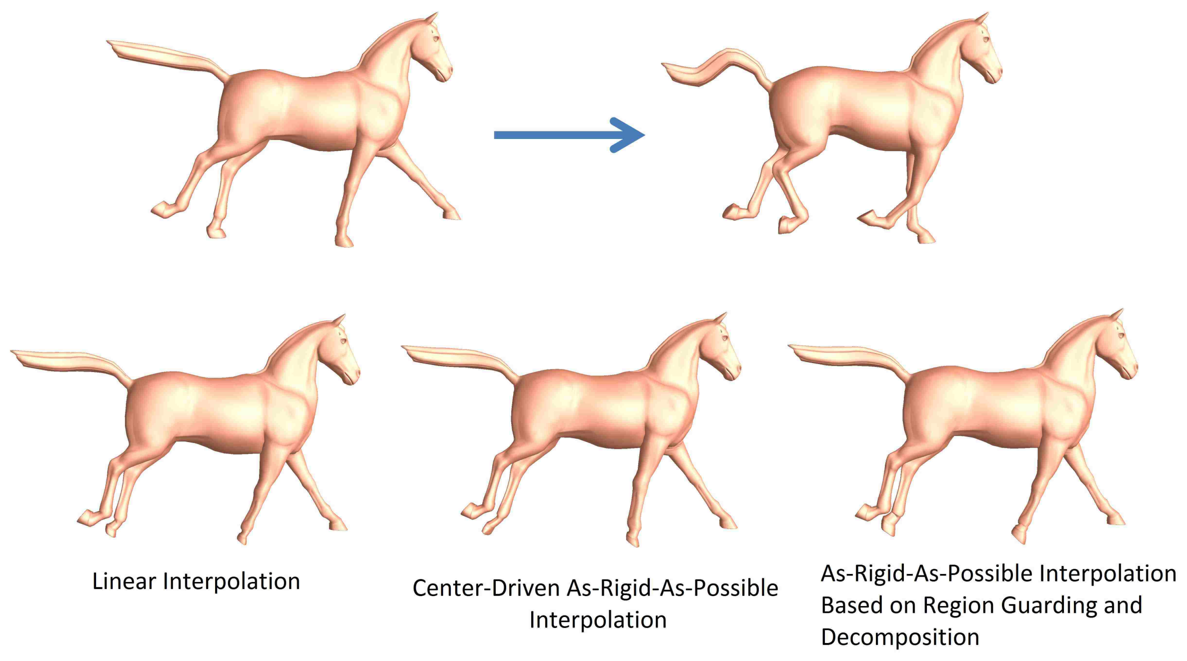

Linear Interpolation
Center-Driven As-Rigid-As-Possible Interpolation
As-Rigid-As-Possible Interpolation Based on Region Guarding and Decomposition

Figure 2.7: Shape Morphing. The top row shows the source and target shapes. The lower row shows the 50% shape interpolation.

Shape Retrieval

For all the shapes in the database we can pre-compute their guarding graphs and node histograms as their descriptors. Given a new object we compute its descriptor, then match it with existing descriptors following the above approach. The descriptor with the smallest matching error indicates the most similar object in the database. We perform shape matching and comparison on the TOSCA dataset. The comparison results are illustrated in Fig. 2.6, where black indicates small difference. The black blocks in this figure indicates several groups of models, although in different postures, share better similarity. For examples, cats−dogs−wolves, David−Michael−Victoria, horses−centaurs, and etc.

2.4.2 Shape Morphing

Given the source surface M_1 and target M_2, we want to compute an interpolation $M(t), 0 \leq t \leq 1, M(0) = M_1, M(1) = M_2$. Interpolation between M_1 and M_2 can be generated through the *consistent guarding*. Consistent guarding of M_1 and M_2 are two isomorphic graphs G_1 and G_2, such that G_1 (G_2) guards M_1 (M_2) respectively. The consistent guarding $\{G_1, G_2\}$ can be computed in three steps:

1) Compute cross-surface parameterization $f_M : M_1 \rightarrow M_2$ using surface mapping techniques (e.g. [49] [50]);
2) Extracting compatible skeletons (e.g. [51]) that bijectively corresponds the first curve skeleton C_1 (of M_1) to the second skeleton C_2 (of M_2), $f_C : C_1 \rightarrow C_2$;
3) Solve PILP simultaneously. We say $v_i \in M_1$ and $f_M(v_i) \in M_2$ are *simultaneously visible* to $p_j \in C_1$ and $f_C(p_j) \in C_2$, if both v_i is visible to p_j and $f_1(v_i)$ is visible to $f_2(p_j)$.

The solution found in Step-3 is two consistent guarding sets $\{G_1, G_2\}$. G_i may contain more guards than necessary to cover M_i, but the guarding points and their images consistently cover both models. This consistent guarding can generate consistent star decomposition which can benefit many applications. An example is shape interpolation.

Conventionally, shape morphing can be generated by linear interpolation: given inter-surface mapping $f_M : M_1 \rightarrow M_2$, the morphing for each vertex is generated by linear interpolation between $v_1 \in M_1$ and its image $f_M(v_1) \in M_2$: $v(t) = (1-t)v_1 + tv_2$.

With star decomposition, we can interpolate the corresponding regions to generate the morphing. A similar idea in 2D, based on star decomposition for 2D polygons and the interpolation of polar coordinates, is introduced in [52]. However, directly generalizing this to 3D by interpolating the spherical coordinates does not work well. We break the interpolation into the rigid part and the non-rigid part.

Rigid part. After the consistent star decomposition, we get the consistent surface segmentation $\{S_1, S_2\}$, for each subregion with m triangles $\{P_1, P_2, P_3, \ldots, P_m\}$ in S_1, there is a corresponding triangle set in S_2, $\{Q_1, Q_2, Q_3, \ldots, Q_m\}$, their guards are $g_{S_1} = (g^x_{S_1}, g^y_{S_1}, g^z_{S_1})$ and $g_{S_2} = (g^x_{S_2}, g^y_{S_2}, g^z_{S_2})$ respectively, for each corresponding triangle pair P, Q, we compute the Jacobian of the affine transformation A_T:

$$\begin{aligned} A_T &= \begin{bmatrix} p^x_1 - g^x_{S_1} & p^y_1 - g^y_{S_1} & p^z_1 - g^z_{S_1} \\ p^x_2 - g^x_{S_1} & p^y_2 - g^y_{S_1} & p^z_2 - g^z_{S_1} \\ p^x_3 - g^x_{S_1} & p^y_3 - g^y_{S_1} & p^z_3 - g^z_{S_1} \end{bmatrix}^{-1} \\ &\cdot \begin{bmatrix} q^x_1 - g^x_{S_2} & q^y_1 - g^y_{S_2} & q^z_1 - g^z_{S_2} \\ q^x_2 - g^x_{S_2} & q^y_2 - g^y_{S_2} & q^z_2 - g^z_{S_2} \\ q^x_3 - g^x_{S_2} & q^y_3 - g^y_{S_2} & q^z_3 - g^z_{S_2} \end{bmatrix} \end{aligned}$$

where $P = \{p_1, p_2, p_3\}$, $p_i = (p^x_i, p^y_i, p^z_i), i = 1, 2, 3$ is the the coordinates of the ith vertex of triangle P. Given a $t, 0 < t < 1$, we interpolate the Jacobian by polar decomposition [53]. Since a vertex may be shared by several triangle pairs, each triangle pair has a transformation, to keep the mesh consistent during the interpolation, we compute the interpolation vertex i position $v^i_r(t)$ by minimizing the quadratic error between the actual Jacobian and the desired ones, as stated in [54].

Non-rigid part. Excluding the rigid part transformation, we use a linear interpolation to blend the non-rigid deformation. For vertex i we compute the $v^i_n(1) = v^i_{M_2} - v^i_r(1)$, where $v^i_{M_2}$ is the target position, $v^i_r(1)$ is the rigid transformed position. Then we compute the non-rigid position $v_n^i(t) = tv_n^i(1)$. So the final interpolated position of vertex i is $v^i(t) = v_r^i(t) + v_n^i(t)$. Compared with linear interpolation and the as-rigid-as-possible interpolation [54] directly computed globally, our morphing based on star-decomposition could lead to less self-intersection and therefore generate more natural interpolation: A comparative example is shown in Figure 2.7. The source and target models are shown in the first row. In the second row, from left to right, the 50% morphing generated by linear interpolation, global center-driven as-rigid-as-possible interpolation, and our as-rigid-as-possible interpolation based on star decomposition are illustrated. Our result is natural, especially can be seen at regions in red circles.

2.5 CONCLUSION

In this paper we present an efficient progressive integer linear programming scheme to compute 3D shape guarding and star-decomposition. The proposed method is efficient and robust, which is demonstrated by extensive experiments. We also explore its effective computer graphics applications in shape retrieval and shape morphing. It is also used in autonomous pipeline inspection [55, 56].

Skeleton shape and skeletal nodes sampling are important for our guarding compaction. We will develop greedy or optimization strategies to further adjust them during the guarding computation. We will also improve our shape matching algorithm, and explore new applications of guarding and star decomposition.

3. GEOMETRIC-AWARE GRAPH PARTITIONING

In recent years, the acquisition or generation of large high-resolution geometric datasets pose new challenges to the design of effective processing algorithms. These big and complex data are expensive to model and analyze using existing sequential algorithms, as the limited CPU and memory are often insufficient to handle billions of elements efficiently. Parallel algorithms utilizing high-performance computers make it possible to solve large and complex problems efficiently on hundreds or thousands of computing clusters and is therefore desirable.

Divide-and-conquer is a natural and effective strategy in parallel mesh generation for large geometric data. The given region or object is first decomposed into a set of solvable and simplified subparts; then each subpart is distributed to a working processor for mesh construction; finally, individually generated meshes are merged to get the final result. Parallel mesh generation strategies such as Delaunay-based methods and advancing front techniques have been used for both triangulation [57–59] or tetrahedralization [60, 61]. Following this general paradigm, in this work we aim to develop a partitioning algorithm on comlex and large-scale 2D regions for parallel quadrilateral mesh generation.

In geometric processing through divide-and-conquer, the partitioning of data often directly dictates the algorithm's efficiency and quality. We formulate **three main criteria** as follows: (1) The areas of the subregions should be similar; (2) The boundary length of each subregion should be small compared with it's area; and (3) Each subregion should have desirable geometric property, and more specifically for quad meshing, it should have corner angles close to $k\pi/2, k \in \{0, 1, 2, 3\}$. The **first two criteria** determine the parallel performance: the balance of work load on different processors, and the communication cost among processors. Efficient parallelization should balance the workload distributed to different processors and should minimize interprocessor communication to reduce synchronization and waiting. The **third criterion** on the subregion's geometry affects the quality of the final quad tessellation. For example, on a square subregion one can generate a quad mesh where every element is uniform and not sheared; but on a triangle subregion, the smallest angle of the resultant quad mesh will inevitably be smaller than the smallest boundary angle of this triangle patch. Hence, for effective quad mesh generation, it is desirable to partition the geometry into subregions whose boundary angles are near $k\pi/2$ to construct less-sheared quad elements. To incorporate the geometric constraint in data partitioning, however, is sometimes prohibitively expensive. We will discuss this issue in Section 3.2 and propose a more efficient strategy to tackle it.

After data partitioning, we distribute subregions to different working processors for mesh generation. In our implementation, we use consistent boundary sampling and *advancing front* for parallel meshing on each subregion. The sub-meshes are finally merged together then a relaxation is performed to get the final result. The *main contributions* of this paper include:

- We study geometry-aware data partitioning for effective parallel mesh generation, and suggest new models to partition large and complex 2D regions into subparts with desirable geometric shapes for quad meshing. Compared with existing partitioning algorithms, our approach could lead to more efficient parallel processing and higher-quality meshing results.

- We develop a parallel computing framework for quad mesh generation of large-scale 2D regions. It can effectively utilize parallel computational resources to handle big geometric data. We demonstrate an application in coastal modeling where massive-size coastal terrains and oceans need to be discretized for simulations.

3.1 RELATED WORK

3.1.1 Data Partitioning

Given a geometric region M, a set of components $\{M_i\}$ is a *partition* of M if (1) their union is M, i.e., $\bigcup_i M_i = M$, and (2) all M_i are interior disjoint, namely, $\forall_{i \neq j} M_i^\circ \cap M_j^\circ = \emptyset$, where $M_i^\circ = M_i \backslash \partial M_i$ is the open set of M_i. Depending on the geometric processing applications, partitioning techniques consider different criteria accordingly. Thorough surveys on geometric partitioning algorithms have been given [18, 62] for computer graphics and geometric modeling applications; some data benchmarks [63] have been built for evaluating these methods in these graphics applications.

Slightly different from the partitioning criteria considered in graphics applications, in order to obtain effective data partitioning for parallel computing, partition strategies can be classified into two categories: *extrinsic space partitioning* and *intrinsic manifold partitioning* methods. We call the first strategy the *extrinsic space partitioning* method, because it partitions data by partitioning the data's embedding space. For example, data can be decomposed by spatial subdivision or partitioning structures such as quad-tree or octree [64], axis/planes [58], or blocks [65]. In parallel data processing literature, a very popular extrinsic space partitioning strategy is the *space filling curve* [66, 67]. The idea is to first fill the N-dimensional space with a 1-dimensional curve and establish a bijective map between cells in the space and curve segments passing them. Different regions (cells) in the space are therefore indexed by this curve, and partitioning of the space (therefore, partitioning of data) is obtained by partitioning the curve accordingly. In general, data (space) partitioning using space-filling curves or other extrinsic space partitioning methods is very efficient, as demonstrated in several successful applications, such as computational physics, algebraic multigrid, PDE solving, adaptive mesh refinement [68,69]. However, algorithms based on spatial partitioning are not suitable to handle data that have complex geometry or nonuniform properties.

We call the second category the *intrinsic manifold partitioning* method, and it partitions the data model on its intrinsic tessellation. The data are discretized into a mesh or a graph, where elements or nodes are clustered into subparts directly [70] or recursively [71]. Among this category, an effective and widely used strategy is called *graph partitioning* [2, 72–75], which usually produce good-quality partitions with balanced load and reduced communication. Solving the graph partitioning is NP-complete, and several effective strategies include spectral bisection [73], Kerninghan-Lin algorithm [72] and the multi-level scheme [2]. The spectral bisection algorithm [73] uses the spectral information to partition the graph. The eigenvector of the Laplacian matrix is computed to bisect the graph. Spectral Bisection usually produces a good partitioning, however it is very expensive to compute, especially for large matrix. On the other hand, the Kerninghan-Lin algorithm [72] is a fast heuristic scheme. It starts with an initial bipartition of the graph, then iteratively swaps a subset of vertices from each part to reduce the energy. This algorithm is fast, however, the initial partition is critical but often not easy to obtain. The multi-level method, including the widely adopted algorithm/software *METIS* [2], uses a three-phase scheme: first, the graph is simplified to a coarse graph; then a partitioning is performed on the coarsened graph; finally, the partitioning is modified during the progressive graph refinement. These existing graph partitioning algorithms focus on only tackling the load balancing and communication reduction issues. However, only considering these two criteria is insufficient. Incorporating extra geometric constraints is often critical in geometric modeling applications. The Medial Axis Domain Decomposition (MADD) algorithm [76] merges triangles to eliminate small angles then solves a graph partitioning on dual graph of the merged mesh. Subregions constructed using MADD partitioning usually possess larger corner angles than METIS. For our problem, it is desirable to have perpendicular corner angles. Therefore, a geometric term to incorporate this angle constraint can be formulated and included into the *graph partitioning* model. Furthermore, an additional connectivity constraint is needed to ensure each subregion form only one connected component. However, solving the original graph partitioning is already NP-hard, and the incorporation of these extra geometric constraints will further significantly increase the complexity of the problem (see Section 3.2 for details). This makes the efficient solving of this problem highly challenging.

3.1.2 Quadrilateral Mesh Generation

Quadrilateral mesh generation has been studied in computer graphics and geometric modeling fields. Quad meshes can be constructed through either the *indirect* or *direct* approaches. The *indirect* approaches first generate an intermediate structure/tessellation that can be easily constructed, e.g., a triangle mesh, then convert it into a quadrilateral mesh [77]. One big drawback of this method is that the layout of the unstructured elements in the intermediate tessellation determines the layout of final quad mesh, and there are usually a large number of singularities (i.e., non-valance-4 vertices) in the resultant quad meshes, which are usually undesirable for efficient simulation. The *direct* approaches construct quads on the model directly. Related techniques include quad-tree ([78]), template-guided decomposition ([79]) and advancing front([80]). A related problem is the quadrilateral mesh generation on curved surfaces. [12] gave a good survey on this topic. However, the curved geometry pose extra challenges in quad mesh generation and many recent surface quad meshing algorithms [81,82] use a cross frame field to guide the construction of the quad meshes. Finding an optimal cross frame field reduces to nonlinear integer programming, which is computationally expensive for large-scale geometric regions. Another related problem is the *quad layout patch* construction [83]. Its goal is to partition a surface into topologically rectangle patches, upon which quad meshes can be constructed. In this paper, we didn't adopt this strategy, because the topological constraint on restricting subregions to be "4-sided polygons" is very expensive to enforce.

3.2 REGION PARTITIONING

Data partitioning is the first step in a divide-and-conquer framework for parallel computational models. A good partitioning with balanced workload and small interprocess communication helps improve computational efficiency. Furthermore, in our parallel meshing problem, a good data partitioning is also critical in generating high-quality quad elements. We use three criteria to quantitatively evaluate a partitioning: **workload balance**, **total separator length**, and **separator angle deviation**. *Graph Partitioning* is a suitable strategy to solve our partitioning, because it can systematically model and optimize these criteria. In this section, we will first introduce the related notation (Sec. 3.2.1) and formulation of the three criteria (Sec. 3.2.2, 3.2.3 and 3.2.4) and the *Connectivity Constraints* (Sec. 3.2.5), then propose our algorithm in Sec. 3.2.6.

3.2.1 Notation

Given a tessellation $M = (V^M, E^M, F^M)$ of a 2D region, where V^M, E^M, F^M are the sets of vertices, edges, and faces (cells) respectively, let $G = (V^G, E^G)$ denotes its dual graph, where V^G, E^G are the node and arc sets respectively. A node $v \in V^G$ corresponds to a cell of M. Two nodes $v_1, v_2 \in V^G$ are connected by an arc if their corresponding cells are adjacent, namely, share an edge. Therefore, each arc of E^G also corresponds to an edge in E^M. The weight of a node $v \in V^G$ is defined to be the area of its associated cell $f \in F^M$, and an arc's weight is defined as the length of its associated edge. The partitioning on G can be computed on the dual graph G. A k-way *Graph Partitioning* divides V^G into k connected components, each of which is a subregion that will be processed individually. In practice, k-way partitioning is usually solved through recursive bisection [2]. Hence, we recursively partition G into two sub-graphs $G_0 = (V^{G_0}, E^{G_0})$ and $G_1 = (V^{G_1}, E^{G_1})$, where $V^{G_1} = V^G \setminus V^{G_0}$. This also partitions cells in the original tessellation M into two sets M_0 and M_1, if an edge $e \in M$ is shared by two cells f_i, f_j from distinct subsets, $f_i \in M_0, f_j \in M_1$, then edge e is called a **separator**.

For each node $v_i^G \in V^G$, we assign a variable x_i,

$$x_i = \begin{cases} 0, & \text{if } v_i^G \in V^{G_0} \\ 1, & \text{if } v_i^G \in V^{G_1} \end{cases}.$$

Then for each arc $e_{ij}^G = [v_i^G, v_j^G]$, we assign a variable $y_{ij} = x_{_i} - x_{_j}$:

$$y_{ij} = \begin{cases} 1 \text{ or } -1, & \text{if } v_i^G \text{ and } v_j^G \text{are in two sub-graphs} \\ 0, & \text{otherwise} \end{cases}.$$

We have $\mathbf{y} = \mathbf{U}\mathbf{x}$, where $\mathbf{x}$ and $\mathbf{y}$ are node and arc variable vector respectively, and $\mathbf{U}$ is a $|E^G| \times |V^G|$ matrix. With these variables, we formulate the three criteria as follows.

3.2.2 Workload Balance

Load balancing refers to the practice of distributing approximately equal amounts of work among tasks, so that all tasks are kept busy all of the time. Unbalanced workload between working processors leads to inefficiency in parallel computing, as the slowest task often determines the overall running time. In our problem, the workload on each working processor can be estimated by the area of each subregion to mesh. On the dual graph, the subregion area can be calculated using the sum of weights of nodes in the corresponding subgraph.

A balanced partitioning should avoid big area difference between subregions. Hence, it is formulated as the following constraint:

$$c_1 \le \mathbf{x}^T \mathbf{w}_v - c \le c_2, \tag{3.1}$$

where $\mathbf{x} = (x_1, x_2, \ldots, x_n)^T$ is the variable vector, $\mathbf{w}_v = (w_{v_1}, w_{v_2}, \ldots, w_{v_n})^T$ is the node weight vector, $c = \frac{1}{2}\sum_i w_{v_i}$, and c_1, c_2 are the constant thresholds (in our experiments, $c_1 = c_2 = 0.1c$).

After a k-way partitioning is obtained, its workload balance can be evaluate by the ratio between the areas of the largest and smallest subregions:

$$R_W = A_{max}/A_{min}, \tag{3.2}$$

where A_{max} and A_{min} are the areas of the largest and smallest subregions respectively. R_W close to 1 means better workload balance.

3.2.3 Total Separator Length

In parallel computing, inter-process communication means overhead. A smaller total separator length usually indicates less communication cost. In our geometric data partitioning, a smaller separator length also indicates (1) smoother subregion boarder lines, (2) better geometric compactness of subregions, and (3) more efficient post-processing refinement (see Section 3.3). Therefore, it is desirable to minimize the *total separator length*

$$L_S = \mathbf{y}^T \mathbf{W}_e \mathbf{y} = \mathbf{x}^T \mathbf{U}^T \mathbf{W}_e \mathbf{U} \mathbf{x}, \tag{3.3}$$

where $\mathbf{y} = (y_{e_1}, y_{e_2}, \ldots, y_{e_n})^T$ is the edge variable vector, $\mathbf{W}_e = \text{diag}(w_{e_1}, w_{e_n}, \ldots, w_{e_n})$ is a diagonal matrix composed of arc weights.

With the above two criteria, solving a graph partitioning can be formulated as:

$$\begin{aligned} \min \quad & \mathbf{x}^T \mathbf{U}^T \mathbf{W}_e \mathbf{U} \mathbf{x}, \\ \text{subject to} \quad & c_1 \le \mathbf{x}^T \mathbf{w}_v - c \le c_2, \\ & x_i \in \{0, 1\}. \end{aligned} \tag{3.4}$$

Optimizing problem (3.4) is NP-hard. For large data, multilevel schemes such as METIS [2] have been widely adopted to obtain good approximate solutions. Figure 3.1 illustrates a simple example. The input tessellation M is shown in (a) and the partitioning result by METIS [2] is shown in (b). The two partitioned subregions colored in blue and red respectively. We can see that the two subregions have the same area with total separator length minimized.

Minimizing the total separator length will makes the boundary of subregions straight, and it can increase the *compactness* of each subregion. On each subregion M_i, the *compactness* can be measured by the ratio between the boundary separator's length of M_i and the area of M_i. Globally, we can compute an *average compactness ratio*,

$$\hat{R}_C = \frac{L_S}{kA_M}, \tag{3.5}$$

where k is number of subregions, and A_M is the total area of the region. Smaller $\hat{R}_C$ comes from a smaller L_S and indicates better compactness.

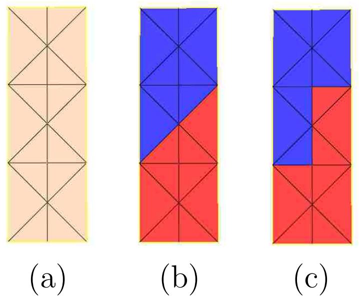


Figure 3.1: Graph Partitioning on a simple example. (a) The original mesh; (b) the two (red and blue) subregions obtained by METIS [2]; (c) the partitioning result with separator angle deviation considered.

