Centroidal Voronoi Tessellation and Its Applications

Wang Xiaoning

School of Computer Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfilment of the requirement for the degree of
Doctor of Philosophy

2015

Abstract

xxxxxxxxxxxxxxx

Acknowledgments

Many people have helped to the production of this thesis, though only my name appears on it. I would like to express my deepest gratitude to all those people who have made this thesis possible.

It gives me immense pleasure to thank my supervisor, Prof. He Ying, for his great efforts of support and continuous guidance. His patience and meticulous suggestions helped me overcome many difficulties and finish this thesis. It's a good fortune for me to have a good advisor as Prof. He. Also many thanks to Prof. Wolfgang Mueller-Wittig, for his kindly support to my research.

I also wish to thank all the people at Fraunhofer research center for their help, and all my friends of the Geometry Modeling Group at gameLab who supported me a lot by providing valuable feedback in many fruitful discussions. This thesis would not be possible in this form without the support and collaboration of them, in particular, Ms. Cheng Peng, Dr. Dao Thi Phuong Quynh, Mr. Fang Zheng, Dr. Hou Fei, Mr. Lai Chi-Fu William, Mr. Le Tien Hung, Dr. Leung Yuen Shan, Dr. Meng Min, Dr. Sun Qian, Dr. Wang Dayong, Ms. Wang Jin, Dr. Xin Shiqing, Ms. Yan Xiaoqi, Dr. Ying Xiang, Dr. Zhang Minqi.

I would like to thank my parents for their support during my study. Last but not least, many thanks to my girl friend, Ms. Wang Anran, for her warm encouragement and trust.

Publications

Published:

Xiang Ying, **Xiaoning Wang**, Ying He. Saddle vertex graph (SVG): a novel solution to the discrete geodesic problem. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH ASIA 2013)*, Vol. 32, No. 6, pp. 170:1-170:12, 2013.

Shi-Qing Xin, **Xiaoning Wang**, Jiazhi Xia, Wolfgang Mueller-Wittig, Guo-Jin Wang, Ying He. Parallel computing 2D Voronoi diagrams using untransformed sweepcircles. *Computer-Aided Design (CAD)*, Vol. 45, No. 1, pp. 483-493, 2013.

Xiaoning Wang, Xiang Ying, Yong-Jin Liu, Shi-Qing Xin, Wenping Wang, Xianfeng Gu, Wolfgang Mueller-Wittig, Ying He. Intrinsic computation of centroidal Voronoi tessellation (CVT) on meshes, *Computer-Aided Design (CAD)*, Vol. 58, No. 1, pp. 51-61, 2015. **(Best Paper Award 3rd Place in Solid and Physical Modeling 2014)**

Yuen Shan Leung, **Xiaoning Wang**, Ying He, Yong-Jin Liu, Charlie C.L. Wang. Robust and GPU-friendly Isotropic Meshing Based on Narrow-banded Euclidean Distance Transformation. Pacific Graphics 2015. (Short Paper)

Peihong Guo, Ergun Akleman, He Ying, **Xiaoning Wang**, and Wei Liu. 2015. Critical points with discrete Morse theory. ACM SIGGRAPH 2015 Posters, 67:1-67:1, 2015.

Ready to submit:

Xiaoning Wang, Le Tien Hung, Qian Sun, Xiang Ying, Wolfgang Mueller-witting, Ying He. User Controllable Anisotropic Shape Distribution on 3D Meshes. . A General Graph-Theoretic Framework for Computing Discrete Geodesics.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Statement

1.1.1 Voronoi Diagram

Voronoi diagram is a special kind of decomposition of a space, determined by distances to a given set of objects in the space. As a fundamental geometric data structure, Voronoi diagrams have been widely applied in various computer graphics and visualization applications, including collision detection [130], meshing [152], vector field visualization [92], artistic effect, image segmentation [140], just to name a few. Among many efficient algorithms for construction of Voronoi diagram,Fortune's sweep line algorithm [49] is popular due to its elegancy and simplicity. Given a set of 2D points (called *sites*), the sweep line algorithm maintains a vertical sweep line moving from left to right across the plane as the algorithm progresses. At any time during the algorithm, the input sites on the left side of the sweep line have already been incorporated into the Voronoi diagram, while the sites right of the sweep line have not been considered yet. Each site to the left of the sweep line generates a parabola of points equidistant from that point and from the sweep line. The wavefronts of all parabolas form a curved beach line, which follows the sweep line moving from left to right. As the sweep line progresses,

the vertices of the beach line, at which two parabolas cross, trace out the edges of the Voronoi diagram. The sweep line algorithm is proven to be optimal with $O(n\log n)$ time complexity and $O(n)$ space complexity.

In spite of its simplicity and popularity, there is no effective technique to parallelize the sweep line algorithm.Thurston [136] pointed out that if the input vertex set is very large, then the sweepcircle technique allows one to compute the Voronoi diagram locally [74]. Dehne and Klein [32] proposed a *sweepcircle* technique to compute the *transformed* Voronoi diagram, where the transformation is made in polar coordinates:

$$x \triangleq (\rho, \theta) \longrightarrow x' \triangleq (\rho + |x - x_i|, \theta), \tag{1.1}$$

where x_i is the nearest site to x. The purpose of this transformation is to guarantee that the site s is touched before the Voronoi cell of s is swept by a sweeping circle. They proved that the transformed diagram has the same combinatorial structure as the original Voronoi diagram [32]. This transformed sweepcircle algorithm is parallel in nature and can be also applied to compute the Voronoi diagram on cones. However, it is difficult to represent the transformed Voronoi edges from the implementation perspective, since they are not straight line segments or conic curves anymore. To our knowledge, there is no practical implementation of the sweepcircle algorithm.

1.1.2 Centroidal Voronoi Diagram

A centroidal Voronoi tessellation (CVT) is a particular Voronoi tessellation of a compact domain in Euclidean space yielded by a set of samples such that each site locate at the same place with the centroid of its Voronoi cell. For example, Figure1.1(a) shows a Voronoi tessellation of a circular domain where the sites do not locate at the same place with the centroids of the Voronoi cells (not a CVT), while Figure1.1(b) shows a CVT of the same domain. CVT generates an evenly spaced distribution of sites in the domain

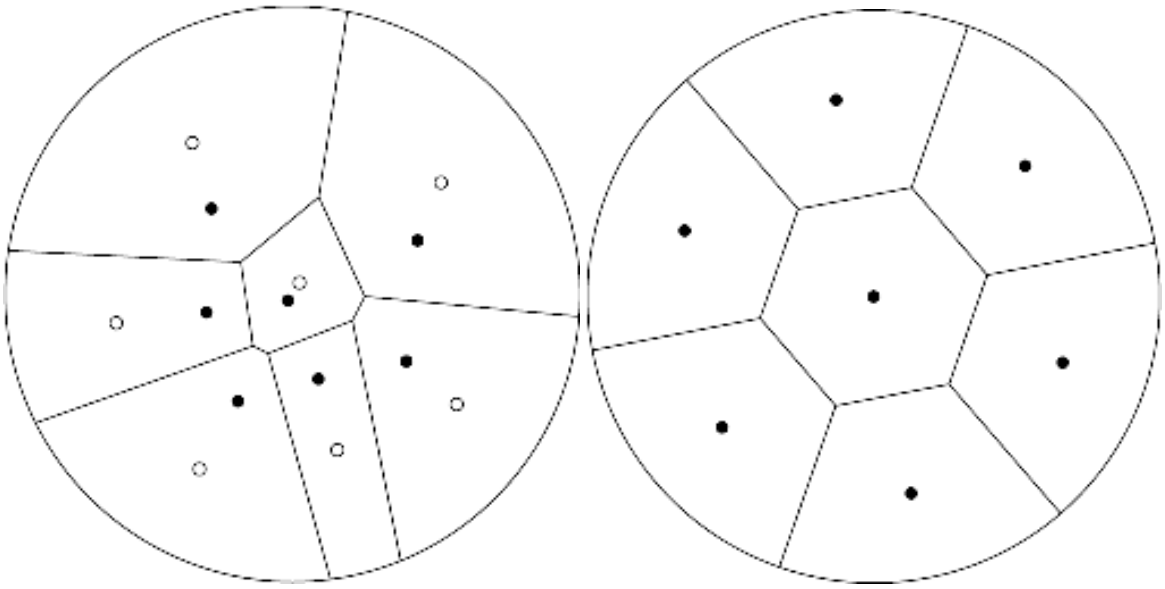


Figure 1.1: (a) An ordinaryVoronoi tessellation of a circular domain with seven sites marked with black dots and the the centroids of the Voronoi cells marked with small circles; (b) a CVT of the same domain with seven sites.Anisotropy sampling is useful in many geometry applications, such as object distribution, vector field visualization. Figures are taken from [83]

with respect to a given density function, and is therefore very useful in many fields, such as optimal quantization, clustering, data compression, optimal mesh generation, cellular biology, optimal quadrature, coverage control, and geographical optimization. An excellent introduction to the theory and applications of CVT is given in Du et al. [38] and Okabe et al. [101].

CVT has recently been applied to the surface remeshing problem. Surface meshing is a fundamental problem due to the dominance of triangle meshes in computer graphics and visualization. The past decade has witnessed significant progress in surface meshing. Among many promising techniques, the centroidal Voronoi tessellation (CVT) has received much attention recently. CVT is a special kind of Voronoi diagram (VD) in which the generating point of every Voronoi cell is also its center of mass [38].

Most of the existing CVT techniques focus on the numerical solver of the CVT energy function. The Lloyd algorithm [89] iteratively moves the generator of Voronoi cell to its mass center. Although being conceptually simple and easy to implement, the Lloyd algorithm has only linear convergence rate. Liu et al. [83] proved that the energy function of CVT is C^2 continuous in general. As a result, we can minimize the CVT energy by the Newton or quasi-Newton solver, which converges much faster than Lloyd algorithm. However, both solvers require computing the Voronoi diagrams in each iteration. It is fairly simple to construct the Voronoi diagrams in Euclidean space (e.g. $\mathbb{R}^2$ and $\mathbb{R}^3$), since many efficient algorithms and software tools are readily available. However, it is technically challenging to compute VD on curved surfaces. Therefore, some researchers tackle this challenge by computing the restricted Voronoi diagrams [153], which is the intersection between a 3D Voronoi diagram and an input triangle mesh. These approaches are embedding space dependent and may fail for models with complicated geometry. Others rely on the global parameterization, where the input surface is parameterized to the Euclidean plane $\mathbb{E}^2$, sphere $\mathbb{S}^2$, or hyperbolic disk $\mathbb{H}^2$. Then the 2-dimensional CVT is computed in the parametric domain, which induces a meshing on the input triangle mesh. It is known that the global parameterization is computational high cost and may cause some numerical issue if the parameterization has large distortion.

1.2 Objectives

This report aims to find a fully parallel sweep-circle algorithm for calculating a Voronoi Diagram,and a parallel and intrinsic algorithm for generating CVT on a surface.

1.2.1 Fast Parallel Algorithm for a Voronoi Diagram

This report aims to tackle the challenge of fast computing a Voronoi Diagram by proposing a new algorithm, called the *untransformed sweep-circle*, for computing a 2D Voronoi diagram. Starting with a degenerate circle (of zero radius) centered on an arbitrary point (not necessarily a site), as the name suggests, our algorithm sweeps the circle by increasing its radius across the plane. At any time during the sweeping process, each site inside the sweep-circle defines an ellipse composed of points equidistant from that point and from the sweep-circle. The union of all ellipses forms the beach curve, a star shape inside the sweep-circle which divides the portion of the plane within which the Voronoi diagram can be completely determined, regardless of what other points might be outside of the sweep-circle. As the sweep-circle progresses, the intersection of expanding ellipses traces out the Voronoi edges.

We show that the sweep-line algorithm is a degenerate form of the proposed sweep-circle when the circle center is at infinity, and our algorithm also has the same time and space complexity as its line counterpart. However, our algorithm fundamentally distinguishes itself from the sweep-line algorithm in that it allows multiple circles at arbitrary locations to sweep the domain simultaneously. This leads to a very natural parallel implementation, while the sweep-line algorithm does not have such a feature. Compared with the traditional sweep-circle technique [32], our algorithm is fairly easy to implement, since the most complicated numerical operation is nothing more than an arc cosine calculation. Furthermore, our algorithm supports the computation of additively weighted Voronoi diagrams in which the Voronoi edges are hyperbolic and straight line segments.

1.2.2 Centroid Voronoi Diagram on a Surface, and Remeshing

This report presents a fully intrinsic and parallel algorithm for calculating CVT on a surface.

This report falls into the category of CVT-based meshing. Instead of using global parameterization or relying on the embedding space, we propose a new algorithm to calculate the CVT on surfaces directly. Our key idea is to use exact geodesic distance to calculate intrinsic CVT. It is worth noting that our method is independent of the embedding space and fully intrinsic. We propose an effective algorithm to compute the approximate CVT on arbitrary surfaces. Compared to the conventional CVT techniques, our algorithm is computationally efficient, conceptually simple, and easy to implement.

Based on our new CVT algorithm, we develop a computational framework to generate high quality isotropic and adaptive triangle meshes. Compared to the existing meshing techniques, our method is completely intrinsic and insensitive to the resolution/tessellation of the input surfaces. We demonstrate the efficacy of our method in a variety of real-world models. Thanks to its intrinsic nature, our method works well for models with arbitrary topology and complicated geometry, where the existing extrinsic approaches often fail.

Meshes are an important way to represent shapes. A large number of these meshes are generated by scanning devices, isosurfacing implicit representations or modeling software. These meshes often lack the properties desired for subsequent geometric processing. For example, a poor triangulation may result in the failure of some numerical methods. Therefore, surface remeshing has been extensively studied in the past 20 years. The readers can refer [109, 135, 3] for a complete survey.

Surface remeshing may have various purposes, depending on the type of application in which it is used. For instance, structured meshes offer a simpler connectivity graph,

thus allowing for efficient localization and traversal in the algorithms. In this report, we aim at proposing a high-quality remeshing algorithm, i.e., approximating the model geometry with a high-quality triangle mesh while preserving geometric features. We are aware that, Liu et al. [83] proposed an elegant method for this purpose. They found that by considering Centroid Voronoi Tesselation (CVT) in the embedding space, it is possible to obtain a group of uniformly distributed seed points that define a high-quality triangle mesh. However, Rong et al. [111] pointed out that their algorithm is not intrinsic and therefore cannot deal with models with narrow and flat parts or in higher dimensional space.

Observing that a discrete exponential map [115] can bridge the gap between 2D CVT and uniform point distribution on 2-manifolds, we propose a new framework for surface remeshing. Initially, we get a collection of random distributed sample points. Then in a parallel style, we locally map the seed points onto the tangent plane by discrete exponential mapping , calculate their centroid in 2D,then map the centroid back to the surface. We repeat this process until the energy magnitude is within a prescribed tolerance. Finally, we extract the re-triangled mesh conforming to the geodesic Delaunay triangulation. By encoding the geometric feature information into a density function, our framework can easily support an adaptive distribution of seed points. Compared with existing remeshing algorithms, our algorithm is parametrization free, adaptive, intrinsic, parallel in nature and insensitive to the original triangle quality.

1.3 Thesis Organization

In the following report, we first review the related work on conformal parametrization for computer graphics in Chapter **??**, then we introduce the mathematical background for conformal map in Chapter **??**. In Chapter **??** we will discuss our work to compute

extremal quasi-conformal map on general surface. Chapter **??** presents the work by using ricci flow and extremal quasi-conformal map to build the surface registration on brainstem for the study of AIS, in addition in Chapter **??** we use the holomorphic 1-form and extremal quasi-conformal map to build the registration of the vestibular system. In Chapter **??**, we propose a solution for 2D/3D sensor network greedy routing by applying a parametrization similar to the conformal map to build the virtual coordinate system first. In Chapter **??**, we use the conformal map as a basic image editing tool to generalize the traditional Escher-Droste effect. Lastly, we conclude our work and provide some future directions for conformal maps in Chapter **??**.

Chapter 2

Related Work

This chapter surveys some related work in Voronoi Diagram , Centroid Voronoi Diagram and discrete geodesics.

2.1 Voronoi Disgram

Voronoi diagrams are a fundamental geometric structure, which can be traced back to René Descartes in 1644. It is named after the Russian mathematician Georgy Fedoseevich Voronoi who defined and studied the general n-dimensional case in 1908. Since then, Voronoi diagrams have been extensively studied and widely applied to many science and engineering fields, including computer graphics, image processing, robot navigation, computational chemistry, materials science, climatology, etc [8, 101], due to its nice geometric properties.

As the dual of Voronoi diagrams, Delaunay triangulations also possess a number of useful properties. For example, among all triangulations of a planar point set the Delaunay triangulation maximizes the minimum angle, which is helpful for quality mesh generation [124, 50]. Also, in all dimensions, the Euclidean minimum spanning tree is a subgraph of the Delaunay triangulation [31]. There are a large array of literature on

the Delaunay triangulation problem. For example, Devillers [34] proposed a hierarchial data structure to compute the Delaunay triangulation on a 2D plane. Both Voronoi diagram and Delaunay triangulation can be computed in the optimal $O(n \log n)$ time for a set of n sites in 2D [74]. Among these algorithms, some are in a divide-and-conquer scheme [123, 61, 41], while others are incremental [56, 128]. The same optimal worst-case bound is also obtained by Fortune's elegant sweep line algorithm [49] and the traditional sweepcircle algorithm [32].

For Voronoi diagram of moving points, Albers et al. [2] showed the upper bound of the number of topological events and presented a numerically stable algorithm for the update of the topological structure of Voronoi diagram, using only $O(\log n)$ time per event. Zhou et al. [160] presented the bi-cell filtering technique to utilize the connectivity coherence of the Delaunay triangulation of moving points, which leads to an efficient algorithm to compute the dynamic Delaunay triangulation.

For 3D Voronoi diagram, Browstow et al. [16] suggested computing the facets of each Voronoi region, from which the edges and vertices of the region can be constructed. Their algorithm is of an $O(n^4)$ time complexity theoretically but runs in $O(n^2 \log n)$ time in practice. Finney [47] proposed an algorithm to compute the vertices of each Voronoi region, and then compute the edges and facets of the region by identifying the sites that define each vertex. Tanemura et al. [90] presented an algorithm that determines the Voronoi region for each site by computing all Delaunay tetrahedra, which have the site as a common vertex. By representing the 3D domain as a 3D tetrahedral mesh, Yan et al. [154] presented an efficient algorithm for computing the 3D clipped Voronoi diagram.

There are also some algorithms [56, 142, 9] available for d-dimensional Voronoi diagrams ($d \geq 3$). The combinatorial complexity of the Voronoi diagram is $\Theta(n^{\lceil d/2 \rceil})$ [73]

and it can be computed in $O(n \log n + n^{\lceil d/2 \rceil})$ optimal time [19, 23, 120]. The duality of the Voronoi diagram and the Delaunay triangulation still holds in higher dimensions. Guibas et al. [60] proved that Delaunay triangulations can be computed in $O(n^{\lceil d/2 \rceil})$ time through randomized incremental point insertion.

There are several techniques, such as randomization [110] and divide-and-conquer [25], for parallel computation of Voronoi diagrams. However, these parallel algorithms are rather complicated and highly non-trivial to implement. To improve the performance of computing the Voronoi diagram, many researchers focus on computing approximate Voronoi diagram on discrete spaces with the help of the modern graphics hardware. Hoff III et al. [63] developed a highly efficient algorithm for computing generalized Voronoi diagrams in 2D and 3D using rasterization hardware. The algorithm first divides the space into regular cells; then for each cell it computes the closest primitive and the distance to that primitive using polygon scan-conversion and Z-buffer depth comparison. Sud et al. [130] computed the 2nd order discrete Voronoi diagram using GPU and applied it to perform N-body culling. Very recently, Rong et al. [112] presented a GPU-assisted Voronoi diagram algorithm for accelerating the computation of Centroidal Voronoi Tessellation (CVT). These graphics hardware accelerated algorithms are efficient, but produce only a discrete approximation of the Voronoi diagram. Our algorithm, in contrast, computes the exact 2D Voronoi diagram in parallel. Besides, Cao et al. [18] proposed a parallel banding algorithm on the GPU to compute the distance map for a binary image.

To compute the geodesic Voronoi diagram on a 2-manifold mesh M, the Euclidean norm is replaced by the geodesic metric. Such a metric change results in fundamental change in the bisector and Voronoi regions between the Euclidean plane and a curved 2-manifold. For example, a bisector in $\mathbb{R}^2$ is a line segment, while a bisector on a triangle mesh may contain both line segments and hyperbola segments. Given a genus-g

mesh M, it was shown in [82] that the bisector of two distinct points on M can have at most $g+1$ separated components and a Voronoi region of a point in $\mathbf{S}$ can be bounded by one or more closed bisectors (see Figure 2.1). Liu [84] proved that the combinatorial complexity of geodesic Voronoi diagram on M is $O((m+g)n)$, where m is the number of points in $\mathbf{S}$ and n is the number of triangles in the mesh. Edelsbrunner and Shah [44] showed that for a general topological space, if a closed ball property is satisfied, then the dual Delaunay triangulation of Voronoi diagram exists. Recently, Liu et al. [86] showed that the intrinsic Delaunay triangulation on mesh M can be obtained by the duality of a geodesic Voronoi diagram on M. They proved that this duality exists under two practical assumptions and proposed an efficient algorithm for constructing the Delaunay triangulation.

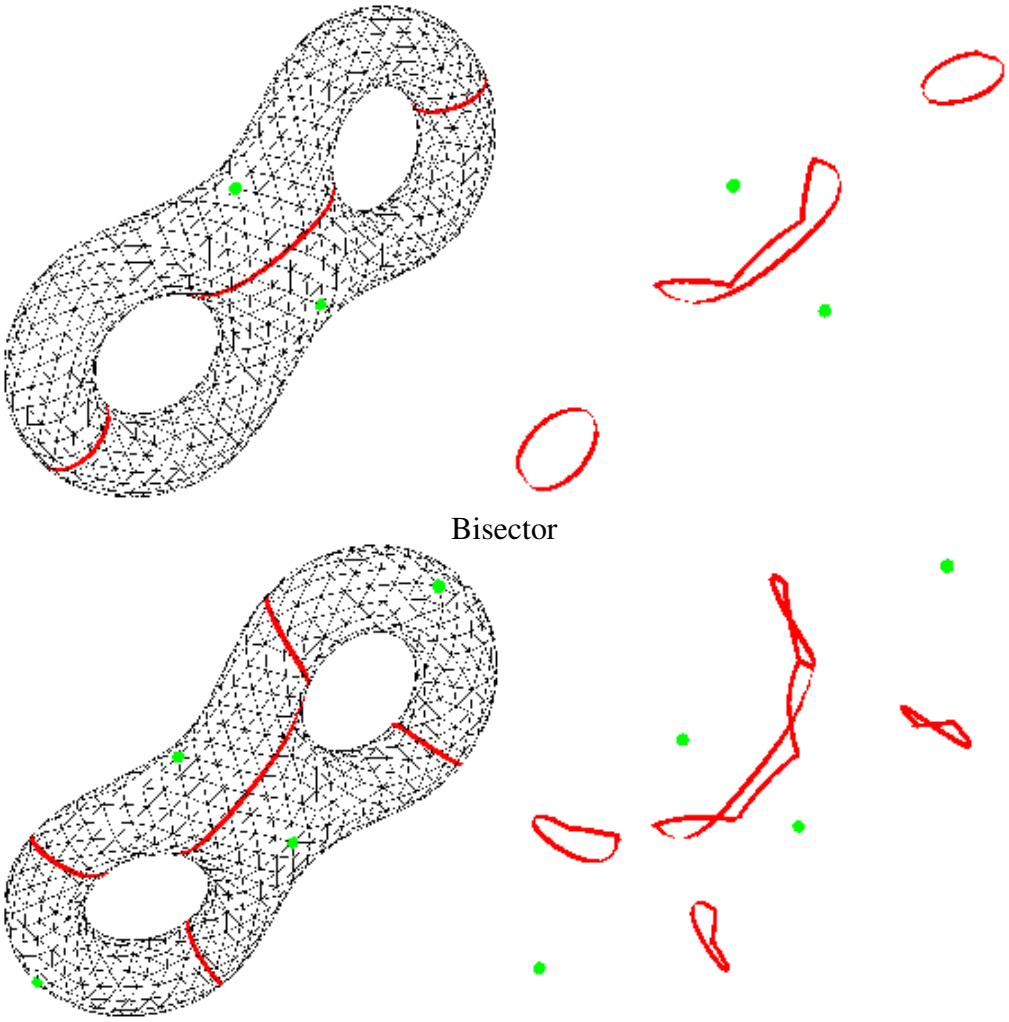
We refer the readers to [42] for in-depth discussion on the properties of Delaunay triangulation and Voronoi diagram on a 2-manifold mesh.

2.2 Centroid Voronoi Diagram

Du et al. [39] first proposed the notion of the centroidal Voronoi tessellation, but long before that the similar notions have been presented in varied areas, e.g. k-means in pattern recognition and optimal quantization in signal processing.

The centroidal Voronoi tessellation (CVT) is a special Voronoi diagram in which every seed x_i locates at the same place with the centroid c_i of its Voronoi cell Ω_i ,and was first introduced by [39]. But long before similar concepts have been studied in various areas, e.g. k-means in computer vision, and pattern recognition.

To compute the CVT, one of the earliest algorithms is introduced by [91], which is a probabilistic algorithm. Although it will surely convergence, its convergence is quite



Bisector

Geodesic Voronoi diagram

Figure 2.1: Bisector and geodesic Voronoi diagram. Row 1: The bisector (red) of two sites (green) on the double-torus has three separated components. Besides line segments, a bisector on triangle mesh may also contain hyperbola segments. Row 2: The geodesic Voronoi diagram of four sites. Each Voronoi cell is bounded by two or three closed bisectors.

slow. Lloyd presented a deterministic method, which is published officially in [89]. Du et al. [37] proved the convergence of Lloyds algorithm later. Because of its robustness and simplicity , Lloyds algorithm is used widely. Ju et al. [66] combined Lloyds algorithm with MacQueens method, and presented a parallel probabilistic Lloyd algorithm. Liu et al. [83] proved that the energy function of CVT has C^2 smoothness in convex domains and in most other commonly encountered domains with smooth density. With this C^2 continuity, one can compute CVT by the quasi-Newton method (e.g. L-BFGS [81]), which has super linear convergence rate, thus, significantly outperforms the Lloyd algorithm. Lévy and Liu [79] presented L_p CVT, which generalizes CVT by minimizing a higher-order moment of the coordinates on the Voronoi cells. This generalization allows for aligning the axes of the Voronoi cells with a predefined background tensor field (anisotropy). Recently, Rong et al. [111] extended CVT from Euclidean space to hyperbolic and spherical space by using global parameterization.

Meshes are an important way to represent shapes. A large body of these meshes are generated by modeling software or scanning devices. These meshes often lack properties desired for subsequent geometric processing. For example, a poor triangulation may result in the failing of some numerical methods. Therefore, surface remeshing has been extensively studied in the past 20 years. The readers can refer [109, 135, 3] for a complete survey.We would only review the algorithms based on CVT [39]. Readers can read citeaurenhammer1991voronoi , citeokabe2009spatial for fundamental concepts and many previous works on Delaunay triangulation and Voronoi diagram.

Geodesic Voronoi diagram(GVD) is the Voronoi diagram on a surface is defined by geodesic distance from an intrinsic point of view. Peyre and Cohen [106] make use of a discrete approximation of geodesic distance to calculate an approximation GVD on a surface. Kunze et al. [76] calculate GVD on parametric surfaces. But exact GVD is

still hard to compute, and the existing approximation methods are quite slow.. And we propose a efficient exact GVD computation algorithm.

Mesh parameterization [6] [7] is another approach for calculating the CVT on a surface. In the approach the input surface is parameterized to the sphere $\mathbb{S}^2$, hyperbolic disk $\mathbb{H}^2$ or Euclidean plane $\mathbb{E}^2$. Then the 2-dimensional CVT is computed in the parametric domain, which induces a meshing on the input surface. It is known that the global parameterization is computational high cost and may cause some numerical issue if the parameterization has large distortion. And a major problem with the remeshing method base on parameterization is that one have to find complicated method to combine parameterized parts together for a high genus surface. Nevertheless, these algorithms have inaccuracies caused by instabilities with poorly shaped triangles and the distortions of the parametrization. Alliez et al. [7] compute a discrete RVD for tetrahedral meshing using dense sample points to approximate a triangle mesh. Valette et al. [139] cluster mesh triangles to compute an approximated RVD. These methods run fast but when the input mesh has degenerate triangles the remeshing result may be poor. .

2.3 Existing Work

2.3.1 Voronoi Diagram & Centroidal Voronoi Tessellation in $\mathbb{R}^2$

Let $\mathbf{S}= (s_i)_{i=1}^m$ be a set of distinct sites in a connected compact region $\Omega \subset \mathbb{R}^2$. The Voronoi region Ω_i of s_i is defined as:

$$\Omega_i = \{x \in \Omega \mid ||x - s_i|| \leq ||x - s_j||, \forall i \neq j\},$$

where $\|\cdot\|$ denotes the Euclidean norm. The Voronoi regions, Ω_i, of all the sites form the Voronoi diagram of $\mathbf{S}$. The VD is a fundamental geometric tool that has a wide range of applications in science, engineering and even art. The classic algorithms for

constructing Voronoi diagram in $\mathbb{R}^2$ are the sweep line algorithm [49] and the divide-and-conquer algorithm [123], which have optimal time complexity $O(m \log m)$.

Let the domain Ω be endowed with a density function $\rho(x) > 0$, which is assumed to be C^2. A typical energy function on Ω with regard to the Voronoi tessellation is defined as follows [39]:

$$F(\mathbf{S}) = \sum_{i=1}^{m} \int_{\Omega_i} \rho(x)||x - s_i||^2 d\sigma \triangleq \sum_{i=1}^{m} F_i,$$

where the term F_i expresses the compactness (or inertia momentum) of the Voronoi cell Ω_i associated with the site s_i. The Voronoi tessellation $\{\Omega_i\}$ is said to be a centroidal Voronoi tessellation if each site s_i coincides with the centroid c_i of its Voronoi cell, that is:

$$s_i = c_i (= \frac{\int_{\Omega_i} \rho(x) x d\sigma}{\int_{\Omega_i} \rho(x) d\sigma}).$$

The Lloyd algorithm [89] iteratively moves the generator of Voronoi cell to its mass center. Although it is conceptually simple and easy to implement, the Lloyd algorithm has only linear convergence rate. Liu et al. [83] proved that the CVT energy function is C^2 continuous. As a result, one can minimize the CVT energy by the Newton or quasi-Newton method, which converges much faster than the Lloyd algorithm.

2.3.2 Computing CVT on Surfaces

Both the Lloyd algorithm and the Newton algorithm require computing the Voronoi diagrams in each iteration. Although it is fairly simple to construct the VD in Euclidean space (e.g. $\mathbb{R}^2$ and $\mathbb{R}^3$), computing VD on curved surfaces is technically challenging. Alliez et al. [4] [5] conformally parameterized genus-0 open surface to a disk and evaluated the centroids over the density function expressed in parameter space rather than on the surface. Thanks to the angle-preserving and local isotropic properties of conformal

parameterization, a well-shaped triangle in parameter space will not be deformed too much once lifted back into $\mathbb{R}^3$, except for its size, which can be easily compensated by the weighted density function in $\mathbb{R}^2$. Rong et al. [111] generalized the CVT energy function from $\mathbb{R}^2$ to spherical space $\mathbb{S}^2$ and hyperbolic space $\mathbb{H}^2$ and then combined all of them into a unified framework - the CVT in universal covering space (UCS). They adapted Lloyd's iteration to compute the CVT in the embedded fundamental domain of the UCS. If a centroid is outside of the fundamental domain by one side, they performed a rigid motion to move it to the oppositeside of the fundamental domain. The adjusted centroids are all inside the fundamental domain and are used as the new sites in the next iteration. Rong et al. [112] proposed a GPU-based method for computing the CVT on the plane and observed significant speedup of these GPU-based methods over their CPU counterparts. Their method also works for some 3D models that can be represented as a geometry image. However, as mentioned above, global parametrization is computationally expensive and may suffer from serious numerical issues if the surface has complicated geometry and non-trivial topology. For example, the Bunny's ears are shrunk to very tiny regions under spherical conformal parameterization [57], which poses a great numerical challenge to compute the CVT on $\mathbb{S}^2$.

Yan et al. [152] proposed a different approach. Rather than constructing the CVT on the mesh directly, they repeatedly computed restricted Voronoi diagrams (RVD), defined as the intersection between the input mesh and a Voronoi diagram in $\mathbb{R}^3$. Their method uses a kd-tree to quickly identify and compute the intersection of each triangle face with its incident Voronoi cells. The time complexity for computing RVD is $O(m \log n)$, where n is the number of seed points and m is the number of triangles of the input mesh. Their method also adopted the quasi-Newton method for fast convergence. They demonstrated that the restricted RVD-based method is flexible for computing the

CVT with a non-constant density function. However, their method is embedding space dependent and may fail on models with complicated geometry/topology. Recently, Lévy and Bonneel [78] proposed an elegant method for constructing curvature-adaptive anisotropic meshes. Their idea is to transform the 3D anisotropic space into a higher dimensional isotropic space, in which the mesh is optimized by computing a CVT. Lévy and Bonneel's method overcomes the d-factorial cost of computing a Voronoi diagram of dimension d by directly computing the restricted Voronoi cells with an algorithm called Voronoi Parallel Linear Enumeration, which can be easily parallelized. Their method is extrinsic due to the computation of intersection between the (higher dimensional) Voronoi cells and the surface.