3.2.4 Separator Angle Deviation

Solving the graph partitioning formulated in Eq. (3.4) will result in balanced area and minimized total separator length. However, in many geometric processing tasks, constraints on the geometry of subregions are important. In our quad meshing problem, ideally, each subregion should look like a square. Less strictly, since we use the advancing front technique to generate quad meshes (Section 3.3), it is desirable to have angles between separators close to $\frac{k\pi}{2}$. Therefore, we include a new separator angle term into graph partitioning to penalize each such angle's deviation from $\frac{k\pi}{2}$.

Angle Deviation Function. Consider the original tessellation M of the given 2D region, suppose two edges $e_i, e_j \in E^M$ share a vertex v and form an angle $\theta_{i,j}$. For concise representation, in the following, we use $Inc(i,j) = 1$ to denote e_i and e_j are incident, and $Inc(i,j) = 0$ means they are not incident. We define an *angle deviation function*

$$\delta_{\theta_{i,j}} = \begin{cases} \min_k\{|\theta_{i,j} - \frac{k\pi}{2}|, k \in \{0,1,2,3\}\}, & \text{if } Inc(i,j) = 1 \\ 0, & \text{if } Inc(i,j) = 0 \end{cases} \tag{3.6}$$

to describe the deviation from angle $\theta_{i,j}$ to the nearest $\frac{k\pi}{2}$ angle.

Accumulated separator angle deviation can then be formulated as

$$D_\theta = \mathbf{y}^T \mathbf{W}_\theta \mathbf{y} = \mathbf{x}^T \mathbf{U}^T \mathbf{W}_\theta \mathbf{U} \mathbf{x}, \tag{3.7}$$

where $\mathbf{y} = (y_{e_1}, y_{e_2}, \dots, y_{e_n})^T$ is the edge variable vector, and $\mathbf{W}_\theta = \begin{pmatrix} 0 & \delta_{\theta_{1,2}} & \delta_{\theta_{1,3}} & \dots & \delta_{\theta_{1,n}} \\ \delta_{\theta_{2,1}} & 0 & \dots & \dots & \delta_{\theta_{2,n}} \\ \dots & \dots & \dots & \dots & \dots \\ \delta_{\theta_{n,1}} & \dots & \dots & \dots & 0 \end{pmatrix}$,

$\mathbf{W}_\theta$ is an $|E^G| \times |E^G|$ matrix storing deviation angles $\delta_{\theta_{i,j}}$. This angle deviation matrix $\mathbf{W}_\theta$ can be precomputed by traversing all the edge pairs of the tessellation M once.

Furthermore, we can show that Eq. (3.7) *will evaluate and only evaluate the angle deviation between adjacent separators.* Suppose two edges $e_i = [u,v]$, $e_j = [u,w]$ are incident separators, sharing vertex u. Then, (1) $y_{e_i} \neq 0$ and $y_{e_j} \neq 0$, and (2) v and w belong to the same subregion and have a same indicator value, $x_v = x_w$. From (1) and (2), it is not difficult to see that y_{e_i} and y_{e_j} are both -1 or both $+1$. Therefore, a *non-negative contribution* $\delta_{\theta_{i,j}}$ will be added to the accumulated separator angle deviation D_θ. To obtain a geometrically desirable partitioning, we can minimize the separator angle deviation term (3.7) together with total separator length (3.3), with the workload balance constraint (3.1).

Finally, after the partitioning, we can numerically evaluate the *average separator angle deviation*:

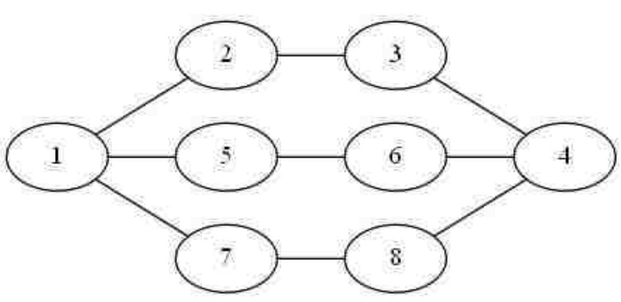
$$\hat{\delta}_\theta = \frac{1}{N_C}(\sum_{Inc(i,j)=1} \delta_{\theta_{i,j}}), \tag{3.8}$$

where N_C is the total number of corners formed by incident separators. The smaller $\hat{\delta}_\theta$ is, the better.

3.2.5 Connectivity Constraints

Currently we have formulated the three criteria. To guarantee the result is a bipartitioning, we need to impose the *connectivity constraint*, which is often not explicitly considered in existing graph partitioning literature, due to its complexity. Specifically, nodes in each subregion should form one connected component. Without explicitly enforcing this, although minimizing total separator length often tends to penalize the partition that produces multiple disjoint subsets, we can sometimes observe that more than one connected components exist in one subregion.

To enforce connectivity of each subregion, we can formulate the following explicit constraint. Given a (dual) graph $G = (V^G, E^G)$, for any pair of non-adjacent nodes u, v, we define a node set $S \subset V \backslash \{u, v\}$ to be a *node-cut set* that separates u and v, if there is no path between u and



h

Figure 3.2: An example graph for connectivity constraint.

v after S is removed from G. For example, in the graph shown in Fig 3.2, for node pair $\{1, 4\}$, $\{2, 5, 7\}$ is a node-cut set and $\{3, 6, 8\}$ is another node-cut set. For $\{u, v\}$ that $[u, v] \notin E^G$, we define $\Gamma(u, v)$ to be the set consisting of all the node-cut sets of $\{u, v\}$. In this example, $\Gamma(1, 4) = \{\{2, 5, 7\}, \{2, 5, 8\}, \{2, 6, 7\}, \{2, 6, 8\}, \{3, 5, 7\}, \{3, 5, 8\}, \{3, 6, 7\}, \{3, 6, 8\}\}$.

The connectivity constraint can be described as: between each pair of nodes u, v that are in the same subregion, any node-cut set in $\Gamma(u, v)$ must have at least one node being assigned to this subregion. Using the binary variable x_i defined previously, the connectivity constraints can be formulated as a set of linear constraints. For any two non-adjacent nodes in subgraph G_1:

$$\sum_{w \in S} x_w \geq x_u + x_v - 1, \forall S \in \Gamma(u, v), \text{for } \forall x_u = x_v = 1, [u, v] \notin E^G, \tag{3.9}$$

meaning that every node-cut set must have at least one node being assigned as 1. Similarly, for any two non-adjacent nodes in subgraph G_0:

$$\sum_{w \in S} x_w \leq x_u + x_v + |S| - 1, \forall S \in \Gamma(u, v), \text{for } \forall x_u = x_v = 0, [u, v] \notin E^G. \tag{3.10}$$

These constraints ensure that there is at least one path connecting non-adjacent nodes u and v if they are grouped to a same subgraph.

By incorporating both the *separator angle deviation* (Eq. 3.6) and *connectivity constraints*

(Eqs. 3.9,3.10) into graph partitioning, the quadratic integer programming problem with linear constraints can be solved through branch-and-bound algorithm. In our implementation, we use the open-source *Basic OpeN-source Mixed INteger* (BONMIN) solver from [84] to obtain a solution. To solve mixed integer non-linear programming problems, BONMIN allows one to choose optimization strategies including branch-bound, outer approximation (OA), Quesada Grossman branch-cut, and Hybrid OA based branch-cut.

Figure 3.1 (c) shows the solution of this new graph partitioning. With the minimization of separator angle deviation and the enforcement of connectivity constraints, a partitioning more suitable for quad mesh generation is obtained. However, because (1) incorporating separator angle deviation significantly increase the numbers of non-zero cross multiplication of indicator variables, and (2) the enforcement of connectivity introduces an exponential number of linear constraints. Solving such a new problem becomes prohibitively expensive for even moderately large problem.

3.2.6 Our Two-Step Partitioning Algorithm

We proposed a two-stage partitioning scheme to incorporate the two new constraints discussed in the last section. A key observation this idea based upon is that: *if the cells of the initial tessellation has near-square geometry, then the partition performed on the dual graph of this tessellation tends to have smaller separator angle deviation.* Hence, first, we use L_∞-CVT to generate a tessellation with cells similar to squares (Sec. 3.2.6); second, we solve our graph partitioning on this tessellation to get a set of subregions with balanced workload and small total separator length, with connectivity constraints enforced on the subgraphs during their refinement.(Sec. 3.2.6) We call this algorithm a L_∞-*CVT-GP* algorithm for short, and the idea is illustrated in Fig. 3.3.

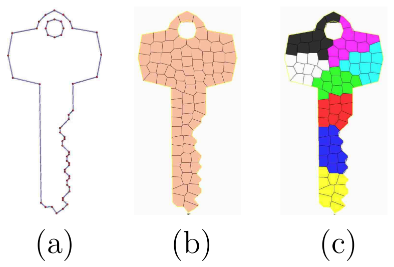


Figure 3.3: Partitioning a 2D "Key" Region for Parallel Quad Mesh Generation. (a) The input 2D boundary; (b) L_∞-norm CVT on the input boundary; (c) our partitioning result, with different subregions indicated using different colors.

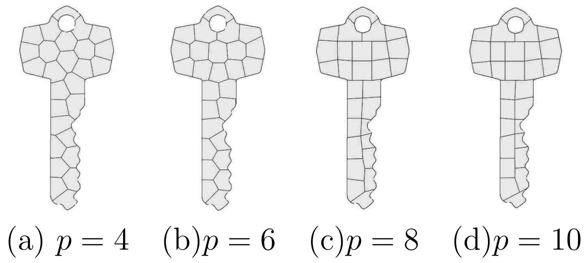


Figure 3.4: L_p-CVT using different p values. The results are similar when $p >= 8$. In our experiments, we use L_8-CVT to approximate L_∞-CVT.

L_∞ Centroidal Voronoi Tessellation The Voronoi diagram is a fundamental geometric structure widely used in various fields, especially in geometric modeling and computer graphics. A 2D *Voronoi Diagram* of a given set of distinct points $\mathbf{X} = \{\mathbf{x}_i\}_{i=1}^n$ in $\mathbf{R}^2$ is defined by a set of Voronoi cells $\{\Omega_i\}_{i=1}^n$ where

$$\Omega_i = \{\mathbf{x} \in \mathbf{R}^2 | \; \|\mathbf{x} - \mathbf{x}_i\| \leq \|\mathbf{x} - \mathbf{x}_j\|, \forall j \neq i\}.$$

These points $\mathbf{X}$ are called *sites*. Each Voronoi cell Ω_i is the intersection of a set of half-planes. A *Clipped Voronoi Diagram* for the sites $\mathbf{X}$ within a given 2D domain Ω is the intersection of the Voronoi Tessellation and the domain Ω, denoted by $\{\Omega_i|_\Omega, i = 1, \ldots, n\}$, where

$$\Omega_i|_\Omega = \{\mathbf{x} \in \Omega | \; \|\mathbf{x} - \mathbf{x}_i\| \leq \|\mathbf{x} - \mathbf{x}_j\|, \forall j \neq i\}. \tag{3.11}$$

In other words, $\Omega_i|_\Omega = \Omega_i \cap \Omega$. The *Centroidal Voronoi Tessellation* (CVT) is that each site of the Voronoi cell is coincident to this cell's centroid.

The L_p Centroidal Voronoi Tessellation Energy [85] can be defined as:

$$F(\mathbf{X}) = F([\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n]) = \sum_i \int_{\Omega_i \cap \Omega} \|\mathbf{y} - \mathbf{x}_i\|_p^p d\mathbf{y} \tag{3.12}$$

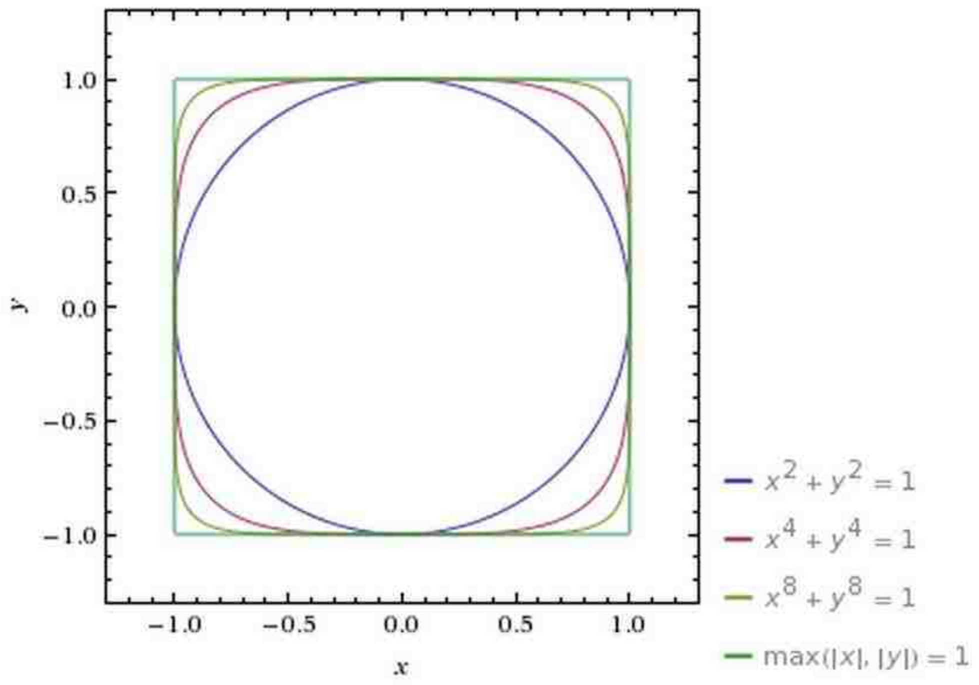


Figure 3.5: The trajectory of $L_p(x, y) = 1$ with different p values. With the increase of p the trajectory gradually approximates the unit square. $L_\infty(x, y) = 1$ gives a square.

where $\|\mathbf{z}\|_p = (\sum_j |z_j|^p)^{\frac{1}{p}}$, z_j is the j-th coordinate of a 2D point $\mathbf{z}$. Figure 3.5 shows the trajectory with different p. With the increase of p, the trajectory approximates to the square. The trajectory of L_∞ is a perfect square, where the L_∞ norm of a d-dimensional vector $\mathbf{z}$ is the maximal component in $\mathbf{z}$, $||\mathbf{z}||_\infty = \max_j |z_j|$. For efficient CVT computation, we choose a sufficiently large p to approximate the L_∞ norm. This also allows us to efficiently compute the gradient of $F(\mathbf{X})$ of Eq. (3.12). We test L_p-CVT on the key model using different p values. And as the results illustrated in Figure 3.4, when $p > 8$, the difference of L_p-CVT energy becomes very small, so we use L_8-CVT to approximate L_∞-CVT in all our experiments. Since we use L_8 which is smooth, the optimization of CVT energy F can be solved efficiently using the quasi-Newton BFGS solver [86]. In summary, the algorithm to compute the L_p-CVT on the input 2D region Ω has four steps.

1. Get a uniformly sampled sites set $\mathbf{X}$. In our implementation, we simply embed Ω on a spatial grid, the grid points inside Ω are the initial sites.
2. Use the sweeping line algorithm [87] to construct the initial Voronoi Diagram.
3. Get the L_∞-CVT tessellation by optimizing CVT energy.
4. Clip the CVT using $\partial\Omega$ to get the L_∞-CVT of Ω.

Graph Partitioning Based on L_∞-CVT

After solving the L_∞-CVT, we get a tessellation $\widetilde{M}$ of the 2D region. Then on $\widetilde{M}$'s dual graph $\widetilde{G}$, we solve the graph partitioning formulated in Eq. (3.4). Following the heuristics used in [2], instead of explicitly enforcing the large number of connectivity constraints in Eqs. (3.9, 3.10), we can adopt a region growing examination to check the connectivity of each subregion, disconnected elements will be grouped into the other subregion automatically. During our optimization, after every K iterations, we perform such an examination and update on $\widetilde{G}$. In practice, this strategy is efficient and effective in enforcing the connectivity of subregions. We test different partitioning results on the Key model using (a) original graph partitioning by METIS, (b) geometry-integrated graph partitioning solver introduced in Sections 3.2.4 and 3.2.5, and (c) this two-step L_∞-CVT-GP algorithm. We also evaluate the workload balancing ratio R_W, average compactness $\hat{R}_C$, and average separator angle deviation $\hat{\delta}_\theta$ on these partition results. Using these three partitioning methods, R_W are 1.13, 1.14, and 1.18, respectively; $\hat{R}_C$ are 1.11, 1.15, and 1.18, respectively, and $\hat{\delta}_\theta$ are 0.26, 0.12, and 0.15, respectively.

As expected, the graph partitioning without geometric constraint focuses on workload balance and separator lengths, and gets best R_W and $\hat{R}_C$, but very bad angle deviation. The expensive geometry-integrated graph partitioning produces smallest angle deviation. This two-step L_∞-CVT-GP produces results with slightly worse R_W, $\hat{R}_C$, and $\hat{\delta}_\theta$ than that from the geometry-integrated graph partitioning. But its separator angle deviation is significantly better than the original graph partitioning, and its speed is significantly faster than geometry-integrated graph partitioning. More thorough comparisons will be given in Section 3.4.

Through the L_∞-CVT, we can also directly obtain a region partitioning. Is such a partitioning sufficient, so that we no longer need to further solve a graph partitioning? This section illustrates that the two-step L_∞-CVT-GP algorithm usually leads to a better data partition for the parallel meshing problem. One observation is that: with the *increase of the number of Voronoi sites/cells, the decomposition from L_∞-CVT will become more uniform (better workload balancing) and more square-like (smaller separator angle deviation).* We have performed extensive experiments on L_∞-CVT to verify this. Table 3.1 is the partitioning statistics on the key and the pipe model. If we directly evaluate the quality of the partitioning suggested by the CVT decomposition, when the number of Voronoi cells increases from 16 to 128, the workload balancing ratio R_W (Eq. (3.2)) reduces from 1.65 to 1.23, while the average separator angle deviation $\hat{\delta}_\theta$ also reduces from 0.26 to 0.18.

Table 3.1: Partitioning on the Key and Pipe models using direct CVT versus using our CVT-GP algorithm. N_S is the number of subregions, $R_W, \hat{R}_C$, and $\hat{\delta}_\theta$ are the workload balance ratio, average compactness, and average separator angle deviation.

	Key			Pipe		
	CVT/CVT-GP			CVT/CVT-GP		
N_S	16	32	128	32	64	256
R_W	1.65/1.15	1.45/1.24	1.23/1.11	1.81/1.31	1.65/1.31	1.41/1.27
$\hat{R}_C$	1.21/1.11	2.21/ 2.01	9.41/9.24	1.82/1.77	4.04/3.88	18.11/17.84
$\hat{\delta}_\theta$	0.26/0.21	0.21/0.16	0.18/0.17	0.23/0.20	0.18/0.15	0.14/0.12

We have the following observations.

- For a k-way partition, direct partitioning through a L_∞-CVT decomposition with k cells will lead to a worse partitioning result than our two-step algorithm (which first generates n cells ($n > k$) then performs a graph partition to get k subregions).
- In order obtain a partition with similar R_W and $\hat{\delta}_\theta$, direct L_∞-CVT decomposition should use more sites. But this will increase the total separator length, resulting in bigger overhead and more singularities.

And we conclude that the L_∞-CVT-GP algorithm offers a better partition than the direct L_∞-CVT decomposition. More comparisons will be given in Section 3.4.

3.3 QUAD MESH GENERATION

3.3.1 Parallel Quad Meshing on Subregions

After the entire 2D region is partitioned, subregions can be sent to different processors for simultaneous mesh generation. Different quad meshing techniques can be applied on the sub-regions. One requirement is to make sure the neighbouring subregion boundaries should have consistent vertices. We use the **advancing front technique** [80] to tile the interior regions with quads. To ensure the individually constructed sub-meshes can be composed directly, we need to sample boundary vertices consistently on the shared edge of adjacent subregions. We use a simple sampling scheme: first, we compute an average edge length $\bar{l}$ from input boundary line segments, then on each interior partitioning boundary curve S we evenly sample $\frac{l_S}{2\bar{l}}$ points, where l_S is the length of S. To ensure the valid generation of a full quad mesh, the number of the points on the subregion boundary must be even [80], therefore, each interior separator line segments is subdivided into two to ensure the number of sampled boundary vertices is even.

The advancing front algorithm initiates a wave front from the boundary of each subregion, along which quadrangles are constructed inwards until all empty regions are tiled. We implement the advancing front following [80]. The wave front propagates until the front has 6 or fewer vertices, by when a template is used to finish the quad mesh generation. Readers are referred to [80] for details.

3.3.2 Post-processing after Composition

After composing meshes of subregions, we perform a Laplacian relaxation on vertices near separators to improve the smoothness of the orientations of mesh elements on the partitioning boundary. Each vertex moves towards the mass center of its neighboring vertices. Since our partitioning algorithm minimizes the total separator length, this relaxation only applies to a small number of vertices and takes a short time to process. In our experiments, up to 50 iterations are applied to each vertices within a five-ring buffer zone surrounding separators. In our implementation, we didn't parallelize this post-processing. But naturally, this refinement can be easily parallelized if needed.

3.4 EXPERIMENTAL RESULTS

We perform experiments on our high performance computing clusters, SuperMIC, which consist of 128 computing nodes. Each node has two 2.6GHz 8-Core Xeon 64-bit Processors and 32GB memory. Five datasets have been tested: a key model with 21.6M boundary segments and 1 inner hole, a pipe model with 57.6M boundary segments and 9 holes, and three coastal ocean/terrain regions : the Gulf of Mexico, Matagorda Bay, and West bay, with 230M, 250M, and 550M boundary vertices, respectively. The generated meshes of the two bays and the Gulf of Mexico have about 3, 4.5, and 10 billion elements, respectively.

We compare algorithms in three aspects: (1) **Decomposition Quality**: the workload balance ratio, total separator length, and separator angle deviation. (2) **Parallel Computation Efficiency**: the running time on each working processors and the total speedup in quad meshing. (3) **Meshing Quality**: the scaled-Jacobian of quad elements and number of singularities of the final mesh.

Our experiments are designed to compare 4 decomposition methods: (1) Partitioning via direct L_∞-CVT, (2) Partitioning by METIS [2], a very widely used Graph Partitioning solver, (3) Partitioning by Medial Axis Domain Decomposition (MADD) [76], and (4) Our L_∞-CVT-GP algorithm.

3.4.1 Partitioning Quality Comparison

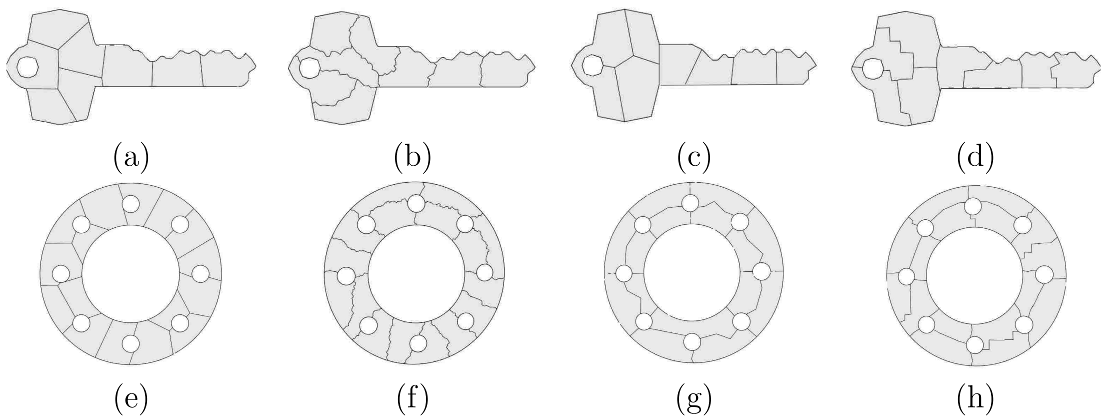


Figure 3.6: (a-d) Partitioning the key model into 8 subregions using direct L_∞-CVT, METIS, MADD, and our L_∞-CVT-GP algorithm. (e-h) Partitioning the pipe model into 16 subregions using direct L_∞-CVT, METIS, MADD and our algorithm.