2.3.3 Discrete Geodesics & Exponential Map

For any two points p and q on a 2-manifold surface, the geodesic path between p and q is a local shortest path connecting p and q on the surface [36]. If the surface is smooth, the geodesic is a curve on the surface whose geodesic curvature is always zero. Since geodesic curvature is only dependent on the first fundamental form, the geodesic is intrinsic to the surface. To compute the discrete geodesic on a triangle mesh of arbitrary topological type, two broad classes of methods exist. The first class treats the mesh as the first-order approximation of a smooth surface and uses numerical methods to solve a characterizing partial differential equation on the mesh; as a typical example, the Eikonal equation is solved on a mesh in [72]. The second class treats the mesh as a polygonal domain and uses computational geometry methods to build the shortest paths. The second class can compute the discrete geodesic exactly, and is typified by the Mitchell-Mount-Papadimitriou (MMP) algorithm [97], the Chen-Han (CH) algorithm [21] and their many variants[133][87][85][148][157]. The time complexities of the MMP and CH algorithms are $O(n^2 \log n)$ and $O(n^2)$, respectively, where n is the number

of mesh vertices. Recently, Ying et al. [156] proposed the saddle vertex graph (SVG), which is a pre-computation technique for efficiently computing various types of discrete geodesics.

With the above-mentioned discrete geodesic algorithms, one can compute the exponential map, which defines a geodesic polar coordinate system on meshes (see Figure 2.2). Let $p \in M$ be an arbitrary point on a smooth surface M. The exponential map $\exp_p : T_pM \to M$ at p is a map from the tangent plane at p to p's local neighborhood. Given a radial line on the tangent plane which originates at p and has direction $\mathbf{v} \in T_pM$, the exponential map sends $p+t\mathbf{v}$ to a unique geodesic curve γ originating at p such that $\gamma'(0) = \mathbf{v}$ and $\|\gamma(t)\| = 1$. Conversely, given an arc-length parameterized geodesic γ originating at p, $\gamma(0) = p$, there is a unique tangent direction $\gamma'(0)$ on the tangent plane $\exp_p^{-1}(\gamma) = \gamma'(0)$. The exponential map has been used in various graphics applications, such as decal compositing [116], texture mapping [131], Poisson disk sampling [158], etc.

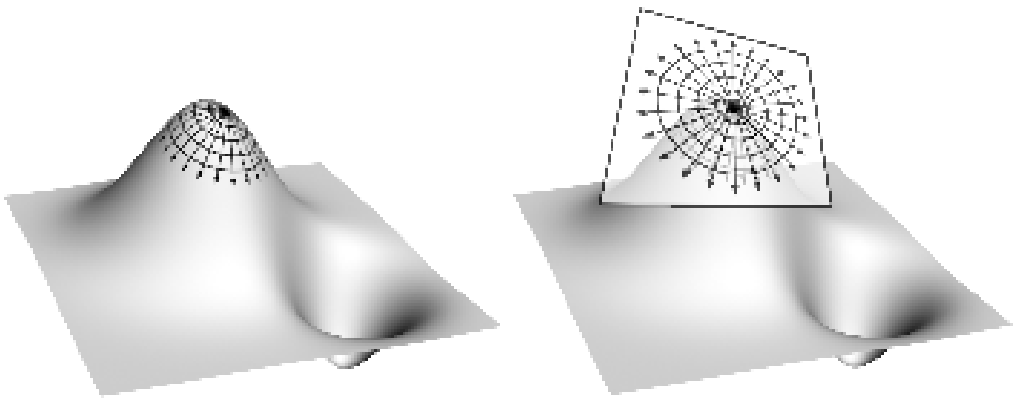


Figure 2.2: Exponential map naturally defines a geodesic polar coordinate system on curved surfaces.

2.4 Discrete Geodesics

The discrete geodesic problem has four variations, namely, single-source single-destination (SSSD), single-source all-destination (SSAD), multi-source all-destination (MSAD), and all-pairs (AP). Among them, solving the SSAD geodesic distances and paths is of particular interest, as it is the foundation of the other three. Sections 2.1 and 2.2 reviews the discrete wavefront propagation methods and the PDE methods, which are two major techniques for solving the SSAD problem. Section 2.3 reviews the $(1+\varepsilon)$-approximation algorithms, which can guarantee the accuracy of the computed geodesic distances. Sections 2.4 discusses other types of discrete geodesics and the methods for computing geodesics on discrete domains other than meshes.

2.4.1 Discrete Wavefront Propagation Methods

Two representative methods are the MMP algorithm [96] and the CH algorithm [22], which simulate the wavefront propagation in a discrete manner. They partition mesh edges into intervals, called windows, where each window encodes the geodesic paths that are in the same face sequence. Then they propagate windows across mesh faces and update the geodesic distance of vertices when they are swept by some window. The MMP algorithm maintains a priority queue for windows and propagates the window closest to the source at a time, whereas the CH algorithm organizes windows in a tree and processes them in a breadth-first search order. Given an n-face mesh, the MMP algorithm runs in $O(n^2 \log n)$ time and the CH algorithm has an $O(n^2)$ time complexity. Both algorithms have many variants, such as performance improvement [134],[149],[84],[88],[151], parallelization [157], and handling degeneracy [88]. It is worth noting that there are at most $O(n^2)$ windows on an n-face mesh [96], so the window propagation algorithms have a worst-case $O(n^2)$ time complexity on general

polyhedral surfaces, which is known as the theoretical quadratic time barrier. In practice, as observed in [134], the MMP algorithm and the improved CH algorithm [149] run in $O(n^{1.5}\log n)$ time on real-world meshes.

Schreiber and Sharir [119] proposed an elegant algorithm for computing geodesic distances on a convex polytope in $O(n\log(n))$ time, reaching the theoretical lower bound. Later, Schreiber [117] generalized the optimal-time algorithm of convex polytope to three realistic scenarios, including terrain, uncrowded polyhedron and self-conforming model, which can be concave. These optimal-time algorithms have significant theoretical values, however, they have limited applications in computer graphics, since most real-world models are nonconvex.

Kapoor [68] proposed an algorithm for computing shortest paths on any polyhedral surfaces in an $O(n\log^2 n)$ time complexity. However, due to lack of technical details, it is not possible to verify the claim. To date, the problem of computing *exact* geodesic distances on general polyhedral surfaces in subquadratic time still remains open.

2.4.2 PDE Methods

The PDE methods solve the Eikonal equation $|\nabla u(x)| = 1$ with boundary condition $u(s) = 0$ on discrete domains, where s is the source. In contrast to the exact methods which are computationally expensive, the PDE methods are very easy to implement and efficient.

Using an upwind finite difference approximation to the gradient and a Dijkstra-like sweep, Sethian [121] proposed the fast marching method (FMM), which provides a solution to the Eikonal equation in time $O(n\log n)$ on a regular grid. A similar algorithm based on a different discretization of the Eikonal equation was developed independently by Tsitsiklis [137]. Later, the FMM was generalized to arbitrary triangulated sur-

faces [71], unstructured meshes [122], implicit surfaces [94], parametric surfaces [129] and even broken meshes [17]. Since the FMM uses a strict updating order and a priority queue to manage the narrow band containing the wavefront, it is non-trivial to parallelize the FMM. Using raster scan on a completely regular structure, Weber et al. [144] developed a parallel FMM on geometry images, which runs in $O(n)$ time. Other parallel methods for solving the Eiknoal equation include [159], [52], and [33].

The heat method [28] is based on a completely different strategy. It places heat at the source vertex and then diffuses the heat in a very short time. Since the normalized gradient of the heat function coincides with the gradient of the geodesic distance function, the heat method computes the geodesic distance by solving a Poisson equation. Using Cholesky factorization of the Laplacian matrix, both the heat flow and the Poisson equation can be solved at linear time.

Recently, Belyaev and Fayolle [12] introduced a variational approach for computing the distance to a surface (a curve in 2D) and proposed efficient iterative schemes for minimizing the energy functionals. They also investigated several PDE-based distance function approximation schemes, including Poisson distance, p-Laplacian distance and L_p-distance.

The PDE methods work for a wide range of discrete domains, including regular grids, point clouds, and unstructured triangular and tetrahedral meshes. However, they provide only the first-order approximation, which is highly sensitive to mesh tessellation.

2.4.3 Other Methods

Kanai and Suzuki [67] proposed an approximate algorithm for computing geodesic paths on polyhedral surfaces. The algorithm constructs a graph using the original vertices and Steiner points added on the edges. It computes an initial shortest path between the two

given points. Then it iteratively improves the graph by adding more Steiner points to the region where the path passes through. The algorithm computes a geodesic path in $O(n \log n)$ time. Since the algorithm determines the location of Steiner points using heuristics, there is no theoretical guarantee of the approximation error. Moreover, it is not practical to extend the algorithm for the single-source geodesic distances.

Xin et al. [147] presented an algorithm for iteratively evolving an initial closed path on a given mesh into an exact geodesic loop. The algorithm has an empirical $O(mk)$ time complexity, where m is the number of vertices in the region bounded by the initial loop and the final geodesic loop, and k is the average number of edges in the edge sequences that the evolving loop passes through. Their algorithm guarantees to terminate within finite steps.

2.5 Sampling and Shape Distribution

The widely used sampling techniques include blue noise [26], point repulsion [138], centroid Voronoi tessellation (CVT) [39], stratified sampling [99], spectral sampling [104], and polyominoes [103], just name a few.

Among them, Poisson disk sampling is popular due to its nice spatial and spectral properties. Dart throwing [26] is a popular technique to generate isotropic blue noise samples due to its simplicity. However, it is time consuming since a large number of trials are discarded due to conflicts. Many techniques have been proposed to improve the performance of dart throwing, including effective spatial data structures [95, 40, 53, 43], kernel density model [45], and tile-based approaches [24, 77, 75, 103]. Wei [145] presented the phase group algorithm which subdivides the sample domain into grid cells and drawing samples concurrently from multiple cells that are sufficiently far apart to avoid conflicts. Li et al. [80] generalized the isotropic blue noise sampling to the anisotropic setting. By

using parameterization, their technique can generate high quality anisotropic sampling on surfaces. Bowers et al. [15] extended the phase group algorithm for Poisson disk sampling on 3D surfaces. Wei [146] introduced multi-class blue noise sampling where each individual class and their union exhibit blue noise properties. Chen et al. [20] presented bilateral blue noise sampling for handling problems with non-spatial features.

Bisides point based sampling, distributing 2D objects has a wide range of graphic applications. For example, line segment distributions are used in rendering applications, such as motion blur, defocus blur and scattering media [132], and rectangle distributions are used to simulate decorative mosaics [62, 11]. Feng [46] generated non-overlapping ellipses with blue noise characteristic. Sun et al. [132] proposed a frequency analysis of line segment sampling, which can be generalized to arbitrary non-point shapes.

In contrast to the 2D counterpart, shape distribution on 3D surfaces remains largely unexplored. Bowers et al. [15] computed Poisson disk distributions on 3D surfaces by space partition. With a global parameterization, Li et al. [80] presented an algorithm to distribute ellipse samples that exhibit anisotropic blue noise properties on 3D surfaces. They also evaluated the spectrum of anisotropic blue noise by warping and sphere sampling.

Chapter 3

Parallel Computing 2D Voronoi Diagrams Using Untransformed Sweepcircles

3.1 Overview

This section presents a new algorithm, called *untransformed sweepcircle*, for constructing Voronoi diagram in $\mathbb{R}^2$. Starting with a degenerate circle (of zero radius) centered at an arbitrary location, as the name suggests, our algorithm sweeps the circle by increasing its radius across the plane. At any time during the sweeping process, each site inside the sweep circle defines an ellipse composing of points equidistant from that point and from the sweep circle. The union of all ellipses forms the beach curve, a star shape inside the sweep circle which divides the portion of the plane within which the Voronoi diagram can be completely determined, regardless of what other points might be outside of the sweep circle. As the sweep circle progresses, the intersection of expanding ellipses defines the Voronoi edges. We show that the sweep line algorithm is the degenerate form of the proposed sweep circle algorithm when the circle center is at infinity, and our algorithm has the same time and space complexity as the sweep line algorithm.

Our untransformed sweepcircle algorithm is flexible in allowing multiple circles at arbitrary locations to sweep the domain simultaneously. The parallelized implementation is pretty easy without complicated numerical computation; the most complicated case is nothing but an arc-cosine operation. Furthermore, our algorithm supports the additively weighted Voronoi diagrams of which the Voronoi edges are hyperbolic and straight line segments. We demonstrate the efficacy of our parallel sweep circle algorithm using GPU.

The rest of the chapter is structured as follows.Section 3.2 briefly reviews some math prelimitaries. Then we introduce the key concepts and properties of the untransformed sweepcircle algorithm in Section 3.3 and then present the parallel implementation on modern GPUs in Section 3.4. We provide the detailed experimental results in Section 3.5 and discuss the fundamental differences between the untransformed sweepcircle algorithm and Dehne & Klein's sweepcircle method [32] in Section 3.6. Finally in Section **??**, we conclude the chapter and suggest several future research directions.

3.2 Preliminaries

This section briefly reviews the theoretical background of Voronoi diagram and Fortune's sweep line algorithm.

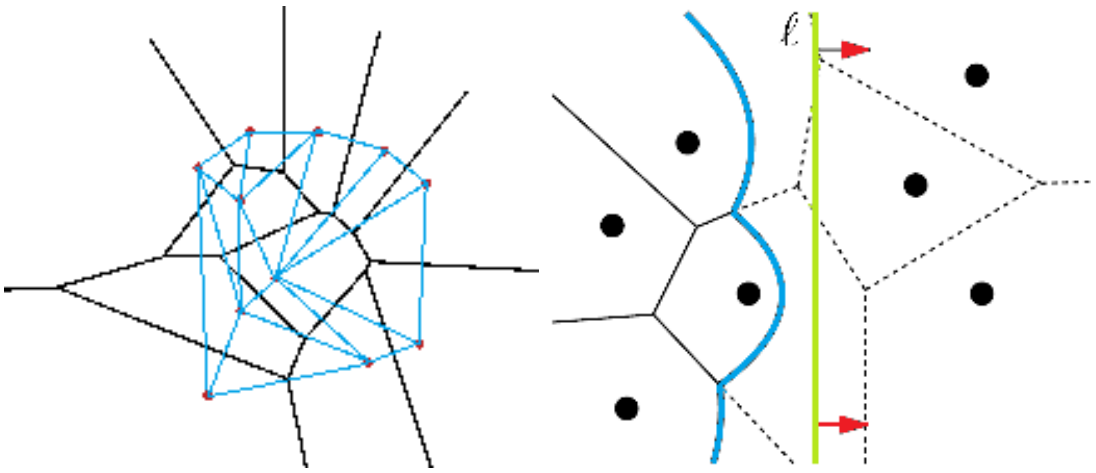
Let $S = \{s_1, s_2, \cdots, s_n\}$ be a set of points (called *sites*) in $\mathbb{R}^d$. The Voronoi cell $Vor(s_i)$ of a site s_i is

$$Vor(s_i) = \{x \in \mathbb{R}^d | \|x - s_i\| \leq \|x - s_j\| \ \forall j, j \neq i\},$$

where $\|p - q\|$ denotes the Euclidean distance between points p and q. The Voronoi cell $Vor(s_i)$ is the set of all the points x that are at least as close to s_i as to any other site s_j in S. All the Voronoi cells form a partition of the given space. It is well known that each Voronoi cell is convex and contains exactly one site. A Voronoi cell is unbounded if and

only if the corresponding site lies on the convex hull of S. The number of Voronoi edges and vertices is $O(n)$.

The Delaunay triangulation is the dual structure of the Voronoi diagram. By dual, we mean two Voronoi sites are connected by a line segment if they share a Voronoi edge. Figure 3.1(a) shows an example of 2D Voronoi diagram and its dual Delaunay triangulation.



(a) (b)

Figure 3.1: (a) An example for the Voronoi diagram (black) and its dual Delaunay triangulation (blue). (b) Fortune's sweep line algorithm [49] maintains a sweep line (green) and a beach line (blue) moving from left to right. The region left to the beach line contains the partial Voronoi diagram with a correctly computed topological structure (the solid black lines). The right region corresponds to the to-be-determined part of the Voronoi diagram.

Fortune's sweep line algorithm [49] maintains a sweep line and a beach line as the algorithm progresses. Without loss of generality, assume the sweep line is a vertical straight line moving from left to right across the plane. Each site left of the sweep line defines a parabola of points equidistant from that point and from the sweep line. The beach line is the boundary of the union of these parabolas, which splits the plane into two regions: the left region contains a partial Voronoi diagram with a correctly computed topological structure, and the right region corresponds to to-be-determined part of the

Voronoi diagram. As the sweep line progresses, the vertices of the beach line, at which two parabolas cross, trace out the edges of the Voronoi diagram, see Figure 3.1(b). Some important properties of the sweep line algorithm are as follows:

- The beach line intersects each horizontal line at exactly one point.
- When a parabolic segment disappears, a Voronoi vertex is generated.
- For the region on the left hand side of the beach line, the Voronoi diagram has been determined, while the region on the right corresponds to the to-be-determined Voronoi diagram.
- The sites on the left of the beach line can be classified into two groups: *fixed* sites whose Voronoi cell has been totally determined and *active* sites whose Voronoi cell is under computation.
- Suppose x_i is an active site, then the parabolic segment determined by x_i is totally inside x_i's Voronoi cell.

In the sweep line algorithm, a balanced binary search tree is used to maintain the combinatorial structure of the beach line, and a priority queue listing potential pending events that possibly change the beach line structure. It can be shown that there are $O(n)$ events to process and $O(\log n)$ time required to process each event, and hence the total time is $O(n \log n)$.

Although the sweep line algorithm is conceptually elegant and widely adopted, it is non-trivial to parallelize and implement it on modern GPUs. As shown in Figure 3.2, to compute the Voronoi diagram for the square domain in parallel, we partition it to a set of square-shaped blocks with side length τ, and then run the sweep line algorithm for each block independently. Without loss of generality, we assume that at least one

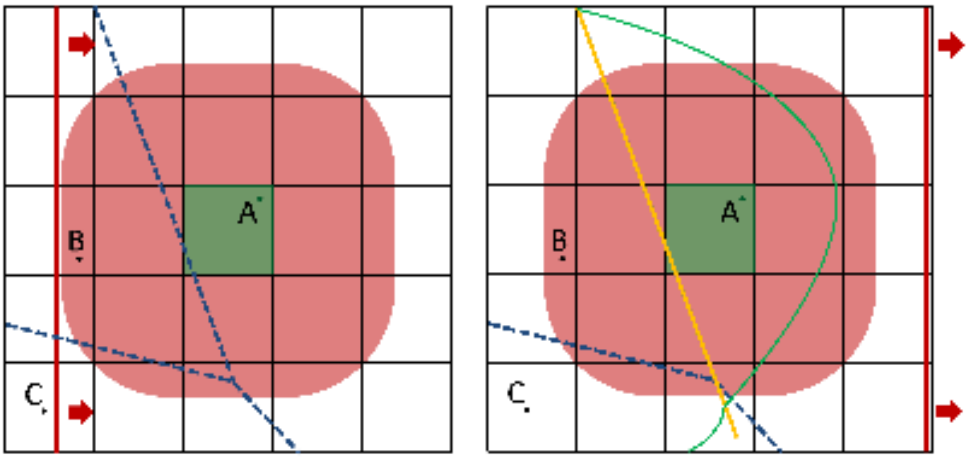
site is contained in the green block. It can be shown that only the sites in the red round-cornered rectangle, $(4\sqrt{2}+2\pi+1)\tau^2$ in size, affect the Voronoi diagram of the green block. Therefore, we need to place a sweep line at the left boundary of the red round-cornered rectangle (see Figure 3.2(a)) and consider the sites located inside this area in the subsequent sweeping process. In this example, as sites A and B are inside the red region, the sweep line algorithm results in a Voronoi edge bisecting A and B (see the yellow line segment in Figure 3.2(b)). However, the correct Voronoi diagram (the dotted blue lines in Figure 3.2(a)) consists of only part of the bisecting line of A and B due to the existence of site C, of which is outside of the red region. In fact, it can be shown that only the Voronoi diagram inside the green region is correct and contributes to the final result. Therefore, we have to trim the computed Voronoi diagram (of the entire red region) with the centered green region to get the final result. In other words, each sweep line algorithm computes the Voronoi diagram for a region of size $(4\sqrt{2}+2\pi+1)\tau^2$, but only the result inside the centered region of size τ^2 is used. Roughly speaking, $1-\frac{1}{(4\sqrt{2}+2\pi+1)} = 92.3\%$ computation is totally wasted. Due to these drawbacks, it is difficult to parallelize the sweep line algorithm in an efficient and effective manner. To our knowledge, there is no such algorithm known to the community.

The proposed untransformed sweepcircle algorithm overcomes the drawbacks of the sweep line algorithm, and can be naturally extended to parallel setting.

3.3 Untransformed Sweepcircle Algorithm

3.3.1 Algorithm Description

In contrast to the sweep line algorithm which propagates the sweep/beach line from left to right, our algorithm starts with a degenerate circle at an arbitrary location and then sweeps the circle by increasing its radius to cover the entire domain. Our untransformed sweepcircle algorithm contains four fundamental elements:



(a) (b)

Figure 3.2: (a) To compute the Voronoi diagram for the green region, one has to apply the sweep line algorithm to a much larger domain (in red). (b) At the conclusion of the sweep line algorithm, one has to trim the computed Voronoi diagram with the centered region. The doted blue lines are the real Voronoi edges.

- The **sweep circle**, centered at the user-specified or a random location c, sweeps the plane by increasing its radius. The initial radius is usually set to zero.
- The **beach curve** is a set of consecutive elliptic segments, each of which bisects a site and the sweep circle. The beach curve splits the plane into two regions: the inside region contains a partial Voronoi diagram with a correctly computed topological structure, and the outside region corresponds to an undetermined part of the Voronoi diagram that will be computed later.
- A **touching event** occurs when the sweep circle touches a new site. As a result, a new elliptic segment is inserted into the current beach curve.
- A **vanishing event** occurs when an elliptic segment disappears. Consequently, a new Voronoi vertex is generated.

Figure 3.3 illustrates the basic idea of sweep circle for a two-site example. Initially, we place a degenerate circle (i.e., a point) at the user-specified or at random location

c. Assume the circle center does not coincide with any sites. The circle grows until it touches the closest site, say s_1, which triggers the first touching event (see (b)). As a result, we construct a degenerate ellipse with focus points s_1 and c, satisfying $\|x-s_1\|+\|x-c\|=R$, where $R=\|c-s_1\|$ is the radius of the current sweep circle. Clearly, the initial beach curve is a line segment (i.e., degenerate ellipse). Throughout the sweeping process, the ellipse always satisfies the constraint $\|x-s_1\|+\|x-c\|=R$ for the increasing radius R. It can be shown that the ellipse is the bisector of s_1 and the sweep circle $\odot(c,R)$. Furthermore, the beach curve is always inside the sweep circle (see (c)). The next event (again, a touching event) occurs when the sweep circle reaches the second site s_2, which creates an ellipse $\|x-s_2\|+\|x-c\|=R$ (now $R=\|c-s_2\|$. This newly generated ellipse is inserted into the current beach curve. Thus, the updated beach curve contains two elliptic segments, each of which is the bisector of the sweep circle and the corresponding site (see (d)). As the sweep circle progresses, the vertices of the beach curve, at which two ellipses cross, trace out the bisector (in red) of s_1 and s_2 (see (e)). Note that the *continuous* animation is used to illustrate the basic concept of sweep circle. In fact, our algorithm sweeps the circle and processes the events in a *discrete* manner.

Figure 3.4 shows the circle sweeping on a domain with five sites. The key idea of our untransformed sweepcircle algorithm is to maintain the (circular) ordered list of elliptic segments and trace out the vertices of the beach curve. The two neighboring (in circular order) elliptic segments correspond to an *active* Voronoi edge and the two neighboring active edges (in circular order) are joined by an elliptic segment. When an elliptic segment vanishes (i.e., a vanishing event occurs), the two neighboring active edges meet at a point, which defines a Voronoi vertex (see the insets in Figure 3.4 (d),(f),(g)). When the sweep circle touches a new site (i.e., a touching event occurs), a new Voronoi edge is generated. The discrete events are maintained in a priority queue sorted by the distances

of the corresponding sites to the center c (from near to far). Throughout the algorithm, the Voronoi diagram inside the beach curve has been determined. The algorithm stops when the priority queue is empty and at this moment the complete Voronoi diagram is known. The essential data structures in the untransformed sweepcircle algorithm include the Voronoi diagram, the beach curve and the event manager, shown as follows:

```
//2D point, vector or interval
typedef pair<double,double> double2;

//Lists of sites and Voronoi vertices
vector<double2> SiteList;
vector<double2> VoronoiVertexList;

struct VoronoiEdge {
    //two endpoints of the edge
    int vert1, vert2;
    //two sites contributing to the bisector
    int site1, site2;
};
vector<VoronoiEdge> VoronoiEdgeList;

//An active edge, shared by two consecutive elliptic
//segments, contains one fixed endpoint and an
//extending direction
struct ActiveEdge {
    int indexFixedEndpoint;
    double2 extendingDirection;
    EllipticSegment* leftSegment;
    EllipticSegment* rightSegment;
```

```
};
struct EllipticSegment {
    //One focus point is the center of the sweep circle
    double2 circleCenter;
    //The other focus point is a site
    int indexActiveSite;
    //Sandwiched by two neighboring active edges.
    ActiveEdge* leftActiveEdge;
    ActiveEdge* rightActiveEdge;
};
//The beach curve contains the elliptic segments in
//circular order
BalancedBinaryTree<EllipticSegment> beachCurve;

//Discrete events are handled from near to far
enum EventType {TOUCH, VANISH};
struct Event {
    EventType type;
    double whenOccur;
    int indexSiteToTouch;
    EllipticSegment* segmentToDisappear;
}
priority_queue<EventType> EventQueue;
```

Since two neighboring elliptic segments trace out an active Voronoi edge and two neighboring active Voronoi edges sandwich an elliptic segment, we use pointers to maintain the relationship between ActiveEdge and EllipticSegment. We also use a balanced binary tree to dynamically maintain the beach curve that contains the elliptic segments

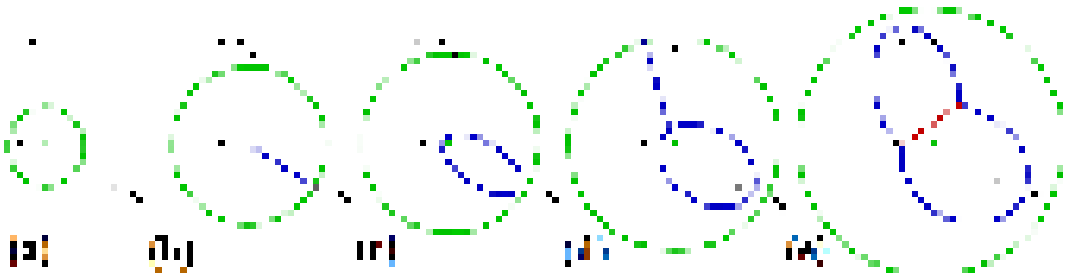


Figure 3.3: Illustration of sweep circles for two sites. The beach curve consists of elliptic segments, each of which corresponds to a site. The elliptic segment is the bisector of the corresponding site and the sweep circle. The vertices of the beach curve trace out the bisector between two sites. The sweep circle, the beach curve and the bisector are drawn in green, blue and red, respectively.

in a circular order. The complete structure of the Voronoi diagram, such as the incident edges of a Voronoi vertex and the bounding edges of a Voronoi cell, can be induced from the above data structures. The pseudo code of our untransformed sweepcircle algorithm is shown in Algorithm 1.

3.3.2 Correctness

This subsection proves the correctness of our untransformed sweepcircle algorithm. Consider a set of 2D sites $S = \{s_1, s_2, \cdots, s_n\}$. Let us denote $\odot(c,R)$ by the sweep circle centered at c with radius R.

Lemma 3.1 *Each elliptic segment of the beach curve bisects the sweep circle $\odot(c,R)$ and the corresponding site s_i. When the center c coincides with a site s_i, the elliptic segment degenerates into a circular arc.*

Proof: Let e_i be the ellipse corresponding to site s_i, i.e., $e_i = \{x | \|x-c\| + \|x-s_i\| = R\}$. For any point $x \in e_i$, the shortest distance between x and the sweep circle is $d\big(x, \odot(c,R)\big) = R - \|x-c\| = \|x-s_i\|$. Thus, the ellipse is the bisector of the sweep circle and s_i. When $c = s_i$, we obtain $\|x-c\| = \frac{R}{2}$, which is a circle.

Algorithm 1 Untransformed sweepcircle algorithm for computing 2D Voronoi diagram

Input: A set of 2D sites $S = \{s_1, \cdots, s_n\}$ and the center of the sweep circle c;
Output: The Voronoi diagram of S
//$\mathscr{T}$: a binary tree to maintain the beach curve;
//$\mathscr{V}$: the Voronoi vertex and edge structure;
//$\mathscr{Q}$: the priority queue of discrete events
Push n touching events into $\mathscr{Q}$ sorted by $\|c - s_i\|$, $1 \leq i \leq n$.
while $\mathscr{Q}.isEmpty() == false$ **do**
    Pop up the top event q from $\mathscr{Q}$;
    if q is a touching event **then**
        Insert a new ellipse E into $\mathscr{T}$;
        //Suppose E splits the existing elliptic segment
        //E' into E'_1 and E'_2, and let p be the intersection
        //point.
        Build two radial edges rooted at p, one shared by E'_1 and E, the other shared by E and E'_2;
        Compute the time when E, E'_1, and E'_2 disappear;
        Push the vanishing event into $\mathscr{Q}$;
    else if q is not out of date **then**
        //q is a vanishing event.
        //Let E denote the vanishing
        //elliptic segment, E_1 and E_2 the two neighboring
        //elliptic segments. If one of them is out of data,
        //so is the event. see Figure 3.6.
        Create a Voronoi vertex v;
        Terminate the two active edges, joined by E, at v;
        Update the Voronoi structure $\mathscr{V}$;
        Create an active Voronoi edge e;
        Push a vanishing event according to e and its neighboring active edges into the queue $\mathscr{Q}$;
    end if
end while

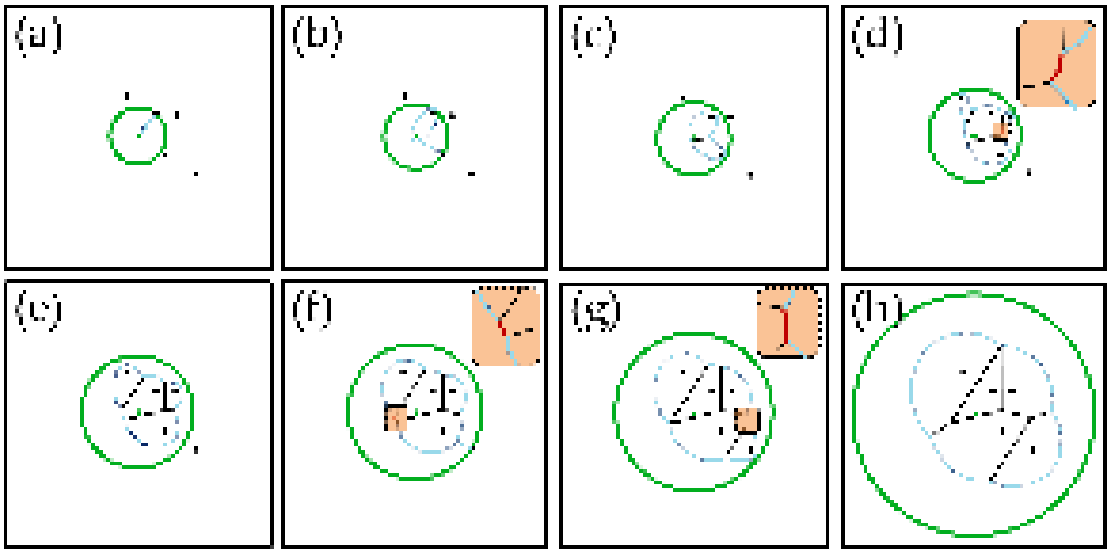


Figure 3.4: Applying the sweep circle algorithm to a domain with five sites. (a) The sweep circle touches the first site. (b) The sweep circle touches the second site. (c) The vertices of the beach curve, at which two ellipses cross, trace out the bisector of the two corresponding sites. (d) An elliptic segment (in red) is vanishing, as a result, the neighboring two segments will eventually meet. The intersection point is a Voronoi vertex. (e) The circle is expanding toward the fifth site. (f) The sweep circle touches the fifth site. (g) The beach curve follows the growing sweep circle and the vertices of the beach curve trace out the Voronoi edges. (h) The algorithm stops when the event queue is empty. At this moment, the Voronoi diagram structure has been fully determined. The beach curve and the Voronoi edges are drawn in blue and black, respectively. The vanishing elliptic segments are highlighted in red (see the insets).