Figure 3.6 illustrates the four partitioning results on the key and pipe models. Table 3.2 gives

Table 3.2: Decomposition quality comparison (the key and pipe model): direct L_∞-CVT, METIS, MADD and our L_∞-CVT-GP method (initialized 4000 CVT Cells). N_S is the number of subregions. Our method has comparable workload balance and average compactness, while our $\hat{\delta}_\theta$ is up to about 50% and 55% smaller than the METIS method on key and pipe model respectively; our $\hat{\delta}_\theta$ is 20% and 15% smaller than MADD on two models respectively. Compared with L_∞ method, our R_W is 40% smaller.

Key	L_∞-CVT		METIS		MADD		Our Method (4000 cells)	
N_S	128	1024	128	1024	128	1024	128	1024
R_W	2.01	1.92	1.03	1.02	1.13	1.08	1.06	1.04
$\hat{R}_C$	13.25	104.3	12.21	93.31	14.26	104.91	13.57	105.21
$\hat{\delta}_\theta$	0.18/0.15	0.15/0.14	0.32/0.26	0.41/0.21	0.32/0.20	0.25/0.19	0.17/0.15	0.15/0.14
Pipe	**L_∞-CVT**		**METIS**		**MADD**		**Our Method (4000 cells)**	
N_S	128	1024	128	1024	128	1024	128	1024
R_W	2.09	1.98	1.05	1.04	1.25	1.13	1.13	1.10
$\hat{R}_C$	4.76	36.51	4.64	36.37	5.06	35.48	4.71	36.42
$\hat{\delta}_\theta$	0.18/0.15	0.15/0.15	0.32/0.25	0.42/0.21	0.32/0.20	0.25/0.21	0.15/0.12	0.13/0.11

the partitioning statistics for key and pipe model respectively. The workload balance ratio R_W is calculated following Eq. (3.2). When R_W is closer to 1, a better workload balance is achieved. The average compactness $\hat{R}_C$ is calculated following Eq. (3.5). The average separator angle deviation $\hat{\delta}_\theta$ is calculated following Eq. (3.8). Note that for each of these three terms, *the smaller the measured value is, the better.* From these experiments, we can see that:

1) For *workload balance*: METIS leads to the smallest R_W. The R_W of our method is about 5% bigger than METIS. R_W of MADD is about 10% bigger than ours. The direct CVT partitioning has worst workload balance and its R_W is about 55% bigger than ours.

2) For *average compactness* $\hat{R}_C$: METIS also leads to the most compact subregions. The $\hat{R}_C$ of our method is about 8% bigger than METIS, but 6% smaller than MADD, and about 2% smaller than direct CVT partitioning.

3) For *average separator angle deviation* $\hat{\delta}_\theta$: our algorithm has the smallest separator angle

deviation. Our $\hat{\delta}_\theta$ is 40% smaller than METIS, 20% smaller than MADD, and about 5% smaller than direct CVT partitioning.

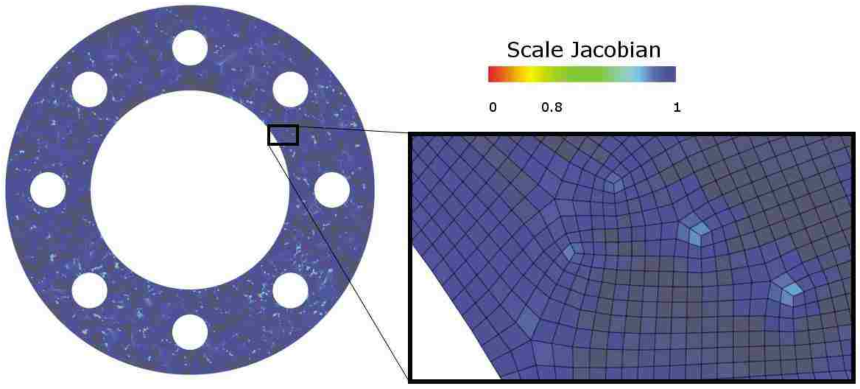


Figure 3.7: Quad meshing result of the Pipe model, color-coded by elements' scaled Jacobian values. The color from blue to red indicates the mesh quality from high to low.

In Conclusion, our algorithm results in significantly smaller separator angle deviation than the METIS and MADD method, while preserving good workload balance and compactness. Compared with the direct CVT decomposition, our algorithm yields 40% smaller workload balance ratio, while have slightly better the compactness and the separator angle deviation small. This indicates that meshing based on our decomposition is about 20% faster than that using CVT decomposition. To achieve similar workload balance, the CVT method needs to use much more (in our experiments, more than 4 times) cells. Then as a side effect, this will lead to a significant increase in L_S and in singularities of the final quad mesh (see Section 3.4.3), which is undesirable.

3.4.2 Parallel Computational Efficiency

Sub-regions are distributed to different working processors for simultaneous quad mesh generation using advancing front. We test the parallel computation efficiency on our datasets:

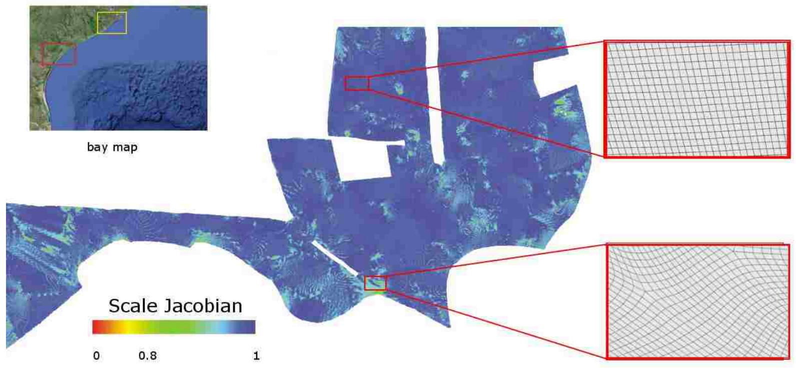
the pipe model (Figure 3.7), the terrain near West Bay (Figure 3.8(a)), and the terrain near Matagorda Bay (Figure 3.8(b)) and the entire ocean region of the Gulf of Mexico (Figure 3.8(c)). Figure 3.9(a) shows the actual meshing time on each working processor. The working time on different processors are very balanced during the parallel execution.

Table 3.3: The Runtime Table for different partitioning algorithms. N_S is the number of subregions. The runtime is in minute, and includes the partitioning, meshing and relaxation time. Usually, the METIS is fastest, and our method has the second best efficiency.

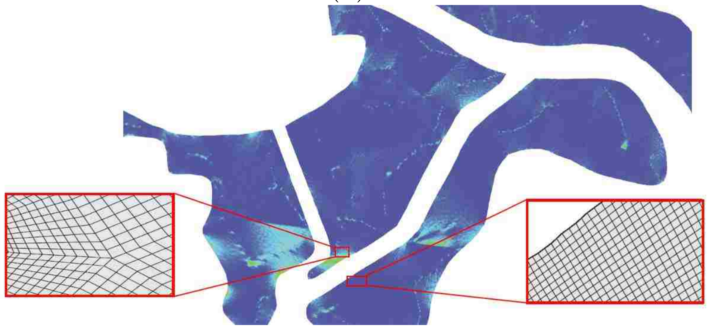
Model	Key				Pipe			
N_S	16	64	256	1024	16	64	256	1024
L_∞-CVT	13.92	4.51	1.59	0.60	16.55	5.43	1.94	0.74
METIS	13.09	4.24	1.49	0.56	15.54	5.10	1.82	0.70
MADD	13.69	4.43	1.56	0.59	16.27	5.34	1.90	0.73
Our	13.60	4.40	1.55	0.58	16.07	5.28	1.88	0.72

Model	Matagorda				West Bay				Gulf of Mexico			
N_S	16	64	256	1024	16	64	256	1024	16	64	256	1024
L_∞-CVT	19.25	6.49	2.26	0.91	19.14	6.50	2.18	0.95	26.07	8.55	3.11	1.28
METIS	18.13	6.11	2.13	0.86	18.08	6.14	2.06	0.90	24.42	8.01	2.91	1.20
MADD	18.94	6.38	2.22	0.90	18.92	6.42	2.15	0.94	25.51	8.36	3.04	1.25
Our	18.78	6.33	2.20	0.89	18.70	6.35	2.13	0.93	25.27	8.29	3.01	1.24

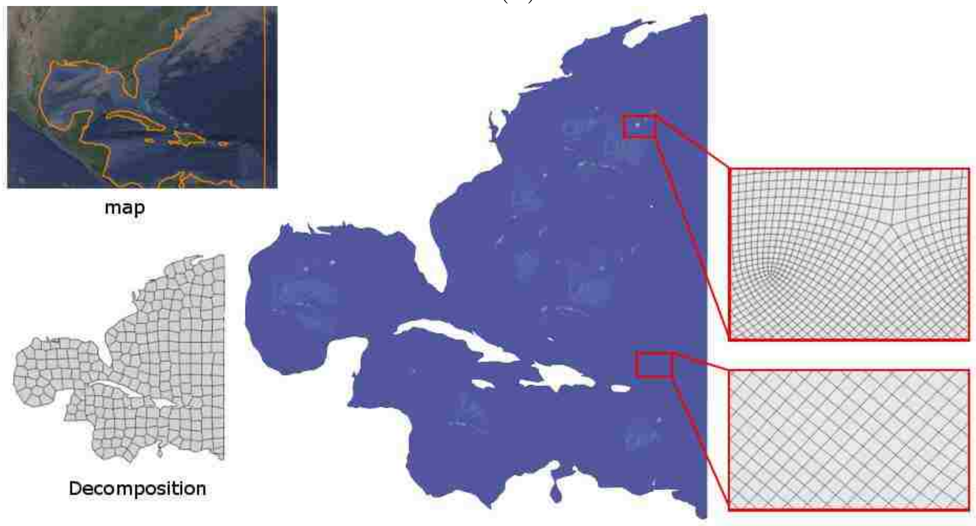
We also run the experiments by changing the number of working processors. The total run time is shown in Table 3.3. METIS leads the fastest total running time, Our algorithm is about 7% slower than METIS, 1% and 3% faster than MADD and direct CVT respectively. Ideally, when the computation is evenly distributed to each processor, the speed up would be T/D, where T is the total time cost of our algorithm without parallelization, and D is the number of processors. But there are overheads of this divide-and-conquer pipeline, including the partitioning, data transmission, and post-processing. The workload balancing also affects the speed-up performance. Figure 3.9 (b) plots the speedup of our algorithm on different models. The yellow dot line is the ideal speedup, and our algorithm s' parallel speed up ranges from 2.78 to 601.5 when 4 to 1024 processors are used, respectively. The



(a)

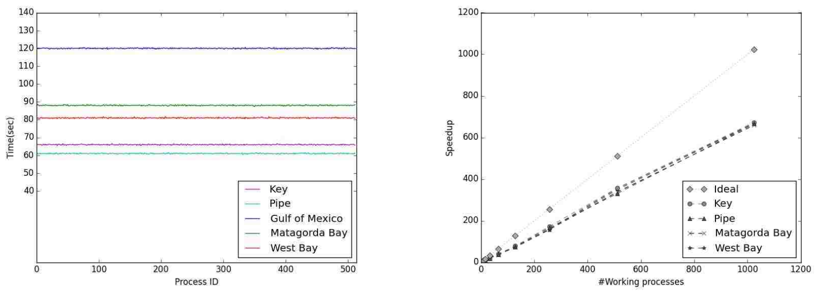


(b)



(c)

Figure 3.8: (a, b, c) The quad meshes for West Bay (yellow region in bay map), Matagorda Bay (red region in bay map), and the Gulf of Mexico respectively. The quad meshes are color-coded in scaled Jacobian.



(a) (b)

Figure 3.9: (a) The load balance for 512 working processes of the meshing of our dataset. (b) The parallel speedup of meshing: we test our algorithm using 4 to 1024 working processes. The yellow dot line is the ideal speedup, and the speed up of our algorithm on different models ranges from 2.78 to 601.5.

partitioning time usually takes up to 30% of the total time. To improve the meshing quality near the separator, we apply a Laplacian relaxation [88] near the separators after individually generated meshes are merged. Hence, a smaller L_S will reduce this post-processing time. In our experiments, the relaxation takes about 4% of the total computation time.

3.4.3 Meshing Quality Comparison

Table 3.4 compares the quality of final quad meshes generated by the sequential algorithm and four parallel algorithms using different partitioning methods. The sequential algorithm applies the advancing front to generate quad mesh without partitioning. For large model such as Matagorda Bay, West Bay and Gulf of Mexico the sequential algorithm failed to get a result. For each quad mesh, we compute four values, the (1) average, (2) standard deviation, and (3) minimum of scaled Jacobian, and (4) the number of singularities. Ideally, the scaled Jacobian should be 1. The scaled Jacobian of our mesh is comparable to the sequential algorithm, but we have 20% more singularities. Compared with other partitioning techniques, our average and minimal scaled Jacobians are 10% and 4% better than METIS

Table 3.4: Mesh Quality Comparison between the sequential meshing algorithm, L_∞-CVT, METIS, MADD, and our algorithm. The sequential algorithm applies the advancing front without partitioning; and it only works on small models such as the Key and Pipe. N_S is the number of subregions. The four values: (1) average, (2) standard deviation, and (3) minimum of the scaled Jacobian, and (4) the number of singularities are listed to show meshing quality. The scaled Jacobian of our mesh is comparable to the sequential algorithm, but we have 20% more singularities. The average and minimal scaled Jacobians of our algorithm are 10% and 4% better than METIS and MADD, respectively. Our singularities are 40% and 20% fewer than METIS and MADD. Compared with L_∞-CVT, our algorithm leads to about 4% better average and minimal scaled Jacobians and 8% fewer singularities.

Model (N_S)	Sequential Algorithm
Key (32)	0.97 / 0.13 / 0.39 / 35
Pipe (16)	0.97/ 0.04 / 0.31 / 34
Matagorda Bay (512)	-
West Bay (512)	-
Gulf of Mexico (1024)	-

Model (N_S)	L_∞-CVT	METIS
Key (32)	0.98 / 0.14 / 0.38 / 233	0.93 / 0.27 / 0.23/ 411
Pipe (16)	0.97 / 0.04 / 0.30 / 55	0.94 / 0.09 / 0.23 / 128
Matagorda Bay (512)	0.97 / 0.04 / 0.35 /325	0.93 / 0.16 / 0.21 / 413
West Bay (512)	0.96 / 0.05/ 0.36 / 235	0.93 / 0.12 / 0.24 / 562
Gulf of Mexico (1024)	0.97 / 0.04 / 0.36 / 3158	0.96 / 0.15 / 0.21 / 3491

Model (N_S)	MADD	Our Method
Key (32)	0.97 / 0.18 / 0.23 / 226	0.98 / 0.13 / 0.38 / 216
Pipe (16)	0.95 / 0.07 / 0.23 / 84	0.97 / 0.04 / 0.30 / 50
Matagorda Bay (512)	0.96 / 0.08 / 0.27 /346	0.97 / 0.04 / 0.35 /305
West Bay (512)	0.95 / 0.07 / 0.28 / 321	0.98 / 0.05/ 0.36 / 225
Gulf of Mexico (1024)	0.96 / 0.09 / 0.23 / 3201	0.98 / 0.04 / 0.36 / 3104

and MADD respectively. Our singularities are 40% and 20% fewer than METIS and MADD. Compared with L_∞-CVT, our algorithm leads to about 4% better average and minimal scaled Jacobians and 8% fewer singularities. These experiments show that our algorithm produces higher-quality quad meshes than other partitioning algorithms.

3.5 CONCLUSION

We present a parallel quad mesh generation pipeline for large-scale 2D geometric regions. A main contribution of this work is the solving of data partitioning with effective incorporation of the geometric constraint on angles between separators. After partitioning, subregions are distributed to different processors for parallel mesh generation through advancing front. Finally, after composition, post-processing is performed near partitioning boundaries for refinement. We evaluate our partitioning and mesh generation algorithm in different experiments. Compared with other data partitioning stratifies, our new algorithm leads to better partition result and final meshing quality.

In the future, we will generalize this parallel pipeline for structured meshing of curved 2D manifolds and 3D solid regions. We will also investigate parallel meshing algorithms with controlled singularity numbers and distributions. For this purpose, singularity estimation needs to be incorporated in partition optimization; tessellation of subregions may be computed through parameterization-based mesh construction algorithms [89] [90] which globally control the singularity distributions; and post-processing that allows the merging of nearby singularities may also be useful.

PART III

HEXAHEDRAL MESH GENERATION

This part is devoted to the PolyCube Parameterization based Hexahedral Mesh Generation. High quality hexahedral meshes benefit many numerical simulations and are preferred over the commonly used tetrahedral meshes in quite a few scientific tasks (see a survey in [91]). Hex meshing algorithms can generally be categorized into three classes: *spatial partitioning* approaches, *plastering* approaches, and *mapping-based* approaches. In *spatial partitioning* approaches, the given model or the space it embeds is first decomposed into a set of cells which will then be projected or deformed to conform to the model's boundary geometry. Marechal [92] generated hex meshes through an octree-based method through dual mesh generation and buffer-layers insertion. Ito et al. [93] developed a set of templates to optimize the octree-based hex meshing. A limitation of octree-based approaches is its pose-sensitivity. A small orientation change (e.g. rotation) of the object can lead to different meshing results. Levy and Liu [85] introduced Lp centroidal tessellation to generate anisotropic hex-dominant meshes. *Plastering* algorithms usually start from the quadrilaterally meshed boundary, then propagate hexahedral elements toward the object center. Staten et al. [94] used an unconstrained plastering to generate the hexahedral mesh. Zhang et al. [95] developed hexahedral meshes based on a skeleton-driven sweeping algorithm. One disadvantage of plastering algorithms is that the growing process could finish with unmeshed region. Another issue is that the singularity configurations inside the hexahedral mesh may be very complicated. *Mapping-based* methods map the input model M to a regular domain N, then transfers the hexahedral grid (induced from N) back to M. Finding a desirable regular domain is usually critical but as difficult as the computation of the lowly distorted map. Due to its natural regularity and geometric similarity to the model, polycube can be a suitable canonical domain for hex mesh generation [3,4]. Instead of using a fixed domain like the polycube, volumetric mappings can produce lowly distorted hex meshes by introducing singularity curves inside the domain [5,96,97]. Challenges of these volumetric parameterizations in hexahedral meshing is that either the cross frame-fields need to be given manually [96] or they need to be solved through expensive optimizations [5,97] that cannot guarantee the finding of valid solutions.

A Polycube is an orthogonal polyhedron (all the edges parallel to coordinate axes) that is a 3D manifold. A polycube parameterization is a bijective map between a 3D model and a polycube domain. The polycube domain shares the same topology with the model and approximates the geometry of the model, as shown in Figure 3.10.

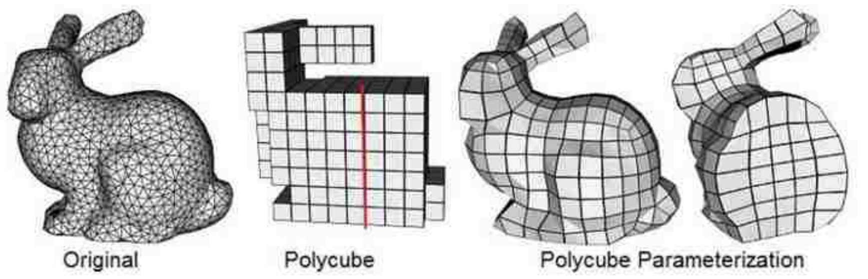


Figure 3.10: The bunny model. From left to right: The original input mesh, polycube domain, the hexahedral mesh via polycube parameterization.

Hence, the polycube map can produce a seamless parameterization that has low metric distortions and regular atlas structure (i.e, each patch/part is a rectangle/cube). Such a parameterization is desirable in many geometric modeling and processing tasks such as spline construction, meshing [3, 4, 98], shape interpolation [99], and texturing [100]. This chapter studies effective automatic polycube parameterization.

High quality hexahedral (simply denoted as *hex* in the following) mesh generation has received much attention in recent years. Tessellating the given solid model using high-quality hex meshes is desirable for effective finite element analysis or isogeometric analysis in structural mechanics, fluid dynamics simulations, etc. Polycube parameterization can be used for automatic hex mesh generation. The basic idea is as follows. For a given solid shape M, whose boundary surface is denoted as S, first construct a polycube domain Ω geometrically similar to M and compute a map $\phi : \Omega \rightarrow M$; then on Ω, construct a uniform hexahedral mesh H, and transform it by $\phi(H)$. The resultant hex mesh $\phi(H)$ conforms with the geometry of M and has regular tessellation. The meshing quality is dictated by the parameterization ϕ. If ϕ has smaller volume distortion, the resultant mesh is more uniform, and if ϕ has smaller angle distortion, the mesh elements are less sheared.

Constructing a good polycube domain is critical in reducing mapping distortions. Earlier polycube parameterization methods build polycubes manually [100,101], which can be labor-intensive and infeasible for complicated shapes. More recently, a few automatic polycube construction methods have been proposed [3, 4, 102]. However, automatic, robust, yet effective polycube construction for general 3D models remains challenging (see Section 2.1). Specifically, a polycube domain having simpler structure and fewer corners usually provides a more desirable (e.g. fewer singularities in spline construction [101] and mesh generation [3,4]) parameterization. In contrast, composing a polycube domain via more subcubes may better approximate the original geometry and result in less distorted mapping. Hence, seeking an effective balance between these two aspects is important, but it was little explored.

This chapter is structured as follows: first is the related work, then is our automatic Polycube construction, next is the PolyCube Parameterization Computation, then the next we propose a simultaneously optimize the PolyCube domain and the paramterization method.

4. AUTOMATIC POLYCUBE DOMAIN CONSTRUCTION

4.1 RELATED WORK

Polycube Construction and Mapping. Polycube mapping was first introduced by Tarini et al. [100] for seamless texture mapping, in which the polycube domain is constructed manually and the map is constructed by spatial projection followed by iterative relaxations of a deformation energy. Wang et al. [101] introduced an intrinsic polycube mapping method using the conformal mapping, and resulted in a bijective map with small angle distortion; the polycube construction, however, is also manual. Xia et al. [103] proposed an editable polycube mapping method which provides a semi-automatic user-controllable polycube construction and editing scheme. Lin et al. [102] proposed a polycube construction algorithm using the Reeb graph. This method can automatically construct polycubes for models with simple topology and geometry. However, for models with complex geometry, their constructed domain can be too simple and has large deviation from the given model, resulting in low-quality mapping results. He et al. [4] suggested a line-scanning strategy to construct the polycube nicely approximating the given model and produce nice hexahedral meshing results. This method, however, is sensitive to the object's orientation and off-axis features and it often generates overly refined polycubes with many corner points. Gregson et al [3] proposed a rotation-driven deformation algorithm to construct polycubes by rotating surface normal to nearest coordinate axes. This approach generates polycube domain shapes with significantly fewer corner points. However, to remove topologically incorrect “wedges” regions generated in deformation, a non-trivial domain correction is necessary. The post-processing suggested in [3] is not robust and could fail to produce valid polycube on models with moderately complex geometric features. Wan et al. [104] developed a polycube optimization algorithm to refine the polycube domain shape according to the stretch distortion of the final surface mapping. They construct an initial polycube through voxelization and the subsequent polycube optimization algorithm is formulated to only refine the polycube geometry without topology changing. In this chapter, we propose a polycube construction and mapping algorithm to construct polycube and optimize its domain shape, which balances the polycube shape's complexity and its geometric similarity to the given model.

4.2 BASIC IDEAS AND ALGORITHM OVERVIEW

Like the octree-based mesh generation algorithms, a polycube domain can be robustly generated through voxelization. However, such a polycube will be sensitive to orientation of the model, and will have overly zigzagged subparts for non-axis-aligned branches. Deforming non-axis-aligned branches to an axis-aligned part on a polycube will be desirable for simplifying the polycube structure and constructing lowly distorted maps. Inspired by [3], we first deform a curved model into a geometrically regular shape whose branches are axis-aligned. This gives us a polyhedron that is geometrically close to a polycube, but topologically not necessarily correct (see Section 4.3 for details). Upon this axis-aligned polyhedron, we can apply the voxelization and get a desirable initial polycube construction. Furthermore, we apply a polycube domain shape optimization algorithm to optimize the polycube. This algorithm optimizes a tradeoff between the simplicity of the polycube structure and its geometric similarity to the given model. It helps improve the polycube mapping quality. Our automatic polycube construction have two steps.

1) **Pre-deformation**. To solve a deformation of the given model so that its surface normals align with coordinate axes and it deforms to a polyhedron geometrically close to a polycube. (Section 4.3)

2) **Polycube Construction and Optimization**. To obtain an initial polycube through a voxelization, then optimize the polycube shape through some novel homotopic morphological operations. (Section 4.4)

Notations. We denote a given solid model as M and its boundary surface as $S = \partial M$. The constructed solid polycube domain and its boundary surface are denoted as Ω and $P = \partial\Omega$ respectively. In Step 1, we call the obtained axis-aligned polyhedron from deformation a *pseudo-polycube*, denoted as Q. It is so-called because its facets are nearly perpendicular to coordinate axes; but topologically, some of its vertices are not shared by three edges. In Step 2, the voxelization of Q results in an initial polycube $\widetilde{P}$, which will be optimized to a final polycube P.

4.3 PRE-DEFORMATION

A polycube is aligned with coordinate axes but the given model is usually not. Automatic polycube construction is often sensitive to the orientation of the model or its branches. An example of the polycube construction [4] that poorly handles non-axis-aligned branches

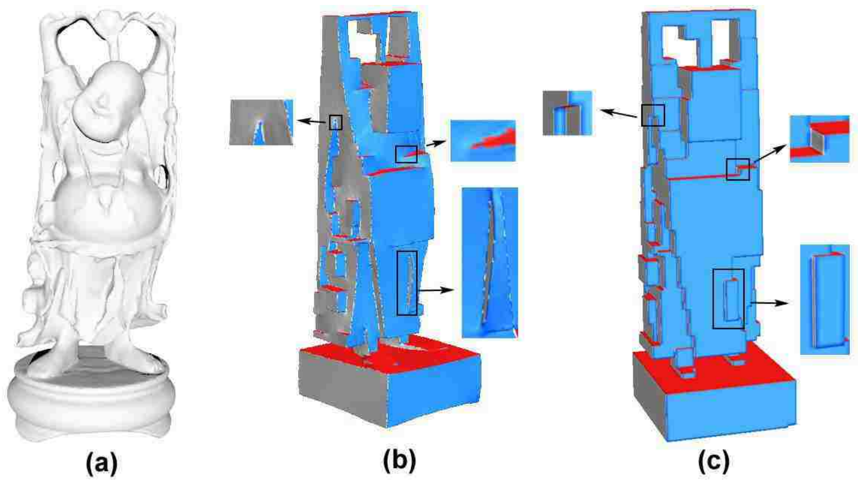


Figure 4.1: Polycube Construction for Buddha model. The input (a) is deformed to a pseudo-polycube (b) with topological irregularities plotted in red; the corrected model by our method is shown in (c).

is shown in Fig. 6.5(a), where the bunny's ears are approximated by zigzagged polycube branches with many corners.

The rotation-driven deformation guided by the surface normal, proposed by [3], can effectively prevent constructing overly zigzagged polycubes. We recap its basic idea and refer readers to [3] for details. (1) Cluster the surface S into different regions following the normal distribution. Normals on vertices are classified based on their minimal rotations to axes $\pm X, \pm Y, \pm Z$. (2) Then, use this minimal rotation to deform the surface so that the clustered normals align with their corresponding nearest axes. This reduces to solving a Poisson equation on each vertex-i:

$$\mathbf{x}_i - \frac{1}{|N_i|} \sum_{v_j \in N_i} \mathbf{x}_j = \frac{1}{|N_i|} \sum_{v_j \in N_i} \frac{R_i + R_j}{2} \cdot (\widetilde{\mathbf{x}}_j - \widetilde{\mathbf{x}}_i),$$

where $\widetilde{\mathbf{x}}_i$ and $\mathbf{x}_i$ are the original and new (after deformation) vertex positions of vertex v_i, R_i is the 3×3 rotation matrix of vertex v_i which rotates its normal to the nearest axis, and N_i

consists of neighboring vertices of vertex v_i and $|N_i|$ indicates v_i's valence. The deformation of the surface S will converge in a few iterations and we will obtain a closed surface Q that is geometrically axis-aligned.

The deformation result Q has each of its clustered regions (classified by normals) being flat and perpendicular to a coordinate axis. However, when considered as a boundary of a 3D polyhedron, the solid region bounded by Q is usually not an exact orthogonal polyhedron. Given a polyhedron, whose boundary is a closed surface made up of polygonal facets. We classify and label these facets based on the minimal rotations of their normals to axes $\pm X$, $\pm Y$ and $\pm Z$. We call the labels of $+X$ and $-X$ as opposite labels (same for Y and Z). Each facet f should be adjacent to 4 facets with different labels; and none of these four labels can either be the same with or opposite to f's label. A *wedge* is a facet whose surrounding labeling violates this rule. A few wedge examples are shown in Fig. 4.1(b).

Such wedge regions are hard to avoid, because strictly enforcing the topological constraints of orthogonal polyhedra during the surface deformation is very difficult and too expensive. Therefore, we adopt this deformed surface Q which is geometrically regular but topologically incorrect, and call it a *pseudo-polycube*.

Note that global orientation of the model sometimes affects the polycube construction. An example is shown in Fig. 4.2, where the rotation method of [3] generates an undesirable polycube (b) for the given model (a). Therefore, before this pre-deformation, we heuristically compute three principal axes [105] of the given model and pre-align principal axes with the global X, Y, Z axes. The preprocessed model may lead to better polycube construction (c).

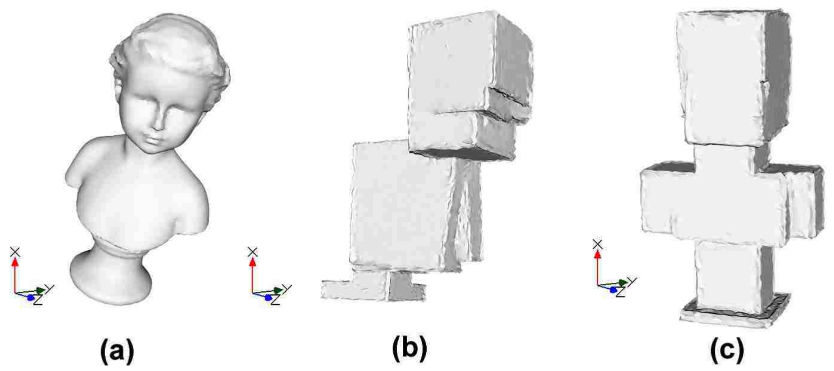


Figure 4.2: Polycube construction with respect to model orientation: (a) a 30-degree-rotated child model; (b) the polycube constructed by [3]; (c) our constructed pseudo-polycube.

4.4 POLYCUBE CONSTRUCTION AND OPTIMIZATION

Topologically irregular regions such as wedges (Fig. 4.1(b), Fig. 4.3(b)) exist in most pseudo-polycubes. Correcting these topological irregularity of a pseudo-polycube is nontrivial. A heuristic patch segmentation postprocessing algorithm was suggested in [3]. However, performing these operations on triangle meshes are also not easy, and getting a valid segmentation for polycube may not be guaranteed. We propose an algorithm through voxelization to produce a valid polycube. Furthermore, we develop a scheme to optimize the polycube domain shape, balancing domain simplicity and its geometric similarity to the pseudo-polycube.

4.4.1 Polycube Extraction by Voxelization

We first embed the pseudo-polycube Q into a spatial grid, then obtain a polycube $\widetilde{P}$ from the voxels which are inside or partially inside Q. A suitable size η of the voxel is computed (See Section 4.4.1) to avoid topological change between Q and $\widetilde{P}$ during this voxelization. We pre-compute a level-set function $D_Q(x)$ to indicate the distance from $x \in \mathcal{R}^3$ to Q using the algorithm of [106]. This $D_Q(x)$ provides efficient spatial query for partial occupation

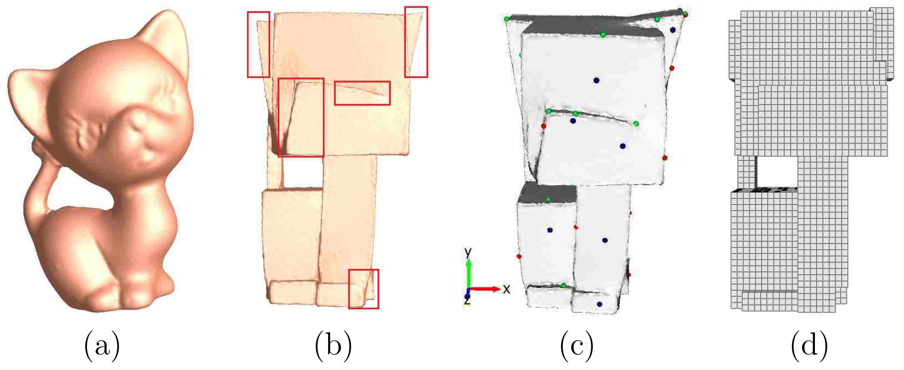


Figure 4.3: Polycube Parameterization of the Kitten Model. (a) The kitten mesh is deformed to a pseudo-polycube (b), where wedges are identified in red boxes. The resolution of voxelization is determined by a Morse analysis: the red, green, and blue points in (c) indicate the critical values in x, y and z directions respectively. The optimized polycube domain is shown in (d);

detection.

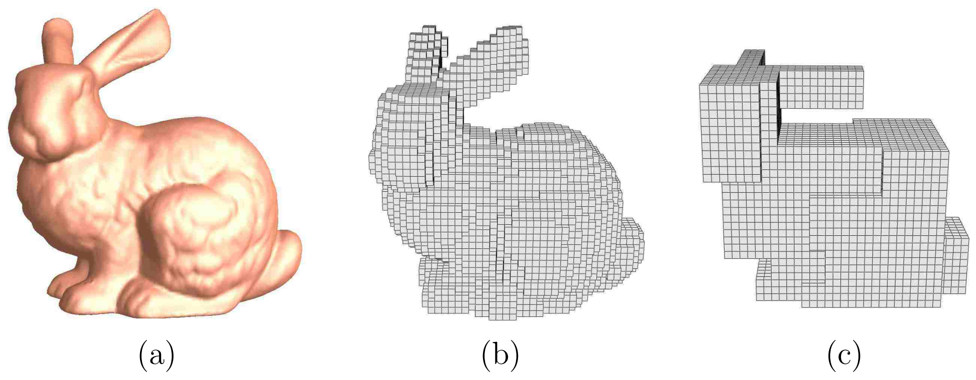


Figure 4.4: With pre-deformation, we can generate the more desirable polycube (c) than the direct voxelization (b) from original model (a).

Note that since our *pre-deformation* transforms the given model S into a pseudo-polycube Q whose geometry is simple and flat, our voxelization will produce a simple and regular polycube. In contrast, directly applying voxelization on the curved original model S will lead to a zigzagged result. An example of voxelization on bunny that is with and without pre-deformation is shown in Fig. 4.4.

Selecting Voxelization Resolution

The selection of the voxelization resolution threshold η affects the final polycube's geometric shape and topology. η should depend on the geometric saliency that we want the polycube to preserve, and be fine enough to encode the topology of the pseudo-polycube Q. Voxelization with a too big η may miss small *topological features* such as handles and voids. Size and locations of topological features can be estimated through a Morse analysis [107].

On the pseudo-polycube Q, we apply a Morse analysis by defining a Morse function $f : Q \to \mathbb{R}$. The purpose of this Morse analysis is to obtain a suitable threshold for the resolution of spatial voxelization, i.e., to capture the extrinsic spatial size (in each of the three axis directions) of each topological handle and void. Note that, using intrinsic scalar functions such as harmonic fields [108] as this function f can result in isolated critical points, but they usually do not precisely capture the spatial size of the topological features. On the other hand, the height function is simple and directly serves as such an extrinsic indicator, and thus has been widely used in Reeb graph construction [109]. Using the height functions results in clusters of critical points (e.g. on each flat facet), therefore, we pick representative critical points from these clusters and use them to determine the suitable voxelization resolution.

For vertices $\{v_i\}$ with coordinate (x_i, y_i, z_i), we first use function values $f(v_i) = x_i$ to extract the critical points $\{v_1, v_2, \ldots, v_{N_x} | x_1 < x_2 < \ldots < x_{N_x}\}$ in the x direction. The value range $[x_1, x_{N_x}]$ splits into $N_x - 1$ intervals: $[x_1, x_2], [x_2, x_3], \ldots, [x_{N_x - 1}, x_{N_x}]$. We use a threshold ε to group critical points from the same facet cluster: critical points are considered to be on the same level if their difference of Morse function values is less than ε. Topological features (handles, voids) that are smaller than ε will be treated as noise and ignored. In our experiments, we choose ε to be the 1% of the diagonal length of the bounding box of M. Similarly, we extract critical points along the y and z directions. Fig. 4.3 (c) shows the Morse analysis conducted on the Kitten model, where red, green, and blue nodes are the critical points along $x-$, $y-$, and $z-$ directions respectively. Finally, we make

$$\eta = \frac{1}{3} \min\{\min_i [x_i, x_{i+1}], \min_j [y_j, y_{j+1}], \min_k [z_k, z_{k+1}]\}$$

to ensure that the voxel representation shares the same topology with the pseudo-polycube. A more thorough discussion on the topology preservation of our polycube construction algorithm is given in section 4.5.1.

4.4.2 Polycube Shape Optimization

The initial polycube $\widetilde{P}$ we get after the voxelization is an orthogonal polyhedron and a topologically valid polycube. But geometrically, $\widetilde{P}$ may still be zigzagged in some regions. Such zigzagged geometry will lead to extra corners and increase the mapping distortion. Inspired by the morphological operations [110] in image processing, we propose the *homotopic morphological operations* to optimize the polycube shape.

Homotopic Morphological Operations

Similar to [110], we can define two basic operations: **erosion** and **dilation** on voxels in 3D. The erosion operation removes a layer of boundary cells (boundary voxels); the dilation inserts a new layer onto the boundary. More rigorously, given $\widetilde{P}$ and a cell $c \in \widetilde{P}$, c has 26 adjacent cells. An erosion on c deletes c if any of its adjacent cell is not in $\widetilde{P}$. A dilation inserts c if any of its adjacent cells is in $\widetilde{P}$.

Then we have two combined operations: **opening** and **closing** [110]. An opening operation performs an erosion followed by a dilation; and a closing performs a dilation followed by an erosion. The openings can remove thin ridges/glitches and the closings can fill small valleys/holes. Fig. 4.5 illustrates two examples.

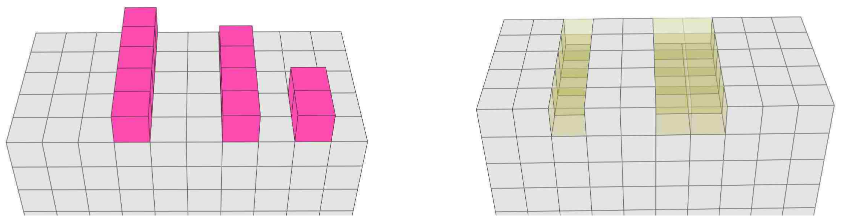


Figure 4.5: Opening and Closing: the red regions are removed by *opening* operations and the yellow regions are filled by *closing*.

Directly applying these morphological operations can simplify the polycube geometry, but does not guarantee the preservation of topology. Hence, we further develop the following homotopy-preserving approaches.

Simple Removal. Consider each k-dimensional element as a k-D cell. If a k-D element (vertex, edge, face) is shared by only one $(k+1)$-D element (edge, face, voxel, respectively). Then this k-D element is called a simple element and its accompanying $(k+1)$-D element is called its witness. To see whether a voxel (the window's center) is removable, we can iteratively remove a simple element and its witness, which is called a *simple removal.* We iteratively conduct simple removals until no further simple removal is possible. If the remained elements on this voxel are non-manifold elements, then this voxel is *not removable*, otherwise, it is *removable.* Fig.4.6 shows an example of simple removal on 3 glued cubes.

The simple removal operation is topology-preserving (see Section 4.5.1). Based on the simple removal, we can define the homotopic morphological operations. We define a **homotopic**

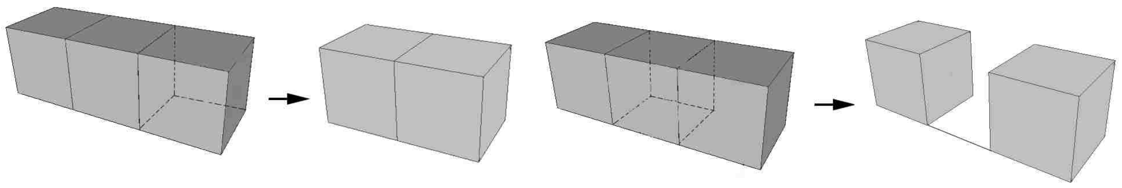


Figure 4.6: A removable right voxel (in the left subfigure) v.s. an unremovable middle voxel (in the right subfigure).

erosion operation as an erosion on a layer of boundary voxels that only performs 2D simple removals, namely, only removable voxels are allowed to be removed. The **homotopic dilation** can be defined similarly. A dilation can be considered as an erosion performed on $\mathcal{R}^3\backslash\widetilde{P}$, i.e. practically, unselected cells outside $\widetilde{P}$ but inside big bounding box of Q. Therefore, on the 3D space outside $\widetilde{P}$ we can perform simple removals during dilations on $\widetilde{P}$.

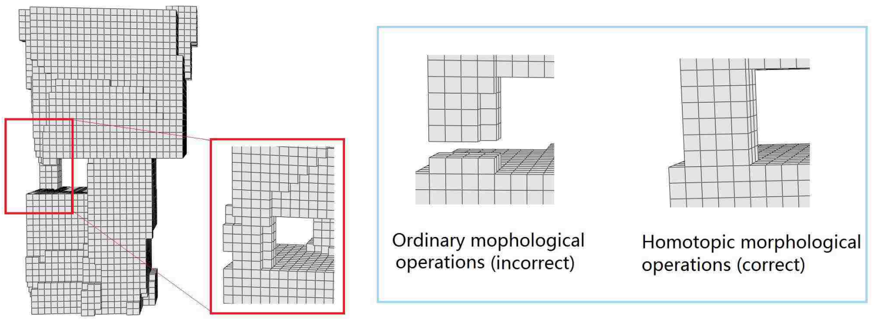

Ordinary mophological operations (incorrect)
Homotopic morphological operations (correct)

Figure 4.7: The homotopic morphological operations on Kitten model. On the tail region, the ordinary morphological operations may change the topology, while the proposed homotopic morphological operations preserve the topology.

To preserve the homotopy of the voxelization, we shall perform **homotopic opening** and **homotopic closing** using homotopic erosions/dilations rather than the originally defined erosions/dilations. A homotopic opening performs an homotopic erosion followed by a homotopic dilation on the polycube; and a homotopic closing can be considered as a homotopic opening on the dual space $\mathcal{R}^3\backslash\widetilde{P}$. Pruning via simple removal will not change the topology of space, hence homotopic erosions will not change the voxelization's topology (also see the discussion in Section 4.5.1). Fig. 4.7 shows an example of applying homotopic morphological operations on the Kitten model.

Polycube Optimization

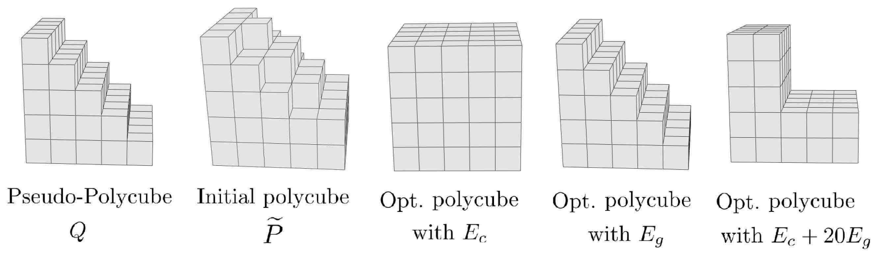


Figure 4.8: Polycube Optimization. By only optimizing E_g we will get the original shape; by only optimizing E_c we will get a simple cube with minimal 8 corners; in practice, a polycube should be optimized following a combined energy.

We use two terms to measure the polycube quality:

- Domain simplicity (Number of corners) E_c. A vertex on polycube P is a *corner* if its valence is not 4. Fewer corners indicates simpler (less zigzagged) domains.
- Geometric deviation E_g. The optimized polycube P should approximate the pseudo-polycube Q. We can accumulate distances from vertices of P to Q, $E_g = \sum_{x \in P} |D_Q(x)|$, where $D_Q(x)$ is the distance from each point polycube boundary x to Q. Less E_g indicates a more geometrically similar domain.

The total energy describing the polycube quality is

$$E_p = E_c + \alpha E_g \tag{4.1}$$

Fig. 4.8 illustrates some examples of this polycube optimization. Based on the variance of E_c and E_g, we set $\alpha = 20$ in most of our experiments. We also discuss the usage of different weighting factors α in the polycube domain optimization and compare the results in Section 4.5.2.

To minimize the objective function (4.1), we can now use a moving window to perform the *homotopic opening* or *homotopic closing* operations on Ω. On each cell c, we have the choices to perform three types of operations: *keep*, *open*, or *close*, where *keep* means we do neither *homotopic opening* nor *closing*. What operation to choose is determined greedily by the change of E_p on c. If a *homotopic closing* (or *homotopic opening*) operation reduces E_p the most, then we choose to do it, if neither reduces the energy E_p more than a threshold value (or even increases the energy), we do nothing on c. Since the distance field $D_Q()$ was pre-computed (Section 4.4.1), and the increasing/decreasing of corner points can be locally updated, this local greedy search on each cell is $O(1)$.

Another issue is the size of the moving window, which controls the coarseness of the opening and closing (e.g., when the window size is 1, each cell has 26 neighbors while when the window size is 2, each cell has 124 neighbors). We start from a big window size (e.g., 5) and gradually reduce it. On each level, we keep moving the window along all boundary cells on $\widetilde{P}$ until the energy E_p doesn't decrease (i.e., no update on boundary cell is performed). The final optimized polycube surface is denoted as P.

4.5 DISCUSSIONS AND IMPLEMENTATION

4.5.1 Topology Preservation in Polycube Construction

The construction of polycube P should preserve the topology of the input model S. In the following, we discuss the topology preservation during each of our three-step algorithm, between (1) S and Q, (2) Q and $\widetilde{P}$, and then (3) $\widetilde{P}$ and P, respectively. The topology preservation is not theoretically guaranteed, but practically, these designs are effective and in all our experiments, the resultant polycube constructed by this pipeline shares the same topology with the given 3D shape.

First, the pre-deformation from the input model S to the pseudo-polycube Q usually does not change the topology. To avoid self-intersections during the deformation, distance preservation constraints can be added to the pre-deformation equations to preserve the distance between nearby charts aligned to the same axis, and avoid self-intersections during deformation. With self-intersection prevented, the degeneracy of small topological handles can be prevented. Note that although it works well in all our experiments, we admit that non-self-intersection in the predeformation has no theoretic guarantee. If self-intersection occurs in Q, topology may change during voxelization in the next step. We believe deformation explicitly avoiding self-intersection can be adopted here to further improve this step's robustness.