Lemma 3.2 *For any two sites satisfying $\|c-s_i\| < R$ and $\|c-s_j\| < R$, the ellipses*

$$e_i = \{x | \|x-c\| + \|x-s_i\| = R\}$$

and

$$e_j = \{x | \|x-c\| + \|x-s_j\| = R\}$$

intersect at exactly two points, which are on the bisector of s_i and s_j.

Proof: The number of intersection points of two ellipses is from 0 to 4. First, the number of intersection points of e_i and e_j is at least 2 since:

- The two ellipses share the focus point c, which is completely inside the ellipses e_i and e_j.
- Let $p_i \in e_i$ (resp. $p_j \in e_j$) be the closest point on e_i (resp. e_j) to the sweep circle $\odot(c,R)$. Then p_i is outside of e_j and p_j is outside of e_i (see the above figure).

On the other hand, the intersection points between e_i and e_j satisfy $\|x-s_i\| = \|x-s_j\|$ that determines a straight line L. Any intersection point in $e_i \bigcap e_j$ must be a subset of $L \bigcap e_i$ (or $L \bigcap e_j$). This implies that the number of intersections cannot exceed 2. Therefore, the two ellipses e_i and e_j meet at exactly two points, which determine the bisector of s_i and s_j.

Lemma 3.3 *The beach curve is a closed star shape with center c, and is always inside the sweep circle $\odot(c,R)$.*

Proof: Let e_i be the ellipse corresponding to the site s_i, i.e., $\|x-c\| + \|x-x_i\| = R$. Clearly, the ellipse e_i is inside the sweep circle $\odot(c,R)$, since $\|x-c\| \leq R$. Let $\Omega = \bigcup_{s_i \in \odot(c,R)} \{x \mid \|x-c\| + \|x-s_i\| \leq R\}$ denote the union of the ellipses which are inside

the sweep circle. The beach curve is the boundary of Ω. Due to the fact $\partial \circ \partial = 0$, the beach curve is closed.

Then we show $\partial\Omega$ is star shaped. Observe that each ellipse $\|x - s_i\| + \|x - c\| = R$ is a star shape with respect to the center c. Assume the beach curve is not a star shape w.r.t c, there must exist two points $p_i \in e_i, p_j \in e_j$ on the beach curve such that p_i, p_j, c are collinear and p_j is between p_i and c. Therefore, p_j is totally inside e_i since e_i itself forms a star shape w.r.t. c. This contradicts to the assumption that $p_j \in e_j$ is on the beach curve, which is the boundary of $\bigcup e_i$.

Lemma 3.4 *The shortest distance between the center c and the beach curve is at least* $\frac{R-d}{2}$*, where d is the distance between c and the nearest site.*

Proof: Suppose the s^* is the nearest site to the center c. Then the ellipse $\|x - s^*\| + \|x - c\| = R$ must be completely inside the beach curve. Observe that the shortest distance between the ellipse and the center c is $\frac{R-d}{2}$. So the shortest distance between the beach curve and c is at least $\frac{R-d}{2}$.

Lemma 3.5 *The elliptic segment corresponding to site s_i is inside the Voronoi cell of s_i.*

Proof: We distinguish the input site set S into the *outside* group $S_O = \{s_j | s_j \notin \odot(c,R)\}$ and the *inside* group $S_I = \{s_j | s_j \in \odot(c,R)\}$.

We first examine the inside group S_I. Let $s_i \in S_I$ be a site inside the sweep circle and e_i be the corresponding ellipse $\|x - s_i\| + \|x - c\| = R$. For any point $x \in e_i$, we have $d(x,s_i) < d(x,s_j)$, $\forall s_j \in S_O$, since the ellipse e_i is the bisector of the sweep circle and the site s_i (by Lemma 3.1).

Then, we prove that $\forall x \in e_i$, $d(x,s_i) \leq d(x,s_j)$ for any $s_j \in S_I$ by contradiction. Assume there is a point $\tilde{x} \in e_i$ and a site $s_j \in S_I$ such that $d(\tilde{x},s_i) > d(\tilde{x},s_j)$. Let e_j be the ellipse

of site s_j, i.e., $e_j = \{x | \|x - s_j\| + \|x - c\| = R\}$. Then the point $\tilde{x}$ must be completely inside the ellipse e_j since $\|\tilde{x} - s_j\| + \|\tilde{x} - c\| < R$. Observing that the beach curve is exactly the boundary of the union of ellipses inside the sweep circle, $\tilde{x}$ is not on the beach curve, which leads to a contradiction!

Putting it together, the elliptic segment e_i is closer to s_i than any other sites in S. Thus, e_i is inside the Voronoi cell of s_i.

Lemma 3.6 *The intersection of consecutive elliptic segments of the beach curve lies on the Voronoi edges.*

Proof: Suppose x is the intersection of two consecutive elliptic segments e_1 and e_2, corresponding to sites s_1 and s_2 respectively. By Lemma 3.5, e_1 (resp. e_2) is inside the Voronoi cells of s_1 (resp. s_2). Thus, the intersection of e_1 and e_2 is of equal distance to s_1 and s_2 (and closer to s_1 and s_2 than any other sites), i.e., it lies on the Voronoi edge.

Lemma 3.7 *The Voronoi diagram inside the beach curve has been completely determined.*

Proof: For every point p inside the beach curve, the site which is closest to p can be determined by the elliptic segment passing through p. Thus, the Voronoi diagram inside the beach curve can be completely determined regardless of the sites outside of the sweep circle.

Lemma 3.8 *The current beach curve contains an elliptic segment corresponding to the site s_i if and only if $s_i \in \odot(c,R)$ and its Voronoi cell is still open.*

Proof: $\Leftarrow$: Assume the current beach curve still contains one of its elliptic segments, say e. This implies that part of this Voronoi cell lies outside the current beach curve,

which cannot be determined by our algorithm (see Lemma 3.7). Thus, the Voronoi cell is not yet complete, which contradicts to the assumption that the Voronoi cell of s_i is closed.

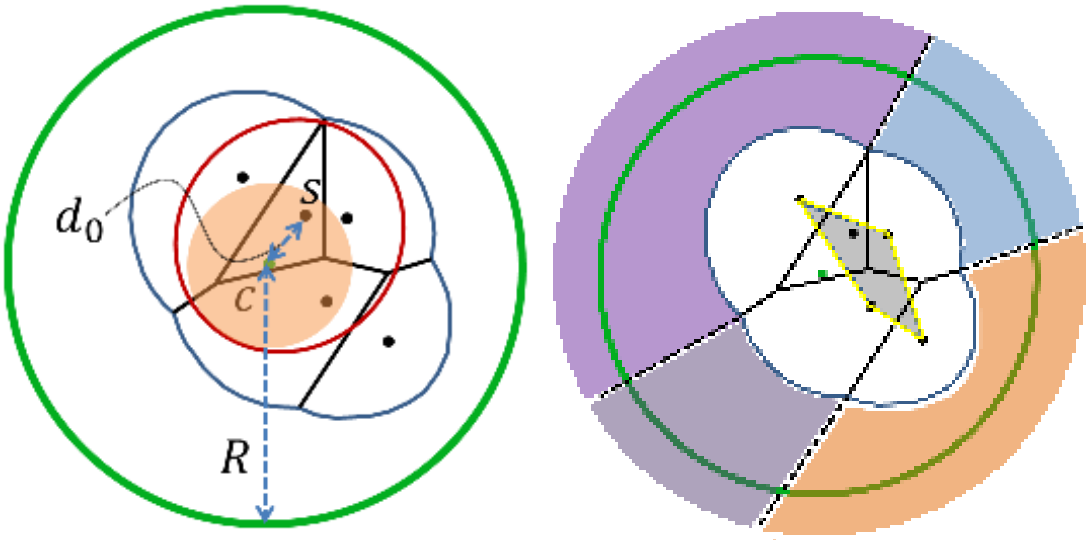
$\Rightarrow$: If the Voronoi cell of s_i is still open, it must intersect the current beach curve at two or more points. Without loss of generality, consider two intersection points p_1 and p_2 joined by one or more elliptic segments that are inside the Voronoi cell of s_i. So every point x on this curved segment is closer to s_i than any other sites, which implies that x satisfies the equation $\|x-s_i\|+\|x-c\|=R$. Thus, the current beach curve must contain the elliptic segment of site s_i.

When the algorithm stops, there are exactly k elliptic segments remaining on the beach curve, corresponding to the k sites on the convex hull of the input sites S, see the above figure.

Lemma 3.9 *Let s be the nearest site to the sweep circle center c and $d_0 = \|c-s\|$. Given the sweep circle $\odot(c,R)$, the disk $\odot(c,\frac{R-d}{2}$*

the already-determined Voronoi diagram inside $\odot(c,\frac{R-d_0}{2})$ has been completely determined. Especially, when c is exactly one of the sites, the known Voronoi region is covered by $\odot(c,\frac{R}{2})$. When the algorithm stops, there are exactly k elliptic segments remaining on the beach curve, corresponding to the k sites on the convex hull of the input sites S.

Proof: According to Lemma 3.3, the ellipse $\|x-s\|+\|x-c\|=R$ must be totally enclosed inside the beach curve. Also, the perihelion of the ellipse (with c being the main focus) is closer to c than any other points on the ellipse. Therefore, as shown in Figure 3.5, the Voronoi diagram inside the disc $\odot(c,\frac{R-d_0}{2})$ has been completely determined. When c is exactly one of the sites, i.e., $d_0=0$, the Voronoi diagram inside $\odot(c,\frac{R}{2})$ is completely known.



(a) (b)

Figure 3.5: (a) Given the sweep circle $\odot(c,R)$ centered at c with radius R, the beach curve is at least $\frac{R-d_0}{2}$ away from the center c (by Lemma 3.4), where d_0 is the distance from the nearest site s to the sweep circle center c. By Lemma 3.7, the Voronoi diagram inside the beach curve has been completely determined, so the disk $\odot(c,\frac{R-d_0}{2})$ (the shaded region) provides a rather conservative bound of the region where the Voronoi diagram has been ready. (b) When the algorithm terminates, there are exactly k elliptic segments remaining on the beach curve, corresponding to the k sites on the convex hull of the input sites S.

Furthermore, with sufficiently large radius R, all the Voronoi vertices can be enclosed in the the disc $\odot(c,\frac{R-d_0}{2})$, which implies that all the Voronoi cells are closed and totally inside the beach curve except those sites locating on the convex hull of the site set S. By Lemma 3.8, there are k elliptic segments remaining on the beach curve, corresponding to the k sites on the convex hull.

Finally, the following theorem reveals the relationship between the sweep line algorithm and our untransformed sweepcircle algorithm.

Theorem 3.1 *The Fortune's sweep line algorithm [49] is a degenerate form of our sweepcircle algorithm if the center c is at infinite distance to the sites.*

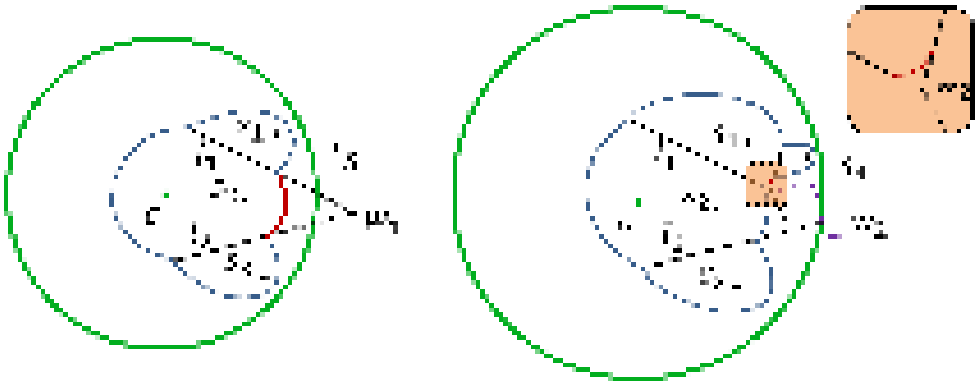


Figure 3.6: Out of date events. The red elliptic segment is sandwiched by two Voronoi edges, say l_1 and l_2, and will vanish when the sweep circle touches the point w_1. As a result, a vanishing event q has been pushed into the priority queue $\mathscr{Q}$. However, when the sweep circle touches the site s_4, the edge l_1 is intercepted by a newly-created Voronoi edge (see the inset). Therefore, the event q is out of date when it is popped from the queue $\mathscr{Q}$, which is simply ignored by the sweep circle algorithm.

Proof: The site s_i contributes to the beach curve *after* the sweep circle hits s_i and *before* the corresponding Voronoi cell becomes a closed polygon. During this period, the elliptic segment of s_i is inside this Voronoi cell (by Lemma 3.5) and it is closer to the focus point s_i than the other focus point c, the center of the sweep circle since c is assumed to be at infinite distance to the sites. The distance between s_i and its elliptic segment is very limited, and cannot exceed the size of s_i's Voronoi cell.

Under this condition, if the center c is at infinite distance to s_i, the eccentricity of the ellipse approaches 1. As a result, the sweep circle becomes a straight line and the elliptic segment becomes a parabola segment. At the same time, the beach curve still serves as the bisector of the sweep line and the active sites, exactly the same situation with that achieved in Fortune's sweep line algorithm. Therefore, we can conclude that Fortune's sweep line algorithm is the degenerate form of our untransformed sweepcircle algorithm.

3.3.3 sweepcircle-Complexity

Theorem 3.2 *The untransformed sweepcircle algorithm computes the Voronoi diagram in $O(n\log n)$ time and requires $O(n)$ space.*

Proof: First of all, sorting the sites according to the distances to the sweep center requires $O(n\log n)$ time. In the following, we come to analyze the time cost of the sweeping process.

First, we show the number of elliptic segments, as well as that of the events, is $O(n)$. There are two kinds of events, namely, the touching event and the vanishing event. The number of touching events is equal to the number of sites n. When the sweep circle touches a new site, the algorithm splits an existing elliptic segment of the current beach curve into two and then add a new elliptic segment corresponding to the new site. Therefore, at most $2n$ elliptic segments are generated, and at most $3n$ vanishing events need to be considered throughout the algorithm. Note that some vanishing events may be out of date and are discarded when the sweep circle progresses (see Figure 3.6).

Second, we show handling each event takes $O(\log n)$ time. For the touching event, a new elliptic segment is inserted into the beach curve, maintained by a balanced binary tree $\mathscr{T}$. The insertion takes $O(\log n)$ time. For the vanishing event, the priority queue $\mathscr{Q}$ is updated, which also takes $O(\log n)$ time.

Third, the Voronoi diagram, the balanced binary tree and the priority queue require $O(n)$ space to store the vertex-edge structure, the $O(n)$ elliptic segments and $O(n)$ events respectively.

Putting it altogether, the sweep circle algorithm takes $O(n\log n)$ time and requires $O(n)$ space.

3.4 Parallel Sweep Circles

To compute the Voronoi diagram in a parallel fashion, we need to partition the domain and then solve each sub-domain independently. To use the sweep line algorithm, one needs to place the initial sweep line at one side of the to-be-computed region, and then moves it in a certain direction across such region. As shown in Figure 3.2, the sweep line algorithm needs to sweep the region much larger than the desired region. Furthermore, as only the centered region is correct, one has to trim the computed Voronoi diagram to get the final result.

Compared to the classical sweep line algorithm, the proposed untransformed sweepcircle algorithm has two unique advantages, which makes it preferable for parallel implementation. First, the sweep circle can be placed anywhere in the domain. Second, the beach curve of our sweepcircle algorithm is closed and the Voronoi diagram inside the beach curve can be completely determined regardless of the sites outside the sweep circle. Thus, each sweep circle thread can work independently. Even though the sweep circles may have overlap, we do not need any postprocessing to trim the Voronoi diagram.

Given a 2D domain Ω containing sites S, we can partition the domain Ω into disjoint sub-regions, $\Omega = \bigcup_{i=1}^{m} \Omega_i$, $\Omega_i \bigcap \Omega_j = \varnothing$, $\forall i \neq j$. Then each sub-region Ω_i is assigned a sweep circle thread, which places a sweep circle at arbitrary location inside Ω_i. In practice, one can set the center c_i the barycenter of Ω_i. Each thread stops when its event queue is empty. If one sub-region contains no sites at all, we can merge that region with neighboring region such that each sub-region contains at least one site. The pseudo code of the parallel sweep circle algorithm is shown in Algorithm 2.

In fact, since each sweep circle thread cares for only the Voronoi structure inside Ω_i, we can modify the sweep circle algorithm such that it can terminate early, i.e., without

Algorithm 2 Parallel sweep circle algorithm

Input: A 2D region Ω containing n sites $S = \{s_1, \cdots, s_n\}$ and a partition of Ω into m disjoint sub-regions Ω_i, $\Omega_i \bigcap \Omega_j = \varnothing \; \forall i \neq j$.
Output: The Voronoi diagram of S
parallel for each sub-region Ω_i **do**
  Compute the barycenter c_i of Ω_i
  Run the sweep circle algorithm centered at c_i
parallel end for

processing all the vanishing events in the event queue. By Lemma 3.7, the Voronoi diagram inside the beach curve is complete. Thus, we can test whether the beach curve contains the entire sub-region Ω_i. If so, we can stop the sweep circle algorithm for Ω_i. In practical application, the sub-region Ω_i is usually modeled by a polygon. So this test can be implemented easily. Let R_s denote the radius of the sweep circle when the algorithm stops. The following theorem provides an upper bound of the stopping radius R_s:

Theorem 3.3 *Given the sub-region Ω_i, let $\odot(c_i, r_i) \supset \Omega_i$ be the smallest covering disc. The site $s \in \Omega_i$ is the nearest one to the center c_i. Define $d = \|c_i s\|$. The sweep circle algorithm for the sub-region Ω_i can stop when the radius $R_s \geq 2r_i + d$ or the event queue is empty, or whichever occurs first.*

Proof: By Lemma 3.3, the ellipse $\|x-s\| + \|x-c_i\| = R$ must be inside the beach curve, since s is the nearest site to the center c_i. By Lemma 3.4, the beach curve can contain a disc $\odot(c_i, \frac{R-d}{2})$ completely. In order to ensure that the Voronoi diagram inside Ω_i has been fully determined, we require the terminating radius R_s satisfying $\frac{R_s-d}{2} \geq r_i$, which completes the proof.

We would like to point out the above upper bound of the stopping radius $R_s = 2r_i + d$ is very conservative. Based on our experiments, we observe that the beach curve usually follows the sweep circle very closely. As a result, the stopping radius is usually slight

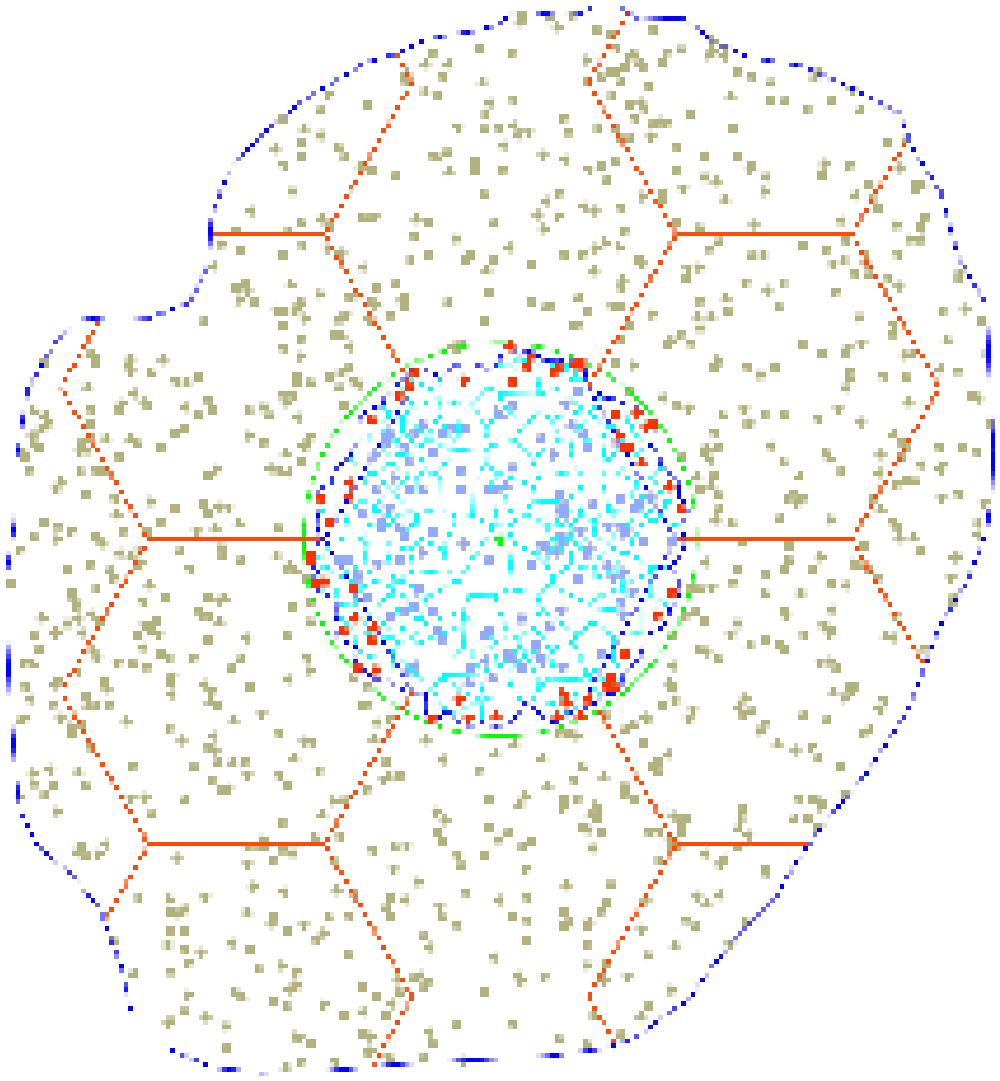


Figure 3.7: In practice, the beach curve follows the sweep circle very closely, which leads to a small stopping radius R_s. For a domain Ω with 1000 randomly distributed sites, we partition Ω into equally sized regular hexagons and compute the ratio of the number of sites inside the hexagon to the number of sites inside the sweep circle. The average ratio of 10 experiments is 71.9%, which implies that the parallel sweep circle algorithm is very effective, since only less than 28.1% computation is "wasted".

larger than the size of the sub-region r_i. In practice, we can partition the domain into equally sized regular hexagon, which is very effective to reduce R_s. The theoretical lower bound of R_s is $R_s = r_i$, which occurs for the uniformly distributed sites with such hexagonal partition.

Furthermore, as the Voronoi diagram inside the beach curve is guaranteed to be correct, the overlap among the sweep circles does not affect the final result. This feature significantly distinguishes our algorithm from the sweep line algorithm, which has to trim the computed voronoi diagrams.

3.5 Experimental Results

We implemented our untransformed sweepcircle algorithm in C++ and tested it on a PC with an Intel Xeon 2.50GHz CPU and 8GB memory. The graphics card is an NIVDIA GTX 580 with 512 cores and 1.5GB memory. Our program is compiled using CUDA4.0 RC2.

We have shown that the sweep circle has the same time and space complexity as the sweep line algorithm. To measure and compare the practical performance, we count the number of discrete events for the sweep circle algorithm and the sweep line algorithm. We randomly generate up to 1000 sites in a unit square and then run the sequential sweep circle and sweep line algorithms. Although based on different sweeping strategies, both approaches have very similar number of events, as shown in Figure 3.8. Due to the trigonometric functions used for computing the joint point between consecutive elliptic segments, its computational cost for processing each event is slightly higher than that of the sweep line algorithm. As a result, the single-core sweep circle algorithm is slower than the sweep line algorithm. However, as mentioned before, the proposed sweep circle algorithm is superior than the sweep line algorithm due to its parallel nature. With

the parallel untransformed sweepcircle algorithm, the average number of events can be significantly reduced by increasing the number of sweep circles.

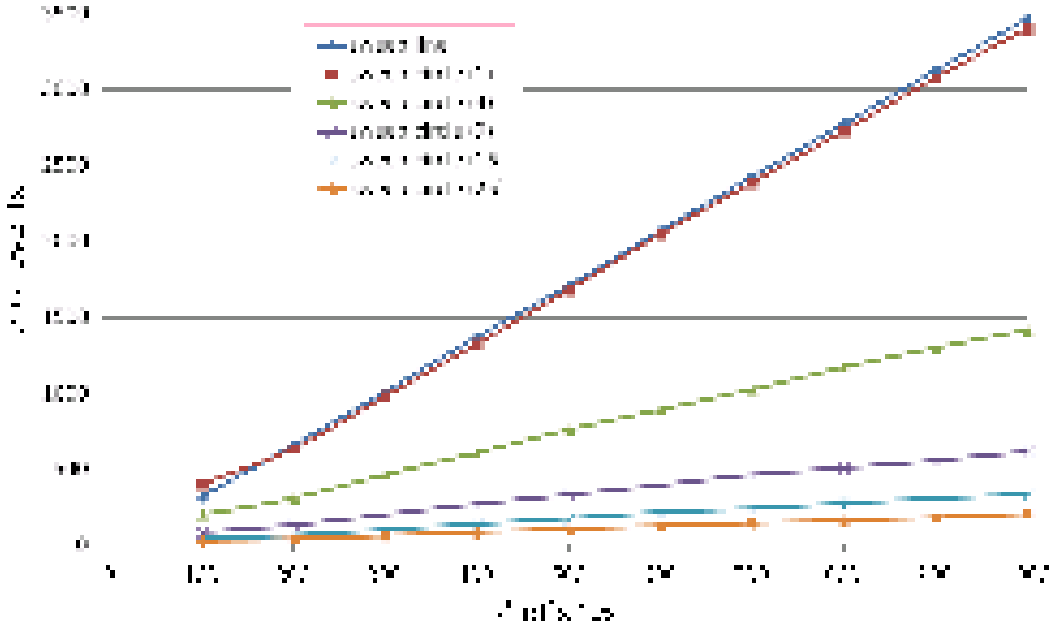


Figure 3.8: Both the sequential sweep circle and sweep line algorithms have very similar number of discrete events. With the parallel sweep circle algorithm, the average number of events for each GPU thread is significantly reduced. The horizontal axis and the vertical axis show the number of sites and the average number of discrete events respectively. The number in the bracket represents the number of sweep circles.

Figure 3.10 shows an example of 250 sites within an irregular domain, which is partitioned into m^2 $(m = 2,3,4,5)$ sub-regions. Each sub-region Ω_i is assigned a GPU based sweep circle thread, which outputs the Voronoi diagram inside the corresponding beach curve. Although the neighboring beach curves have overlap when the sweep circles progress, all the threads can run independently without any data conflicts or synchronization. Figure 3.9 demonstrates a comparison of our sweep circle algorithm with CGAL [1] and Triangle [126]. In this experiment, the number of sweep circles of our algorithm is 25. Due to the limitation of time, our implementation has not been optimized. But our sweep circle algorithm still shows satisfactory parallel speedup ratio. Considering that modern Graphics Computing Devices always have hundreds of cores, our algorithm has great potential of speedup.

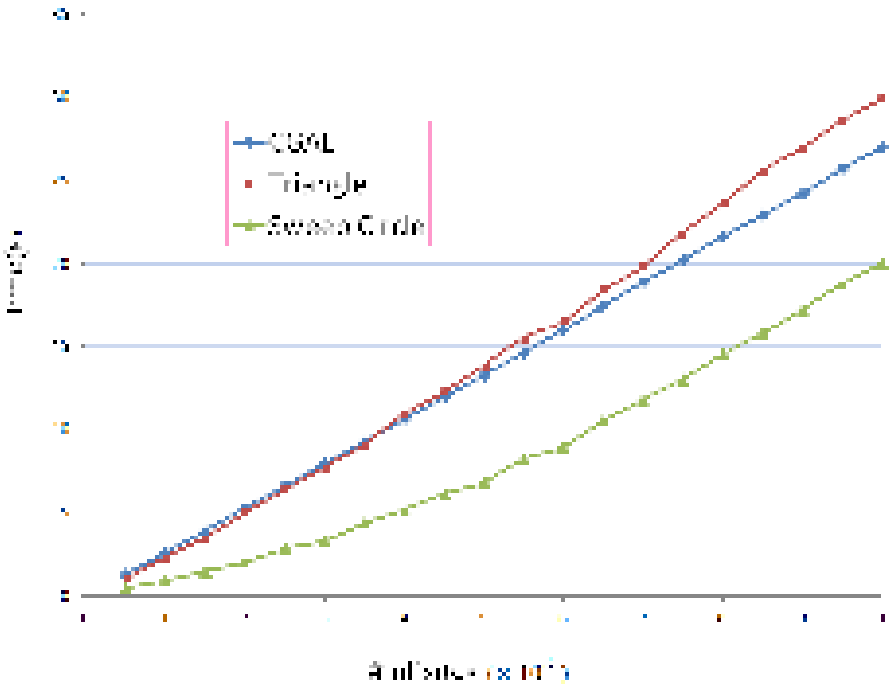


Figure 3.9: The comparisons with CGAL [1] and Triangle [125] show that our sweep circle algorithm has parallel speedup ratio of 1.63 and 1.73 respectively. The horizontal axis and the vertical axis shows the number of sites and computing time respectively.

Our algorithm can be applied to 3D surface by using parameterization. Figure 3.12 shows two genus-0 3D models which are conformally parameterized to disc and square using the harmonic map [57] and the holomorphic 1-form method [58]. We generate 200 and 600 random sites on the 3D face and sheep model. Then we run the parallel sweep circle algorithm on the 2D parametric domain using the Euclidean distance and the parameterization induces the Voronoi diagram on the 3D surface. See Figure 3.13. An interesting work is to use the geodesic distances instead of the Euclidean distance, which would result in an intrinsic geodesic Voronoi diagram on surface. As this is beyond the scope of this report, we will address it in the future work.

The proposed sweep circle framework can be easily extended to the additively weighted Voronoi diagram, for which the conventional Euclidean distance is modified by the weights assigned to each site. We represent each site s_i as a disk of radius r_i centered at

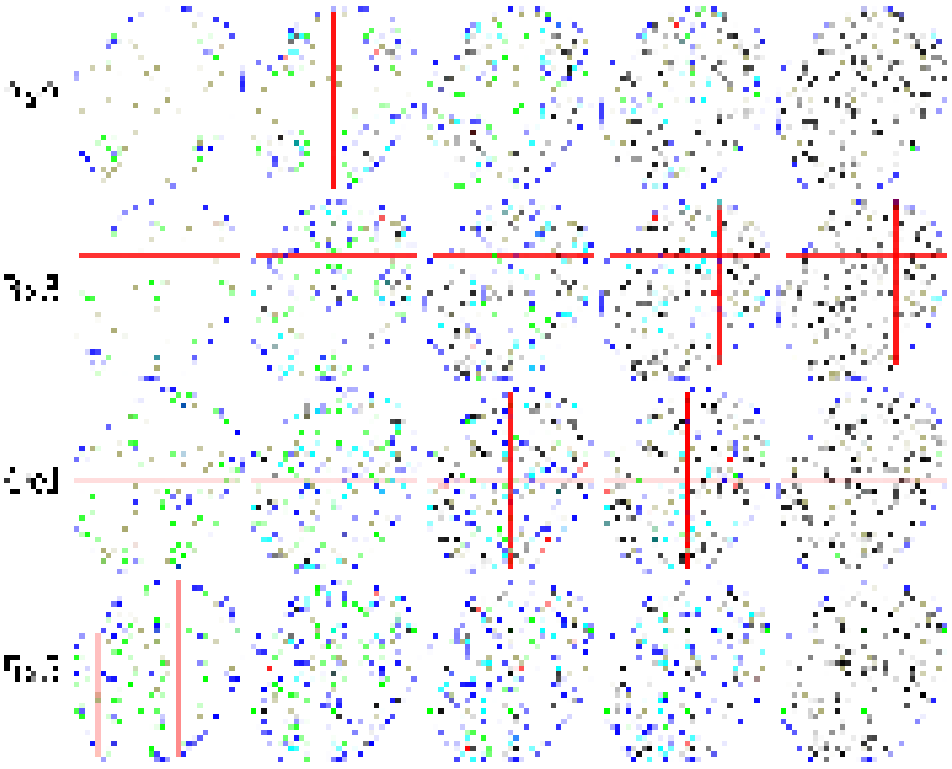


Figure 3.10: Parallel computing Voronoi diagram using GPU based sweep circle algorithm. The entire domain is uniformly divided into m^2 sub-regions (where $m = 2,3,4,5$ for this experiment), and a sweep circle thread is run for each sub-region independently. The intermediate results are shown in the left and the right most column is the final result. The active edges, the sweep circle and the beach curve are drawn in cyan, green and blue, respectively.

s_i, where r_i is the weight of site s_i. The active edge structure is also modified to contain both the line segments and hyperbolic segments, as shown in Figure 3.11.