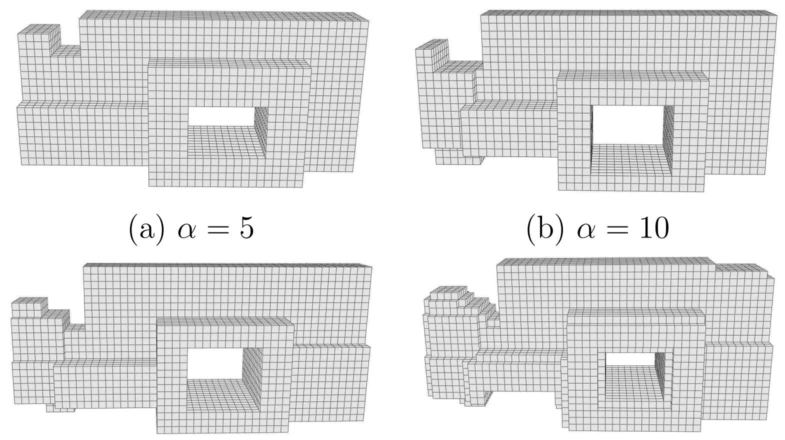
Next, the construction of initial polycube $\widetilde{P}$ is done by voxelizing the pseudo-polycube Q. The threshold η of the voxel is determined by Morse analysis. η is set to be $\frac{1}{3}$ of the minimum interval, with which critical points are usually separated in individual non-adjacent voxels, preventing the topological degeneracy of handles and voids.

Finally, the optimization from $\widetilde{P}$ to the final polycube P uses homotopic morphological operations to preserve the topology. The topology of 3-manifold can be characterize by three Betti numbers: $\beta_0, \beta_1, \beta_2$. They represent the numbers of connected components, handles and voids of the volume, respectively. The simple removals are equivalent to elementary simplicial collapse in algebraic topology [111], which preserves βs of the 3-manifold [112]. The morphological operations based on simple removal are topology-preserving. This can also be verified using the topological invariant Euler characteristics χ of the volumetric region: $\chi = \beta_0 - \beta_1 + \beta_2$, which can also be computed by

$$\chi = k_0 - k_1 + k_2 - k_3 \tag{4.2}$$

where k_i denotes the number of elements of dimension i (k_0, k_1, k_2, and k_3 corresponds to the vertex, edge, face, and cube number, respectively). When conducting a simple removal on face, $\Delta k_2 = -1, \Delta k_3 = -1, \Delta k_0 = \Delta k_1 = 0$. Hence, χ, and the topology, will not change.

4.5.2 Different Weights in Polycube Optimization



(a) $\alpha = 5$ (b) $\alpha = 10$

(c) $\alpha = 20$ (d) $\alpha = 40$

Figure 4.9: Different α in Polycube Optimization and the Resultant Domain Shapes.

The weight α in Equation (4.1) offers a flexible optimization scheme balancing a trade-off between the simplicity of the optimized polycube P and its geometric similarity to $\widetilde{P}$. Fig. 4.9 shows the different optimized domains for the Rocker-Arm model using different α's. With the increase of the α , the corner number E_c increases while the geometric deviation E_g decreases. Table 4.1 shows the different parameterization and meshing results under different choices of αs. The mapping distortion, measured by average Scaled Jacobian $\bar{\zeta}$ of the generated hexahedral mesh (the closer to 1 the better, to be defined in Section 6.6). In general, increasing α to emphasize geometric similarity can improve the parameterization quality. But this makes the polycube more complicated and has more corner singularities. Also, overly large number of corner singularities makes the polycube surface mapping difficult to optimize; the unnecessary rotations of iso-parametric curves introduced on these corners could also increase the parameterization distortion undesirable. For different models, the optimal α could be different. Based on empirical results, in most of our experiments, we set $\alpha = 20$.

Table 4.1: Different Weights in Polycube Optimization. E_c is the corner number of the polycube; E_g is the accumulated distance of the vertices on surface P to surface Q. $\bar{\zeta}$ is the average Scaled Jacobian.

Weight (α)	E_c (# of Corners)	E_g	$\bar{\zeta}$
5	41	6.82	0.861
10	60	5.51	0.897
20	80	4.10	0.931
40	134	3.12	0.921

4.5.3 Feature Preserving

Feature preserving can be implemented in our polycube mapping framework. In applications such as meshing, feature points or curves (e.g. ridges/values with salient principal curvatures) on the original model should be sampled. We can normalize the scaling of the polycube, so that the smallest cell has the unit length. Then the parameters of the feature vertices/edges should be restricted to integers. The above optimization problem (5.1) becomes a mixed-integer problem [96, 97]. We use a simple greedy strategy to solve it. First, we solve the parameterization without considering these integer restrictions, and get the initial parameters (u^1, u^2, u^3) for each vertex i. Then, for each feature edge, we project it onto an integer iso-

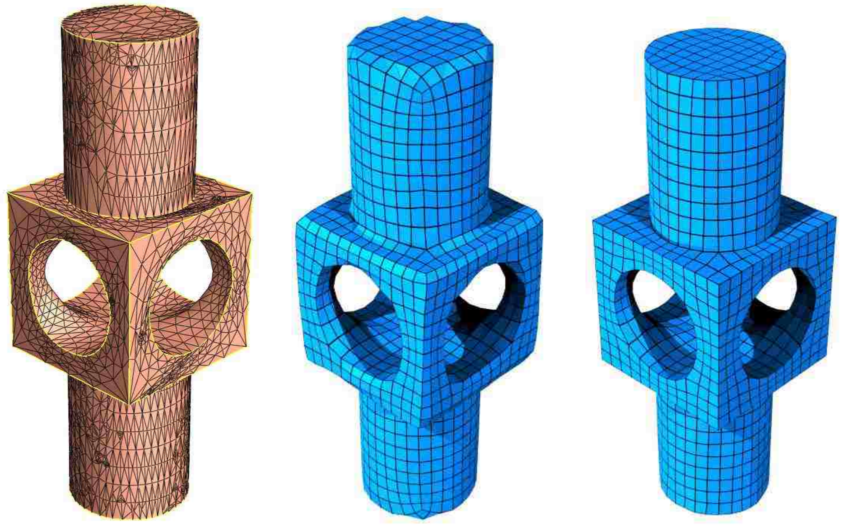


Figure 4.10: Feature Preserving. (a) The input mechanical part model has feature curves (in yellow). (b) The result without feature-preserving. (c) The result with feature preserving.

line parallel to one of the coordinate axes on the parameter domain. Feature points can be projected onto nearest parameter grids easily. After the projection of all features, we fix their parameters and solve the system again to get parameters on other vertices. Fig. 4.10 shows an example of feature-aligned meshing result.

5. POLYCUBE MAPPING COMPUTATION

5.1 RELATED WORK

Volumetric Parameterization. Volumetric parameterization have been studied recently in computer graphics and geometric modeling. Wang et al. [113] computed the discrete volumetric harmonic mapping over tetrahedral meshes for volumetric mappings on solid spheres. Li et al. [98, 114] developed meshless methods using the fundamental solution method in computing harmonic and biharmonic volumetric maps. Martin et al. [115] parameterized volumetric models onto cylinders using the finite element method, and later generalized the algorithm to more complicated models with medial surfaces [116]. Nieser et al. [96] proposed a *cube-cover* mapping algorithm for hexahedral meshing, and the mapping is guided by a user-designed frame field. Huang et al. [97] designed a boundary-aligned 3D frame field optimization algorithm that can automatically generate a smooth frame field from a given surface frame field. But the resultant frame field is not guaranteed to be valid (to induce valid mapping). Li et al. [5] solved singularity-restricted frame fields to fix the singularity errors in the direct rotational-symmetry solving. However, the generation of valid cross frame-field (hence valid mapping) is not guaranteed.

5.2 POLYCUBE VOLUMETRIC PARAMETERIZATION

After polycube construction, we compute volumetric parameterization of volume M over the solid polycube domain Ω. We first use a fast algorithm to compute polycube surface mapping $\phi|_{\partial M} : S \rightarrow P, S = \partial M, P = \partial\Omega$ (Section 5.2.1), which indicates the initial boundary constraints of the volumetric parameterization $\phi : M \rightarrow \Omega$ (Section 5.2.2).

5.2.1 Polycube Surface Mapping

First, we project every vertex v of Q along its normal direction onto P. This simple projection $f : Q \rightarrow P$ is not necessary bijective, and some region could have big metric distortion. We perform a *local relaxation* similar to [4, 100] to reduce the mapping distortion:

- If a vertex v_i's projection $f(v_j)$ and its projected one-ring neighbors $f(v_j), \forall v_j \in Nbr(v_i)$ are co-planar, we directly move $f(v_i)$ to the weighted (using the constant weight or mean-value harmonic weight) average center of $\{f(v_j), \forall v_j \in Nbr(v_i)\}$.
- If $f(v_i)$ and some of its projected neighbors $f(v_j), v_j \in Nbr(v_i)$ are on different polycube facets, we flatten these facets locally [101] and relax $f(v)$ on that plane.

Finally, $f(v)$ is projected back to the polycube surface P. Fig. 5.1 (c,d) show the mapping before and after relaxation on the kitten polycube. This relaxation can eliminate flip-overs and reduce the surface mapping distortion. In our experiments, all the flip-overs introduced in projections are corrected after a few iterations. On the other hand, note that, since surface mapping will be only used to cluster patches in the subsequent volumetric parameterization rather than hard constraints, local flip-overs within each cluster (facet) does not affect the volumetric mapping computation.

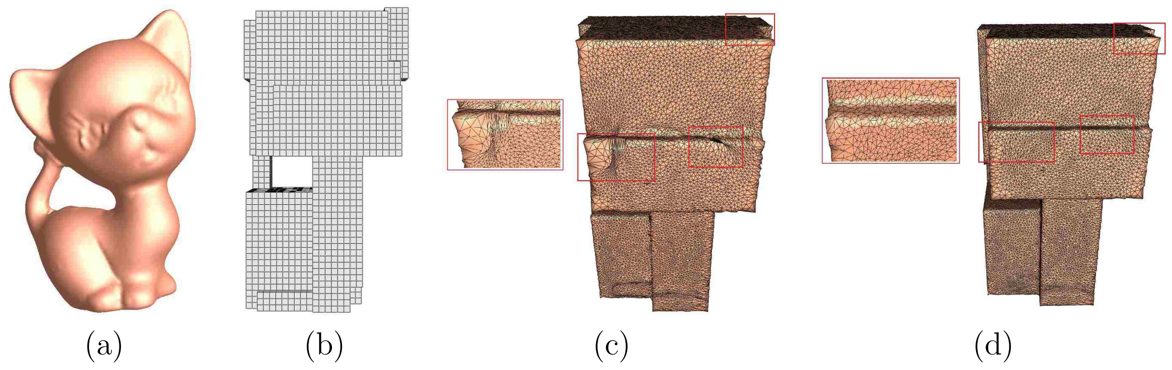


Figure 5.1: Polycube Parameterization of the Kitten Model. (a) The kitten mesh, (b) is the PolyCube domain, the polycube surface mapping is computed by a projection (c) followed by iterative local relaxations (d).

With this refined surface map $f : Q \to P$, we can compose the previously computed pre-deformation $g : S \to Q$ to get the polycube surface map from S to P, $\psi = \phi|_{\partial M} : S \to P = f \circ g$.

5.2.2 Volumetric Polycube Parameterization

Suppose the solid model M is represented by a tetrahedral mesh. If the input is a boundary surface $S = \partial M$ represented by a triangle mesh, we can simply generate a tetrahedral tessellation for M using existing algorithms/software [117]. In the following, we also use M to denote this tetrahedral mesh. Now, we want to compute a piecewise linear volumetric parameterization $\phi : M \to \Omega$, composed of three piecewise linear scalar fields (u^1, u^2, u^3) defined on tetrahedral vertices using a global chart. Their gradient fields $(\nabla u^1, \nabla u^2, \nabla u^3)$ are piecewise constant on different tetrahedra. To reduce metric distortion on ϕ, the Jacobian $\nabla \phi = (\nabla u^1, \nabla u^2, \nabla u^3)$ should be closed to a unitary matrix on every tetrahedron. We use a *volumetric frame field* [96] to guide the computation of $\nabla \phi$. A frame field X on M is composed of 3 perpendicular unit vector fields. In each tetrahedron, it can be represented by three unit vectors $X = (\vec{x}^1, \vec{x}^2, \vec{x}^3)$ that are pairwise orthogonal and form a local coordinate system. We compute a volumetric polycube parameterization as follows.

1) Set the initial frame field on the boundary tetrahedra $X|_{\partial M}$ (Section 5.2.2);

2) Solve a volumetric frame field X by optimizing its smoothness (Section 5.2.2);

3) Set boundary positional constraint $\phi|_{\partial M}$ following polycube surface mapping (Section 5.2.2);

4) Solve the polycube volumetric mapping ϕ using the directional constriant X and boundary positional constraint $\phi|_{\partial M}$ (Sectioin 5.2.2).

This computation has similar spirit of the general volumetric parameterization computation based on volumetric frame field construction [5,96,97]. However, to construct a bijective map between two solid shapes, we use a global chart, do not consider the rotational-symmetry [96], and enforce a different boundary constraint. Due to the carefully designed polycube domain shape, our parameterization results are comparable to that from general boundary-free volumetric parameterization methods. Meanwhile, this parameterization is much more efficient (no mixed-integer constraints in frame smoothing, much fewer unknowns in mapping computation) and usually does not introduce interior singularities.

Setting the Boundary Frame Field

The initial polycube construction and polycube surface mapping $\psi : S \rightarrow P$ suggests a natural initial boundary frame field for volumetric parameterization: On each boundary tetrahedron i, its frame field X_i has one vector x_i^k following the normal direction of its associated boundary face F. We preprocess each tet mesh so that each boundary tetrahedron only has one boundary face (which can be done by splitting each tetrahedron that has two

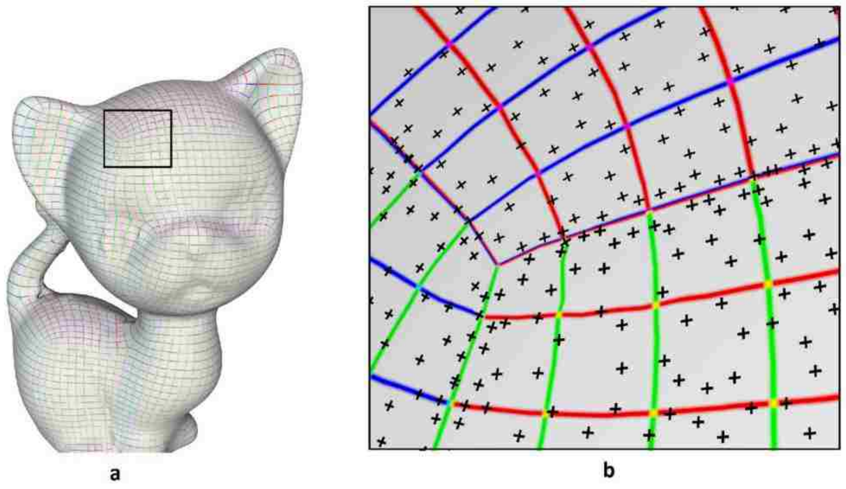


Figure 5.2: Initial Boundary Surface Frame Field on Kitten. (a) A texture map of iso-parametric lines on the boundary surface of Kitten, where red, green, and blue lines indicate 3 parametric coordinates respectively; (b) the zoom-in of surface frame field: crosses are illustrated at the baricenters of (a subset of) boundary triangles.

or more boundary faces). F is mapped onto the iso-u^k facet on P, hence ∇u^k defined on F should following F's normal direction. The other two inherent iso-parametric directions on this facet of P defines the other two components in X_i. However, these two directions, initially defined by polycube surface mapping, are not fixed and will be optimized during the frame field smoothing. Fig. 5.2 illustrates the surface frame fields on the kitten model. In (a), a texture mapping on boundary surface of Kitten is shown, where red, green and blue lines indicate iso-u^1, u^2 and u^3 parametric lines respectively. In (b), the two frame directions (excluding the one along the normal direction) on the boundary tetrahedra, rendered as crosses on their corresponding boundary faces, are zoomed and shown in (b). This region corresponds to a polycube corner.

Solving the Smooth Frame Field

With boundary frame field initially set up, we then solve a smooth volumetric frame field. We can use three Euler angles $(\alpha_i, \beta_i, \gamma_i)$ about the x, y, and z axes to represent the $i-$th frame. Intuitively, smaller change on Euler angles on frames defined on adjacent tetrahedra indicates better smoothness. We define the smoothness energy as:

$$C_S = \sum_i \sum_{j \in N(i)} [(\alpha_i - \alpha_j)^2 + (\beta_i - \beta_j)^2 + (\gamma_i - \gamma_j)^2]$$

The interior Euler angles that minimizes this energy can be computed by solving a linear system. The volumetric frame fields then can be derived from the computed Euler angles $\{\alpha_i, \beta_i, \gamma_i\}$. Note that we solve ∇u^k and u^k on a global parametric chart on Ω instead of using different local charts like [5,96,97]. Also, without considering rotational symmetry, our global frame fields are like the traditional vector fields: discretely, surrounding any interior tet edge, the composed rotation of local frames is always $2N\pi, N \in \mathbb{Z}$ (instead of $\frac{N\pi}{2}$ in [5, 96, 97]). Thus, with the boundary condition of polycube parameterization, the computed harmonic fields of Euler angles will most likely result in 3 irrotational vector fields, each of which has $N = 0$ everywhere. Under properly given boundary conditions, it is unlikely that our frame field smoothing will introduce interior singularities in the constructed gradient fields, but indeed, we admit the singularity-free and non-degeneracy of the parameterization are not theoretically guaranteed here.

Setting Boundary Positional Constraints

The compute polycube volumetric mapping $\phi : M \to \Omega$ should map the boundary surface $S = \partial M$ to domain boundary $P = \partial\Omega$. Following the previously computed polycube surface mapping, each boundary vertex v_i is mapped onto a corner, edge, or facet of the polycube surface, $\psi(v_i) = (u_i^1, u_i^2, u_i^3)$. If $\psi(v_i)$ is a corner on the polycube, we enforce all these three coordinates; if $\psi(v_i)$ is on a polycube edge perpendicular to axis x^j, then we fix the two coordinates except u_j^k; otherwise, $\psi(v_i)$ is on a cube facet perpendicular to an axis x^k, and we only fix its k coordinate u_i^k. This allows the surface mapping $\psi(v_i)$ to flow within facets and along edges on the polycube during volumetric mapping computation.

Solving Volumetric Parameterization

Given the optimized frame field X and the boundary positional constraints, we solve the volumetric parameterization ϕ by minimizing the least square energy

$$E = \sum_j \sum_{k=1}^{3} (\|\nabla u_j^k - X_j^k\|^2) \cdot vol(t_j), \tag{5.1}$$

where $vol(t_j)$ is the volume of tetrahedron t_j, superscript k and subscript j in $\nabla u_j^k, X_j^k$ indicate the kth component of parameters on tetrahedron t_j, and $\|\cdot\|^2$ is the Euclidean 2-norm. Minimizing E aligns the Jacobian of the parametrization with the cross-frame field. The gradient ∇u_j^k can be assembled from unknown scalar values u_i^k defined on vertex i contained by this tetrahedron t_j. Assume that in the tetrahedron t_j, the four vertices are $v_{jk}, k = 0, 1, 2, 3$. $u_{jk} = (u_{jk}^1, u_{jk}^2, u_{jk}^3)$ is the scalar value on vertex k. The Jacobian matrix on tetrahedral t_j is $\nabla u_j = (\vec{g_j}^1, \vec{g_j}^2, \vec{g_j}^3)$. According to [113],

$$\vec{g_j}^r = \frac{1}{3\text{vol}(t_j)} \sum_{k=0}^{3} \vec{s}_{jk} u_{jk}^r, r = 1, 2, 3$$

where $\vec{s}_{jk} = \text{Area}_{jk} \cdot \vec{n_{jk}}$, Area_{jk} is the area of the triangle face opposite to v_{jk} in tet j, and

$\vec{n}_{jk}$ is the normal of the triangle face pointing outside of the tet. Therefore we have:

$$E = \sum_j \sum_{k=1}^{3} (\|\vec{g_j}^k - X_j^k\|^2),$$

The derivatives of E_j over u_{ji}^k can be derived analytically. Hence, minimizing the quadratic objective function reduces to solving a sparse linear system.

5.3 RESULTS AND COMPARISON

We compute polycube parameterization for various solid models. The experiments are conducted on a workstation with 2.27 GHz CPU and 4GB memory. The runtime statistics are shown in Table 6.1. Generally, for a 10k-vertex mesh, the pre-processing roughly takes 120 seconds, the voxelization and polycube optimization take about 240 seconds, and the parameterization takes about 45 seconds.

Table 5.1: Runtime Table (in seconds). N_V is vertex number of the input mesh S; $t_{pre}, t_{opt}, t_{smap}, t_{vmap}$ are computational times on pre-processing, polycube construction/optimization, surface mapping, and volumetric mapping, respectively.

Model(N_V)	t_{pre}	t_{opt}	t_{smap}	t_{vmap}
3-Torus (9.1K)	124.12	234.53	40.52	0.40
Bunny (34.8K)	294.21	521.24	67.83	0.78
Rocker arm (45K)	312.42	609.97	72.97	1.32
Hand (32K)	260.16	525.88	80.42	1.01
Fertility (34K)	262.21	485.53	65.43	1.57
Kitten (35K)	284.21	515.88	85.42	1.26

Hexahedral Remeshing. Regular hex structure Ω_H can be generated on the polycube domain Ω. With the parameterization $\phi : M \rightarrow \Omega$ computed on the tetrahedral mesh of M, we simply resample all the vertices of Ω_H on M by ϕ^{-1} using barycentric interpolation. We evaluate hex meshing quality using several terms:

- The scaled Jacobian [118]. Given a vertex x in a hex element with three neighboring vertices x_1, x_2, x_3, the three edge vectors defined on this corner are $e_i = x_i - x, i = 1, 2, 3$. The scaled Jacobian of x at this element is $\zeta(x) = \det([e'_1 e'_2 e'_3])$, where e'_i is the normalized edge vector. $\zeta \in [-1, 1]$ measures a combination of angle and area distortions from a hex element to a unit cube, where 1 is its optimal value.

- The average and standard deviation of dihedral angles, α and σ_α. Ideally, hex elements should have most dihedral angles close to 90-degree, with small σ_α.
- The average volume distortion $D_v = \frac{1}{|T|}\frac{V_t}{V_s}\sum_i \frac{v_s^i}{v_t^i}$, where V and v^i are the volumes of the mesh and i-th tet element; s and t indicate the source and target meshes, both having $|T|$ number of tet elements.
- The Hausdorff distance d_H measures the difference between the original model S and the generated mesh $\partial\phi^{-1}(\Omega_H)$. This *geometric deviation* is normalized using the ratio of Hausdorff distance d_H to M's bounding box diagonal l_{diag}, denoted as d_H/l_{diag}.

Upon the direct hex meshing from polycube parameterization, we also develop a simple and efficient post-processing refinement (without introducing any singularity) to improve the hexahedral mesh quality. We optimize the positions of mesh vertices using the Untangle and Jacobian metric defined in [119]. In our implementation, in each iteration we first optimize vertices in $\mathbb{R}^3$, then project them back to the original surface. The movement is accepted if there is significant energy reduction and no flip-over. This postprocessing optimization increases the meshing quality ($\bar{\zeta}$) by roughly 0.3%.

Fig. 6.6 illustrates our hex meshing results (also see the accompanying video for better visualization). We document our mapping/meshing distortions, and the geometric deviation in Table 6.2.