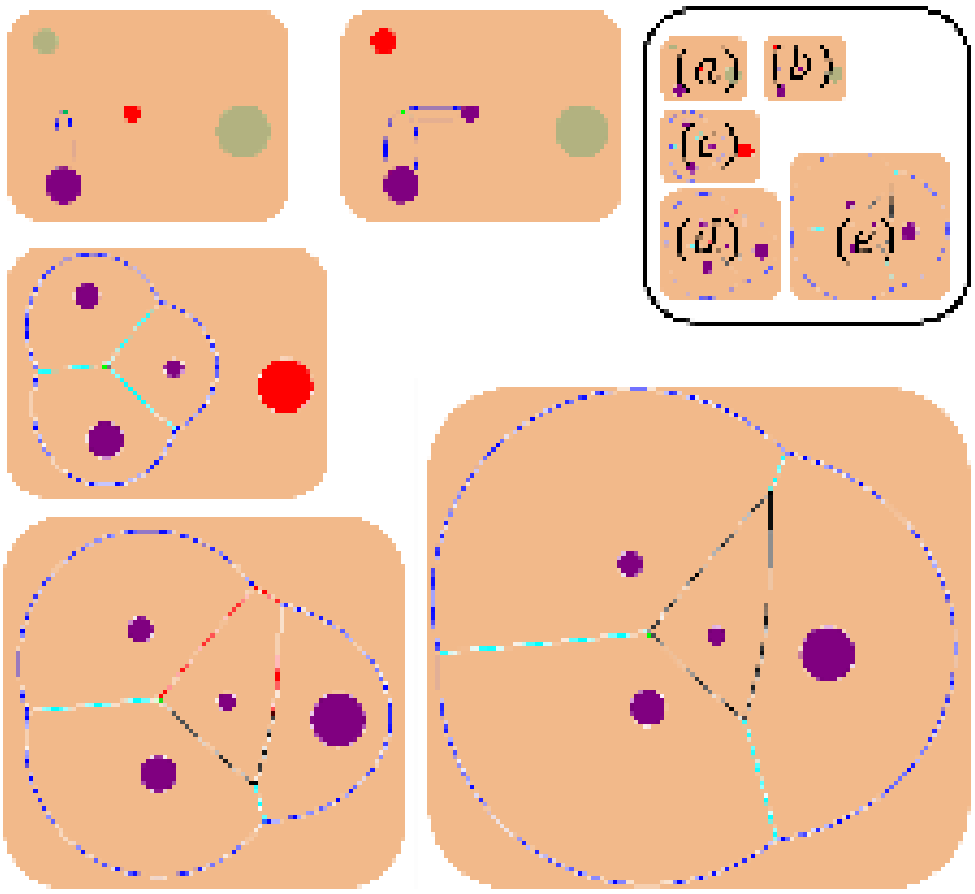


Figure 3.11: Computing the additively weighted Voronoi diagram, where the conventional Euclidean distance is modified by the weights assigned to the generator sites. The weights are illustrated by the size of the site. The resulting Voronoi diagram contains both hyperbolic segments and line segments. The animation from (a) to (e) illustrates the sweeping process.

3.6 Discussion

Generally speaking, a divide-and-conquer Voronoi algorithm needs to overcome two difficulties: 1) computing the Voronoi diagram in a sub-domain; and 2) merging the separate results into a complete diagram. Taking Figure 3.2 for an example, a parallelized sweepline algorithm needs to distinguish the diagram in the green square from

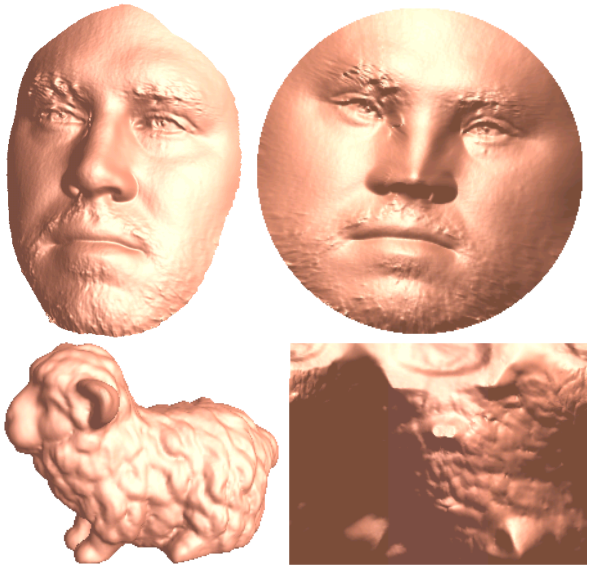


Figure 3.12: 3D models and their parameterization.

other part for merging purpose, because the computed result outside the green square may be incorrect. However, our algorithm are different than the sweepline algorithm to this point, due to the property that our algorithm guarantees the computed Voronoi diagram behind the beach curve is globally correct.

The robustness is very important for computing Voronoi diagram and Delaunay triangulation. For example, *In-Circle* test is a frequent routine in computing Delaunay triangulation. The operation, however, is error-prone. Different than the In-Circle operation, we use a sweepcircle to determine the priority in handling sites. Suppose s_1 and s_2 are in a nearly equal distance to the sweepcircle center. Even if the handling orders of s_1 and s_2 are reversed due to numerical problems, it will not affect the final Voronoi diagram since s_1 and s_2 lead to changes in different parts of the beach curve. Instead, we found two cases are critical to the robustness of the sweepcircle algorithm: 1) When we insert a new elliptic arc segment e, it intersects an existing elliptic arc e_i that nearly vanishes; and 2) two consecutive elliptic arc segments vanish in a very short time gap. These cases happen when more than 3 Voronoi cells share a common vertex. In our implementation,

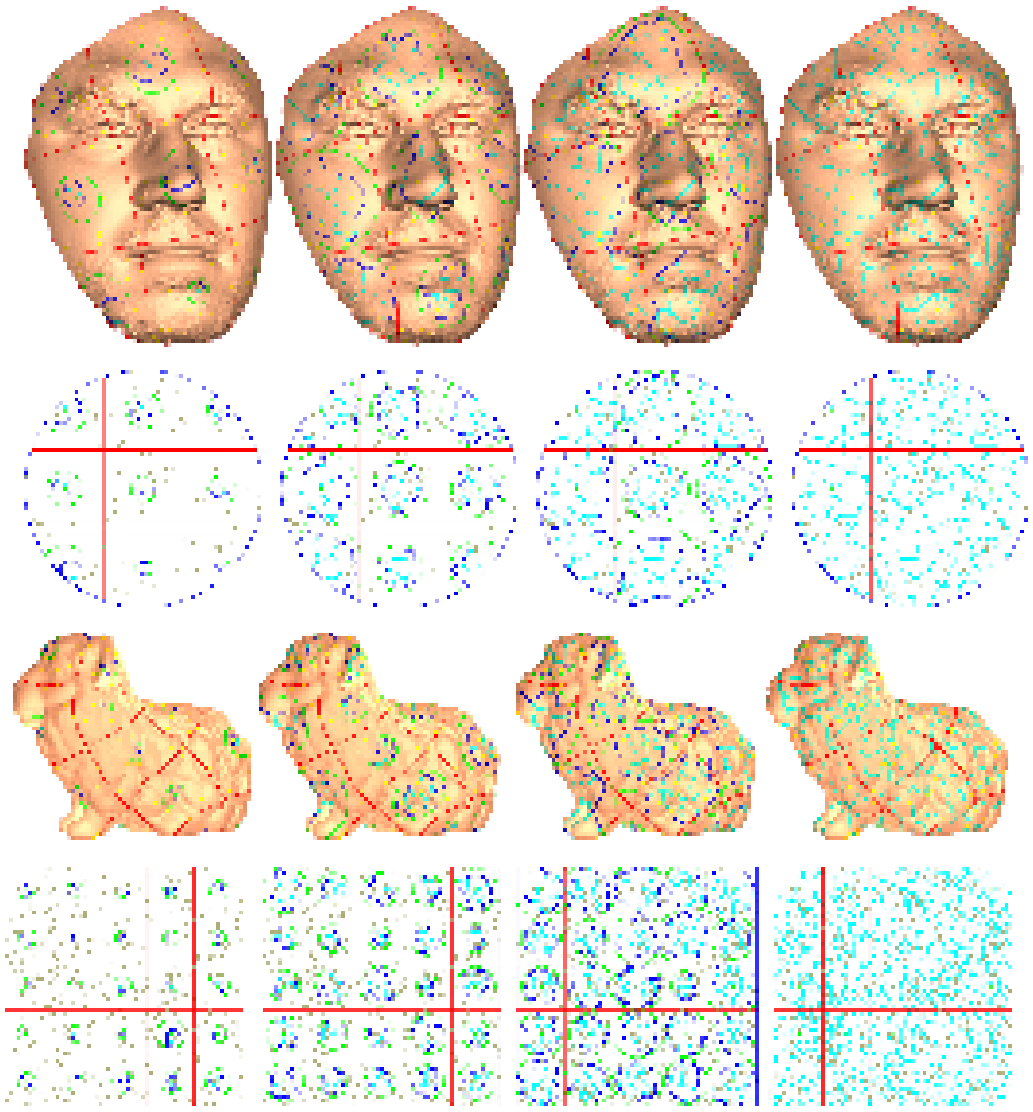


Figure 3.13: Our method can be applied to 3D surfaces by using parameterization.

we use a tolerance $1e-7$ to identify if the vanishing point is a high-order Voronoi vertex.

Thurston [136] observed the sweepcircle technique allows to compute the Voronoi diagram locally, which shows the Voronoi diagram can be parallelized in nature. Dehne and Klein [32] applied the sweepcircle to compute a type of transformed Voronoi diagram. Although their algorithm also supports multiple sweep circles, it is fundamentally different with our untransformed sweepcircle algorithm.

(i) In the traditional sweepcircle algorithm, the transformed edges are very complicated in representation form. By contrast, our algorithm is easy to implement since the most complicated operation is nothing but an arc-cosine calculation.

(ii) The beach curve consists of elliptic segments in our untransformed sweepcircle algorithm, while the beach curve is rather complicated in Dehne and Klein's sweepcircle algorithm (they didn't mention the representation form of the beach curve in [32]).

(iii) Our algorithm is very natural to extend onto 2D weighted Voronoi diagrams, while it is not clear whether it is easy or not for Dehne and Klein's sweepcircle algorithm to support 2D weighted Voronoi diagrams.

(iv) Dehne and Klein's sweepcircle algorithm can be easily extended to the Voronoi diagram on the surface of a cone, while it seems not easy for our algorithm.

3.7 Summary

This chapter presents the untransformed sweepcircle algorithm for 2D Voronoi diagram. Our algorithm has the optimal $O(n \log n)$ time complexity and $O(n)$ space complexity. The classical sweep line algorithm is the degenerate form of our algorithm when the circle center is at infinity. Our algorithm is flexible in that it allows multiple circles at

arbitrary locations to sweep the domain simultaneously, which naturally leads to a parallel implementation. It is easy to implement, without complicated numerical calculation. We demonstrate the efficacy of our parallel sweep circle algorithm using GPU.

Chapter 4

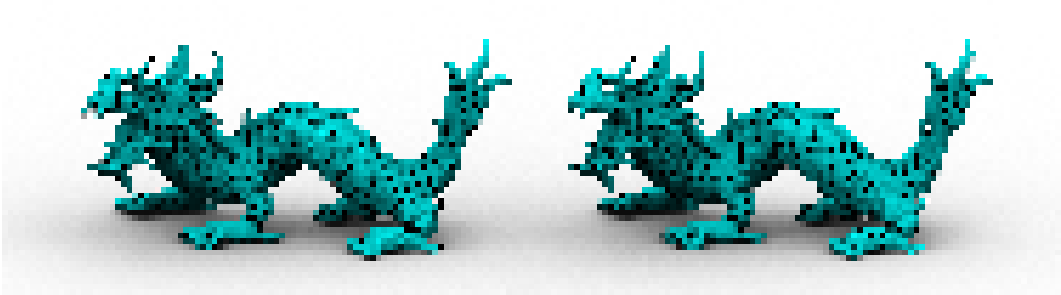
Saddle Vertex Graph (SVG): A Novel Solution to the Discrete Geodesic Problem

4.1 Overview

This chapter presents the Saddle Vertex Graph (SVG), a novel solution to the discrete geodesic problem. The SVG is a sparse undirected graph that encodes complete geodesic distance information - for example a geodesic path on the mesh is equivalent to a shortest path on SVG - which can be solved efficiently using the shortest path algorithm (e.g., Dijkstra algorithm). The SVG method solves the discrete geodesic problem from a local perspective. We have observed that the polyhedral surface has some interesting and unique properties, such as the fact that the discrete geodesic exhibits a strong local structure, which is not available on the smooth surfaces. The richer the details and complicated geometry of the mesh, the stronger such local structure will be. Taking advantage of the local nature, the SVG algorithm breaks down the discrete geodesic problem into significantly smaller sub-problems, and elegantly enables information reuse. It does not require any numerical solver, and is numerically stable and insensitive to the mesh resolution and tessellation. Users can intuitively specify a model-independent

parameter K, which effectively balances the SVG complexity and the accuracy of the computed geodesic distance. More importantly, the computed distance is guaranteed to be a metric. The experimental results on real-world models demonstrate significant improvement to the existing approximate geodesic methods in terms of both performance and accuracy.

4.2 Introduction



Exact polyhedral distance: ICH 127s; MMP 144s
Our result: 1.96s, max err. 0.11%, rms err. 0.05%, mean err. 0.04%

Figure 4.1: It takes 39.1 seconds to construct an approximate SVG for the 1.5M-face Dragon on an Nvidia Tesla K20 GPU. Then any subsequent computation of the single source geodesic distance takes less than 2.0s on a 2.66GHz Intel Xeon machine using a single core. Therefore, our method is highly desirable to the applications that require frequent geodesic computations. Moreover, our method guarantees the computed geodesic distance is a metric, which distinguishes itself from all the other approximate geodesic algorithms.

The discrete geodesic problem has attracted a great deal of attention since Mitchell, Mount and Papadimitriou published their seminal paper in 1987. The MMP algorithm [97] computes the single-source geodesic distance in $O(n^2 \log n)$ time, where n is the number of vertices of the input mesh. In 1990, Chen and Han [21] improved the time complexity to $O(n^2)$, which remains the best-known complexity. However, extensive experiments have shown that the CH algorithm often runs 10^3 to 10^4 times slower than

the MMP algorithm. Xin and Wang realized that such slowness is due to the extremely large number of operations to the useless windows (a data structure to that will be explained later), so they modified the CH algorithm by adopting window filtering and maintaining windows in a priority queue. Although the improved CH (or ICH) algorithm has $O(n^2 \log n)$ time complexity, it outperforms both the MMP and the CH algorithms.

So why can the theoretical time complexity not indicate the real performance in the discrete geodesic problem? The exact geodesic algorithms, such as MMP, CH and ICH, all adopt the same computational framework: each edge is partitioned into a set of intervals, called windows, and each window encodes the geodesic distance from the source to any point in the interval. The windows are maintained in either a sorted data structure (like a priority queue in the MMP and ICH algorithms) or a hierarchical data structure (like a tree in the CH algorithm). In each iteration, the algorithms propagate one window across a triangle by computing the window's children windows. The algorithms terminate when all windows have been processed. Thus, window complexity is a key factor to measure the real performance of the algorithm. Mitchell et al. [97] and Chen and Han [21] noted that the number of windows is $O(n^2)$, which means that none of the the window propagation algorithms can break the theoretical $O(n^2)$ time barrier. On the other hand, Surazhsky et al. [133] observed that the window complexity grows as approximately $O(n^{1.5})$ on real-world models. In practice, the MMP and the ICH algorithms run in sub-quadratic time. Surazhsky et al. [133] also observed that given a smooth surface, adding a random noise at each vertex can reduce the window complexity significantly. This interesting observation is confirmed by the experience that computing the geodesic on a bumpy real-world model usually takes less time than that for a smooth synthetic model of similar complexity. However, the existing literature has not addressed the question of the relation between the algorithm's time complexity and the surface's geometric features.

All of the existing algorithms compute the globally shortest geodesic by propagating the distance information in the wavefront order, thereby tackling the discrete geodesic problem in a global manner. In this paper, we take a completely different approach, observing the discrete geodesic's very strong local structure. The richer the geometric features of the mesh, the stronger such local structures are. This allows us to break down the original problem into easily solvable sub-problems.

In moving towards this goal, we propose the *Saddle Vertex Graph (or SVG)*, a novel method for computing the geodesic distance on a triangle mesh. The SVG is a sparse undirected graph that has the same vertex set as the input mesh. Each SVG edge corresponds to a direct geodesic path that does not pass through any saddle vertices. The SVG encodes complete geodesic information in that a geodesic path on the mesh is a shortest path on the corresponding SVG; therefore, the discrete geodesic problem is converted into a shortest path problem.

The SVG has several unique features that distinguish it from other algorithms.

- Firstly, it is highly efficient, since computing the single-source geodesic distances on a real-world model takes $O(Dn\log n)$ time, where $n = |V|$ is the number of mesh vertices and D ($D \ll n$) measures the complexity of the SVG. Moreover, as the Dijkstra algorithm does not involve any numerical computation, the SVG is much faster than the fast marching method (FMM).

- Secondly, users can intuitively control the accuracy by specifying the local covering range of the saddle vertices. The computed geodesic distance is guaranteed to be a metric, i.e., satisfying the symmetry condition and the triangle inequality. Since it does not require any numerical solver, the SVG method is numerically stable and robust to the mesh triangulation and tessellation.

- Thirdly, the SVG exhibits a strong local structure that enables it to be constructed in a completely local and parallel manner. Furthermore, the SVG's average degrees depend on the model's details, and is insensitive to the mesh size. The more details and complicated geometry the model has, the stronger such local characteristics are, and the *smaller* the average degree of the SVG vertices. Therefore, the results for SVG are much better for real-world models with details than the poor results that existing approximate algorithms often produce.
- Last but not least, it naturally links the discrete geodesic problem with the shortest path problem, which has been studied extensively. Therefore, the discrete geodesic problem can immediately take advantage of existing resources as well as new developments in the shortest path problem (such as parallelization).

4.3 Preliminary

Let $M = (V, E, F)$ be a triangle mesh representing an orientable 2-manifold where V, E and F are the vertex, edge and face sets respectively. Let $n = |V|$ be the number of vertices. Throughout the paper, we always assume the mesh M is connected.

The total vertex angle of a vertex $v \in V$ is the sum of interior angles formed by the edges incident at v. A vertex v is called *spherical* if its total vertex angle is less than 2π, *Euclidean* if the total vertex angle equals 2π, and *saddle* if the total vertex angle is greater than 2π. Let $\gamma(p,q)$ denote a geodesic path between p and q. Note that there may be multiple geodesic paths between p and q. We denote $d(p,q)$ the *globally* shortest geodesic distance between p and q.

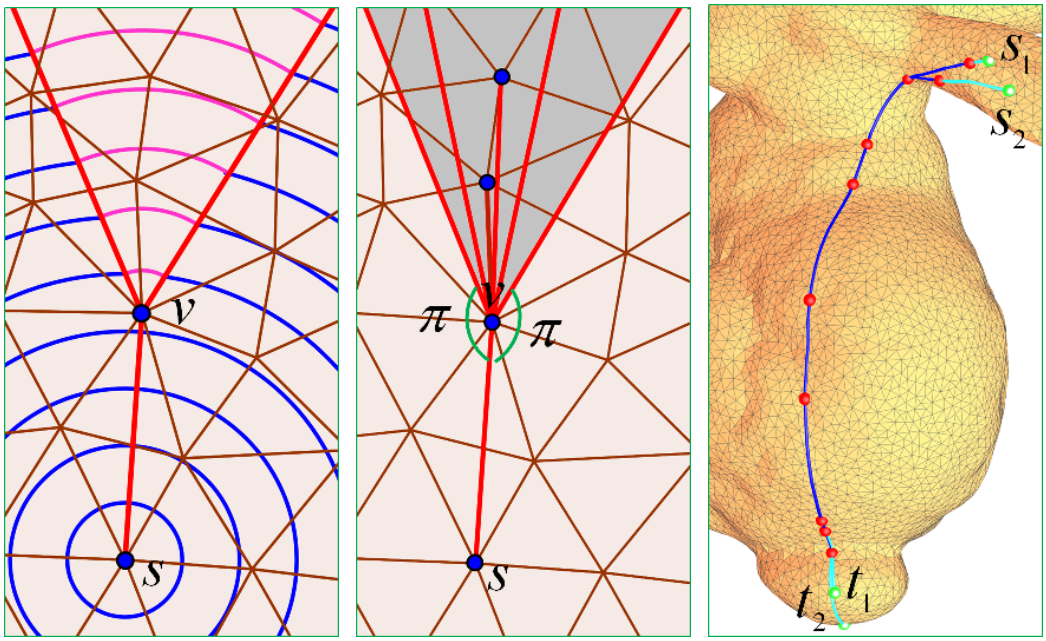
Clearly, a geodesic path must be a straight line inside a triangle. When crossing over an edge, the geodesic path must also correspond to a straight line if the two adjacent faces

are unfolded into a common plane. Mitchell et al. [97] proved the following theorems of discrete geodesics.

- There exists a geodesic path from the source to any other point on the surface.
- A geodesic path goes through an alternating sequence of vertices and edges such that the unfolded image of the path along any edge sequence is a straight line segment and the angle of the path passing through a vertex is greater than or equal to π.
- A geodesic path cannot pass through a spherical vertex unless it is an endpoint or a boundary point.

Both the MMP and CH algorithms adopt a similar window propagation framework, in which the saddle vertices play a critical role by acting as "relays" that propagate the windows from the source towards the destination. As Figure 4.2(a) shows, when a geodesic wavefront passes through a saddle vertex v, it splits into three arcs; the left and right arcs remain centered at the original source, but the central arc is originated at v. These arcs are then propagated with respect to their own centers, causing the geodesic path to be split at v (see Figure 4.2(b)). This is the reason that saddle vertices are called pseudosources in the MMP, CH, and ICH algorithms. As Figure 4.2(c) shows, a typical geodesic path γ may pass through one or more saddle vertices, and two different geodesic paths $\gamma(s_1,t_1)$ and $\gamma(s_2,t_2)$ may share common segments.

Remark. A geodesic path $\gamma(s,t)$ on a smooth surface is both straightest and locally shortest. Therefore, two geodesic paths cannot share a common segment. However, the locally shortest geodesic path is not equivalent to the straightest geodesic path on a piecewise linear surface [108]. Due to this fundamental difference between the smooth surfaces and the polygonal surfaces, the discrete geodesic has some unique features that are not available in the smooth counterpart. For example, the shortest geodesic cannot



(a) (b) (c)

Figure 4.2: Saddle vertex and geodesic path. (a) A saddle vertex v has the total vertex angle $\sum \theta_i > 2\pi$. When passing through v, the geodesic wavefront, a circle centered at the source s, splits into three arcs. The middle arc (in pink) is centered at v, and the left and right arcs remain centered at s. (b) The incoming geodesic path γ splits at the saddle vertex v. All the outgoing geodesic paths are in the fanned area (shaded in dark grey), which has angle $\sum \theta_i - 2\pi$. (c) A long geodesic path γ usually passes through one or more saddle vertices (the red points), which partition γ into several segments. Two geodesic paths $\gamma(s_1,t_1)$ and $\gamma(s_2,t_2)$ may share a large portion of their paths.

be extended through a spherical vertex since it could be shortened by moving off the corner. There exists a family of extension of a geodesic through a saddle vertex. Two or more geodesic paths can share common segments with saddle vertices as endpoints. Taking advantages of these unique features of the discrete geodesic, we define our saddle vertex graph in the next section.

4.4 Saddle Vertex Graph

4.4.1 Definition

Definition 1. Consider $\gamma(p,q)$ a globally shortest geodesic path between p and q. γ is called *direct* if it does not pass through any saddle vertices, and *indirect* otherwise.

An indirect path contains one or more saddle vertices, by which the geodesic path is partitioned into several segments, each of which is a direct path.

Let V_S denote the set of saddle vertices and V_N denote the set of non-saddle vertices (i.e., Euclidean and spherical vertices), then we have $|V| = |V_S| + |V_N|$.

Definition 2. Consider a non-saddle vertex $v \in V_N$. The *saddle neighbors* of v, denoted by $\mathscr{S}(v)$, are the saddle vertices which can be reached from v via a direct geodesic path. Similarly, the *non-saddle neighbors* of v, denoted by $\mathscr{N}(v)$, are the non-saddle vertices which can be reached from v via direct geodesic paths.

Definition 3. The *direct neighbors* of a vertex $v \in V$ are the vertices that can be reached from v via direct geodesic paths.

Definition 4. An S-S edge is a *direct* geodesic path between two saddle vertices. An N-S edge is a *direct* geodesic path between a non-saddle vertex and a saddle vertex. An

N-N edge is a *direct* geodesic path between two non-saddle vertices. Let E_{SS}, E_{NS}, and E_{NN} denote the set of S-S edges, N-S edges, and N-N edges respectively,

$$E_{SS} = \{\gamma(p,q) | p,q \in V_S \text{ and } \gamma(p,q) \text{ is direct}\},$$
$$E_{NS} = \{\gamma(p,q) | p \in V_N,\ q \in V_S \text{ and } \gamma(p,q) \text{ is direct}\},$$
$$E_{NN} = \{\gamma(p,q) | p,q \in V_N \text{ and } \gamma(p,q) \text{ is direct}\}.$$

Clearly, the sets E_{SS}, E_{NS} and E_{NN} are disjoint. Figure 4.3 shows examples of S-S, N-S and N-N edges on the Bimba model. We are now ready to define the Saddle Vertex Graph.

Definition 5. The SVG associated to a triangle mesh $M = (V,E,F)$ is an undirected graph $S = S_1 \bigcup S_2 \bigcup S_3$ formed in three tiers.

- The tier 1 sub-graph $S_1 = (V_S, E_{SS})$, which is the core network, consists of all the S-S edges connecting the saddle vertices via direct geodesic paths.
- The tier 2 sub-graph $S_2 = (V_S \bigcup V_N, E_{NS})$ contains the N-S edges, which are the direct geodesic paths connecting the non-saddle vertices to the saddle vertices.
- The tier 3 sub-graph $S_3 = (V_N, E_{NN})$ contains the N-N edges, which are the direct geodesic paths connecting the non-saddle vertices.

The SVG is connected and encodes the complete geodesic paths between any two vertices.

Proposition 1. The SVG of a connected triangle mesh is connected.

Proof. If $|V_S| \geq 1$, $S_1 \bigcup S_2$ is connected, since S_1 is connected and each non-saddle vertex is connected to at least one saddle vertex. If $V_S = \varnothing$, there are no tier 1 and tier 2 graphs, and every geodesic path is direct. Therefore, S_3 is connected. In either case, the SVG $S_1 \bigcup S_2 \bigcup S_3$ is connected.

Proposition 2. A globally shortest geodesic path on M is a shortest path on its associated SVG.

Proof. Let $p \in V$ and $q \in V$ be two vertices. If there is a direct geodesic path connecting p and q, that path is an edge of SVG. Otherwise, the geodesic path $\gamma(p,q)$ is indirect and can be partitioned into several segments, each of which is direct, thus, an edge in SVG. As the globally shortest geodesic path minimizes the distance, the geodesic path $\gamma(p,q)$ is also a shortest path on the SVG. ∎

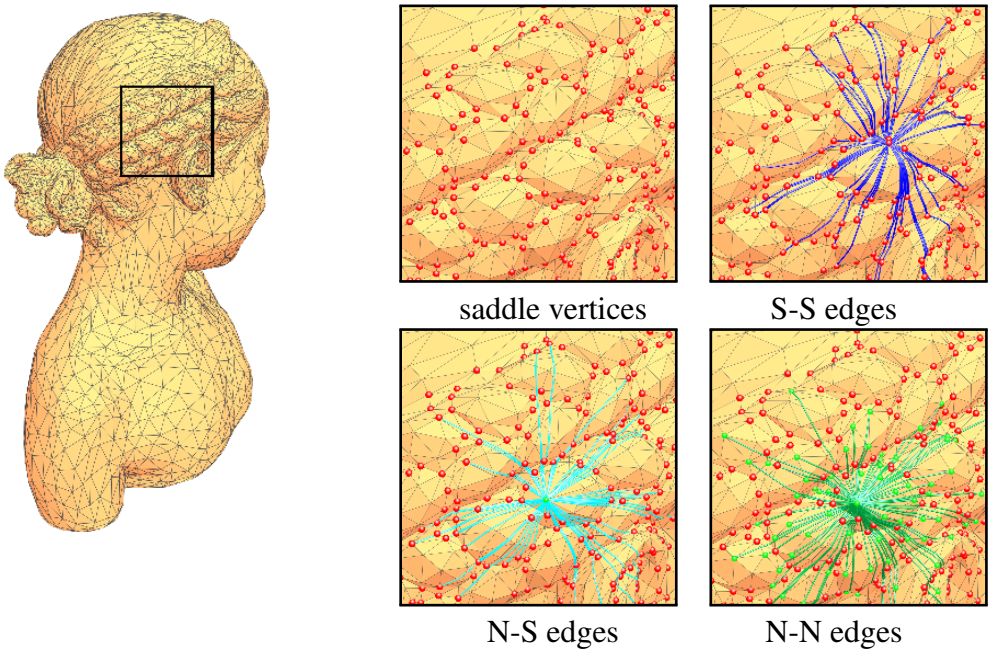


Figure 4.3: The SVG on the 9K-face Bimba. The saddle vertices are drawn in red. For illustration purpose, we show only the S-S, N-S and N-N edges incident to a vertex.

Throughout this paper, we use $\{p,q\}$ to denote the undirected SVG edge between p and q, and $\|\{p,q\}\|$ to denote its length, i.e., the geodesic distance $d(p,q)$, and $\|pq\|$ to denote the Euclidean distance between p and q.

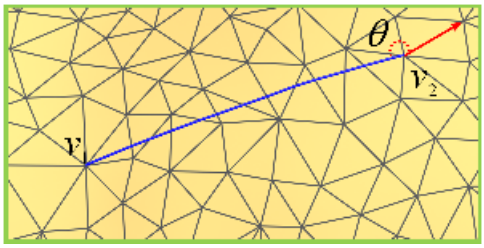
Remark. If the mesh is viewed as a planet, the vertices are the cities, and the geodesic path between two vertices is a flight route. So the SVG is indeed a flight route map, which covers every city on the planet. The saddle vertices are the hub cities and the

non-saddle vertices are the small cities. The S-S edges are the major flight routes connecting the hubs. Each N-S edge is a regional flight route connecting a small city to the nearest hub. Each N-N edge is a local flight route between two nearby small cities. Travelers moving between cities not served by direct flights change planes en route to their destinations.

4.4.2 Data Structure

We use the conventional incidence list to store the SVG. Each vertex stores its incident edges (i.e., the direct geodesic paths) and each edge stores one incident vertex and its length (i.e., the geodesic distance).

```
struct SVGVertex {
    int id;
    bool is_saddle;
    vector<SVGEdge> edges_incident_to_nonsaddle_vertices;
    vector<SVGEdge> edges_incident_to_saddle_vertices;
};
struct SVGEdge {
    int v2;
    double length, theta;
};
```



In order to find the geodesic path, we also associate each SVG edge with an angle θ, which measures the direction of the geodesic path with respect to its endpoint v_2. The reference direction (the red edge in the figure on the right) is defined by using v_2's local information, such as its first incident mesh edge. To obtain the geodesic path $\gamma(v_1, v_2)$, we back-trace the curve from the endpoint v_2: with the reference direction and the angle θ, we can obtain the first segment of γ, which is a straight line in triangle T, where T is one of v_2's incident triangles. We then unfold T's adjacent triangle - say, T' - into a common plane

to extend the line to T'. We repeat this unfolding until the geodesic path reaches v_1. As the direct geodesic path does not pass through the saddle vertex, this unfolding leads to a unique face/edge sequence and thus produces a unique geodesic path. The procedure Backtrace(e) is defined to compute the direct geodesic path for an SVG edge e. The pseuocode of the procedure Backtrace is shown in the Supplementary Material.

4.4.3 Complexity Analysis

The complexity of SVG obviously depends on the number of saddle vertices and their distribution. Consider an extreme case in which every vertex is either spherical or Euclidean (such as a convex polyhedron or a developable surface), $V_S = \varnothing$ and $|E_{NN}| = \binom{n}{2}$, since any geodesic path between two vertices is direct. In this case, S_3 is dense and the whole SVG is also dense.

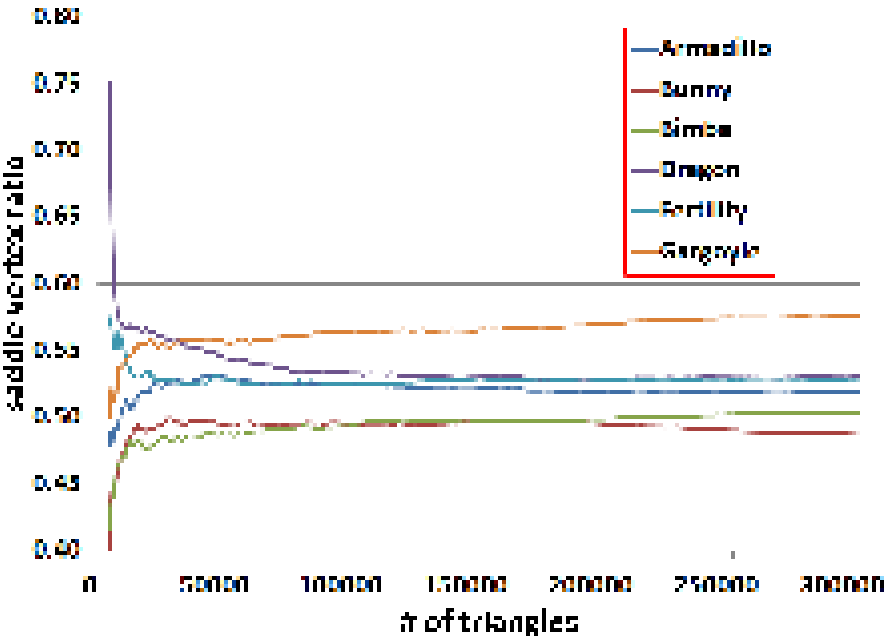


Figure 4.4: The saddle vertex ratio $r = \frac{|V_S|}{|V|}$ is fairly stable with respect to the mesh resolution and tessellation.

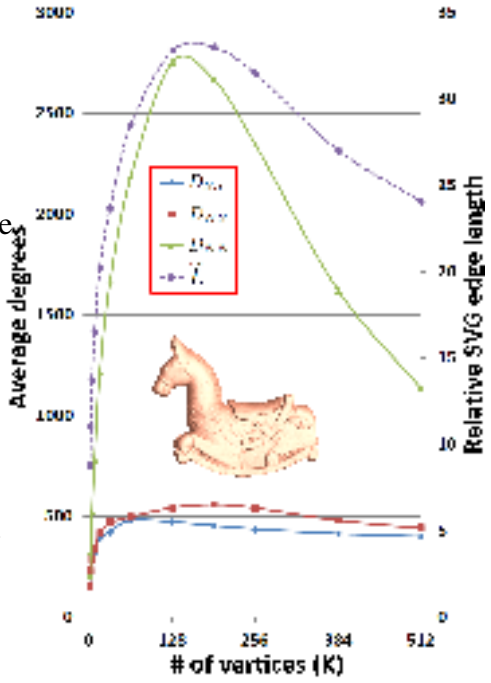
Fortunately, such extreme cases are rare in real-world applications (see Section 6.4 for

the handling of such cases). We tested the saddle vertex ratio $r = \frac{|V_S|}{|V|}$ on common models of various resolutions. For each model, we generate triangle meshes ranging from 1K triangles to 300K triangles. We have observed that the value of r may vary at low resolution, but it becomes fairly stable when the mesh resolution is high. The typical value of r for real-world models ranges from 40%-60%. See Figure 4.4. Thanks to such a large proportion of saddle vertices, most of the geodesic paths are indirect and the SVG is a sparse graph. See Table 4.1.

Through extensive experiments on a wide range of real-world models, we have observed that the SVG exhibits a strong local structure. The more details and complicated geometry the model has, the stronger the local structure. To quantitatively measure the local structure of the SVG, we consider the length of the SVG edges. We have observed that the length of the direct geodesic path is inversely proportional to the local vertex density (the number of vertices in a region divided by the region's area), the higher the density, the shorter the length, and vice versa. This is because the length of the direct geodesic path depends on the number and distribution of saddle vertices, which in turns depends on the mesh resolution. The lower the resolution, the lower the density, the smoother the surface, the less number of saddle vertices, and the longer the direct geodesic paths. On the other hand, the higher the resolution, the higher the vertex density, the more bumpy the surface, the greater the number of saddle points, and the shorter the direct geodesic paths. To obtain a resolution-independent measure, we define the *relative* SVG edge length $\widetilde{L}$, as the ratio of the average length of the direct geodesic path and the average mesh edge length. We also define the average SVG degrees for the sub-graphs S_1, S_2 and S_3 to measure the SVG complexity:

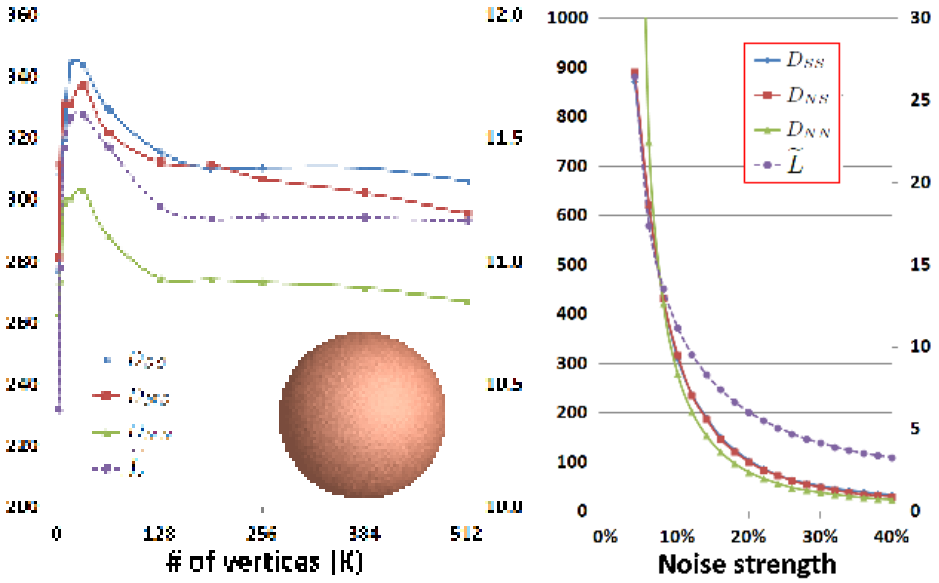
$$D_{SS} = \frac{2|E_{SS}|}{|V_S|}, D_{NS} = \frac{|E_{NS}|}{|V_N|}, \text{ and } D_{NN} = \frac{2|E_{NN}|}{|V_N|}.$$

The coefficient 2 in D_{SS} and D_{NN} is due to the double counting of the E_{SS} and E_{NN} edges.



By investigating how the mesh complexity affects the SVG's local structure and its complexity, we have observed that all the tested real-world models exhibit an inverted U-shaped curve. The figure on the right shows the curves for the Isidore Horse model. The horizonal axis is the mesh complexity in terms of the number of vertices, while the left and right vertical axes are the average SVG degrees and the relative SVG edge length $\widetilde{L}$. When the resolution is low, the model has almost no details and its body is relatively smooth. Therefore, the degrees increase when the mesh size is increased. When the resolution is sufficiently high, the surface becomes bumpy due to the addition of details and has many saddle vertices. Consequently, the S-S and N-S edges become shorter. Furthermore, as each non-saddle vertex is surrounded by more saddle vertices, and the direct geodesic path that originates from the non-saddle vertices cannot reach too far away, the N-N edges are also shorter. Therefore, the average degrees and the relative SVG edge length $\widetilde{L}$ all tend to decrease at the high resolution, which confirms that the SVG has strong local structure when the mesh is of high resolution and has details. We also test the SVG complexity and its local structure on the sphere model. Because all vertices of the sphere are non-saddle, we add geometrical noise to perturb the vertex position. We have found that the SVG of the bumpy sphere exhibits the same characteristics as the real-world models. See Figure 4.5.

Let $D = \max\{D_{SS}, D_{NS}, D_{NN}\}$ denote the maximal degree, which is significantly less than the mesh size $D \ll n$ for real-world models. Therefore, the SVG is in general a sparse graph and the space complexity is $O(Dn)$.



(a) (b)

Figure 4.5: (a) Adding geometrical noise to the sphere, which moves each vertex in a random direction by a uniform random distance 0 to 10% of average edge length. (b) Increasing the noise strength makes the sphere bumpier. As a result, both the average degrees and the relative SVG edge length drop. The left and right vertical axes show the average degrees and the relative SVG edge length $\widetilde{L}$, respectively.

4.5 SVG Construction

The exact SVG contains *all* direct geodesic paths, by which we can compute the exact polyhedral distance between any two vertices. In practice, we have found that an approximate[1] SVG with only a subset of the direct geodesic paths can still lead to highly accurate geodesic distance, but requires much less memory and is more efficient to construct than the exact SVG.

4.5.1 Parameter

Our goal is to find a model and resolution insensitive parameter that can intuitively and effectively control the SVG complexity. Therefore, the maximal SVG edge length is not suitable, since it is resolution-dependent. The relative SVG edge length cannot be used either, since it is model dependent and may not be effective for anisotropic meshes.

In our paper, we use the maximal number of mesh vertices covered by a geodesic disk, denoted by K, as the parameter for controlling the SVG complexity. Given a vertex $v \in V$, we consider a geodesic disk $\odot(v,R)$ centered at v with a radius R that contains no more than K mesh vertices. Then the direct geodesic paths within $\odot(v,R)$ are taken as the SVG edges.

There are a few advantages of using K as the parameter. First, it is a good measure of the computational cost. Computing the geodesic disk $\odot(v,R)$ with the ICH/MMP algorithm takes the worst-case $O(K^2 \log K)$ time and empirical $O(K^{1.5} \log K)$ time. Second, the parameter K *directly* controls the number of direct geodesic paths in the disk $\odot(v,R)$, which is insensitive to the mesh resolution and tessellation. Let us examine the geodesic

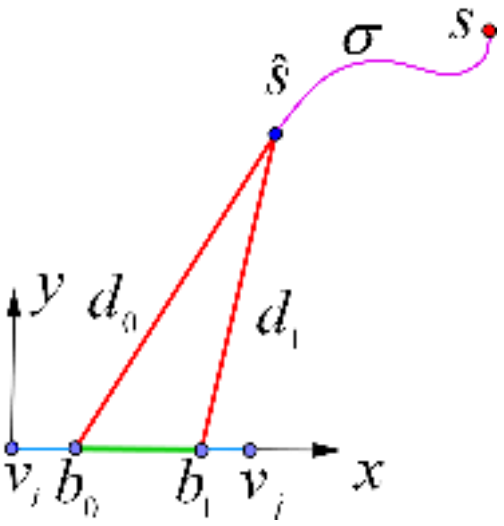
[1] By *approximate*, we mean that the SVG's edge set is a subset of the exact SVG's edge set. Each SVG edge (p,q) in the approximate SVG is still an exact direct geodesic path, which measures the accurate geodesic distance between p and q.

disks $\odot(v,R)$ on a model M in low resolution and high resolution, denoted by M_l and M_h, respectively. Fixing the value of K of course leads to geodesic disks of different radii on M_l and M_h. The radius R_h is small, so the vertex density is high and the direct geodesic paths are short. On the other hand, the radius R_l is big, so the vertex density is low and the direct geodesic paths are long. Regardless of the size, the large and small disks roughly contain the same number of direct geodesic paths originated from v, since each geodesic path uniquely corresponds to a tangent direction at v and these tangent directions are intrinsic and insensitive to the mesh resolution and tessellation. Therefore, the parameter K is model independent and can be used to control the SVG complexity and analyze the time complexity of SVG construction.

Model	$\|V\|$	r	D_{SS}	D_{NS}	D_{NN}	$\widetilde{L}$	ind.%
Bimba	74,764	51.6%	1279.1	1179.3	1168.6	30.4	96.9%
Buddha	488,217	56.1%	861.7	862.3	715.3	28.2	99.6%
Bunny	72,020	54.3%	1029.8	998.2	1295.9	22.5	97.4%
Dragon	422,558	54.2%	1044.2	1043.6	924.1	29.8	99.5%
Fertility	29,994	52.3%	1483.4	1225.4	1172.1	26.4	91.6%
Golfball	122,882	31.0%	693.3	694.9	1586.4	20.9	98.2%
Gargoyle	349,998	58.0%	711.2	699.9	501.2	18.6	99.6%
Kitten	137,098	50.9%	991.3	914.6	981.8	20.2	98.6%
Lucy	262,909	54.1%	390.5	325.6	1003.1	22.9	99.5%
Ramses	826,266	60.4%	365.8	322.8	204.6	12.5	99.9%

Table 4.1: Statistics of the SVG complexity on common models. The last column shows the percentage of the indirect geodesic paths.

4.5.2 Computing Direct Geodesic Paths



To construct the SVG, we must compute the direct geodesic paths. We adopt the half-edge data structure to store the triangle mesh M. Each mesh edge is decomposed into two half-edges, with opposite directions. A window w associated to a half-edge $e = (v_i, v_j)$ is a 5-tuple $(b_0, b_1, d_0, d_1, \sigma)$, where b_0, b_1 define the endpoints of w (b_0 is closer to the vertex v_i where the half-edge is originated), d_0, d_1 are the corresponding distances to the pseudosource, and σ is the geodesic distance from the source s to the pseudo source $\hat{s}$. We define the direct window as follows:

Definition 6. A window $w = (b_0, b_1, d_0, d_1, \sigma)$ is said *direct* if $\sigma = 0$.

Intuitively, a direct window is an interval on a half-edge for which there is a direct geodesic path from the source to any point in the interval. To obtain the direct geodesic paths, we run the single source geodesic algorithm (like the ICH or MMP algorithm) for each vertex. The local structure of the SVG means that it is not necessary for the geodesic algorithm to run for the entire mesh, which would be very time-consuming. When a window w propagates, the value σ of w's child windows cannot decrease. Therefore, the geodesic algorithm can simply stop when all direct windows have been processed.

4.5.3 Algorithm

Our SVG construction algorithm is shown in Algorithm 1. For each vertex $v_i \in V$, we run the procedure ComputeDirectGeodesicPaths (see the Supplementary Material for the pseudocode), which takes v_i as the source and computes a geodesic disk $\odot(v_i, R)$

that contains no more than K vertices. Then all the directed geodesic paths in $\odot(v_i, R)$ are taken to form the SVG edges.

Computing the direct geodesic paths is the bottleneck of the entire SVG construction algorithm. In our implementation, we choose the ICH algorithm to compute the direct geodesic paths due to its better performance and less memory requirement than the MMP algorithm. Moreover, the geodesic disks can be computed in parallel on the GPU: the mesh data structure is accessible to all threads in the read-only manner, and each thread of ComputeDirectGeodesicPaths maintains its own data (for example, the windows and the priority queues). To avoid the written conflicts, the thread for vertex v_i produces *directed* SVG edges originated from v_i. These directed edges are then converted into undirected edges by checking whether the opposite edges are available. See lines 5-9. Let N be the number of threads. The worst-case time complexity for computing the direct geodesic paths is $O(nK^2 \log K/N)$ and the empirical time complexity is $O(nK^{1.5} \log K/N)$.

Algorithm 3 Constructing the SVG

Require: A triangle mesh $M = (V, E, F)$ and the maximal number of vertices K in each geodesic disk;
Ensure: The saddle vertex graph S;
1: **if** each $v_i \in V$ **then**
2: **parallel** $S_i \leftarrow$ ComputeDirectGeodesicPaths(M, v_i, K);
3: **end if**
4: $S \leftarrow \bigcup_{i=1}^{|V|} S_i$;
5: **for** each edge $(s,t) \in S$ **do**
6: **if** edge $(t,s) \notin S$ **then**
7: add (t,s) into t's incident edges;
8: **end if**
9: **end for**

Algorithm 4 Computing the SSSD Geodesic Distance

Require: A triangle mesh $M = (V, E, F)$, the associated saddle vertex graph $S = S_1 \bigcup S_2 \bigcup S_3$ and two vertices $p_1, p_2 \in V$.
Ensure: The geodesic distance $d(p_1, p_2)$.

```
if p1, p2 are direct neighbor then
    return ||{p1, p2}||;
end if
U ← BidirectionalDijkstra({V, E}, p1, p2);
if S is exact then
    S'1 ← S1;
    for i=1,2 do
        if pi ∉ VS then
            S'1 ← S'1 ⋃ {pi and its N-S edges};
        end if
    end for
    A*Dijkstra(S'1, p1, p2, U);
else
    A*Dijkstra(S1 ⋃ S2 ⋃ S3, p1, p2, U);
end if
return dp1(p2);
```

4.6 Computing Discrete Geodesics Using SVG

4.6.1 Dijkstra's Algorithm

If the SVG is exact, according to Proposition 2, computing a geodesic path on the mesh is equivalent to find a shortest path on the SVG. On the other hand, the shortest path on an approximate SVG provides an approximated geodesic path. On each occasion, we adopt the Dijkstra algorithm [35] to compute the shortest path on the SVG.

Consider an undirected graph $G = (V, E)$ and a set of source points $\mathfrak{S} \subset V$. The procedure Dijkstra$(G, \mathfrak{S})$ assigns each vertex $v \in V$ a value $d(v)$, which is the shortest distance from v to the *closest* source $s \in \mathfrak{S}$.

The Dijkstra's algorithm can be implemented with various data structures, such as queue, priority queue, binary heap, Fibonacci heap, etc. We adopt the priority queue

based Dijkstra's algorithm due to its simplicity and good performance.

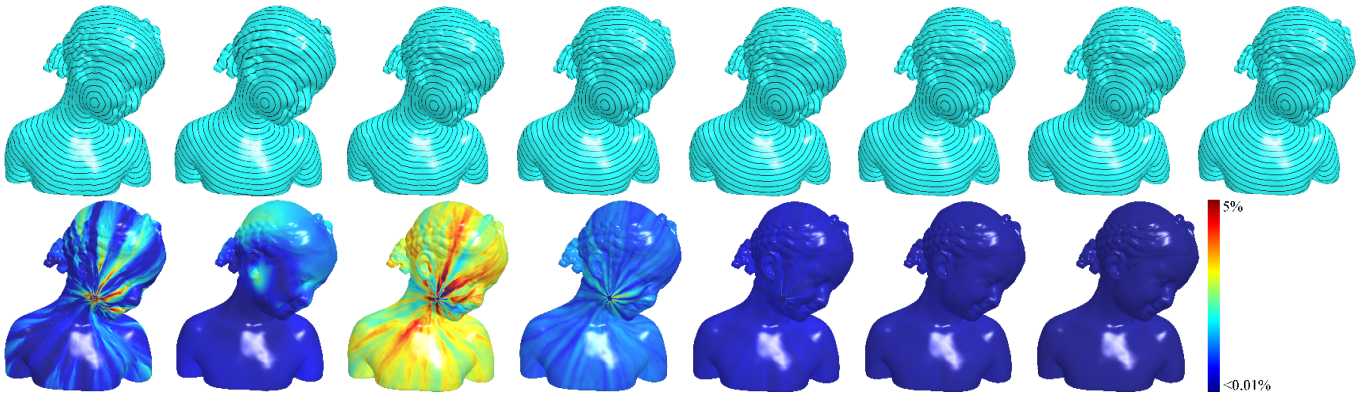
We also adopt two variants of the Dijkstra search. The procedure BidirectionalDijkstra(G,p,q) runs two simultaneous Dijkstra searches on the graph G: one forward from p and one backward from q, stopping when the two meet in the middle. The procedure returns the shortest distance between p and q. The procedure A$*$Dijkstra(G,p,q,U_{pq}) is the A$*$ search [107] [55] on the graph G, where U_{pq} is the heuristic estimate of the upper bound of the shortest distance between p and q. The A$*$ search is guided by an estimate of the remaining distance to the destination.

4.6.2 Single-source Single-destination (SSSD)

To compute the single-source single destination geodesic distance $d(s,t)$, we adopt the widely used A$*$ search [107] [133], which searches only a thin region surrounding the geodesic path $\gamma(s,t)$. Our SSSD algorithm contains two steps: first, we use BidirectionalDijkstra on the mesh edges to get the shortest distance U_{st}, which is the *upper* bound of the geodesic distance $d(s,t)$. Second, we use the A$*$ search on the SVG guided by the upper bound U_{st}. If the SVG is exact, the A$*$ search applies to the tier 1 graph S_1, otherwise, the search applies to the entire saddle vertex graph $S_1 \bigcup S_2 \bigcup S_3$. During the bidirectional A$*$ search, we prune the search by allowing only the point p satisfying the inequality $d_s(p) + \|pt\| \leq U_{st}$, where $d_s(p)$ is the shortest distance from the point s to p and $\|pt\|$ is the Euclidean distance from p to t. The figure on the right shows an example of computing the single-source single-destination geodesic distance on the 263K-vertex Lucy model. The exact polyhedral distance is 0.8188. Given an approximate SVG with $K = 100$, the upper bound U obtained by bidirectional Dijkstra search is 0.8410 and our computed geodesic distance is 0.8191. The bidirectional Dijkstra search region, colored in blue, contains 89.5K vertices, and the A$*$ search region, colored in

red, contains only 17.3K vertices. Simply running the bidirectional Dijkstra search on the SVG takes 0.078s. However, running the BidirectionalDijkstra on the mesh and the A$*$ search on the SVG take only 0.028s and 0.021s, respectively. Therefore, the $A*$ search can improve the performance for the SSSD problem, especially when p and q are far away.

The geodesic path $\gamma(p,q)$ can also be obtained easily. Let $\{e_1, e_2, \cdots, e_k\}$ be the shortest path from p to q on the SVG. Then the geodesic path $\gamma(p,q) = \bigcup_i \text{Backtrace}(e_i)$. The time complexity to obtain $\gamma(p,q)$ is $O(k)$, where k is the number of triangles crossed by the path $\gamma(p,q)$.



FMM $\varepsilon = 0.78\%$ Heat $\varepsilon = 0.72\%$ $(8, 1.96\%)$ $(16, 0.61\%)$ $(30, 0.13\%)$ $(50, 0.08\%)$ $(100, 0.035\%)$ exact

Figure 4.6: Visual comparison of the accuracy. The 2-tuple under the SVG result is (K, ε), where ε is the mean relative error. $K = 50$ leads to high quality result, where the difference to the exact result is hardly visible.

4.6.3 Multiple-sources All-destinations (MSAD)

The single-source and multiple-sources geodesic algorithms have the same computational framework. If the SVG is exact, we first run the Dijkstra search to the tier 1 graph S_1 and then update the geodesic distance for each non-saddle vertex q by using the shortest distance from q's saddle neighbors. Since it takes $O(|E_{NS}|)$ time

to update the geodesic distance for the non-saddle vertices, the time complexity is $O(|E_{SS}|\log|V_S| + |E_{NS}|) = O(Drn\log(rn) + D(1-r)n)$. If the SVG is approximate, we run the Dijkstra search to the entire graph $S_1 \bigcup S_2 \bigcup S_3$, which can reach all mesh vertices. The time complexity is then $O((|E_{SS}| + |E_{NS}| + |E_{NN}|)\log|V|) = O(Dn\log n)$.

Algorithm 5 Computing the MSAD Geodesic Distance

Require: A triangle mesh $M = (V, E, F)$, its saddle vertex graph $S = S_1 \bigcup S_2 \bigcup S_3$ and the set of sources $\mathfrak{S} = \{s_i\}_{i=1}^{m}$.

Ensure: $\forall t \in V$, $d(t)$ is the geodesic distance from t to its closest source $s_i \in V$.

```
if S is exact then
    S'_1 ← S_1;
    for each s_i ∈ 𝔖 do
        if s_i ∉ V_S then
            S'_1 ← S'_1 ⋃ {s_i and its N-S edges};
        end if
    end for
    Dijkstra(S'_1, 𝔖);
    for each q ∈ V_N do
        d(q) = min_{t∈𝒮(q)} {d(t) + ‖{t, q}‖};
    end for
else
    Dijkstra(S_1 ⋃ S_2 ⋃ S_3, 𝔖);
end if
```

4.7 Experimental Results

Performance. Due to the computation of the exact geodesic paths, the SVG construction is expensive. Fortunately, the SVG can be constructed in a parallel manner. We implemented the SVG construction algorithm on CUDA 5.0 and run it on an Nvidia Tesla K20 graphics card with 2496 CUDA cores and 5GB memory to produce all the SVGs used in this paper. Take the 263K-vertex Lucy for example. It takes 3.68s, 12.84s, and 381.1s to construct the SVG with $K = 30$, 100, and 1000, respectively. The computed SVGs are stored in a binary file format, which will be used in the Dijkstra search.

We implemented the priority queue based Dijkstra algorithm and its two variants, bidirectional Dijkstra search and A$*$ search, and tested our algorithms, SSSD, SSAD, MSAD on an Intel Xeon 2.66GHz CPU machine. Only a single core was used to compute the various types of geodesic distances. Table 4.2 shows the mesh complexity and performance of our method and other approximate algorithms. Since all the other geodesic algorithms are CPU-based, we also show the CPU pre-computation time of our method to make a fair comparison. We have found that our CPU-based program is usually 10 to 40 times slower than the parallel program on the Tesla K20 GPU.

Our method solves the SSAD and MSAD geodesic distances in a unified framework. The MSAD algorithm allows us to compute the geodesic Voronoi diagram and the geodesic distances to curve sources. See Figure 4.15.

Table 4.2: Statistics of speed and accuracy. T_p^c (resp. T_p^g): time for pre-computing on the CPU (resp. GPU); T: time for solving the single-source geodesic distance; ε: mean relative error; S: memory required for storing SVG.

Model ($\|V\|$)	ICH	FMM		GTU ($m=3000$)			Heat Method ($t=h^2$)			SVG ($K=100$)					SVG ($K=1000$)				
	T (s)	T (s)	ε	T_p^c (s)	T (s)	ε	T_p^c (s)	T (s)	ε	S (Mb)	T_p^c (s)	T_p^g (s)	T (s)	ε	S (Mb)	T_p^c (s)	T_p^g (s)	T (s)	ε
Armadillo (173K)	10.45	7.57	1.34%	3878	7.28	0.27%	2.28	0.26	0.92%	313.3	168.1	8.82	0.29	0.047%	1905	4895	247.1	0.72	0.0011%
Bunny (72K)	6.33	2.45	0.66%	2315	2.03	0.26%	1.09	0.10	0.85%	124.4	68.0	3.77	0.10	0.041%	966.5	2041	105.5	0.22	0.0011%
Fertility (30K)	1.90	0.47	1.18%	684	0.63	0.11%	0.45	0.03	0.56%	54.4	29.1	1.69	0.04	0.041%	417.8	843.3	43.9	0.12	0.0012%
Gargoyle (350K)	80.1	46.1	1.57%	28743	16.83	0.29%	12.03	0.53	0.96%	633.2	340.1	18.20	0.44	0.048%	5282	9911	508.7	1.41	0.0009%
Lucy (263K)	18.2	15.8	1.78%	6872	11.69	0.23%	8.10	0.37	0.91%	477.8	259.4	12.84	0.39	0.056%	3957	7451	381.1	0.97	0.0016%
Dragon (3M)	194.3	155.1	1.87%	65599	128.1	0.31%	Out of memory			5057	2894	151.2	4.11	0.048%	Out of memory				

Accuracy. We compare our approximate geodesic distance to the exact polyhedral distance [97] [148] and measure the relative difference $|d(x,y) - \widetilde{d}(x,y)|/d(x,y)$, where $d(x,y)$ and $\widetilde{d}(x,y)$ are the exact and approximate distances, respectively. We report the maximal, root-mean-square error, and mean relative differences. As Figure 4.7 shows, increasing K makes an approximate SVG approaching the exact SVG. Figure 4.7(b) plots the mean relative error as function of K. Figure 4.7(c) shows the error vs K curves with respect to the mesh resolution and tessellation. We down-sample the 144K-face Bunny to 6 isotropic and anisotropic meshes. All the error vs K curves exhibit the same pattern with slightly different ranges. Figure 4.12 shows the Happy Buddha model from

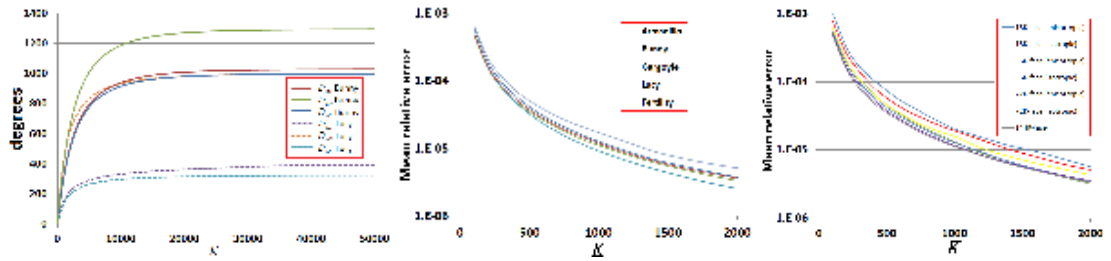
a low-resolution mesh with many obtuse triangles to a high-resolution mesh with regular triangulation. The same parameter $K = 30$ produces consistently high quality results for all three meshes. This again justifies that the parameter K is resolution and triangulation insensitive. Figure 4.6 shows the geodesic distances on the Bimba with various K. Figure 4.10 shows the results on the Bunny and the Armadillo. See more results in the Supplementary Material.

Following [29], we also examine the errors relative to the mean edge length h on the unit sphere, where the analytic geodesic distance is available. We observe that the heat method provides a linear convergence and the exact polyhedral distance converges quadratically. As shown in Figure 4.20, our method with $K = 500$ has the same quadratic convergence rate as the exact polyhedral distance for large mean edge length h, while the convergence speed becomes slower when h is small. This result is not a surprise. Our SVG with a large K does compute the *exact* polyhedral distance when the mesh resolution is low, since the SVG is exact and contains all direct geodesic paths. On the other hand, when the mesh resolution is high and the parameter K is relative small, the corresponding SVG becomes approximate since it contains only a subset of the direct geodesic paths and the resulting geodesic distance is approximate. An interesting observation shows that the approximate SVG converges slightly faster than the exact polyhedral distance when the mean edge length $h \in [0.03, 0.15]$. This is due to the fact that the approximate SVG always produces a *longer* distance than the exact polyhedral distance on arbitrary meshes. On the mesh representing the sphere, the exact polyhedral distance is always less than the analytic geodesic distance. Therefore, when the mesh resolution is in the aforementioned range, the distance produced by the SVG tends to be more accurate than the exact polyhedral distance.

The existing approximation algorithms usually work well for smooth surfaces. However, their accuracy could be very low if the models have rich features. In sharp contrast, our

method works remarkably well for real-world models with geometric features, since our method does take advantage of these details: the richer the geometric features, the stronger the local characteristic of the discrete geodesic is, the more accurate SVG we can construct (with a fixed K), and the *more* accurate geodesic we obtain. Our result on the 1.5M-face Dragon has mean relative error less than 0.04%. Comparing to the result of the exact algorithm, such a small error is not visually noticeable. See Figure 1.

Metric. It is well known that given an undirected connected graph G, the set V of vertices of G forms a metric space by defining $d(x,y)$ to be the length of the shortest path connecting the vertices x and y. Since we compute the geodesic distance by using the shortest path distance on the SVG, the resulting distance is guaranteed to be a metric, i.e., satisfying the symmetry condition and triangle inequality. All the other approximate algorithms, such as the FMM, the AMMP algorithm, the heat method and the GTU method, exhibit violations of metric properties.

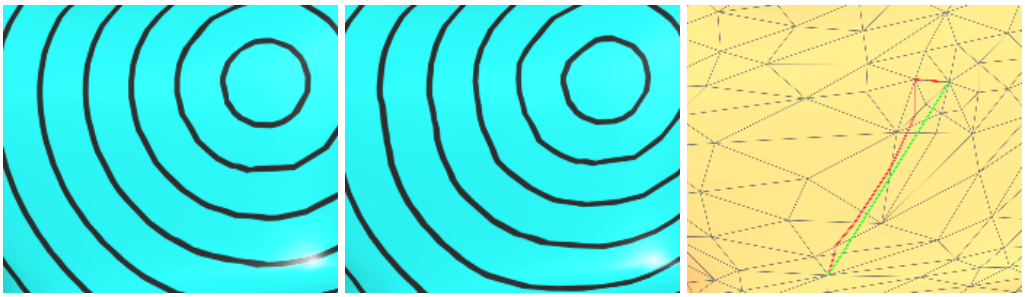


(a) Degrees vs K (b) Mean relative error vs K (c) Mean relative error vs K

Figure 4.7: The parameter K effectively controls the SVG complexity and the accuracy of the approximate geodesic distance. (a) A sufficiently large K produces the exact SVG. (b) The mean relative error as a function of K on various models. (c) The mean relative error as a function of K on Bunny of various resolutions and tessellations.

Convex or developable surfaces. Our method works remarkably well for real-world models which contain large number of saddle vertices. However, our method is not efficient for computing geodesic distances on convex polyhedrons and developable surfaces, which do not have saddle vertices at all. Therefore, every geodesic path is direct

and the associated corresponding SVG is a dense graph with $|E_{NN}| = \binom{n}{2}$ and $D = n$. It is very expensive to construct and store such a dense graph. Alternatively, one can seek other efficient techniques for these special cases. Shreiber and Sharir [118] presented an optimal time $O(n \log n)$ algorithm to compute the single source geodesic on convex polyhedra. For the developable surfaces, one can adopt the FMM with spherical wavefront propagation [100], which can compute the exact polyhedral distance in $O(n \log n)$ time too.



Our result Heat method Geodesic paths

Figure 4.8: The SVG method computes the geodesic path by unfolding a sequence of triangles, which is stable and accurate. The other approximate algorithms computes the geodesic path by tracing the gradient of the distance, which is sensitive to the triangulation. The geodesic paths obtained by our method and the heat method are colored in green and red, respectively.

Geodesic path. Like the other approximate algorithms, the heat method computes the geodesic path by tracing the gradient of the distance function. Note that the discrete gradient operator computes the exact gradient only if the underlying function is piecewise linear. However, the distance function is non-linear. Therefore, the gradient tracing usually leads to incorrect result when the triangulation is poor. Our method computes the geodesic path by triangle unfolding. Let $\gamma(s,t) \in M$ be the geodesic path on M and $\{p_0, p_1\}, \{p_1, p_2\}, \cdots, \{p_k, p_{k+1}\}$ be the corresponding shortest path on the SVG, where $p_0 = s$ and $p_{k+1} = t$. Then the procedure $\bigcup_{i=0}^{k} \text{Backtrace}(\{p_i, p_{i+1}\})$ can recover the geodesic path $\gamma(s,t)$ accurately. See Figure 4.8.