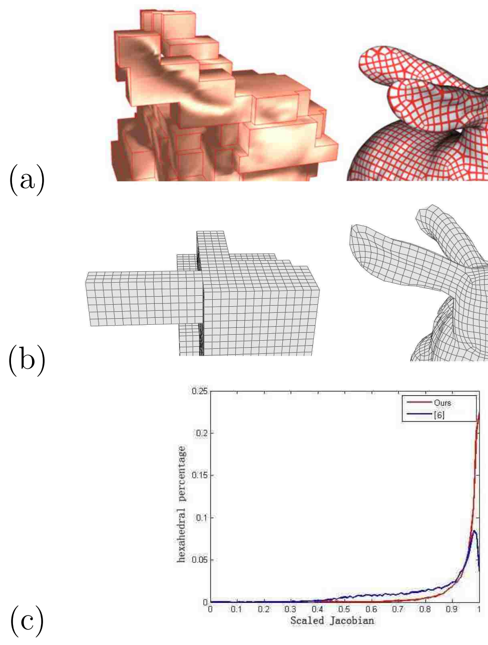


Figure 5.3: Polycube construction and map of Bunny . Our construction (b) is insensitive to non-axis-aligned branches like ears, and generates a better polycube than the method of [4] (a). Our mapping distortion is also smaller (c).

Comparison with Existing Methods. We compare our parameterization and meshing results with existing polycube mapping methods [3, 4] and volumetric parameterization methods [5, 96]. The statistics are shown in Table 6.2. We measure the following criteria, which indicate the angle and volume distortions of the mapping and meshing: (1) the scaled Jacobian $\bar{\zeta}$, (2) the average $\bar{\alpha}$ and standard deviation σ_α of dihedral angles, (3) average volume distortion D_v, and (4) the number of singularities in hex-meshes N_h (number of non-valence-6 interior vertices plus non-valence-4 boundary vertices). Since the average values are affected by the mesh resolution (better when the mesh is denser), we also include the number of elements in this comparison. When the number of hex elements are similar, the average values are directly comparable. Compared with [4], our results have significantly fewer corner numbers and smaller mapping distortion, as illustrated in Fig. 6.5 and Fig. 5.4.

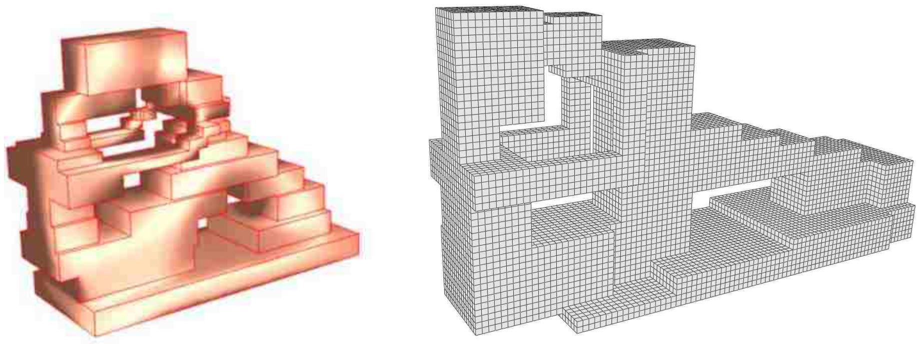
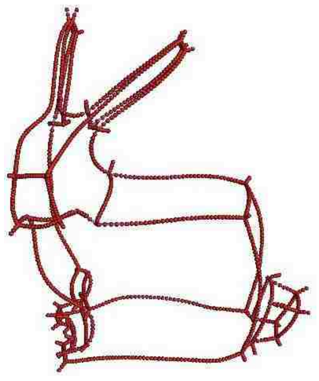
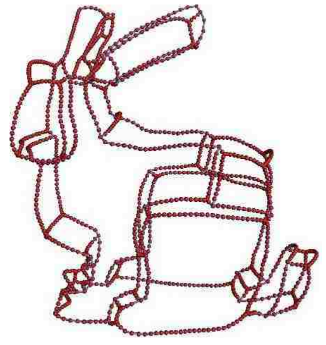


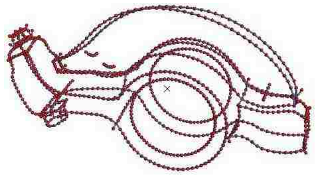
Figure 5.4: Polycube of Fertility. A simpler polycube domain can be obtained than the method of [4] (a) through our construction (b).

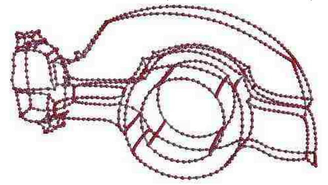
Fig. 6.5 (c) shows an example. The average Scale Jacobian of the Bunny model parameterized by [4] is 0.84, while ours is 0.94. Compared with [3], our algorithm also results in less distorted mapping and meshing results.

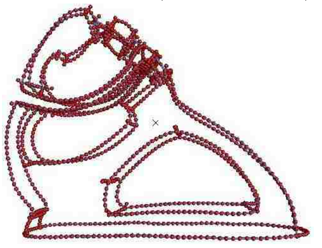
Compared with other volumetric parameterization algorithms, the scaled Jacobian and dihedral angle of our results are also comparably low. Through a free-boundary parameterization of M, Cube-cover [96] generates hexahedral meshes with smaller angle distortion (smaller deviation). Compared with [5], since [5] aims in improving worst stretched elements, their results have larger smallest scaled Jacobian values while ours have slightly better average scaled Jacobian, as shown in Table 6.2. However, there are usually complicated singularity lines inside the hex mesh whose distribution cannot be controlled. They need to perform a postprocessing to reduce singularities during mesh generation. Figure 5.5 illustrates the singularity distribution of the final generated hex-meshes of [5] and our results. The singularity numbers in the final meshes are similar.


(a) [5]: $N_h = 234$
$\bar{\zeta} = 0.935$ (134K hexes)


(b) Our result: $N_h = 211$
$\bar{\zeta} = 0.938$ (80K hexes)


(c) [5]: $N_h = 188$
$\bar{\zeta} = 0.866$ (11K hexes)


(d) Our result: $N_h = 200$
$\bar{\zeta} = 0.931$ (18K hexes)


(e) [5]: $N_h = 339$
$\bar{\zeta} = 0.911$ (14K hexes)

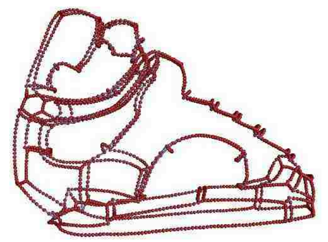

(f) Our result: $N_h = 366$
$\bar{\zeta} = 0.914$ (18K hexes)

Figure 5.5: Comparison with [5]. $\bar{\zeta}$ is the Scaled Jacobian. Our results demonstrate smaller distortion (larger scaled Jacobian) while the number of singularities is similar to what is in [5].

Table 5.2: Comparison with other methods. # Hexes indicates the number of hex elements in the final mesh. $\bar{\zeta}$ is the average Scaled Jacobian; $\bar{\alpha}$ and σ_α are the average and standard deviation of dihedral angles. D_v is the volume distortion of the parameterization. $D_g = d_H/l_{diag}$ is the geometric deviation. N_h is the number of singularities in the hex-mesh.Some data/statistics are not available from the original papers and thus are labeled as -.

Models	#Hexes	$\bar{\zeta}$	$\bar{\alpha}$ / σ_α	D_v	$D_g(10^{-7})$	N_h
Bunny [3]	82k	0.930	89.99 / 29.97	-	-	405
Bunny [5]	134k	0.935	89.99 / 27.79	-	-	234
Bunny (ours)	80k	0.938	89.99 / 11.43	0.997	0.04	211
Rocker-arm [96]	36k	0.950	90.00 / 8.40	-	-	-
Rocker-arm [3]	18K	0.899	-	-	-	-
Rocker-arm [5]	11K	0.866	89.98 / 37.31	-	-	866
Rocker-arm (ours)	18k	0.931	90.00 / 12.75	0.997	0.02	200
Fertility [3]	20k	0.911	90.00 / 29.62	-	-	432
Fertility [5]	14K	0.911	90.00 / 29.36	-	-	339
Fertility (ours)	18k	0.914	89.99 / 10.41	0.993	4.23	366
Hand [96]	5k	-	90.00 / 10.30	-	-	-
Hand [3]	12k	0.928	-	-	-	-
Hand (ours)	10k	0.937	89.88 / 12.88	0.994	2.06	110
3-Torus (ours)	25k	0.927	89.99 / 10.31	0.996	2.36	944
Kitten (ours)	16k	0.923	89.98 / 12.18	0.997	0.12	941

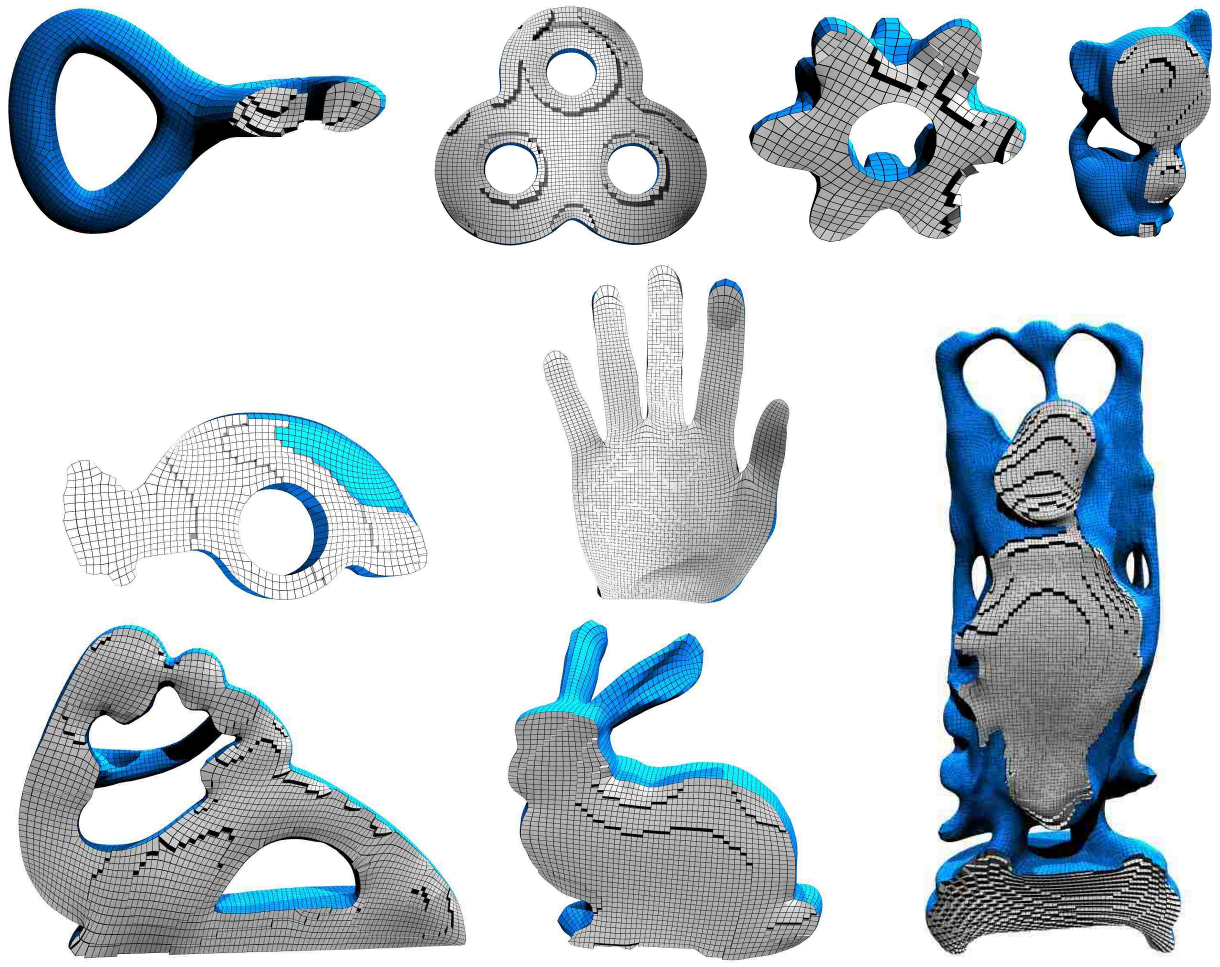


Figure 5.6: Hex Meshing Results for Double Torus, 3-Torus, Bump Torus, Hand, Rocker arm, Kitten, Fertility, Bunny and Buddha.

5.4 CONCLUSIONS

We develop an automatic and effective polycube parameterization pipeline for general 3D solid models. Our domain optimization can generate a desirable polycube balancing the domain simplicity and adequate resemblance to the input model. Upon this polycube, a volumetric parameterization with small distortion can be computed. This usually does not have interior singularities and hence is desirable for many computer-aided design/engineering tasks such as spline construction. We show this parameterization's application in high-quality hexahedral mesh generation for 3D solid geometric models.

Limitations. First, our polycube construction and volumetric mapping are computed separately. Since the domain shape will affect the mapping distortion, solving them together may lead to a better parameterization. However, their simultaneous optimization is very expensive, especially when the topological structure (corner numbers) of the polycube domain needs to be modified during the optimization. A polycube surface domain optimization algorithm without changing polycube corner numbers was given in [104], which is already expensive for surface parameterization and prohibitive in volumetric parameterization. Second, modeling complex feature curves are still challenging on volumetric polycube domains. More sophisticated scheme and efficient mixed-integer optimization algorithms are needed. Third, hex meshes generated by polycube mapping may have big distortion near corners. Introducing interior singularities can reduce such distortion. Postprocessing such as adaptive insertion of padding layers [92] (by splitting highly sheared hex elements) or other general frame field and mapping optimization methods [5, 96, 97] can be directly adopted upon our initial meshing results.

6. SIMULTANEOUSLY OPTIMIZING THE POLYCUBE DOMAIN AND THE PARAMETERIZATION

In previous chapters, we compute the PolyCube parameterization in two steps: first construct and optimize a suitable PolyCube domain, then base on the domain we compute a parameterization.

Because the parameterization depends on two aspects, the domain shape and the image on the domain, our natural thought to simultaneously optimize them, so we can get a lower distorted PolyCube parameterization.

In previous section, we apply a pre-deformation step to get a pseudo-PolyCube, Here we can extend the deformation from surface to volume to get the PolyCube shape and the volumetric domain.

6.1 NOTATION AND OVERVIEW

We follow the notations defined in previous chapters. The input is a solid tetrahedral mesh solid model as M and its boundary surface as $S = \partial M$. The vertex coordinate set is denoted as $\widetilde{X} = \{\widetilde{x_i}\}$ The output is the deformed tetrahedral mesh with the normal of each boundary triangle is axis-aligned for an arbitrary orthonormal coordinate system, i.e., a polycube. The deformed vertex coordinate set is denoted as $X = \{x_i\}$.

This output will be found through the constrained minimization of an energy containing terms to enforce axis-alignment of the normal field (Section 6.2) and term to control the paramterization distortion (Section 6.3), an numeric approach to solve this optimization problem is introduced in Section 6.4, after get the solution, a process to further reduce the number of the corner number of the PolyCube is introduced in Section 6.5.

6.2 L_1 FORMULATION OF THE POLYCUBE

A simple geometric characterization of the PolyCube domain is that the normal of each boundary faces is aligned with one of the axes of an orthonormal coordinate frame. That means the l_1-norm of every l_2-unit normal vector on the boundary is 1, which is the minimum [120].

For a triangle with L_2-unit normal $\mathbf{n}$, the L_1-norm is:

$$\|\mathbf{n}\|_1 = |\mathbf{n}_x| + |\mathbf{n}_y| + |\mathbf{n}_z|$$

the deviation from being axis-aligned can be defined using L_1-norm through $\|\mathbf{n}\|_1 - 1$.

So that, for a boundary triangle t, the normal energy E^{NORMAL} can be written as:

$$E^{NORMAL}(t) = \|\mathbf{n}_t\|_1 - 1. \tag{6.1}$$

However, directly use this energy will possibly cause the triangle degenerate to a point. So we use an area to prevent the shrinkage:

$$E^{NORMAL}(t, X) = \widetilde{A}_t \|\mathbf{n}_t(X)\|_1 - A_t(X). \tag{6.2}$$

where $\widetilde{A}_t$ is the input area of triangle t, which is a constant, $A_t(X)$ is the deformed area of triangle t.

With this, the normal term over the whole tetrahedral mesh M is

$$E^{NORMAL}(X) = \sum_{b_i} \widetilde{A}_{b_i} \|\mathbf{n}_{b_i}(X)\|_1 - \sum_{b_i} A_{b_i}(X),$$

6.3 PARAMTERIZATION DISTORTION TERM

We are not only need to generate the PolyCube shape. It desirable for a PolyCube paramterization to be low distortion. This is not only for surface but also for the volumetric. There are several distortion measurement such as As-Rigid-As-Possible [53], As-Similar-As-Possible [121] and As-Killing-As-Possible [122], and Most Isometric ParamterizationS(MIPS) [123]. The stand 2D MIPS energy measures the conformality of the mapping $\sigma_1\sigma_2^{-1} + \sigma_2\sigma_1^{-1}$ where σ_1, σ_2 are the singular values of the Jacobian of the mapping associated with a triangle. Based on the singular values, there are other type of distortion measurement such as Dirichlet energy $\sigma_1^2 + \sigma_2^2$, Green-Lagrange energy $(\sigma_1^2 - 1)^2 + (\sigma_2^2 - 1)^2$. Here we adopt a modified MIPS energy called Advanced MIPS (AMIPS) [124] which can also measure the isometric distortion.

In 2D case, given a triangle t with points p_1, p_2, p_3 with coordinates $(x_1, y_1), (x_2, y_2), (x_3, y_3)$ respectively, and with the corresponding points in parametric domain are q_1, q_2, q_3 with coordinates $(u_1, v_1), (u_2, v_2), (u_3, v_3)$.

The Jacobian Matrix is

$$\begin{aligned} J_t &= \frac{1}{2A_t} \begin{pmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \end{pmatrix} \begin{pmatrix} 0 & 1 & -1 \\ -1 & 0 & 1 \\ 1 & -1 & 0 \end{pmatrix} \begin{pmatrix} u_1 & v_1 \\ u_2 & v_2 \\ u_3 & v_3 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad (6.3) \\ &= \frac{1}{2A_t} C^X U^T B \end{aligned}$$

where $A_t = \frac{(u_2-u_1)(v_3-v_1)-(u_3-u_1)(v_2-v_1)}{2}$ is the area of triangle t,

$$C^X = \begin{pmatrix} x_3 - x_2 & x_1 - x_3 & x_2 - x_1 \\ y_3 - y_2 & y_1 - y_3 & y_2 - y_1 \end{pmatrix} \tag{6.4}$$

$$B = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{6.5}$$

Here x_i, y_i are known, u_i, v_i are unknowns.

On this triangle t, the MIPS energy E^{MIPS} is:

$$\begin{aligned} E_t^{MIPS} &= \frac{\sigma_1}{\sigma_2} + \frac{\sigma_2}{\sigma_1} \\ &= \sqrt{\sigma_1^2 + \sigma_2^2}\sqrt{\frac{1}{\sigma_1^2} + \frac{1}{\sigma_2^2}} \\ &= \sqrt{\mathrm{Tr}(\Sigma_t^2)}\sqrt{\mathrm{Tr}((\Sigma_t^{-1})^2)} \\ &= \sqrt{\mathrm{Tr}(V\Sigma^T U^T U \Sigma V^T)}\sqrt{\mathrm{Tr}(U(\Sigma^{-1})^T V^T V \Sigma^{-1} U^T)} \\ &= \sqrt{\mathrm{Tr}\, J_t^T J_t}\sqrt{\mathrm{Tr}\,(J_t^{-1})^T (J_t^{-1})} \\ &= \|J_t\|_F \|J_t^{-1}\|_F \\ &= \frac{\mathrm{Tr}\,(J_t^T J_t)}{\det J_t} \end{aligned} \tag{6.6}$$

where σ_1, σ_2 are the singular values, $J_t = V\Sigma U^T$ is the Singular Value Decomposition.

For the isometric control, the E^{det} term can be added:

$$E_t^{det} = \frac{\det(J_t) + \det(J_t^{-1})}{2} \tag{6.7}$$

So that the total energy (Advanced MIPS, AMIPS) on the whole mesh is:

$$E^{AMIPS} = \sum_t E_t^{MIPS} + \sum_t E_t^{det} = \sum_t \frac{\mathrm{Tr}\,(J_t^T J_t)}{\det J_t} + \sum_t \frac{\det(J_t) + \det(J_t^{-1})}{2} \tag{6.8}$$

Note that the definition of AMIPS is easy to extend to 3D case.

6.4 SOLVING THE OPTIMIZATION PROBLEM

With the formulation of the boundary normal term E^{NORMAL} measuring the PolyCube geometric shape, and the AMIPS term E^{AMIPS}measuring the paramterization distortion, we can assemble the total energy for solving the PolyCube paramterization:

$$E^{PCP}(X) = \alpha E^{NORMAL} + E^{AMIPS} \tag{6.9}$$

where α is the weight to adjust the two energy term.

6.4.1 Gradient Computation

To solve this optimization, we apply the gradient-decent method. However, the E^{NORMAL} contains the L_1-norm which is not smooth. To deal with this case, one can use interior point methods [125]. Here we employ a smooth approximation of the L_1-norm for simplicity.

Given a component $c \in [-1, 1]$ of the normalized normal, we simply replace the absolute value $|c|$ by $\sqrt{c^2 + \varepsilon} := \widetilde{c}$, where $\widetilde{c} \leq 0$ is a regularizing parameter to balance smoothness and accuracy.

The gradient (with respect to node positions) of this approximate absolute value is expressed as a function of the original component:

$$\nabla\widetilde{c} = \frac{c}{\widetilde{c}}\nabla c$$

The gradient of second term E^{AMIPS} can be derived as follows.

The derivative with respective to variable p is:

$$\frac{\partial E}{\partial p} = \frac{2\,\mathrm{Tr}(J_t^T \partial_p J_t)}{\det J_t} - E^{MIPS}\,\mathrm{Tr}(J_t^{-1}\partial_p J_t) + \frac{(\det(J_t) - \det(J_t^{-1}))\,\mathrm{Tr}(J_t^{-1}\partial_p J_t)}{2} + \frac{1}{2}(\det J - \det J^{-1})\,\mathrm{Tr}(J^{-1}\cdot \partial_p J). \tag{6.10}$$

6.4.2 Weighting Schedule

Since our energy is consisted of two terms, we need to choose the weight α to balance the E^{NORMAL} and E^{AMIPS}. Also, since we use an approximation of the L_1-norm function, the parameter ε also needs to be determined.

Obviously the boundary normal will become axis-aligned only if the weight α is large. However, starting a large α will remove the effect of E^{AMIPS}. So we begin solving the optimization with a small α and relative large ε to make the approximation of L_1-norm smooth without be trapped in local minimum so fast. After this converge, we gradually increase the α and reduce the ε to get an more accurate solution.

In our experiments, we start with $\alpha = 0.1, \varepsilon = 1$, then after each convergence, we double the α and half the ε. The processes will converge in 20 iterations in our experiments.

Figure 6.1 shows an simple example on 2D:

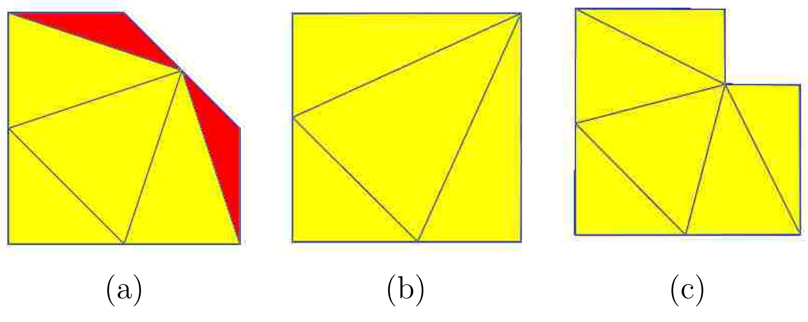


Figure 6.1: (a) Input (b) Deform only considering boundary normal term, the red triangles in (a) will degenerate. (c) Considering the AMIPS mapping distortion term to prevent degeneration.