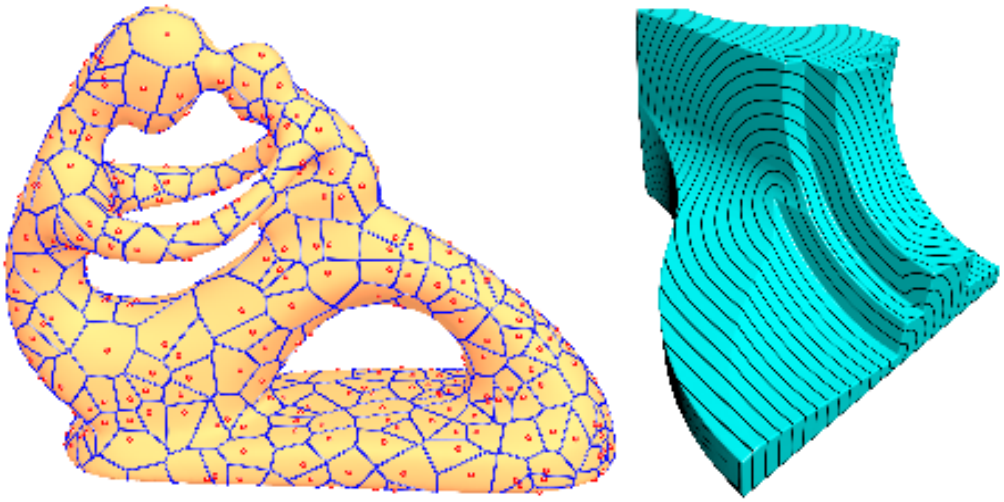
Scalability. It is worth noting that our method has advantages in terms of the scalability. Fixing the parameter K, both the time and space complexities of the pre-computing are linear to the number of vertices n. The direct geodesic paths are computed on the GPU, and the graph is formed on the CPU. Our GPU-based program on Tesla K20 can process a mesh with up to fifty (resp. twelve) million triangles when $K = 30$ (resp. 100). If the entire pre-computation is done on the CPU, our program can process even larger meshes, since the CPU RAM is usually much bigger than the GPU RAM. Note that the heat method prefactors the Laplacian matrix, which has non-linear time and space complexities. Given a PC with 12GB CPU memory, the existing open-source Cholesky factorization package (e.g.,Cholmod [30]) can process meshes with up to five million triangles.

Domain. The heat method takes advantages of the well-established discrete Laplacian, and can easily adapt to a variety of geometric domains, including high-degree nonplanar polygonal meshes, and point clouds. Our method is limited to the triangle meshes.

4.8 Summary

The proposed SVG method is highly efficient, accurate, numerically stable and robust to the mesh resolution and tessellation. Unlike the other approximate algorithms that usually produce less accurate results for models with rich geometric features, our method works remarkably well for such models. The user can intuitively control the accuracy as well as the SVG complexity by the parameter K, which specifies the maximal number of points in a geodesic disk. The parameter K is independent to the scale, model, mesh resolution and tessellation. Moreover, our method guarantees the computed distance is a metric. Experiments on a wide range of real-world models demonstrate that our method

significantly outperforms the existing approximate algorithms in terms of accuracy and speed.

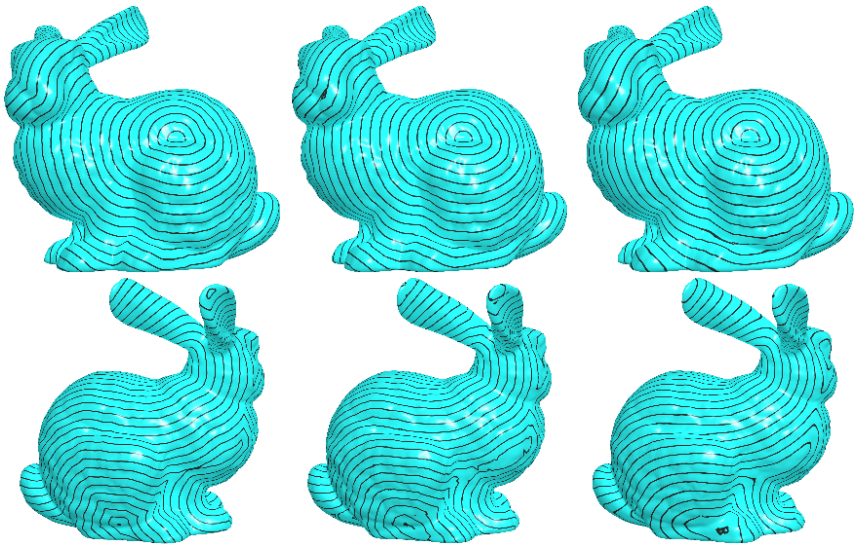


(a) Geodesic Voronoi diagram (b) Curve sources

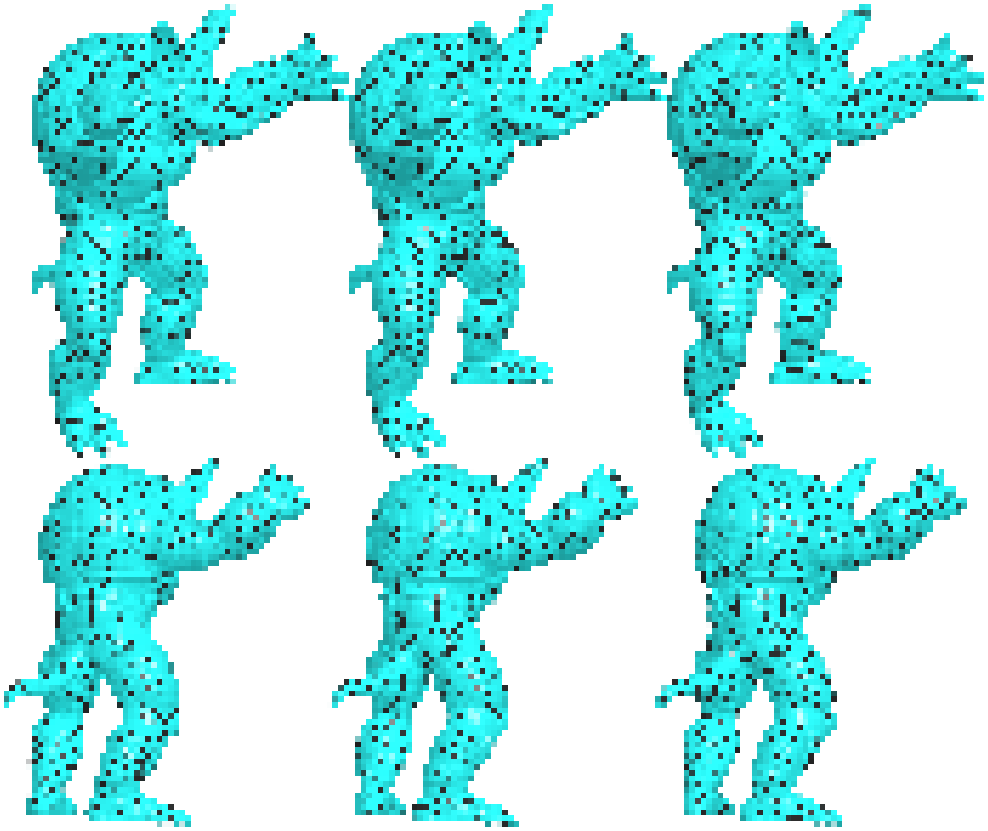
Figure 4.9: The MSAD algorithm allows us to compute the geodesic Voronoi diagram and the geodesic distances to curve sources. (a) 500 random selected vertices are used as the seeds for the Voronoi diagrams. (b) The vertices on the curves are used as the sources.

4.9 Introduction

Centroidal Voronoi tessellation (CVT) is a special type of Voronoi diagram (VD) such that the generating point of each Voronoi cell is also its center of mass [39]. The CVT has broad applications in computer graphics, such as meshing, stippling, sampling, etc. Although the CVT in Euclidean space has been extensively studied, relatively little progress has been reported towards computing the CVT on curved surfaces. A key step in computing the CVT is to construct the Voronoi diagrams in each iteration. It is fairly simple to construct the Voronoi diagrams in Euclidean space (e.g. $\mathbb{R}^2$ and $\mathbb{R}^3$), since many efficient algorithms and software tools are readily available. However, it is technically challenging to compute VD on curved surfaces. Some researchers tackle



SVG $K = 30$, 0.045s GTU $m = 3000$, 2.03s Heat $t = h^2$, 0.10s
$(0.253\%, 0.17\%, 0.12\%)$ $(0.97\%, 0.45\%, 0.26\%)$ $(1.84\%, 0.93\%, 0.83\%)$



SVG $K = 100$, 0.29s GTU $m = 3000$, 7.28s Heat $t = h^2$, 0.26s
$(0.13\%, 0.055\%, 0.047\%)$ $(0.59\%, 0.44\%, 0.27\%)$ $(1.96\%, 1.05\%, 0.92\%)$

Figure 4.10: Experimental results. The 3-tuple shows max relative error, root-mean-square relative error and mean relative error. The exact results are not shown here since our results are visually identical to the exact results.

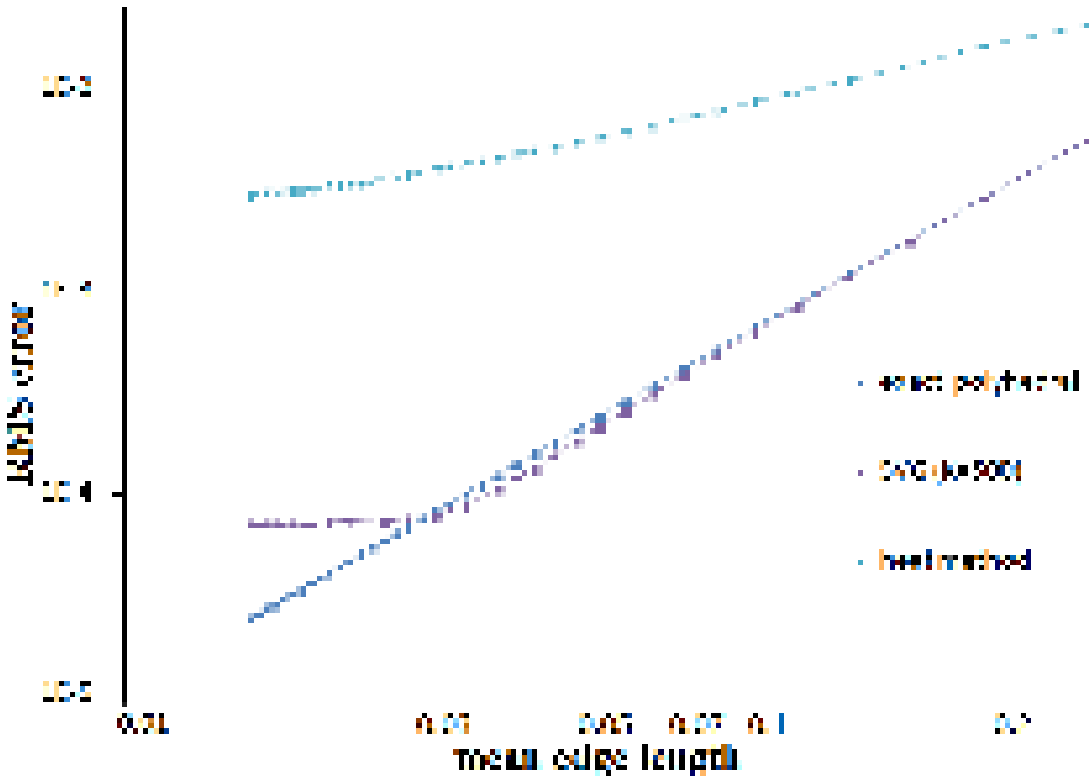
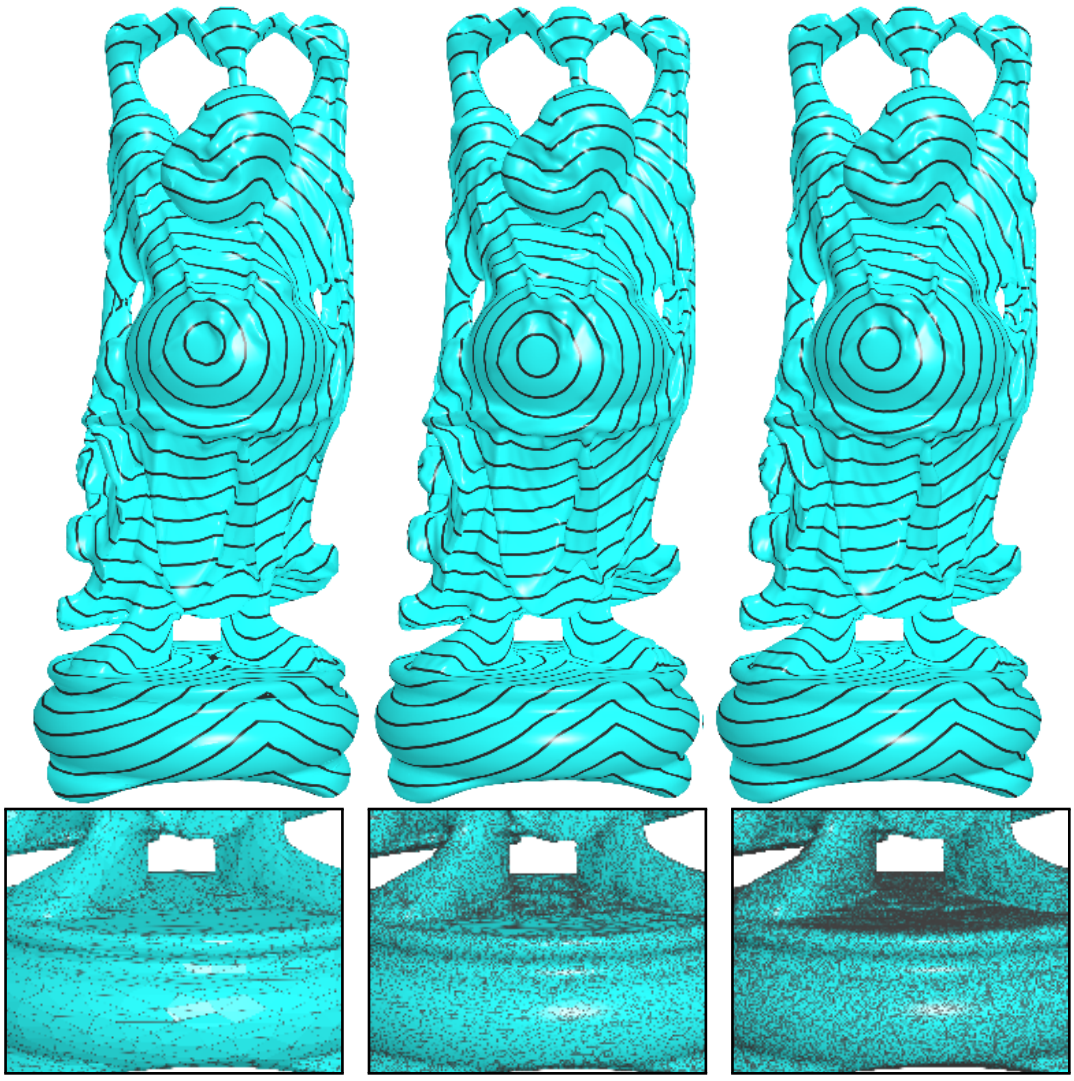


Figure 4.11: Convergence of distance functions on the unit sphere. The vertical and horizontal axes show the logarithm of the root-mean-square error and the mean edge length respectively. The exact polyhedral distance converges quadratically while the heat method converges linearly. Our method has the same convergence rate as the exact algorithm when the mesh resolution is low and K is big. The convergence rate becomes linear when the mesh resolution is sufficiently high.



(0.37%, 0.15%, 0.11%) (0.35%, 0.13%, 0.11%) (0.29%, 0.13%, 0.11%)

Figure 4.12: The SVG method is numerically stable and the approximated geodesic distances are insensitive to the mesh tessellation and resolution. From left to right: 40K-face, 300K-face and 600K-face. The same parameter $K = 30$ applying to all three cases produces consistently high quality results.

this challenge by computing the restricted Voronoi diagrams [152], which is the intersection between the input mesh and the Voronoi diagram in $\mathbb{R}^3$. These approaches are embedding space dependent and may fail on models with complicated geometry and/or topology. Others adopt the global parametrization to map the surface to the parametric domain, such as Euclidean plane $\mathbb{E}^2$, the sphere $\mathbb{S}^2$, or hyperbolic disk $\mathbb{H}^2$, in which the 2-dimensional CVT is computed [127]. It is known that parameterizing models with complicated geometry and/or topology is computationally expensive and often suffers serious numerical issues. To our knowledge, there is no method for computing the CVT on *arbitrary* surfaces.

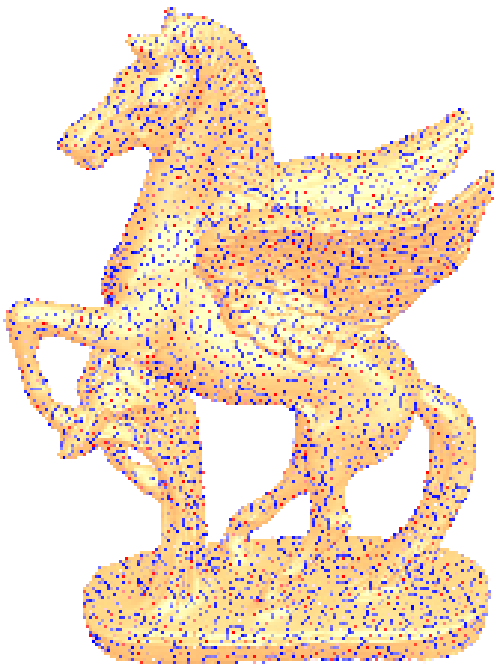


Figure 4.13: Our intrinsic method can compute a high-quality centroidal Voronoi tessellation on model with complicated geometry and topology. The CVT on the Pegaso model was created by 3000 sites.

To tackle the above-mentioned challenge, this chapter presents two *intrinsic* algorithms for computing the centroidal Voronoi tessellation on arbitrary triangle meshes. Our first algorithm adopts the Lloyd framework, which iteratively moves the generator of each

geodesic Voronoi diagram to its mass center. Based on the discrete exponential map, our method can efficiently compute the Riemannian center and the center of mass for any geodesic VD. Our second algorithm uses the L-BFGS method (limited-memory BFGS), which uses the CVT energy function and its gradients to approximate the Hessian matrix. The L-BFGS method has better performance due to its super-linear convergence rate. Thanks to the intrinsic feature, our methods are independent of the embedding space, and work well for models with arbitrary topology and complicated geometry, where the existing extrinsic approaches often fail. Figure 5.1 shows our result on the genus-5 Pegaso model. Moreover, our methods are insensitive to the mesh resolution and tessellation, and can be applied to surfaces embedded in arbitrary dimensional space. The promising experimental results demonstrate the efficacy of our methods.

The rest of the chapter is organized as follows. Then Section 4.10 and section 4.11 presents our intrinsic Lloyd and L-BFGS CVT algorithms in details. Section 6.4 shows the experimental results, compares our method to the existing techniques and discusses its advantages and limitations. Finally, Section 6.5 concludes the chapter.

4.10 The Lloyd Framework

4.10.1 Overview

Let $M = (V, E, F)$ be a triangle mesh representing a 2-manifold surface, where V, E and F are the set of vertices, edges and faces, respectively. Let $S = \{s_i | s_i \in M, i = 1, \cdots, m\}$ denote the set of sites on M.

Our algorithm adopts the Lloyd framework, which iteratively computes the geodesic CVT on meshes. For each iteration, we first compute the multiple-source-all-destination geodesic distance with the sites s_i, $i = 1, \cdots, m$, as the sources. This geodesic distance

Algorithm 6 Intrinsic computation of centroidal Voronoi tessellation on meshes

Require: A triangle mesh $M = (V, E, F)$, the set of sites $S = \{s_i | s_i \in M, i = 1, \cdots, m\}$ and the convergence threshold ε;

Ensure: The centroidal Voronoi tessellation on M;

```
do
    Compute geodesic distance field with {s_i}_{i=1}^m as sources;
    Form the geodesic Voronoi diagram;
    for i = 1 to m do
        Compute the Riemannian center r_i for Voronoi cell Ω_i;
        Compute the center of mass c_i for Ω_i;
        d_i ← d(s_i, c_i);
        s_i ← c_i;
    end for
while (Σ_{i=1}^m d_i)/m > ε
```

field on M induces a geodesic Voronoi diagram. Then for each geodesic Voronoi cell, say $\Omega_j \in M$, we compute its Riemannian center r_j, which is defined as the average of its corners. Next, we compute the exponential map $\exp(r_j)$ at the Riemannian center r_j, and map the Voronoi cell Ω_j to the tangent plane T_{r_j}, on which we can compute the center of mass c_j. Finally, we map the mass center from the tangent plane to the mesh using the exponential map $\exp(r_j)$. We update each site s_i to the new mass center and then repeat the above-mentioned procedure until the offsets of the sites are below the user-specified threshold.

4.10.2 Computing the Geodesic Voronoi Diagram

Taking $\{s_i\}_{i=1}^m$ as sources, we apply the ICH algorithm [148][2] to compute the multiple-source geodesic distance field. As a result, each mesh vertex is assigned a geodesic distance to its *closest* source. Then we label an edge *LE* if its two endpoints have different sources. Clearly, a *LE* edge is passed by a bisector. We further collect into

[2]The ICH algorithm in [148] computes the single-source geodesic distance. But it can be extended to multi-source case easily by adding a stoping criteria during the window propagation procedure: when two windows from different sources cover the same vertex, both windows stop propagation.

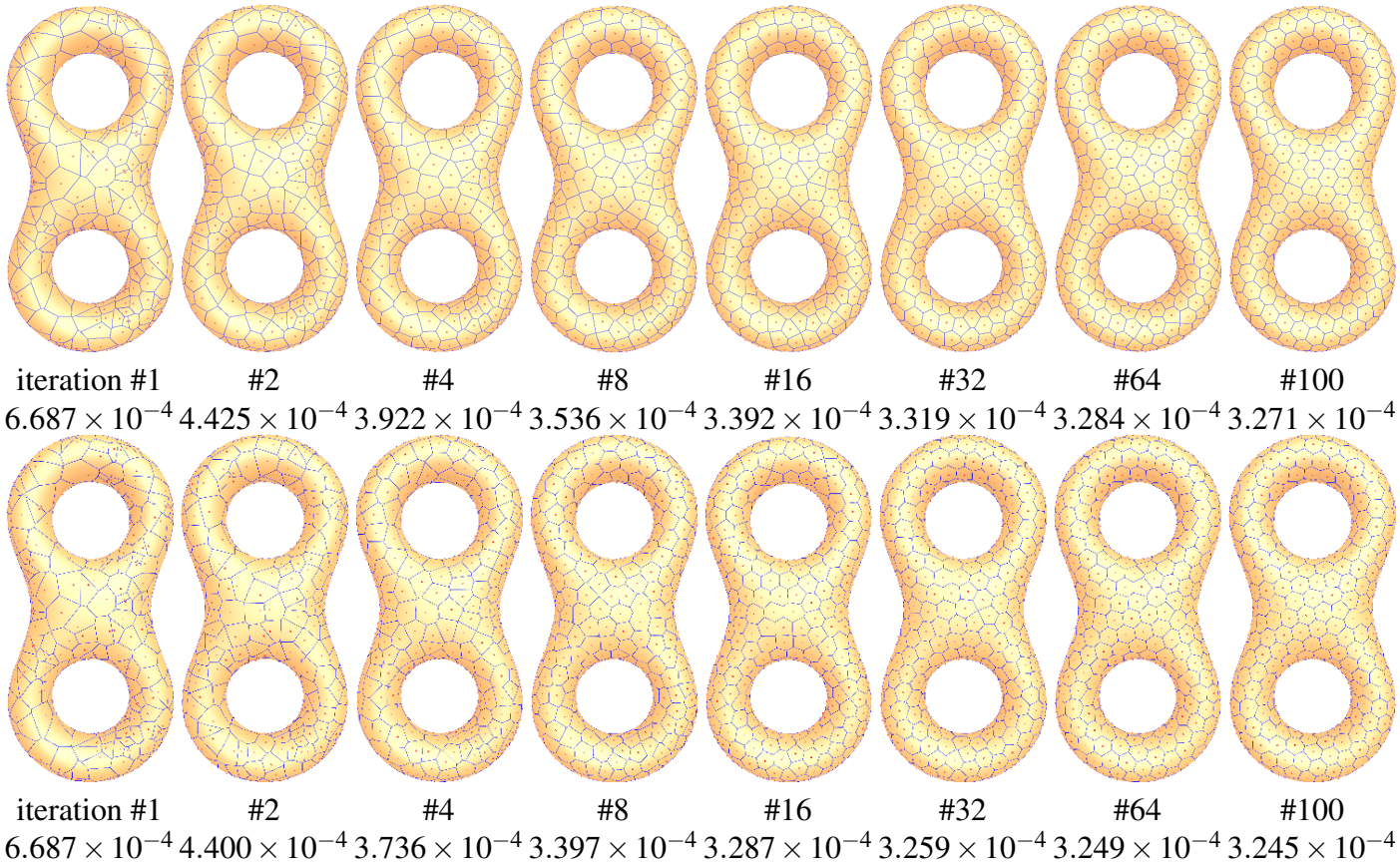
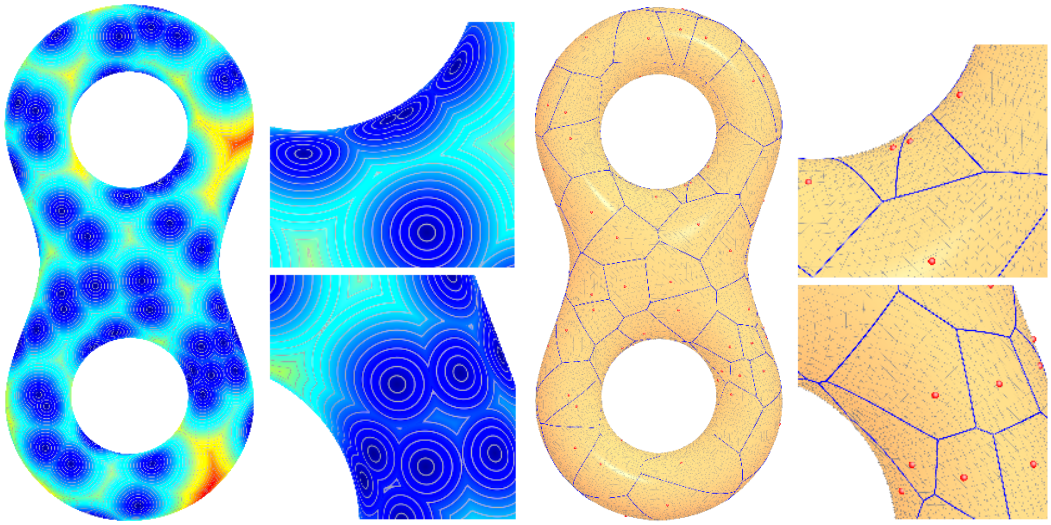


Figure 4.14: Iteratively computing geodesic CVT on meshes. Row 1: the Lloyd algorithm; Row 2: the L-BFGS algorithm.



(a) Geodesic distance field (b) Geodesic Voronoi diagram

Figure 4.15: The multiple-source geodesic distance field induces a geodesic Voronoi diagram. The cold (resp. warm) color in (a) indicates the small (resp. large) geodesic distance.

a list LT all the triangles in M that are incident to any LE edge. As [82] shows, if a triangle $t \in LT$ has all its three edges labelled LE, then t contains a branch point in the geodesic Voronoi diagram; otherwise t is passed through by a single piece of a bisector. Based on the lists of LE and LT, we run the marching algorithm [82] for extracting the geodesic Voronoi diagram on M.

Assume the sites s_i are uniformly distributed. This assumption is reasonable, since the distribution of the sites is improved after only a few Lloyd iterations (see Figure 4.14). The ICH algorithm takes worst-case $O(\frac{n^2}{m}\log(\frac{n}{m}))$ time and empirical $O(\frac{n^2}{m})$ time, where n is the number of mesh vertices. The geodesic Voronoi diagram is then built in $O(k \log k)$ time, where k be the number of triangles in LT. Figure 4.15 shows the geodesic distance field and its induced geodesic Voronoi diagram on the double-torus model.

4.10.3 Computing the Riemannian Center

Let v_1, v_2, $\cdots$, v_k be the corners of a Voronoi cell $\Omega_i \in M$. The Riemannian center is defined as the local minima x of the following function

$$U(x) = \sum_{i=1}^{k} d^2(x, v_i), \tag{4.1}$$

where $d(p,q)$ is the geodesic distance between p and q. If M has zero Gaussian curvature (that is, developable), the Riemannian center exists and is unique. However, in general, the function $U(x)$ is not convex, and the minimizer may not be unique. Kendall [70] and Karcher [69] showed the conditions to ensure the existence and uniqueness of the Riemannian center of mass. Intuitively speaking, if the points v_i are not too far from each other, there exists a *unique* Riemannian center of mass. Refer to [105][113] for the rigorous results.

Let $x^* \in M$ be the local minimal of Equation (4.1). Then x^* satisfies

$$\vec{0} = \nabla U(x^*) = \sum_{i=1}^{k} \nabla d^2(x^*, v_i) = 2 \sum_{i=1}^{k} d(x^*, v_i) \nabla d(x^*, v_i). \tag{4.2}$$

Since $d(,)$ is the geodesic distance, $\nabla d(x^*, v_i) \in T_{x^*}M$ is a unit tangent vector. Therefore, $d(x^*, v_i)\nabla d(x^*, v_i)$ represents a tangent vector with length $d(x^*, v_i)$, denoted by $\vec{t}_i$. Equation (4.2) requires $\sum_{i=1}^{k} \vec{t}_i = \vec{0}$, which means x^* is the center of the terminal points of $\vec{t}_i$.

We iteratively compute the local minimal x^*. Let x be the initial point, which could be either one of the corner points v_i or the site's location s_i. We compute the exponential map $\exp_x$ at x. The exponential map $\exp_x : T_xM \to M$ builds a geodesic polar coordinate system at x. The inverse map $\exp_x^{-1}$ maps the point $v_i \in M$ to the tangent plane T_xM. Let $\hat{x} \in T_xM$ be the average of the points $\exp_x^{-1}(v_1)$, $\cdots$, $\exp_x^{-1}(v_k)$. If $\hat{x}$ does not equal x, we send $\hat{x}$ to the mesh M by the exponential map $\exp_x(\hat{x})$. Setting $x = \exp_x(\hat{x})$, we then

Algorithm 7 Computing the Riemannian center

Require: A set of points v_1, v_2, $\cdots$, v_k on M; the convergence threshold δ;
Ensure: The Riemannian center x;

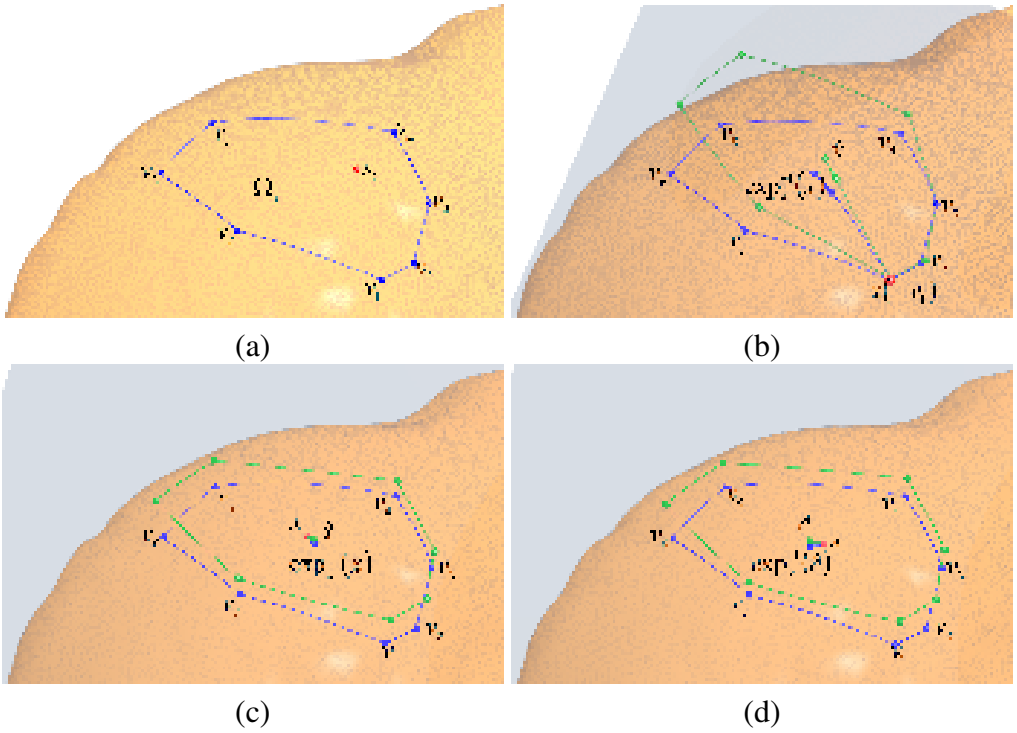
```
x ← v_1;
do
    x_0 ← x;
    Compute the exponential map exp_x at x
    The inverse map exp_x^{-1} brings the points v_i, i = 1, ··· ,k, back to the tangent plane T_xM;
    x̂ ← (Σ_{i=1}^{k} exp_x^{-1}(v_i)) / k;
    x ← exp_x(x̂);
while d(x, x_0) > δ
```

repeat the above procedures until the average $\hat{x}$ agrees with x. Note that the exponential map, in general, does not preserve the area. However, when the Voronoi cells are small with respect to the injectivity radius[105], we observe that our algorithm can generate fairly good results. In our implementation, we set the initial point $x = s_i$, which is the center of the Voronoi region in the previous iteration. During the CVT iterations, the sites are getting closer to the Riemannian center, making finding the Riemannian center faster. We have observed that the iterative algorithm for finding Riemannian center converges very fast, took only two or three steps for all test models in our paper.

4.10.4 Computing the Center of Mass

Although the Riemannian center r is not the center of mass, it is *close* to all corners of the Voronoi cell. Therefore, it is very natural to use r to compute the center of mass. Let $\exp_r$ be the exponential map at the Riemannian center and $\hat{v}_i = \exp_r^{-1}(v_i)$ the pre-image of v_i, which lies on the tangent plane T_rM. Since the points $\hat{v}_i$, $i = 1, \cdots, k$, form a polygon on the tangent plane, its center of mass $\hat{c} \in T_rM$ is given by

$$x = \frac{1}{6A} \sum_{i=1}^{k-1} (x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i)$$



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

Figure 4.16: Computing the center of mass for the Voronoi cell Ω_i. It takes two iterations (b) and (c) to obtain the Riemannian center c. The blue dot $\exp_x^{-1}(\hat{c})$ in (d) is the center of mass.

$$y = \frac{1}{6A}\sum_{i=1}^{k-1}(y_i + y_{i+1})(x_i y_{i+1} - x_{i+1} y_i)$$

where (x,y) are the coordinates of $\hat{c}$, (x_i, y_i) are the coordinates of $\hat{v}_i$, and A is the area of the polygon

$$A = \frac{1}{2}\sum_{i=1}^{k-1}(x_i y_{i+1} - x_{i+1} y_i)$$

Finally, the center of mass for Voronoi cell Ω is given by $c = \exp_r(\hat{c})$.

4.11 The L-BFGS Framework

Liu et al. [83] proved that the CVT energy function F has C^2 smoothness, thus, one can use the Newton or quasi-Newton method to optimize the energy F. In this Section, we adopt the L-BFGS method to accelerate the CVT computation. To compute the numerical integration on meshes, we modify the ICH algorithm [148] for computing the geodesic distance between any point (not necessarily a vertex) to the source point. We use [54] to compute numerical integration on each triangle. The detail of the modified ICH algorithm is in Section 4.12.1. The L-BFGS method requires the gradient of the energy function for approximating the approximated Hessian matrix. Given the energy function F, the gradient of the CVT energy is [65][39]:

$$\frac{\partial F}{\partial x_i} = 2m_i(x_i - c_i),$$

where $m_i = \int_{\Omega_i} \rho(x) d\sigma$, c_i is the center of mass of the Voronoi cell Ω_i. We use the methods in Section 4.10.3 and Section 4.10.4 to compute c_i. Note that the seeds are restricted on the input mesh M, and the gradients are also constrained on the tangent space T_x. Using the exponential map $exp_x : T_x M \to M$, we can compute the projection $\hat{c}_i$ of c_i on T_x. During the L-BFGS optimization process, we use $2m_i(x_i - \hat{c}_i)$ as the gradient, so that it is on the tangent plane at point x_i. During each iteration in L-BFGS

method, we get $\hat{x_i'}$ on T_x for each Voronoi cell, and use the inverse map exp_x^{-1} to get $x_i' = exp_x^{-1}(\hat{x_i'})$. Figure 4.17 shows the energy plot comparison between Lloyd method and L-BFGS method.

Algorithm 8 The L-BFGS algorithm for intrinsic CVT

Require: A triangle mesh $M = (V, E, F)$, the set of sites $S = \{s_i | s_i \in M, i = 1, \cdots, m\}$ and the convergence threshold ε;

Ensure: The centroidal Voronoi tessellation on M;

do
  Compute geodesic distance field with $\{s_i\}_{i=1}^m$ as sources;
  Form the geodesic Voronoi diagram;
  for $i = 1$ to m **do**
    Compute the Riemannian center r_i for Voronoi cell Ω_i;
    Compute the center of mass $\hat{c}_i$ on tangent plane T_i;
    Compute energy $F_i(x)$ and gradient $\frac{\partial F}{\partial x_i}$ for Ω_i;
  end for
  Using L-BFGS method to compute all seeds $\hat{s_i}$ on their tangent plane T_i ;
  Compute all updated seeds s_i' on Ω_i using exp map;
while $\|\nabla F(x)\| > \varepsilon$

4.12 Experimental Results

4.12.1 Implementation

The ICH algorithm [148] has linear space complexity and can compute the exact single-source geodesic distance in an $O(n^2 \log n)$ time (The empirical time complexity is $O(n^{1.5} \log n)$), where n is the number of vertices. The ICH algorithm computes the exact geodesic distances between any mesh vertex to the source vertex. However, it cannot compute the exact geodesic distance between any mesh points, i.e., non-vertex points on the mesh. We modify the ICH algorithm by sacrificing its space complexity: we store all the windows (a data structure that carries the geodesic distance from an edge interval to the source) generated in the window propagation procedure. When computing the distance from the source to a non-vertex point, say $p \in t$, which is inside a triangle t, we consider

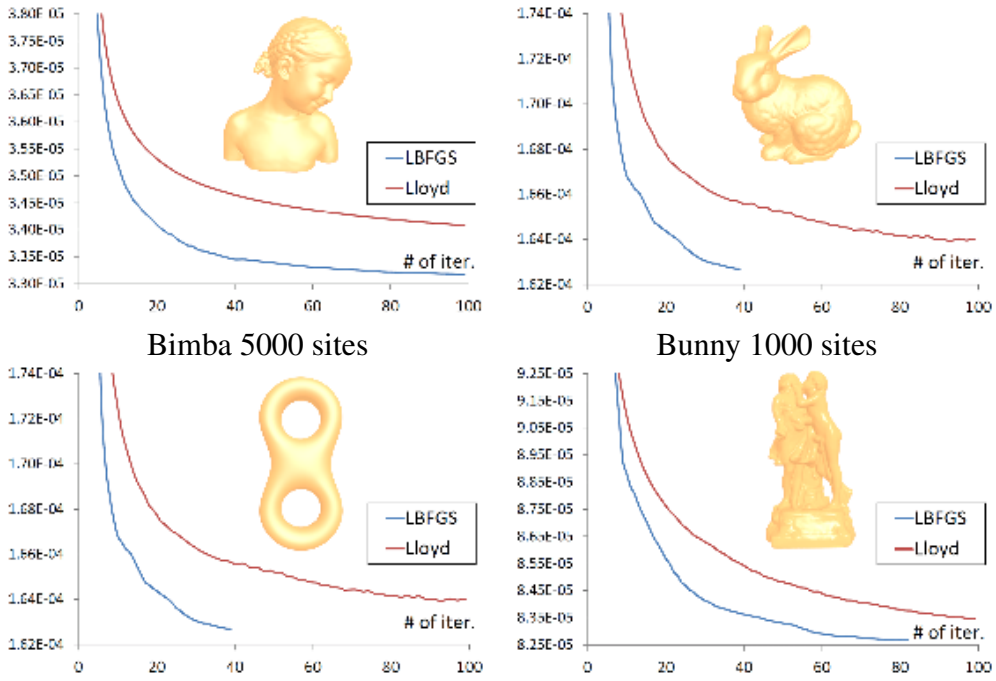


Figure 4.17: Convergence rate comparison between the Lloyd methd and the L-BFGS method.

all windows covering t's sides, and find the one which can provide the shortest distance to p.

The modified ICH algorithm can also compute the discrete exponential map on triangle meshes. Thanks to the parallel structure of the Lloyd iteration, our algorithm can be easily implemented in parallel. Each ICH thread takes a point (not necessarily a mesh vertex) as the source, the ICH algorithm partitions each mesh edge into a set of intervals, called windows, which encode both the geodesic distance and the direction of the geodesic path emanating from the source. The windows are maintained in a priority queue according to the distance from the source and are propagated across the mesh faces: pops a window from the queue and then computes its children windows which can add, modify, or remove existing windows, and updates the queue accordingly. When a window reaches a vertex v, it updates v's distance and direction, which are used for

the polar coordinates. The ICH algorithm terminates if the wavefront has reached the user-specified radius.

The input mesh is encoded in the half-edge structure and stored in the CPU's global memory in a read-only manner. Each CPU thread maintains its own data (i.e., the source point, the wavefront windows and the priority queue) in its own memory pool. Even though two or more ICH threads may compute on overlapped regions, they do not have any data and control conflicts, so each thread can proceed independently.

4.12.2 Results & Comparison

We adopted OpenMP to implement our method on an Intel 2.50 GHz CPU with four cores. Our program asks the user to specify the number of sites, then it generates the sites on the mesh randomly. We set the convergence threshold $\varepsilon = 10^{-6}$ in our implementation. Table 5.3.2 lists the model complexity and the performance of our algorithm and Figure 4.18 shows the computed CVT on some 3D models. Figure 4.19 shows CVT on high genus models.

To evaluating the quality of our results, we compute the Delaunay triangulation, which is the dual graph of CVT. Then we adopt the following measures [51] [152]:

- Triangle quality: Let $Q(t) = 6S_t/(\sqrt{3}p_t h_t)$ be the quality of a triangle t, where p_t, S_t and h_t are the inradius, area, and the length of the longest edge of t, respectively. Let Q_{min} (resp. Q_{avg}) be the minimal (resp. average) quality measure. The closer the value to 1.0, the more isotropic of the Delaunay triangulation, therefore, the higher quality of the CVT one obtains.

- Minimal angle: Let θ_{min} be the minimal of the smallest angle of all triangles and θ_{avg} the average of minimal angles of all triangles. The closer the values of θ_{min} and θ_{avg} to 60 degrees, the more isotropic of the triangulation one obtains.

Figure 4.20 shows the quality improvement via the Lloyd iteration.

Compared to the parameterization-based methods [5], [111] [112], our method avoids the inaccuracy due to the approximation and metric distortion in parameterization. Furthermore, our method can apply to models of arbitrary geometry and topology, for which the parameterization is not easy to obtain. As Figure 4.22 shows, our method outperforms the UCS method [111] and the RVD method [152] in terms of quality (higher angle measure Q_{ave} and lower number of singularities).

The restricted Voronoi diagram methods [152] [154] approximate the CVT on surface by computing the intersection of a 3D CVT and the input mesh. Although it works fairly well for models with simple geometry, this approximation is extrinsic, that is, embedding space dependent. Figure 4.21 shows a Coil Spring model, where the coil almost touches itself and leaves very small gap. The RVD method cannot distinguish the geometrically-close-but-topologically-far pieces, and produces the wrong result. Our method is completely intrinsic in that all the computations are based on the metric only. So it can clearly distinguish these geometric "ambiguity". To further demonstrate the efficacy of our intrinsic method, we apply it to the Lion model in various poses. As Figure 4.23 shows, the computed CVTs are consistently among the near-isometric poses. Figure 4.24 shows the CVTs with very few sites. Since each Voronoi cell is big, we can clearly see the difference between the extrinsic RVD and our intrinsic CVT.

4.13 Conclusion

This paper presents an intrinsic algorithm for computing centroidal Voronoi tessellation on arbitrary triangle meshes. Our algorithm adopts the Lloyd framework, which iteratively moves the generator of each geodesic Voronoi diagram to its mass center.

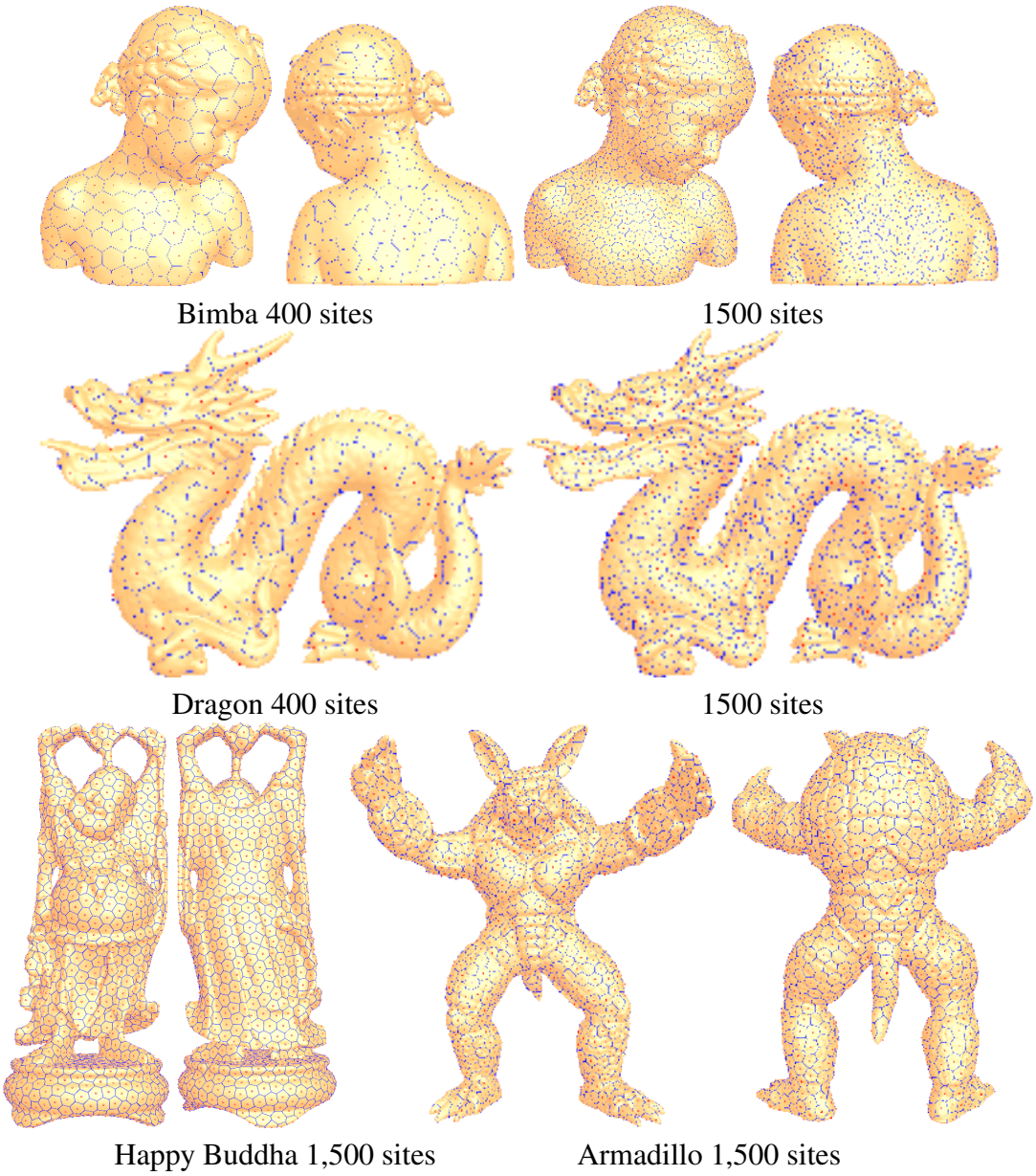
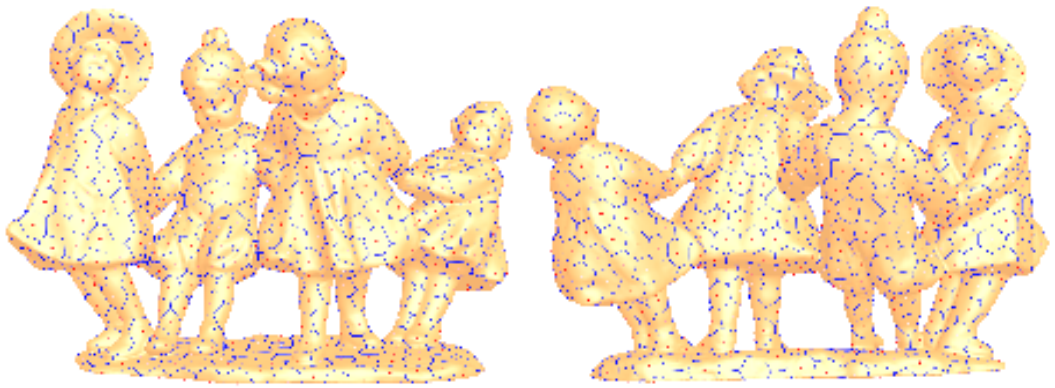
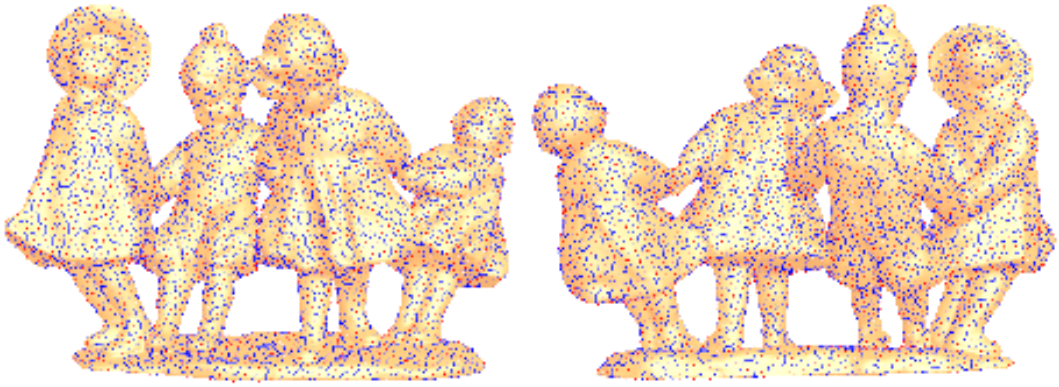


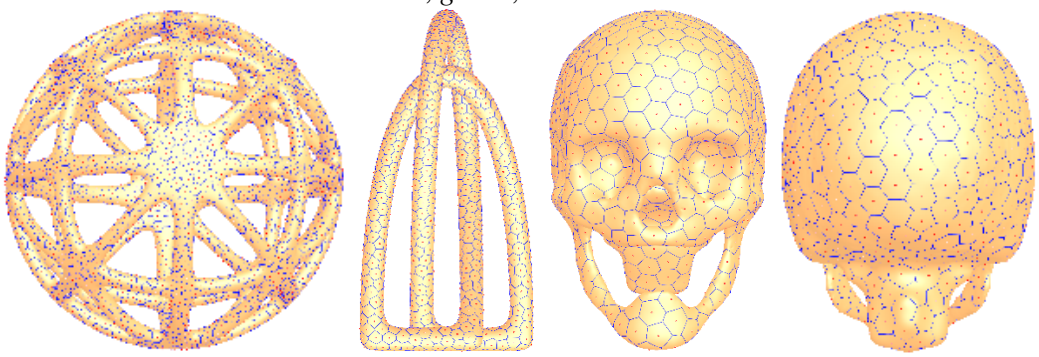
Figure 4.18: Experimental results. Images are rendered in high resolution that allows close-up examination.



1,000 sites

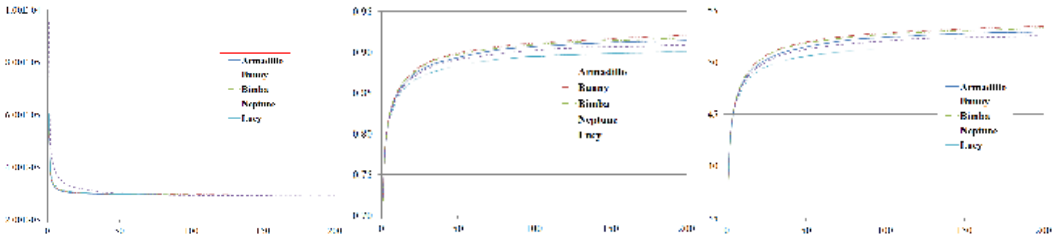


4-kid, $g = 8$, 5000 sites



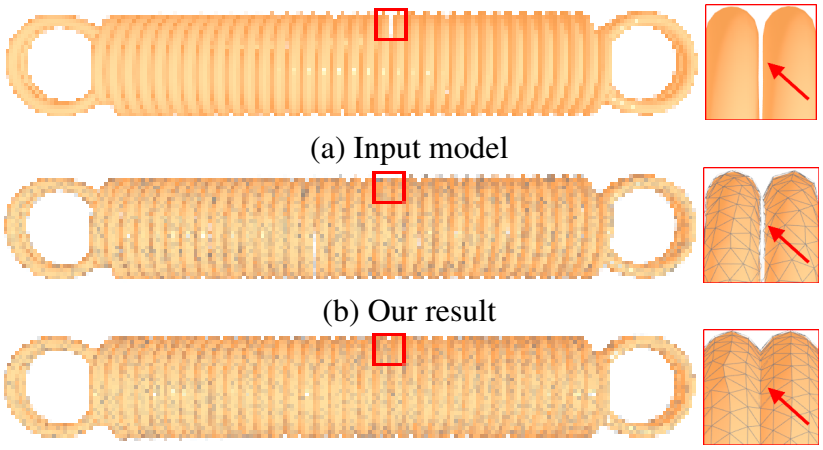
$g = 47$, 4000 sites Knot, $g = 2$, 1000 sites Skull, $g = 2$, 1000 sites

Figure 4.19: Experimental results on high-genus models



(a) CVT energy function (b) Triangle quality Q_{avg} (c) Minimal angle θ_{avg}

Figure 4.20: Energy function and quality measures. The horizontal axis in the plots shows the iteration number. The vertical axis in (a) is the *normalized* CVT energy function, that is, $\frac{F(\mathbf{S})}{A^2}$, where A is the area of the model.



(a) Input model

(b) Our result

(c) D.-M. Yan et al.'s result[152]

Figure 4.21: Intrinsic vs. extrinsic. Consider the Coil Spring model, where the pitch of the helix equals the diameter of the coil. Therefore, the coil almost touches itself and leaves very small gap. See the closeup view. As an intrinsic method, our method is independent of the embedding space and it can correctly separate the coil. The extrinsic method [152] computes the CVT by intersecting a 3D CVT with the model, which cannot distinguish the two geometrically-close-but-topologically-separate pieces. The Delaunay triangulations, the dual of the computed CVTs, are shown in this figure.

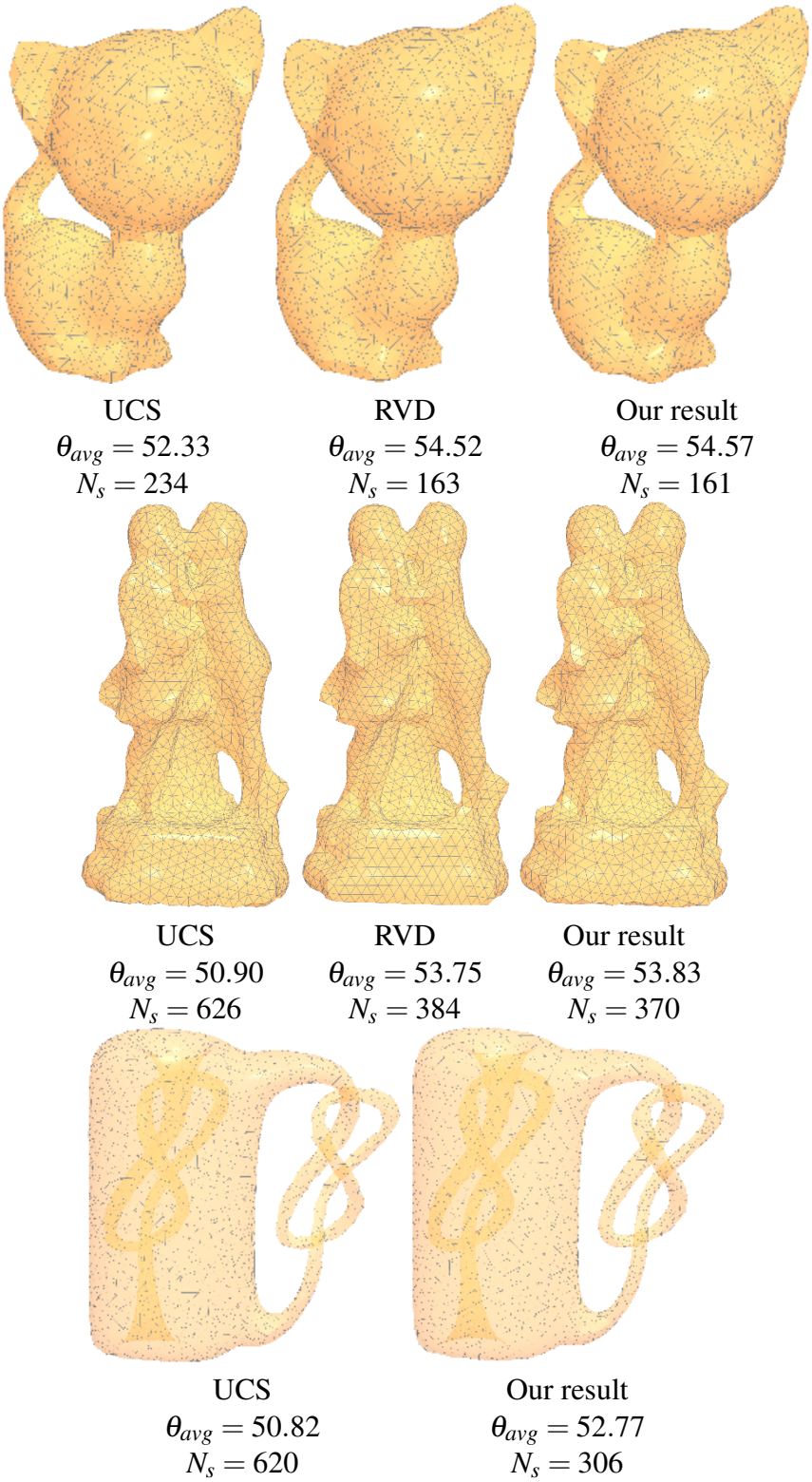


Figure 4.22: Comparison to the RVD method [154] and the UCS method [111]. N_s denotes the number of singularities (i.e., vertices whose valence is not six).

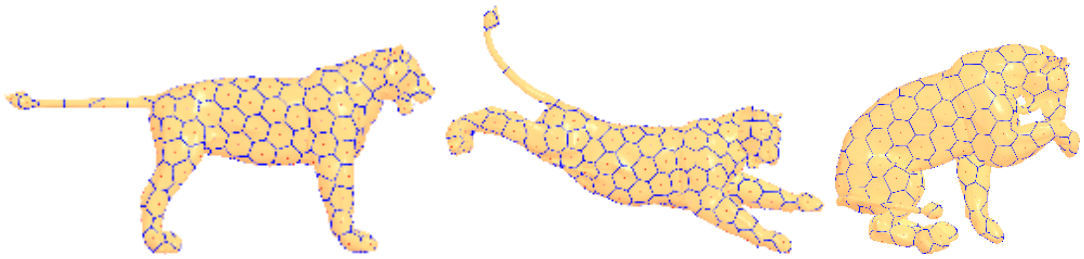
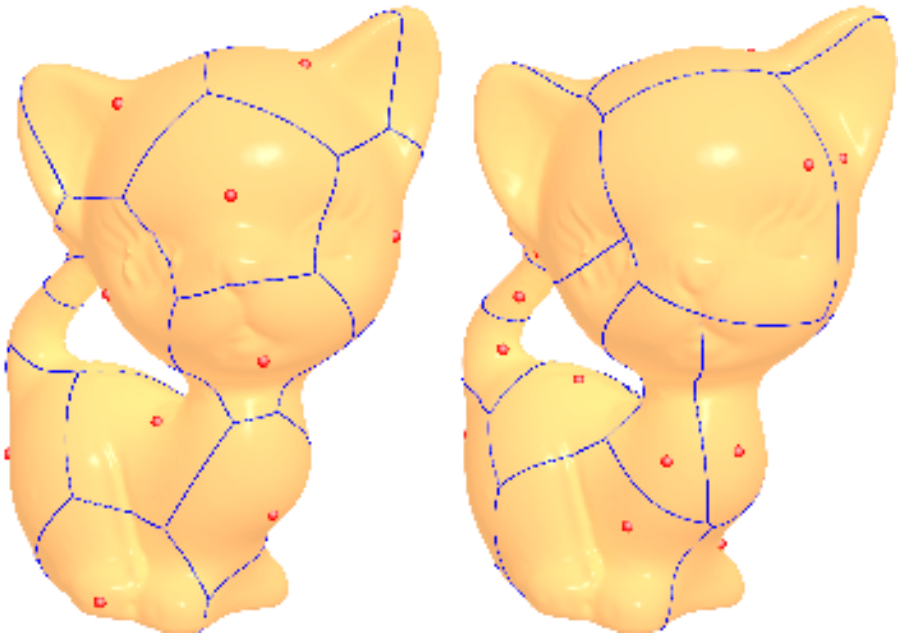


Figure 4.23: Thanks to its intrinsic property, our method can produce consistent results on the various poses of the Lion model.



Kitten, 20 sites

Our result RVD [152]

Figure 4.24: The intrinsic CVT and extrinsic RVD with very few sites.

Model	g	$\|V\|$	m	T	Q_{min}	Q_{avg}	θ_{min}	θ_{avg}
Armadillo	0	172,974	1,500	2.12	0.555	0.914	23.5	52.9
Bimba	0	74,764	1,500	0.91	0.639	0.926	35.4	53.9
Happy Buddha	6	488,217	1,500	8.27	0.456	0.901	21.5	51.4
Bunny	0	72,020	5,000	0.63	0.665	0.918	32.2	53.4
Double-torus	2	12,286	500	0.076	0.651	0.935	29.8	54.6
Dragon	0	422,558	1,500	6.18	0.445	0.901	21.9	51.9
Knotty-bottle	2	96,830	2,000	1.44	0.401	0.913	25.3	52.8
Pegaso	5	333,727	3,000	3.61	0.401	0.913	23.4	52.7
Sculpture	3	199,837	2,000	1.92	0.658	0.923	35.7	54.1
Spring	0	313,874	20,000	5.25	0.455	0.904	22.6	52.1

Table 4.3: Statistics of the mesh complexity and the timing. g: genus; $|V|$: the number of vertices; m: the number of sites; $\#iter$: total number of iterations; T: *average* time for each Lloyd iteration measured in seconds on an Intel 2.50GHz CPU with four cores. The last four columns are the quality measures for the dual Delaunay triangulation.

Based on the discrete exponential map, our method can efficiently compute the Riemannian center and the center of mass for any geodesic VD. Thanks to its intrinsic feature, our method works well for models with arbitrary topology and complicated geometry, where the existing extrinsic approaches often fail. The promising experimental results show the advantages of our method.

Chapter 5

Application:anisotropic shape distribution

5.1 Overview

This paper presents an automatic method for computing anisotropic 2D shape distribution on arbitrary 2-manifold meshes. Our method allows the user to specify the direction as well as the density of the distribution. Using a pre-computed lookup table, our method can efficiently detect collision among the to-be-distributed shapes on 3D meshes. In contrast to the existing approaches, which usually assume the 2D objects are isotropic and of simple geometry, our method applies for complex 2D objects and can guarantee the distribution is conflict-free, which is a critical constraint in many applications. It is able to compute multi-class shape distribution and support maximal distribution so that no additional shapes can be inserted. Moreover, it can also be implemented in parallel. Our method does not require global parameterization of the input 3D mesh. Instead, it compute local parameterization on the fly using geodesic polar coordinates. Thanks to the recent breakthrough in geodesic computation, the local parameterization can be computed at little cost. As a result, our method can be applied to models of complicated geometry and topology. Experimental results on a wide range of 3D models and 2D

anisotropic shapes demonstrate the good performance as well as the effectiveness of our method.

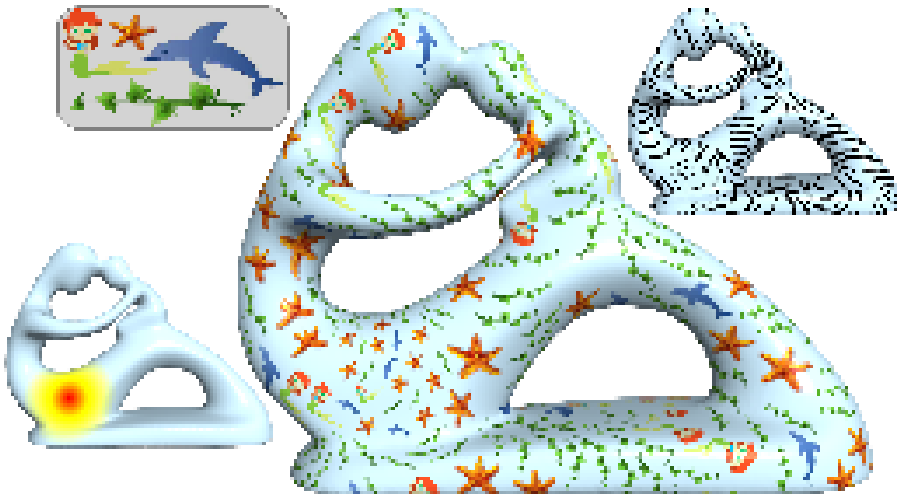


Figure 5.1: Given a 3D triangle mesh M and a set of 2D anisotropic shapes, the user first specifies a vector field for directional control (see the top-right inset) and a scalar field for density control (see the bottom-left inset). Then our method automatically distributes the given 2D shapes onto M, satisfying the user-specified directional and density constraints. It takes only 2.9 seconds to distribute 4 classes of objects on this 400K-face Fertility model. Timing was measured on a quad-core CPU at 2.66GHz.