6.5 SIMPLIFYING THE POLYCUBE DOMAIN

After solving the optimization, we get the PolyCube domain with the paramterization. Recall that a PolyCube domain is preferable if it has fewer number of corner points. However our deformation energy formulation lacks control on the number of the corner points. So we add a step to simplify the PolyCube domain by removing corner points. First, we use voxelization to embed the resultant PolyCube domain in a grid space, then we use a moving boundary plane in the grid space to get a PolyCube domain with fewer corner points and the paramterization. The first step voxelization we apply the same algorithm described in 4.4.1. After the voxelization, the PolyCube domain can be defined by the boundary patches. By moving the boundary patches in the grid space, we can modify the PolyCube domain shape and parameterization.

The quads in $\partial\widetilde{P}$ can be clustered into 6 types of patches by their normals $(\pm X, \pm Y, \pm Z)$. Given a direction $\vec{d}$, a patch is called a *Positive Patch* if $\vec{n} \cdot \vec{d} = 1$, where $\vec{n}$ is the normal of the patch. It's a *Negative Patch* if $\vec{n} \cdot \vec{d} = -1$. In our algorithm we use $+X, +Y, +Z$ as directions. Fig. 6.2 illustrates an example.

In the voxelized grid, each quad has an integer height on its normal direction. Corners can be eliminated by moving patches on the normal direction. Fig. 6.3 illustrates an example. After moving the upper patch down and merge with the lower one, four corners are eliminated.

We can optimize the PolyCube domain by assigning a new height to each boundary patch. The two objectives are: first, we would like to reduce the corner number to get a simpler domain; second, the geometric change should not be large from the initial polycube.

The pipeline of our polycube optimization algorithm is shown in Algorithm 1.

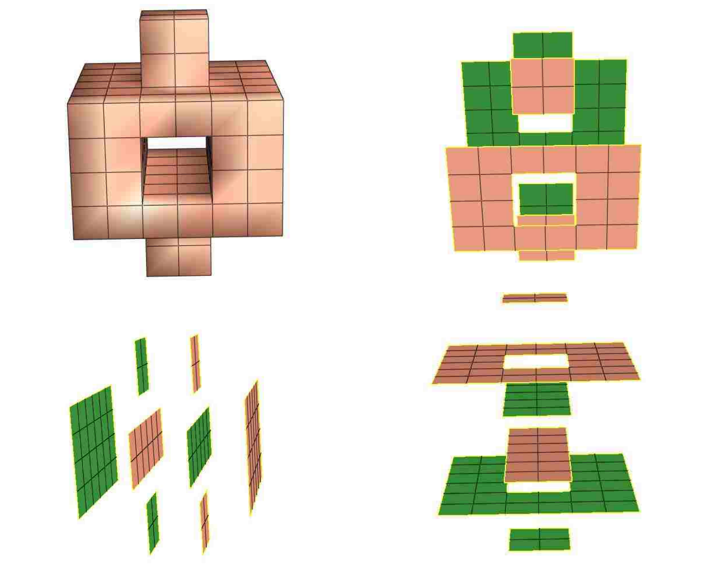

(a) Polycube surface (b) $+X$ and $-X$ patches
(c) $+Y$ and $-Y$ patches (d) $+Z$ and $-Z$ patches

Figure 6.2: Six types of patches in three direction. Negative patches are in green.

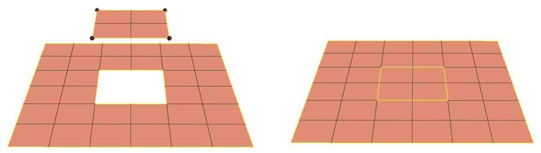


Figure 6.3: Left: Original two patches. The upper patch has four corners in red color. Right: After Moving the up patch down, the four corners are eliminated.

Algorithm 1: Polycube Optimization Algorithm

```
   input: Initial polycube $\widetilde{P}$
1  foreach quad $q \in \partial\widetilde{P}$ do
2      $h_q^0 \leftarrow$ its initial height ;
3  end foreach
4  foreach patch $p \in \partial\widetilde{P}$ do
5      $h_p^0 \leftarrow$ its initial height ;
6  end foreach
7  Compute valid moving integer interval $\{H_p^{min}, \ldots, H_p^{max}\}$ for each patch $p$ ;
8  $k \leftarrow 0$ ;
9  repeat
10     Compute patch heights which decrease the PolyCube Energy $E_P$
11 until no patch changes height;
```

Computing the Valid Moving Integer Interval. When optimizing the height of each polycube patch, the topology of the polycube should not change (i.e., a handle should not disappear). Merging a positive patch with a negative patch will lead to a non-manifold or a topological change of the polycube. For example, in Fig. 6.2 (d), merging the lowest positive patch and highest negative patch will eliminate the handle, which should be prevented during the polycube optimization. So, we enforce a valid moving interval for each patch to avoid this problem.

For a patch p perpendicular to an axis direction d, first we compute a patch set $P_p = \{p_i\}$, composed of all other patches that are perpendicular to d, such that 1) p_i has a different orientation with p; and 2) p and p_i, after projected onto a plane perpendicular to d, intersect with each other. Then the height interval $H_p^{min}, \ldots, H_p^{max}$ for patch p can be determined from P_p: $H_p^{min} = \max\{h_{p_i} | h_{p_i} < h_p, p_i \in P_p\}$, $H_p^{max} = \min\{h_{p_i} | h_{p_i} > h_p, p_i \in P_p\}$ where h_p is the height of patch p, h_{p_i} is the height for patch $p_i \in P_p$.

6.5.1 PolyCube Energy Formulation

We define a PolyCube energy E_P to evaluate the quality of the PolyCube Parameterization. The energy is consisted of two terms: the domain complexity term and the parameterization distortion term.

The domain complexity term reflects the corner numbers of the PolyCube domain, which can be computed by the normals of the adjacent boundary patches. For a patch p_i, $N(p_i)$ is it's neighbor patches, $\mathbf{n}(p)$ is the normal of the patch p, we formulate the complexity energy as follows:

$$E_C(p_i) = \sum_{p_j \in N(p_i)} 1 - (\mathbf{n}_{p_i} \cdot \mathbf{n}_{p_j}) \tag{6.11}$$

The second term of PolyCube Energy is paramterization distortion E^{AMIPS}. The total energy can be written as

$$E_P = \sum_{p_i} E_C(p_i) + \alpha_{PC} E^{AMIPS} \tag{6.12}$$

where the α_{PC} is the weight to balance two terms. In experiments we choose $\alpha = 20$.

Solving the Optimal New Height $\hat{h}_p$. Within the moving interval for each patch p, we can calculate, when we move p to each new height $h' \in (H_p^{min}, H_p^{max})$, (1) the change of corner number $\Delta E_c(p, h') = E_c(p, h') - E_c(p, h^0)$, and (2) the new polycube's geometric deviation from the initial polycube $\Delta E_g(p, h') = \sum_{q \in p} A(q)|h' - h_q^0|$, where $q \in p$ are the quads in p, and h_q^0 is the initial height of quad q. The height that minimizes the PolyCube Energy E_P is the optimal new height $\hat{h}_p$ in this iteration.

The algorithm can be shown as:

```
1  $k \leftarrow k + 1$ ;
2  foreach direction $\vec{d}$ in directions $\{+X, +Y, +Z\}$ do
3      foreach patch $p$ perpendicular to $\vec{d}$ do
4          foreach $h' \in \{H_p^{min}, \ldots, H_p^{max}\}$ do
5              Compute the PolyCube Shape Energy
6          end foreach
7          Picking the Optimal New Height $\hat{h}_p^k$ which minimizes $E(h')$ ;
8          Locally update the polycube mapping
9      end foreach
10 end foreach
```

Parameterization Update. When the optimal height $\hat{h}_p$ is obtained in the $(k+1)$th iteration, we update this modified patch as follows. (1) Set the new height for patch p and its quad facets: $h_p^{k+1} = \hat{h}_p$, and $h_q^{k+1} = \hat{h}_p, \forall q \in p$; (2) Locally update the polycube paramterization (3) Update the *moving range* of related neighboring patches.

6.5.2 Solving the Optimal Patch Heights using Beam Search

Greedily solving the patch heights one by one is slow, and can be easily trapped in local minimum. We apply the *Beam Search* to find a better solution. Beam Search algorithm is a heuristic search algorithm that can be used to solve optimization problem.

Suppose we have a state-space Ω. For a state $S \in Q$, we can use n parameters to describe it, $S = S(x_1, x_2, \ldots, x_n)$. In addition, we have an evaluation function $f(x_1, x_2, \ldots, x_n)$ to estimate how good or how bad this state is. Our goal it to find a state $S^*(x_1^*, x_1^*, \ldots, x_n^*)$, its evaluation value $f(x_1^*, x_1^*, \ldots, x_n^*)$ is the best one of all states in Ω.

The general beam search stragtegy is as follows, also in Figure 6.4.

1. Begin with an initial state S^0
2. Generate all successors of S^0: $\{S_1^1, S_2^1, \ldots, \}$
3. Select the k best states $\{\bar{S}_1^1, \bar{S}_2^1, \ldots, \bar{S}_k^1\}$
4. For each $\bar{S}_1^1$, keep steps 2 and 3 until reach an optimal.

First, for each boundary patch we assign an integer variable x_i for the height in the grid space on it's normal direction. In each iteration of the beam search, k best states are selected by evaluating the PolyCube Energy E_P. Iteratively update the patch heights until reach a local minimum.

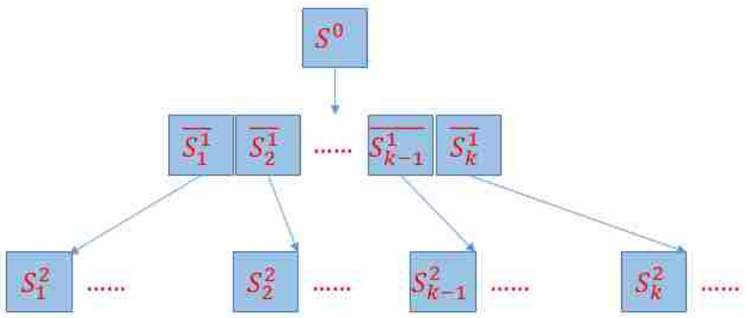


Figure 6.4: The general beam search strategy.

6.6 RESULTS AND COMPARISON

We compute polycube parameterization for various solid models. The experiments are conducted on a workstation with 2.27 GHz CPU and 4GB memory.

We compare our parameterization and meshing results with existing polycube mapping methods [3, 4, 126] and volumetric parameterization methods [5, 96]. The statistics are shown in Table 6.2. The following criteria indicating the angle and volume distortions of the mapping and meshing are measured: (1) the scaled Jacobian $\bar{\zeta}$, (2) the average $\bar{\alpha}$ and standard deviation σ_α of dihedral angles, (3) average volume distortion D_v, and (4) the number of singularities in hex-meshes N_h (number of non-valence-6 interior vertices plus non-valence-4 boundary vertices). Compared with [3], our algorithm more robust and results in close distorted mapping and meshing results. Compared with [4], our result has significantly fewer corners and smaller mapping distortion, as illustrated in Fig. 6.5. Compared with [126], our method gets close corner numbers in polycube and mapping distortion in meshing results. Our geometric deviation is slightly smaller than [126]. Generally, our polycube optimization method is 30% faster than the morphological-based method in [126]. The runtime statistics are shown in Table 6.1.

Compared with other volumetric parameterization algorithms, the scaled Jacobian and dihedral angle of our results are also comparably low. Through a free-boundary parameterization of M, Cube-cover [96] generates hexahedral meshes with smaller angle distortion (smaller deviation). Compared with [5], since [5] aims in improving worst stretched elements, their results have larger smallest scaled Jacobian values while ours have slightly better average scaled Jacobian, as shown in Table 6.2. However, there are usually complicated singularity lines inside the hex mesh whose distribution cannot be controlled. They need to perform a postprocessing to reduce singularities during mesh generation.

Table 6.1: Polycube Optimization Runtime Table (in seconds).N_V is vertex number of the input mesh S.

Model(N_V)	[126]	Ours
3-Torus (9.1K)	234.53	150.21
Bunny (34.8K)	521.24	321.57
Rocker arm (45K)	609.97	420.52
Hand (32K)	525.88	355.28
Fertility (34K)	485.53	330.81
Kitten (35K)	515.88	335.12

Fig. 6.6 illustrates our hex meshing results. We document our mapping/meshing distortions, and the geometric deviation in Table 6.2.

Table 6.2: Comparison with other methods. # Hexes indicates the number of hex elements in the final mesh. $\bar{\zeta}$ is the average Scaled Jacobian; $\bar{\alpha}$ and σ_α are the average and standard deviation of dihedral angles. D_v is the volume distortion of the parameterization. $D_g = d_H/l_{diag}$ is the geometric deviation. The order of magnitude of D_g is 10^{-7}. N_h is the number of singularities in the hex-mesh.Some data/statistics are not available from the original papers and thus are labeled as -.

Models	#Hexes	$\bar{\zeta}$	$\bar{\alpha}$ / σ_α	D_v	D_g	N_h
Bunny [3]	82k	0.930	89.99 / 29.97	-	-	405
Bunny [5]	134k	0.935	89.99 / 27.79	-	-	234
Bunny [126]	80k	0.938	89.99 / 11.43	0.997	0.04	211
Bunny (ours)	80k	0.933	89.99 / 18.42	0.997	0.02	222
Rocker-arm [96]	36k	0.950	90.00 / 8.40	-	-	-
Rocker-arm [3]	18K	0.899	-	-	-	-
Rocker-arm [5]	11K	0.866	89.98 / 37.31	-	-	866
Rocker-arm [126]	18k	0.931	90.00 / 12.75	0.997	0.02	200
Rocker-arm (ours)	18k	0.910	90.00 / 11.50	0.998	0.02	240
Fertility [3]	20k	0.911	90.00 / 29.62	-	-	432
Fertility [5]	14K	0.911	90.00 / 29.36	-	-	339
Fertility [126]	18k	0.914	89.99 / 10.41	0.993	4.23	366
Fertility (ours)	18k	0.911	89.99 / 11.22	0.994	3.23	402
Hand [96]	5k	-	90.00 / 10.30	-	-	-
Hand [3]	12k	0.928	-	-	-	-
Hand [126]	10k	0.937	89.88 / 12.88	0.994	2.06	110
Hand (ours)	10k	0.929	89.99 / 15.28	0.994	1.76	140
Kitten [126]	16k	0.923	89.98/12.18	0.997	0.12	941
Kitten (ours)	16k	0.910	89.99 / 13.41	0.998	0.08	960

(a)

(b)

(c)

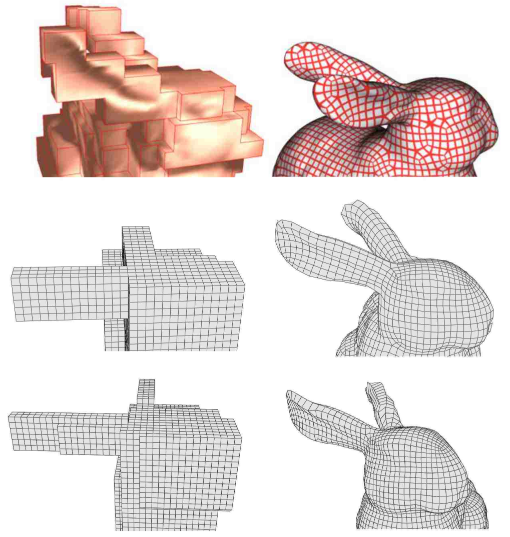


Figure 6.5: Polycube construction and map of Bunny. Compared with [4] (a), our polycube construction (c) can get a simpler domain. Compare with citeYZWL13CAD (b) is 30% faster. The average scale Jacobian are close and our geometric deviation is slightly better. Table 6.2

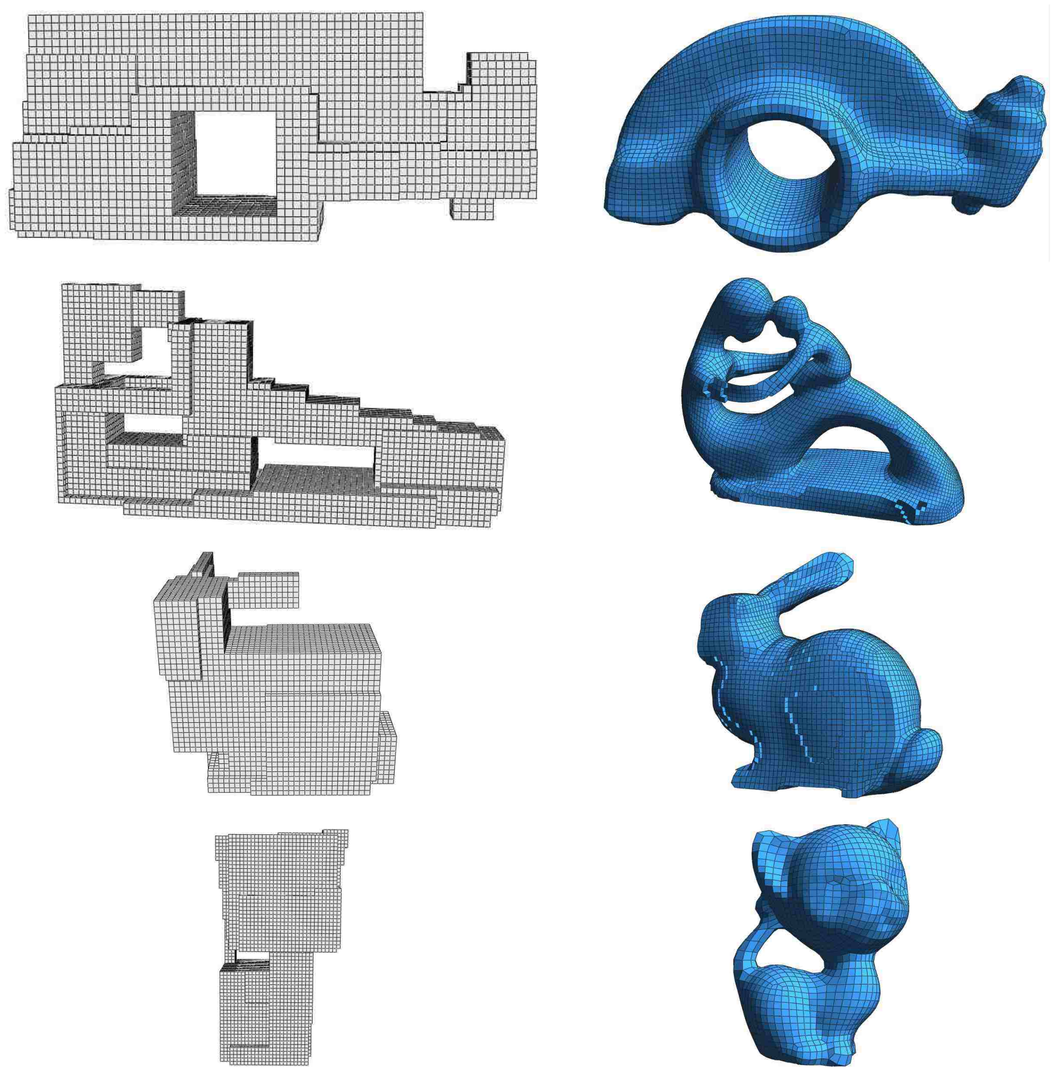


Figure 6.6: Polycube (left) and Hex Meshing Result (right) for Rocker arm, Fertility, Bunny and Kitten.

6.7 CONCLUSION

In this part we explore the PolyCube parameterization based hexahedral mesh generation. The PolyCube construction and parameterization computation can be separate or simultaneously. In the separate framework, a PolyCube is constructed from a pseudo-PolyCube, then optimized by voxelization and homotopic morphological operations. After the PolyCube domain is fixed, a frame-field guided parameterization is computed. In the simultaneous framework, a volumetric deformation is applied on the input solid model, then is optimized by moving planes using beam search algorithm. The simultaneous framework can provide lower distortion parameterization and hiqher quality hexahedral mesh.

7. CONCLUSION AND FUTURE WORK

High-quality mesh generation is important in finite element analysis. When compared with unstructured meshes, the structural meshes are favored in many scientific/engineering tasks. However, generating high-quality structural mesh remains challenging, especially for large-scale geometric data. This dissertation studies the effective generation of structural mesh on large and complex geometric data. We study a general geometric computation paradigm to solve this problem via model decomposition and divide-and-conquer.

To demonstrate the efficiency and effectiveness of our framework, we test our algorithm on various sets of scientific data. We compare our decomposition results with existing partitioning methods; also, we compare meshing quality with the results from other popular meshing algorithms. We also show this pipeline scales up efficiently on HPC environment.

A possible extension of current work is the parallel structured mesh generation framework for 3D data. First a geometric-aware graph partitioning is applied to segment the input 3D data into a set of sub-regions, then for each sub-region using polycube-mapping based method to generate the structured meshes. The extension / improvement of current work includes:

- Extend the geometric-aware graph partitioning to 3D.
- To generate the structured mesh for a large-scale geometry via polycube mapping in divide-and-conquer framework, a parallel polycube domain optimization is desirable.

REFERENCES

[1] TOSCA(Tools for non-rigid shape comparison and analysis) Project 3D Datasets, "*http://tosca.cs.technion.ac.il/book/resources_data.html*,"

[2] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[3] J. Gregson, A. Sheffer, and E. Zhang, "All-hex mesh generation via volumetric polycube deformation," *CGF*, vol. 30, no. 5, 2011.

[4] Y. He, H. Wang, C. Fu, and H. Qin, "A divide-and-conquer approach for automatic polycube map construction," *Comput. & Graph.*, vol. 33, no. 3, pp. 369 – 380, 2009.

[5] Y. Li, Y. Liu, W. Xu, W. Wang, and B. Guo, "All-hex meshing using singularity-restricted field," *ACM Trans. Graph.*, vol. 31, pp. 177:1–177:11, Nov. 2012.

[6] J. R. Shewchuk, "Triangle: Engineering a 2d quality mesh generator and delaunay triangulator," in *Applied computational geometry towards geometric engineering*, pp. 203–222, Springer, 1996.

[7] H. Si and A. TetGen, "A quality tetrahedral mesh generator and three-dimensional delaunay triangulator," *Weierstrass Institute for Applied Analysis and Stochastic, Berlin, Germany*, 2006.

[8] J. Schöberl, "Netgen an advancing front 2d/3d-mesh generator based on abstract rules," *Computing and visualization in science*, vol. 1, no. 1, pp. 41–52, 1997.

[9] L. Piegl and W. Tiller, "The nurbs book. 1997," *Monographs in Visual Communication*, 1997.

[10] Y. He, H. Wang, C.-W. Fu, and H. Qin, "A divide-and-conquer approach for automatic polycube map construction," *Computers & Graphics*, vol. 33, no. 3, pp. 369–380, 2009.

[11] X. Roca, *Paving the path towards automatic hexahedral mesh generation.* PhD thesis, PhD thesis, Universitat Politecnica de Catalunya, 2009.

[12] D. Bommes, B. Levy, N. Pietroni, E. Puppo, C. Silva, M. Tarini, and D. Zorin, "Quad-mesh generation and processing: A survey," *Computer Graphics Forum*, vol. 32, no. 6, pp. 51–76, 2013.

[13] D. R. White, S. Saigal, and S. J. Owen, "Meshing complexity of single part cad models.," in *IMR*, pp. 121–134, 2003.

[14] R. J. Meyers, T. J. Tautges, and P. M. Tuchinsky, "The" hex-tet" hex-dominant meshing algorithm as implemented in cubit.," in *IMR*, pp. 151–158, Citeseer, 1998.

[15] Y. Li, Y. Liu, W. Xu, W. Wang, and B. Guo, "All-hex meshing using singularity-restricted field," *ACM Transactions on Graphics (TOG)*, vol. 31, no. 6, p. 177, 2012.

[16] B. Chazelle and L. Palios, "Decomposition algorithms in geometry," *Algebraic Geometry and Its Applications*, pp. 419–447, 1994.

[17] J. M. Keil, "Polygon decomposition," *Handbook of Computational Geometry*, 2000.

[18] A. Agathos, I. Pratikakis, S. Perantonis, N. Sapidis, and P. Azariadis, "3d mesh segmentation methodologies for cad applications," *Computer-Aided Design*, vol. 4, no. 6, pp. 827–841, 2007.

[19] A. Shamir, "A survey on mesh segmentation techniques," *Computer Graphics Forum*, vol. 27, no. 6, pp. 1539–1556, 2008.

[20] X. Li and S. Iyengar, "On computing mapping of 3d objects: A survey," *ACM Computing Surveys (CSUR)*, vol. 47, no. 2, p. 34, 2015.

[21] C. Bajaj, S. Schaefer, J. Warren, and G. Xu, "A subdivision scheme for hexahedral meshes," *The visual computer*, vol. 18, no. 5, pp. 343–356, 2002.

[22] S. Dong, P.-T. Bremer, M. Garland, V. Pascucci, and J. C. Hart, "Spectral surface quadrangulation," *ACM Transactions on Graphics (TOG)*, vol. 25, no. 3, pp. 1057–1066, 2006.

[23] C.-H. Peng, M. Barton, C. Jiang, and P. Wonka, "Exploring quadrangulations," *ACM Transactions on Graphics (TOG)*, vol. 33, no. 1, p. 12, 2014.

[24] A. C. de Oliveira Miranda and L. F. Martha, "Hierarchical template-based quadrilateral mesh generation," *Engineering with Computers*, pp. 1–15, 2015.

[25] D. Bommes, H. Zimmer, and L. Kobbelt, "Mixed-integer quadrangulation," SIGGRAPH '09, 2009.

[26] D. Bommes, M. Campen, H.-C. Ebke, P. Alliez, and L. Kobbelt, "Integer-grid maps for reliable quad meshing," *ACM Transactions on Graphics (TOG)*, vol. 32, no. 4, p. 98, 2013.

[27] F. Kälberer, M. Nieser, and K. Polthier, "Quadcover-surface parameterization using branched coverings," in *Computer Graphics Forum*, vol. 26, pp. 375–384, Wiley Online Library, 2007.

[28] J. Gregson, A. Sheffer, and E. Zhang, "All-hex mesh generation via volumetric polycube deformation," in *Computer graphics forum*, vol. 30, pp. 1407–1416, Wiley Online Library, 2011.

[29] J. Xia, Y. He, S. Han, C.-W. Fu, F. Luo, and X. Gu, "Parameterization of star-shaped volumes using green's functions," in *Geometric Modeling and Processing*, pp. 219–235, 2010.

[30] B. Ben-Moshe, M. J. Katz, and J. S. B. Mitchell, "A constant-factor approximation algorithm for optimal terrain guarding," in *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pp. 515–524, 2005.

[31] J. O'Rourke and K. Supowit, "Some np-hard polygon decomposition problems," *Information Theory, IEEE Transactions on*, vol. 29, pp. 181 – 190, mar 1983.

[32] D. T. Lee and A. K. Lin, "Computational complexity of art gallery problems," *IEEE Trans. Inf. Theor.*, vol. 32, no. 2, pp. 276–282, 1986.

[33] D. Schuchardt and H.-D. Hecker, "Two np-hard art-gallery problems for ortho-polygons," *Mathematical Logic Quarterly*, vol. 41, no. 2, pp. 261–267, 1995.

[34] A. Efrat and S. Har-Peled, "Guarding galleries and terrains," *Inf. Process. Lett.*, vol. 100, no. 6, pp. 238–245, 2006.

[35] J.-M. Lien, "Approximate star-shaped decomposition of point set data," in *Eurographics Symposium on Point-Based Graphics*, 2007.

[36] J.-M. Lien and N. M. Amato, "Approximate convex decomposition of polygons," *Computational Geometry*, vol. 35, no. 1-2, pp. 100–123, 2006. Special Issue on the 20th ACM Symposium on Computational Geometry.

[37] Y.-K. Lai, S.-M. Hu, R. R. Martin, and P. L. Rosin, "Rapid and effective segmentation of 3d models using random walks," *Comput. Aided Geom. Des.*, vol. 26, no. 6, pp. 665–679, 2009.

[38] D. Cohen-Steiner, P. Alliez, and M. Desbrun, "Variational shape approximation," in *ACM SIGGRAPH*, pp. 905–914, 2004.

[39] A. P. Mangan and R. T. Whitaker, "Partitioning 3D surface meshes using watershed segmentation," *IEEE Transactions on Visualization and Computer Graphics*, vol. 5, no. 4, pp. 308–321, 1999.

[40] S. Shlafman, A. Tal, and S. Katz, "Metamorphosis of polyhedral surfaces using decomposition," in *Computer Graphics Forum*, pp. 219–228, 2002.

[41] N. Gelfand and L. J. Guibas, "Shape segmentation using local slippage analysis," in *Proc. Symposium on Geometry processing*, pp. 214–223, 2004.

[42] N. Cornea, D. Silver, and P. Min, "Curve-skeleton properties, applications, and algorithms," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 13, pp. 530 –548, may. 2007.

[43] T. Dey and J. Sun, "Defining and computing curve-skeletons with medial geodesic function," in *Proc. Eurographics Symp. on Geometry Processing*, pp. 143–152, 2006.

[44] J. Kahn, M. Klawe, and D. Kleitman, "Traditional galleries require fewer watchmen," *SIAM Journal on Algebraic and Discrete Methods*, vol. 4, no. 2, pp. 194–206, 1983.

[45] D. S. Johnson, "Approximation algorithms for combinatorial problems," in *STOC '73: Proceedings of the fifth annual ACM symposium on Theory of computing*, (New York, NY, USA), pp. 38–49, ACM, 1973.

[46] K. Holmstrom, "Tomlab – a general purpose, open matlab environment for research and teaching in optimization," 1998.

[47] H. Hoppe, "Progressive meshes," in *SIGGRAPH*, pp. 99–108, 1996.

[48] W. Yu, M. Li, S. S. Iyengar, and X. Li, "Efficient 3d region guarding for multimedia data processing," in *Multimedia and Expo (ICME), 2011 IEEE International Conference on*, pp. 1–6, IEEE, 2011.

[49] X. Li, Y. Bao, X. Guo, M. Jin, X. Gu, and H. Qin, "Globally optimal surface mapping for surfaces with arbitrary topology," *IEEE Trans. on Visualization and Computer Graphics*, vol. 14, no. 4, pp. 805–819, 2008.

[50] X. Li, X. Gu, and H. Qin, "Surface mapping using consistent pants decomposition," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 4, pp. 558–571, 2009.

[51] Q. Zheng, A. Sharf, A. Tagliasacchi, B. Chen, H. Zhang, A. Sheffer, and D. Cohen-Or, "Consensus skeleton for non-rigid space-time registration," *Computer Graphics Forum*, vol. 29, no. 2, pp. 635–644, 2010.

[52] M. Shapira and A. Rappoport, "Shape blending using the star-skeleton representation," *Computer Graphics and Applications, IEEE*, vol. 15, pp. 44 –50, mar 1995.

[53] M. Alexa, D. Cohen-Or, and D. Levin, "As-rigid-as-possible shape interpolation," in *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '00, pp. 157–164, 2000.

[54] W. Baxter, P. Barla, and K. Anjyo, "Rigid shape interpolation using normal equations," in *Proc. Non-Photorealistic Animation and Rendering*, 2008.

[55] X. Li, W. Yu, X. Lin, and S. Iyengar, "On optimizing autonomous pipeline inspection," *Robotics, IEEE Transactions on*, vol. 28, no. 1, pp. 223–233, 2012.

[56] W. Yu, M. Li, and X. Li, "Optimizing pyramid visibiliy coverage for autonomous robots in 3d environment," in *Computer Science & Education (ICCSE), 2013 8th International Conference on*, pp. 1023–1028, IEEE, 2013.

[57] L. Linardakis and N. Chrisochoides, "Delaunay decoupling method for parallel guaranteed quality planar mesh refinement," *SIAM Journal on Scientific Computing*, vol. 27, no. 4, pp. 1394–1423, 2006.

[58] R. Löhner, “A 2nd generation parallel advancing front grid generator,” in *Proceedings of the 21st international meshing roundtable*, pp. 457–474, Springer, 2013.

[59] T. Panitanarak and S. M. Shontz, “Mdec: Metis-based domain decomposition for parallel 2d mesh generation,” *Procedia Computer Science*, vol. 4, pp. 302–311, 2011.

[60] D. Nave, N. Chrisochoides, and L. P. Chew, “Guaranteed: quality parallel delaunay refinement for restricted polyhedral domains,” in *Proceedings of the eighteenth annual symposium on Computational geometry*, pp. 135–144, 2002.

[61] Y. Ito, A. M. Shih, A. K. Erukala, B. K. Soni, A. Chernikov, N. P. Chrisochoides, and K. Nakahashi, “Parallel unstructured mesh generation by an advancing front method,” *Mathematics and Computers in Simulation*, vol. 75, no. 5, pp. 200–209, 2007.

[62] A. Shamir, “A survey on mesh segmentation techniques,” in *Computer graphics forum*, vol. 27, pp. 1539–1556, Wiley Online Library, 2008.

[63] X. Chen, A. Golovinskiy, and T. Funkhouser, “A benchmark for 3D mesh segmentation,” *ACM Transactions on Graphics (Proc. SIGGRAPH)*, vol. 28, pp. 1–12, Aug. 2009.

[64] M. O. Freitas, P. A. Wawrzynek, J. B. Cavalcante-Neto, C. A. Vidal, L. F. Martha, and A. R. Ingraffea, “A distributed-memory parallel technique for two-dimensional mesh generation for arbitrary domains,” *Adv. Eng. Softw.*, vol. 59, pp. 38–52, May 2013.

[65] J. Gould, D. Martineau, and R. Fairey, “Automated two-dimensional multi-block meshing using the medial object,” in *Proceedings of the 20th International Meshing Roundtable*, pp. 437–452, Springer, 2012.

[66] K. Brix, S. S. Melian, S. Müller, and G. Schieffer, “Parallelisation of multiscale-based grid adaptation using space-filling curves,” in *ESAIM: Proceedings*, vol. 29, pp. 108–129, EDP Sciences, 2009.

[67] M. F. Mokbel and W. G. Aref, “Irregularity in high-dimensional space-filling curves,” *Distributed and Parallel Databases*, vol. 29, no. 3, pp. 217–238, 2011.

[68] P. MacNeice, K. M. Olson, C. Mobarry, R. de Fainchtein, and C. Packer, “Paramesh: A parallel adaptive mesh refinement community toolkit,” *Computer physics communications*, vol. 126, no. 3, pp. 330–354, 2000.

[69] H. Ji, F.-S. Lien, and E. Yee, “A new adaptive mesh refinement data structure with an application to detonation,” *Journal of Computational Physics*, vol. 229, no. 23, pp. 8981–8993, 2010.

[70] A. Maximo, L. Velho, and M. Siqueira, “Adaptive multi-chart and multiresolution mesh representation,” *Computers & Graphics*, vol. 38, pp. 332–340, 2014.

[71] M. Attene, M. Mortara, M. Spagnuolo, and B. Falcidieno, “Hierarchical convex approximation of 3d shapes for fast region selection,” in *Computer graphics forum*, vol. 27, pp. 1323–1332, Wiley Online Library, 2008.

[72] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell system technical journal*, vol. 49, no. 2, pp. 291–307, 1970.

[73] A. Pothen, H. D. Simon, and K.-P. Liou, "Partitioning sparse matrices with eigenvectors of graphs," *SIAM Journal on Matrix Analysis and Applications*, vol. 11, no. 3, pp. 430–452, 1990.

[74] E. Vecharynski, Y. Saad, and M. Sosonkina, "Graph partitioning using matrix values for preconditioning symmetric positive definite systems," *SIAM Journal on Scientific Computing*, vol. 36, no. 1, pp. A63–A87, 2014.

[75] H. Meyerhenke, "Dynamic load balancing for parallel numerical simulations based on repartitioning with disturbed diffusion," in *Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on*, pp. 150–157, 2009.

[76] L. Linardakis and N. Chrisochoides, "A static medial axis domain decomposition for 2d geometries," *ACM Transactions on Mathematical Software*, vol. 34, no. 1, pp. 1–19, 2005.

[77] S. OWEN, M. STATEN, S. CANANN, and S. SAIGAL, "Q-morph: An indirect approach to advancing front quad meshing," *Int. J. Numer. Meth. Engng*, vol. 44, pp. 1317–1340, 1999.

[78] X. Liang, M. S. Ebeida, and Y. Zhang, "Guaranteed-quality all-quadrilateral mesh generation with feature preservation," *Computer Methods in Applied Mechanics and Engineering*, vol. 199, no. 29, pp. 2072–2083, 2010.

[79] S.-W. Chae and J.-H. Jeong, "Unstructured surface meshing using operators," in *Proc. 6th Int. Meshing Roundtable*, pp. 281–291, 1997.

[80] T. D. Blacker and M. B. Stephenson, "Paving: A new approach to automated quadrilateral mesh generation," *International Journal for Numerical Methods in Engineering*, vol. 32, no. 4, pp. 811–847, 1991.

[81] F. Kälberer, M. Nieser, and K. Polthier, "Quadcover-surface parameterization using branched coverings," in *Computer Graphics Forum*, vol. 26, pp. 375–384, Wiley Online Library, 2007.

[82] J.-F. Remacle, F. Henrotte, T. Carrier-Baudouin, E. Béchet, E. Marchandise, C. Geuzaine, and T. Mouton, "A frontal delaunay quad mesh generator using the l norm," *International Journal for Numerical Methods in Engineering*, vol. 94, no. 5, pp. 494–512, 2013.

[83] M. Campen and L. Kobbelt, "Quad layout embedding via aligned parameterization," in *CGF*, vol. 33, pp. 69–81, 2014.

[84] P. Bonami, L. T. Biegler, A. R. Conn, G. Cornuéjols, I. E. Grossmann, C. D. Laird, J. Lee, A. Lodi, F. Margot, N. Sawaya, *et al.*, "An algorithmic framework for convex mixed integer nonlinear programs," *Discrete Optimization*, vol. 5, no. 2, pp. 186–204, 2008.

[85] B. Lévy and Y. Liu, "Lp centroidal voronoi tessellation and its applications," *ACM Trans. Graph.*, vol. 29, no. 4, pp. 119:1–119:11, 2010.

[86] Y. Liu, W. Wang, B. Lévy, F. Sun, D.-M. Yan, L. Lu, and C. Yang, "On centroidal voronoi tessellationÂ¡Âªenergy smoothness and fast computation," *ACM Trans. on Graphics*, vol. 28, no. 4, p. 101, 2009.

[87] S. Fortune, "A sweepline algorithm for voronoi diagrams," *Algorithmica*, vol. 2, no. 1-4, pp. 153–174, 1987.

[88] W. Buell and B. Bush, "Mesh generationÂ¡Âªa survey," *Journal of Manufacturing Science and Engineering*, vol. 95, no. 1, pp. 332–338, 1973.

[89] S. Wan, Z. Yin, K. Zhang, H. Zhang, and X. Li, "A topology-preserving optimization algorithm for polycube mapping," *Computers & Graphics*, vol. 35, no. 3, pp. 639–649, 2011.

[90] W. Yu, K. Zhang, S. Wan, and X. Li, "Optimizing polycube domain construction for hexahedral remeshing," *CAD*, vol. 46, pp. 58–68, 2014.

[91] J. Shepherd, "Topologic and geometric constraint-based hexahedral mesh generation," *PhD Dissertation*, 2007.

[92] L. Marechal, "Advances in octree-based all-hexahedral mesh generation: Handling sharp features," in *Proceedings of the 18th Int'l Meshing Roundtable*, pp. 65–84, 2009.

[93] Y. Ito, A. M. Shih, and B. K. Soni, "Octree-based reasonable-quality hexahedral mesh generation using a new set of refinement templates," *I.J. for Numerical Methods in Eng.*, vol. 77, no. 13, 2009.

[94] M. L. Staten, R. A. Kerr, S. J. Owen, T. D. Blacker, M. Stupazzini, and K. Shimada, "Unconstrained plastering - hexahedral mesh generation via advancing-front geometry decomposition," *Int'l Journal for Numerical Methods in Engineering*, vol. 81, no. 2, pp. 135–171, 2010.

[95] Y. Zhang, Y. Bazilevs, S. Goswami, C. Bajaj, and T. Hughes, "Patient-specific vascular NURBS modeling for isogeometric analysis of blood flow," *Computer Methods in Applied Mechanics and Engineering*, vol. 196, no. 29-30, pp. 2943–2959, 2007.

[96] M. Nieser, U. Reitebuch, and K. Polthier, "Cubecover - parameterization of 3d volumes," *CGF*, vol. 30, no. 5, 2011.

[97] J. Huang, T. Y, H. Wei, and H. Bao, "Boundary aligned smooth 3d cross-frame field," *ACM Trans. Graph.*, vol. 30, no. 6, p. 143, 2011.

[98] X. Li, X. Guo, H. Wang, Y. He, X. Gu, and H. Qin, "Harmonic volumetric mapping for solid modeling applications," in *Proc. ACM SPM*, pp. 109–120, 2007.

[99] Z. Fan, X. Jin, J. Feng, and H. Sun, "Mesh morphing using polycube-based cross-parameterization: Animating geometrical models," *Comput. Animat. Virtual Worlds*, pp. 499–508, 2005.

[100] M. Tarini, K. Hormann, P. Cignoni, and C. Montani, "Polycube-maps," *ACM Trans. Graph.*, vol. 23, no. 3, pp. 853–860, 2004.

[101] H. Wang, Y. He, X. Li, X. Gu, and H. Qin, "Polycube splines," *Computer Aided Design*, vol. 40, no. 6, pp. 721–733, 2008.

[102] J. Lin, X. Jin, Z. Fan, and C. C. L. Wang, "Automatic polycube-maps," in *Proc. GMP*, pp. 3–16, 2008.

[103] J. Xia, I. Garcia, Y. He, S. Xin, and G. Patow, "Editable polycube map for gpu-based subdivision surfaces," in *Symposium on Interactive 3D Graphics and Games*, pp. 151–158, 2011.

[104] S. Wan, Z. Yin, K. Zhang, H. Zhang, and X. Li, "A topology-preserving optimization algorithm for polycube mapping," *Comput. Graph.*, vol. 35, no. 3, pp. 639–649, 2011.

[105] I. Jollife, *Principal Comp. Analysis.* Springer-Verlang, 1986.

[106] E. Larsen, S. Gottschalk, M. Lin, and D. Manocha, "Fast distance queries with rectangular swept sphere volumes," in *Proc. Int'l Conf. Robotics and Automation*, pp. 3719 –3726, 2000.

[107] S. Biasotti, L. De Floriani, B. Falcidieno, P. Frosini, D. Giorgi, C. Landi, L. Papaleo, and M. Spagnuolo, "Describing shapes by geometrical-topological properties of real functions," *ACM Comput. Surv.*, vol. 40, no. 4, pp. 1–87, 2008.

[108] H. Wang, Y. He, X. Li, X. Gu, and H. Qin, "Geometry-aware domain decomposition for t-spline-based manifold modeling," *Comput. Graph.*, vol. 33, no. 3, pp. 359–368, 2009.

[109] V. Pascucci, G. Scorzelli, P.-T. Bremer, and A. Mascarenhas, "Robust on-line computation of reeb graphs: simplicity and speed," *ACM Trans. Graph.*, vol. 26, no. 3, pp. 58.1–58.9, 2007.

[110] J. Serra, *Image Analysis and Mathematical Morphology.* Academic Press, Inc., 1983.

[111] S. Matveev, *Algorithmic Topology and Classification of 3-Manifolds.* Springer-Verlag Berlin, 2003.

[112] Q. Zhou, T. Ju, and S. Hu, "Topology repair of solid models using skeletons," *IEEE Trans. on Visualization and Comput. Graphics*, vol. 13, no. 4, 2007.

[113] Y. Wang, X. Gu, T. F. Chan, P. M. Thompson, and S. T. Yau, "Volumetric harmonic brain mapping.," in *IEEE Int'l Symp. on Biomedical Imaging: Macro to Nano.*, pp. 1275–1278, 2004.

[114] H. Xu, W. Yu, S. Gu, and X. Li, "Biharmonic volumetric mapping using fundamental solutions," *IEEE Trans. on Visualization and Computer Graphics*, vol. 19, pp. 787–798, 2013.

[115] T. Martin, E. Cohen, and R. Kirby, "Volumetric parameterization and trivariate b-spline fitting using harmonic functions," in *Proc. ACM Solid and Physical Modeling*, 2008.

[116] T. Martin, G. Chen, S. Musuvathy, E. Cohen, and C. D. Hansen, "Generalized swept mid-structure for polygonal models," *Comput. Graph. Forum*, vol. 31, no. 2, pp. 805–814, 2012.

[117] TetGen, "http://tetgen.berlios.de/," 2007.

[118] Y. Zhang and C. Bajaj, "Adaptive and quality quadrilateral/hexahedral meshing from volumetric imaging data," in *Comp. Methods in A.M.E.*, 2006.

[119] M. Brewer, L. Diachin, P. Knupp, T. Leurent, and D. Melander, "The mesquite mesh quality improvement toolkit," in *IMR*, 2003.

[120] J. Huang, T. Jiang, Z. Shi, Y. Tong, H. Bao, and M. Desbrun, "&ell; 1-based construction of polycube maps from complex shapes," *ACM Transactions on Graphics (TOG)*, vol. 33, no. 3, p. 25, 2014.

[121] O. Sorkine and M. Alexa, "As-rigid-as-possible surface modeling," in *Symposium on Geometry processing*, vol. 4, 2007.

[122] J. Solomon, M. Ben-Chen, A. Butscher, and L. Guibas, "As-killing-as-possible vector fields for planar deformation," in *Computer Graphics Forum*, vol. 30, pp. 1543–1552, Wiley Online Library, 2011.

[123] K. Hormann and G. Greiner, "Mips: An efficient global parametrization method.," in *Curve and Surface Design*, pp. 153–162, 2000.

[124] X.-M. Fu, Y. Liu, and B. Guo, "Computing locally injective mappings by advanced mips," *ACM Transactions on Graphics (TOG)*, vol. 34, no. 4, p. 71, 2015.

[125] L. Lukšan, C. Matonoha, and J. Vlček, "Trust-region interior-point method for large sparse l 1 optimization," *Optimisation Methods and Software*, vol. 22, no. 5, pp. 737–753, 2007.

[126] W. Yu, K. Zhang, S. Wan, and X. Li, "Optimizing polycube domain construction for hexahedral remeshing," *Computer-Aided Design, (Proceedings SIAM/ACM Conference on Geometric Design and Modeling 2013)*, vol. 46, pp. 56–68, 2013.

VITA

Wuyi Yu was born in the city of Xiamen, Fujian Province in China in 1983. He graduate from Xiamen University, China in July 2005 with the bachelor's degree in Automation. Later he joined the doctoral program in the Automation Department from Xiamen University. In 2011, he began the doctoral program in the Department of Computer Science from Louisiana State University, Baton Rouge, Louisiana. Wuyi Yu will graduate in May 2016 with the degree of Doctor of Philosophy in Computer Science.