5.2 Introduction

Sampling has a wide range of applications in computer graphics, such as geometry processing, texturing, and rendering. Among many sampling techniques, blue noise is popular due to its excellent spatial and spectrum properties. In the past decade, many elegant blue noise sampling algorithms have been proposed. Representative works include parallel sampling [145], maximal sampling [43, 155], multi-class sampling [146], and bilateral sampling [20], just name a few. Some algorithms (e.g., [15][158]) can also be directly extended from Euclidean space to curved surfaces. However, most of the existing approaches compute *isotropic* sample distributions meeting certain constraints,

such as efficiency, spectral properties, maximal distribution, and so on. To date, little research effort has been reported for anisotropic sampling on 3D models.

Li et al. [80] pioneered the anisotropic blue noising sampling. They extended dart throwing and relaxation for isotropic blue sampling to anisotropic setting. To evaluate sample distribution quality, they proposed uniform-isotropic reversible warping for the plane case and spherical harmonics for the spherical domain. Aided by global parameterization, their method can also be applied to 3D surfaces. Li et al.'s algorithm is elegant and theoretically sound. However, it has two limitations that could diminish the usage in real-world applications: first, their algorithm is based on traditional dart throwing, which, in nature, is a sequential process. Although some parallelization techniques (such as the phase grouping method [145]) could improve its performance, the implementation may be difficult, since anisotropy poses a challenge in sampling domain partitioning. Second, their method is mainly designed for Euclidean plane $\mathbb{R}^2$ and sphere $\mathbb{S}^2$, where the parameterization is readily available. Although it can be applied to 3D surfaces with global parameterization, computing a high-quality parameterization (i.e., with low angle and/or area distortion) for surfaces of complicated geometry and arbitrary topology is non-trivial.

This paper proposes a practical method for parallel computing the distribution of anisotropic shapes on arbitrary manifold meshes. Given a 3D model M and a set of 2D anisotropic shapes, the user specifies the desired distribution density as well as the orientation of each class. Then our algorithm automatically distributes the given 2D objects on M satisfying the density and direction constraints. Unlike the existing approaches, which usually assume the 2D objects are isotropic and of simple geometry, our algorithm applies for complex 2D objects and can guarantee the distribution is conflict-free, which is a critical constraint in many applications. Moreover, our method does not require

the global surface parameterization of the input 3D models. Instead, it computes local parameterization on the fly using geodesic polar coordinates. Thanks to the recent breakthrough in geodesic computation, the local parameterization can be computed at little cost. Furthermore, our method has a natural parallel structure and it is also intrinsic in that it depends only on the mesh metric instead of the embedding space. We evaluate our algorithm on real-world models with non-trivial topology and observe promising results.

The rest of the paper is organized as follows: Section 2 reviews the related work on sampling, shape distribution and discrete geodesics. Section 3 presents our implementation of efficient computation of discrete geodesics and geodesic polar coordinates, followed by our parallel algorithm on anisotropic shape distribution in Section 4. Section 5 shows the experimental results and discusses the merits and limitations of our method. Finally, Section 6 concludes the paper.

5.3 Efficient Computation of Discrete Geodesic Distances Between Arbitrary Points

We notice that both the GTU method and the SVG method are graph-theoretic algorithms: the former forms on a dense graph with $O(m^2+n)$ edges, where m is the number of samples (specified by the user), and the latter forms a sparse graph with $O(Dn)$ edges, where $D(\ll n)$ is a model-dependent-but-resolution-independent constant. Each of them has its own merits and limitations. The SVG method is promising since it is able to compute highly accurate geodesic distances between any pair of mesh vertices. However, many applications require the distances between arbitrary points on the input mesh. As Figure 5.2(b) shows, linearly interpolating the vertex distances produces poor results on meshes with large and/or irregular triangles, since distance is a non-linear

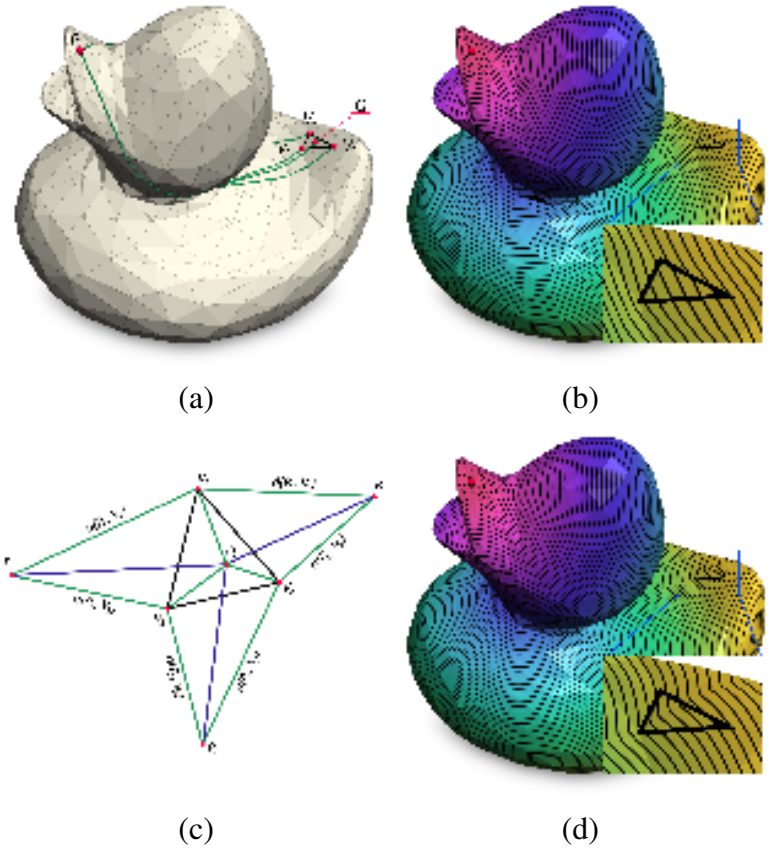
function. On the other hand, the GTU method can efficiently compute the geodesic distance between two arbitrary points in $O(1)$ time. Unfortunately, the price of such a constant-time algorithm is a very high memory usage and a long pre-computing time.

In this section, we first show that the SVG method and the GTU method can be *naturally* combined so that we can take advantage of the merits of both and avoid their limitations. Then we adopt the label correcting method to improve the running time performance of shortest path computation in SVG. Finally, we show that the computed geodesic distances induce a high-quality polar coordinate system, which will be used for local parameterization.

5.3.1 Combination of SVG and GTU

Consider a triangle mesh $M = (V, E, F)$, where V, E and F are the set of vertices, edges and faces, respectively. A vertex v is called *saddle* if the sum of its interior angles exceeds 2π. Mitchell et al. [97] proved that a discrete geodesic path cannot pass through a spherical vertex unless it is an endpoint or a boundary point and the unfolded image of the path along any edge sequence is a straight line segment.

The geodesic path $\gamma(v_i, v_j)$ between two vertices $v_i, v_j \in V$ is called direct, if it does not pass through any saddle vertices. Otherwise, it can be partitioned into several segments so that each segment is direct. Let $\Gamma = \{\gamma(v_i, v_j) | \gamma(v_i, v_j) \text{ is direct}, \forall v_i, v_j \in V\}$ denote the set of all direct geodesic paths on M. Then the saddle vertex graph associated to mesh M is an undirected graph $G = (V, \Gamma)$. All the existing exact geodesic algorithms, such as the MMP algorithm [97], the ICH algorithm [148], and most recently, the FWP-enhanced algorithm [151], can be used to construct SVG. Ying et al. [156] observed that it is *not* necessary to compute *all* direct geodesic paths, in fact, a small subset of Γ can lead to a pretty good result. Therefore, they suggested a simple-yet-effective heuristic to



(a) (b)

(c) (d)

Figure 5.2: Computing the single-source all-destination geodesic distances on a low-resolution mesh. (a) Let p be the source point (also a mesh vertex) and q a point inside a triangle $\triangle v_1v_2v_3$. (b) Using the saddle vertex graph, we can accurately compute the geodesic distances from p to any mesh vertex. With the geodesic distances defined on each vertex, one can easily estimate the distances inside a triangle using linear interpolation. However, the interpolated distances have very low accuracy, since the distance is not a linear function. (c) The geodesic triangle unfolding method can significantly improve the accuracy. With known geodesic distances $d(p,v_i)$, $i = 1,2,3$, we can unfold the geodesic triangles, $\triangle pv_2v_3$, $\triangle v_1pv_3$ and $\triangle v_1v_2p$, onto $\mathbb{R}^2$. Then the geodesic distance $d(p,q)$ is approximated by the minimal distance of three Euclidean distances $d(p_i,q)$. (d) The texture mapping reveals the high quality result by the GTU method.

control the SVG size using a parameter K: for a vertex v, only the direct geodesic paths within a geodesic disk containing K or less vertices are considered. They observed that a typical $K \in [50, 200]$ produces geodesic distances of accuracy $10^{-3} \sim 10^{-4}$, which are good for most applications.

The SVG (V, Γ) allows us to compute the geodesic distance between any pair of mesh vertices. To combine SVG and GTU, we take all mesh vertices as the samples, i.e., $m = n$. This strategy has three advantages: First, we totally avoid the construction of geodesic triangulation on M, since each $f \in F$ is a geodesic triangle. Second, we don't need to explicitly compute and store the *entire* dense weighted graph for the GTU method, since the SVG method allows us to compute the geodesic distance between any pair of samples (i.e., vertices) on the fly. Third, as pointed out by Xin et al. [150], the larger the value of m, the higher the accuracy of the computed geodesic distance. The best accuracy of the GTU method is obtained by setting $m = n$.

Now we are ready to compute the geodesic distance between arbitrary surface points $p, q \in M$. To ease presentation, we first address the simple case when one point, say $p \in V$, is a vertex, and the other $q \notin V$ is not. Let $\triangle v_1 v_2 v_3$ be the triangle containing q. See Figure 5.2(a). To compute the geodesic distance $d(p,q)$, we first apply the SVG method to compute geodesic distances $d(p, v_i)$, $i = 1, 2, 3$. Note that the three geodesic distances together with three mesh edges v_1v_2, v_2v_3, v_3v_1, form three geodesic triangles. We unfold the geodesic triangles $\triangle p v_2 v_3$, $\triangle v_1 p v_3$ and $\triangle v_1 v_2 p$, onto $\mathbb{R}^2$, and obtain three images of p, namely, p_1, p_2 and p_3. Finally, the geodesic distance $d(p,q)$ is approximated by the minimal distance of three Euclidean distances $d(p_i, q)$. See Figure 5.2(c).

The general case $p, q \notin V$ can be solved by unfolding both of the mesh triangles containing p and q, respectively. Readers can refer to [150] for the details.

5.3.2 Improving the SVG Performance

SVG naturally links the discrete geodesic problem on polyhedral surfaces and the shortest path problem on graphs, since computing the geodesic distance between p and q is equivalent to finding a shortest path on the corresponding saddle vertex graph. Dijkstra's shortest path algorithm [35] is a widely used technique for computing the shortest path. In the following, we review some fundamental concepts of the Dijkstra algorithm and its improvements.

Let $G = (N,A)$ be an undirected graph, where N and A are the sets of nodes and arcs, respectively. Let $w_{ij} \geq 0$ be the non-negative weight of the arc (n_i, n_j). To compute the shortest paths from a single node, say n_1, to all of the other vertices, the Dijkstra algorithm maintains a label vector $(d_1, d_2, ..., d_{|N|})$ and a set of nodes $\mathscr{C}$, called the candidate list, starting with $d_1 = 0$, $d_i = \infty$ for $i \neq 1$, $\mathscr{C} = \{1\}$. The Dijkstra algorithm iteratively processes the nodes from the candidate list $\mathscr{C}$ and terminates when $\mathscr{C}$ is empty. Upon termination, each label d_i is the shortest distance from the source n_1 to node n_i.

The Dijkstra algorithm takes the node with the smallest label in the candidate list $\mathscr{C}$. Since each node enters and exits $\mathscr{C}$ exactly once, Dijkstra's algorithm takes $|V|$ iterations. The Dijkstra algorithm has many variants, which is distinguished by the data structures used to compute the minimal label node from $\mathscr{C}$. Examples include binary heap, Fibonacci heap, etc.

The Dijkstra algorithm is known as a *label setting* (LS) method, since the node removed from the list is permanently labeled and never enters the list again. Methods that do not follow this node selection policy are called label correcting (LC). These methods maintain a queue for the candidate list $\mathscr{C}$ and the nodes can enter and exit the queue in constant time $O(1)$. Therefore, selection of the node to be removed from $\mathscr{C}$ is faster than Dijkstra's algorithm, at the expense of multiple entrances of nodes in $\mathscr{C}$.

We adopt the Small Label to the Front (SLF) scheme [13] for node insertion and the Large Label Last (LLL) scheme [14] for extraction: When a node n_j enters the queue $\mathscr{C}$, its labels d_j is compared with the label d_i of the top node n_i of $\mathscr{C}$. If $d_j \leq d_i$, node n_j is entered at the top of $\mathscr{C}$; otherwise n_j is entered at the bottom of $\mathscr{C}$. The top node n_i exits the queue $\mathscr{C}$ if its label is less than a threshold; otherwise send n_i to the bottom of $\mathscr{C}$. The threshold is usually set as the mean value of the labels of all the nodes in $\mathscr{C}$.

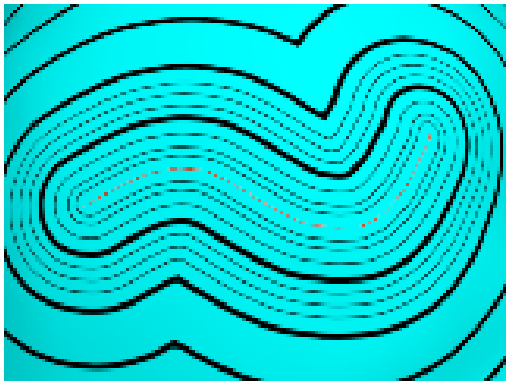
The label setting algorithm performs at most one iteration per node, but requires some extra overhead (e.g., extracting the node with minimum label) per iteration. The label correcting methods, in contrast, takes more iterations than the Dijkstra algorithm, but the overhead per iteration is smaller. We evaluate the performance of the SLF-LLL based label correcting method on SVGs of a wide range of real-world models. As Table 5.2 shows, the label correcting driven SVG can almost double the runtime performance of the Dijkstra driven SVG. It is worth noting that the LC-enhanced SVG has an empirical $O(Dn)$ time complexity to compute the single-source-all-destination geodesic distances, since the graph has $O(Dn)$ edges and the overhead per iteration is $O(1)$.

Method	Pre-computing time	Space	SSAD	Query points
GTU [150]	$O(mn^2 \log n)$	$O(m^2)$	$O(n)$	arbitrary
SVG [156]	$O(nK^2 \log K)$	$O(Dn)$	$O(Dn \log n)$	mesh vertices only
LC-SVG+GUT	$O(nK^2 \log K)$	$O(Dn)$	empirical $O(Dn)$	arbitrary

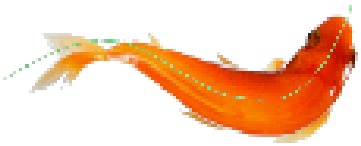
Table 5.1: Time and space complexities. n: # of vertices; m: # of sample points specified by the user; K: the size of geodesic disk containing the direct geodesic paths; D: model-dependent-but-resolution-insensitive constant; SSAD: single-source-all-destination.

5.3.3 Geodesic Polar Coordinates

A key component for computing the shape distribution is to map the 2D object onto 3D models. Global parameterization, adopted in the existing anisotropic sampling algorithm [80], is not a good choice, since global parameterization is computationally expensive



(a) Geodesic offsets



(b) Parameterization result

Figure 5.3: Geodesic polar coordinates. (a) Our LC-enhanced SVG method is able to compute the geodesic distances for curved sources. Each offset curve is then parameterized by arc-length. (b) The offset distance δ together with the normalized arc-length $\hat{s}$ uniquely determine a point on the 3D surface. The geodesic polar coordinate system, formed by the 2-tuple $(\delta, \hat{s})$, defines a bijective map between $\mathbb{R}^2$ and the patch on 3D surface.

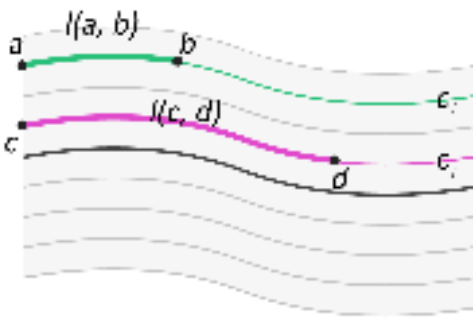
and it also produces large distortion, which may lead to numerical issues and/or visual artifacts. A possible solution is the exponential map induced local parameterization, which builds a geodesic polar coordinate system on 3D surfaces. Given a *smooth* surface S and consider a point $p \in S$. Each tangent direction $\mathbf{v} \in T_p$ corresponds to a unique geodesic $\gamma_\mathbf{v}$ passing through p in direction $\mathbf{v}$, i.e., $\gamma_\mathbf{v}(0) = p$ and $\gamma'_\mathbf{v}(0) = \mathbf{v}$. Thus, a point $q \in \gamma_\mathbf{v}$ can be represented by a 2-tuple (ρ, θ), where ρ is the geodesic distance between p and q, and θ, the polar angle, corresponds to the tangent direction $\mathbf{v}$. Differential geometry can guarantee that on a sufficiently small neighborhood of p[1], the exponential map is a diffeomorphism, i.e., both the function and its inverse are smooth.

Discrete exponential map has many applications in computer graphics and digital geometry processing, for example, surface decaling [116], Possion disk sampling [158], intrinsic CVT computation [141], just name a few. Moreover, exponential map can be extended to a general setting, where the source is a curve [131]. In spite of its pop-

[1]When ρ is less than the injective radius of p, the minimizing geodesic is unique, so that the geodesic distance ρ and the polar angle can *uniquely* determine a point in the neighborhood.

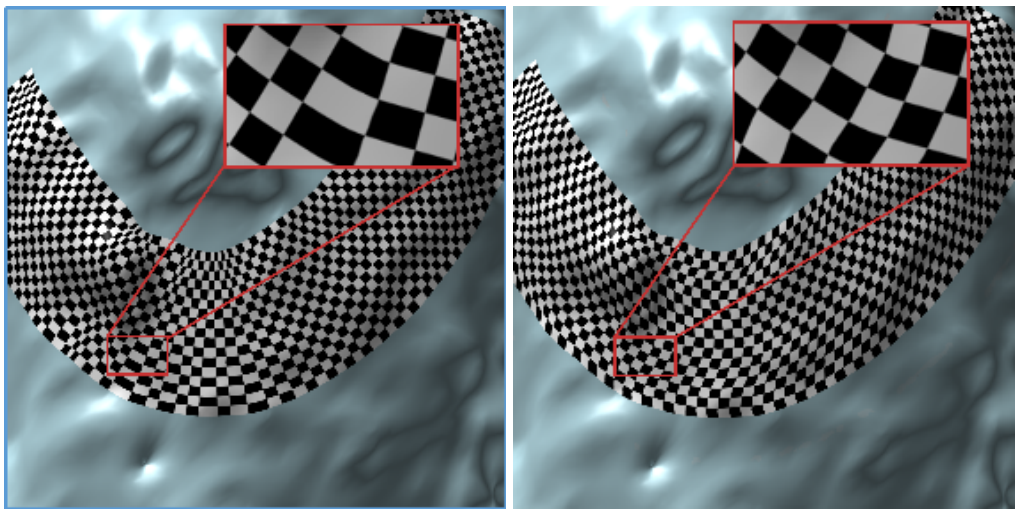
ularity, the exponential map induced parameterization, in general, is not bijective due to two reasons: First, geodesics are not unique when their lengths exceed the injective radius. Consider two geodesic paths $\gamma_{\mathbf{v_1}}$, $\gamma_{\mathbf{v_2}}$ of equal length ρ meet at a point q, then both (ρ, θ_1) and (ρ, θ_2) refer to the same point q. Second, as shown in [156], when a geodesic path passes through a saddle vertex (whose cone angle is more than 2π), it splits into many outgoing geodesic paths, meaning that all the outgoing geodesic paths share the same tangent direction. To fix the above-mentioned issues, Sun et al. [131] adopted a two-step strategy: They observed that the non-bijective issue occurs usually on part of the to-be-parameterized patch. Thus, they proposed a method for quickly detecting such regions. Then they computed a harmonic function to send these regions to rectangular domain. Since the target domain is convex, the harmonic map is guaranteed to be bijective. Schmidt [114] applied Dijkstra's algorithm to spread out the local parameters and his method can parameterize self-intersecting strokes.

Both Sun et al's method and Schmidt's method require local remeshing, which is a significant overhead, especially for the to-be-parameterized patch is big. In this section, we propose a simple-yet-effective method for computing the polar coordinates. Our method is completely integrated into the geodesic computation framework, and the induced parameterization is guaranteed to induce a bijective map.



Note that the combined SVG and GTU method can compute geodesic distances with both point sources and curve sources. Here we discuss the case of curve source, since the point source is a special case. Consider a curve c_s as the source. The u-axis is along the source curve c_s and u values range $[0, 1]$ after the normalization by the length of c_s. Each geodesic offset curve is then parameterized by the arc-length to get

the corresponding u values. As shown in the above inset, we define c_i and c_j are two isoline curves, u_a, u_c are the two corresponding start points of curve c_i and c_j, u_b, u_d are two arbitrary points on curve c_i and c_j. The u value at point u_b is $\frac{l(u_a,u_b)}{l(c_i)}$ and is $\frac{l(u_c,u_d)}{l(c_j)}$ at point u_d, where $l(u_c,u_d)$ represents the length of curve between u_c and u_d. Note that the u values of each point are only related to the arc-length on the corresponding iso-curve, meaning that the polar coordinate is guaranteed to be bijective. In contrast to [131], our method does not solve any linear system, thus, it is numerically stable and efficient. Computational results show that our method is up to 10 times faster than [131]. See Figure 5.4.



(a) Extended exponential map [131] (b) Our result

Figure 5.4: Comparison with extended exponential map [131]. Our result has comparable quality as Sun et al.'s method [131]. Since their method computes a harmonic map to fix the non-bijective issue of the exponential map, it is more computationally expensive than ours. It takes their method 1.7 seconds to parameterize the 10K-vertex patch, whereas our method spends only 0.18 seconds.

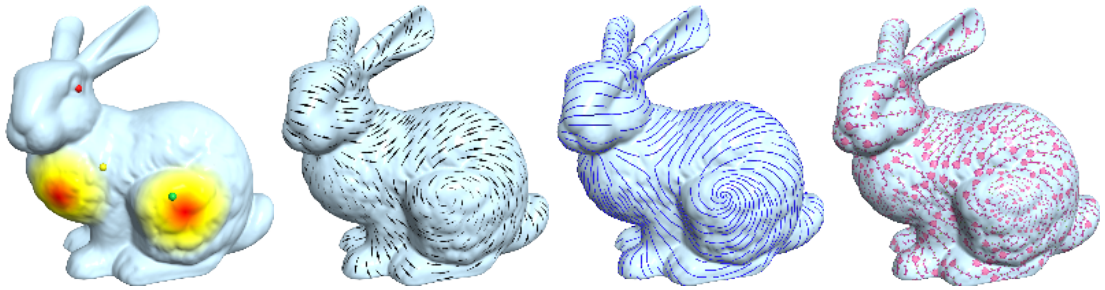
5.4 Parallel Distribution of Anisotropic Shapes

5.4.1 Motivation

A simple algorithm for distributing shapes is the traditional dart throw method. Dart throw method can generates uniformly and randomly distributed samples but is not efficient. It throws a large number of darts and only accept a small percentage of them. And it's not easy to parallel. Thus we present an improved algorithm inspired from Ying et al. [158]. We first randomly generate a large number of points on the surface following Osada et al. [102]'s algorithm, which are unbiased and insensitive to mesh tessellation. Then we start to integrate the vector field from the point to both direction to get a large number of curves. The curve serve as the center line of geodesic polar coordinates which is used to parameterize the shape on surface. Once a curve is accepted, we use the curve as source and compute a geodesic region with distance $2r$. The shapes with distance larger than $2r$ are guaranteed to be free of conflicts. While shapes within 2r may or may not conflict. Then we use a technique introduced in Section 5.4.3.1 called *Minimum Safe Distance Table* to check conflicts and rejected all the shapes which have conflicts. The algorithm terminates when all predefined curves have been processed. To parallelize the algorithm, we assigning a *unique random* value to each shape, which represents the priority of the shape. Then the candidate shapes can be processed by multiple threads at the same time. Each thread checks the collision within a shape's neighbors. The shape with the higher priority will be accepted.

5.4.2 User Input

Our algorithm allows the user to control the density and the orientation of each class. The user can specify density field by choosing the density center and specify vector



(a) Density control & singularities(b) Vector field (c) Central lines (d) Result

Figure 5.5: Algorithmic pipeline of single-class shape distribution. (a) User specifies a scalar field to control the shape density and the locations of singularities and their indices. (b) We adopt Crane et al.'s method [27] to compute the vector field which controls the shape orientation. (c) Our method automatically determines the locations, sizes and orientations of the 2D objects. Each blue curve represents the central line of one object. (d) Thanks to the minimal safe distance table, all distributed objects are guaranteed to be collision-free.

field by choose singularities. Then we will generate a vector field using Crane et at.'s method [27]. We distribute shapes along the vector field adaptively and anisotropically. Please refer to Figure 5.5 for the algorithm overview.

After density field is generated, we can get a user specified spatial variety function $g(\star) \rightarrow [1, M]$, then an adaptive distribution is defined. We need to compute a geodesic region with radius $(1+M)r$ for each point. For point p and q, we use the new min safe distance(the detail of min safe distance will explained in Section 5.4.3.1):

$$\frac{g(p)+g(q)}{2} D(\alpha, \beta)$$

.

For multi-class, user can specify $g_i(\star) \rightarrow [1, M_i]$ for each shape. We need to compute a geodesic disk with radius $r + \max(M_i r, r_m)$. we use

$$\frac{g_i(p)+g_i(q)}{2} D_{i,i}(\alpha, \beta)$$

if p, q are the same class. We use $D_{i,j}(\alpha, \beta)$ directly if p, q are different classes.

To make the surface shape distribution organized and controllable, we use the user-controlled vector field to control the directions of shapes. We adopted a globally optimal vector field generalization method from Crane et al. [27]. Our system allows the user to set singularities and direction constraints on the surface directly, as shown in figure 5.5(a). Each singularity has an index indicating the number of full rotations along a small loop around the vertex. The sum of index of singularities of a vector field on 3D mesh is equal to the Euler characteristic [27]. The user also can sketch stokes on the surface to specify the desired vector filed direction. According to the stokes' direction of corresponding faces, we fix the vector field direction of the specific area. The vector field is then computed by solving a convex optimization problem with linear constraint.

5.4.3 Algorithm

We use parallel sampling algorithm to generate a set of shapes centered as the curves following the vector field. Here we describe how to generate curves following the distance field. Each time we throw a point on the triangle mesh M, we trace the curve starting from each point along the vector field. We use curve integration method, starting from a surface point, we walk along the direction field on the triangle which the point belong to. When we encounter an edge, we set the intersection point of the tracing curve and the edge as the new starting point. We repeat this until the length of the curve reach a given length l_{max}. The resulting curves are piecewise linear polylines on the mesh M.

5.4.3.1 Single-class Shape Distribution

A 2D object S is mapped to the surface along a curve, to make sure they do not intersect each other we need an efficient method to check the conflict. For long shapes, we splite the long shape into equal size parts and check each part using precomputed distance table. Fig. 5.6 gives a brief illustration.

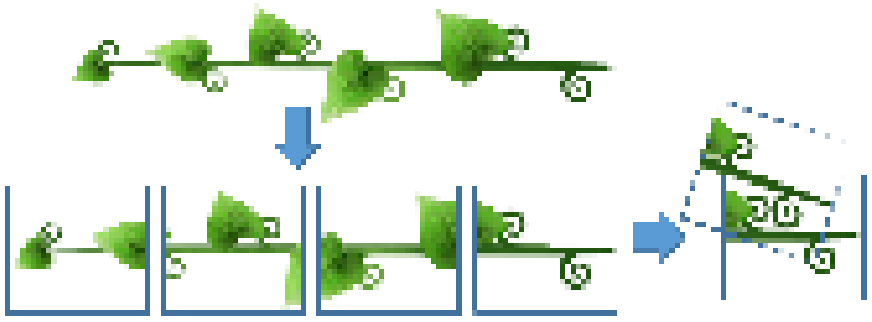


Figure 5.6: For long shapes, we divide it to several parts and test the intersection seperatly.

Now we introduce the details of distance table technique. Given a 2D object S, we compute the smallest covering circumcircle $\odot(c,r)$, where c is the center and r is the radius (see Figure 5.8). The circle can be computed in linear time using [93]. We then choose a ray L from the center c to a fixed direction as the polar axis. Clearly, objects that are $2r$ apart (distance between their center points) are guaranteed to be free of conflicts. However, objects within $2r$ may or may not conflict, depending on their orientations.

To efficiently determine whether or not two objects conflict, we pre-compute a look up table, named *Minimum Safe Distance Table*, or MSDT for short.

As shown in Figure 5.8(right), the relation between two objects A and B is determined by the distance between two center points $d(c_A,c_B)$ and the orientations of each object. The latter is characterized by the angles between the polar axis and the line c_Ac_B, denoted by α and β, respectively.

We uniformly divide $[0,2\pi)$ into D intervals, and build a $D\times D$ look up table. For each pair of angles $\{\alpha,\beta\}$, the entry $D(\alpha,\beta)$ is the minimum distance between the center points that guarantees no conflict, which is computed as follows:

If the object is a 2D polygon with e edges, we can check the polygon intersection in $O(e^2)$ time. Thus, the *Minimum Safe Distance* $D(\alpha,\beta)$ can be found by the binary

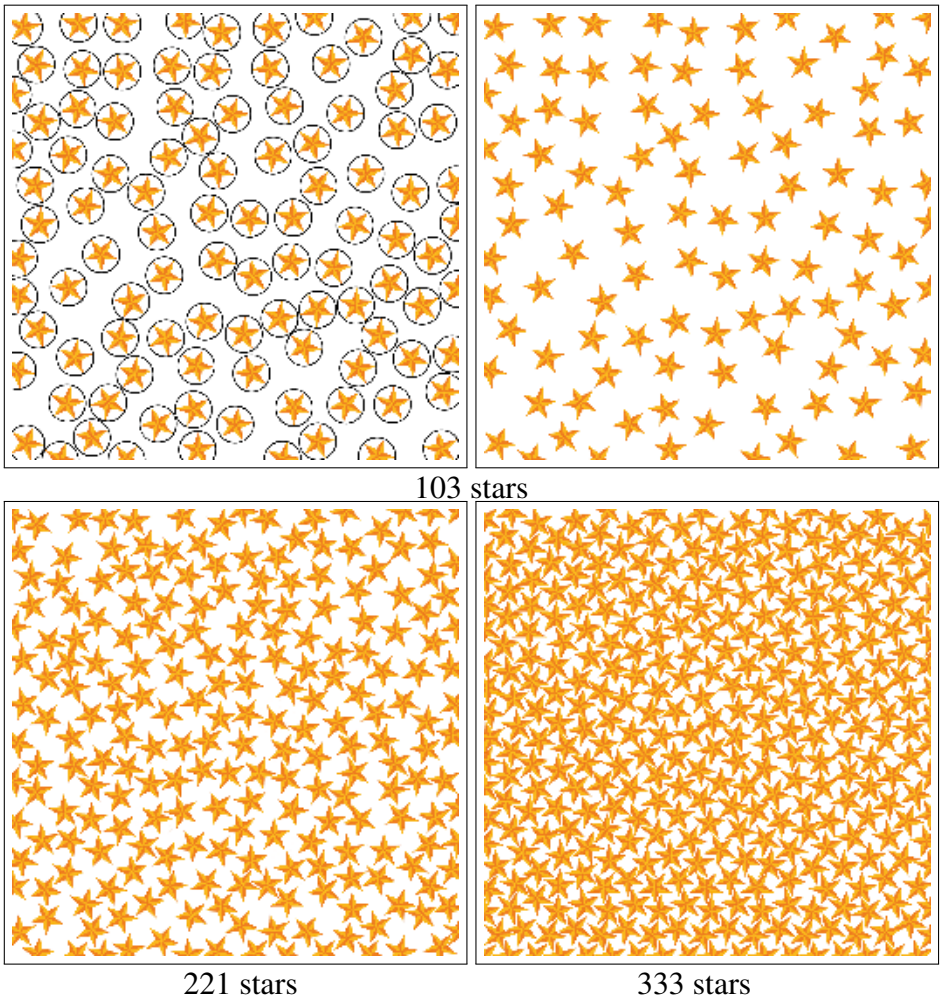


Figure 5.7: Comparison with Poisson disk sampling. Row 1: Poisson disk sampling considers each sample as a disk and does not allow overlapping disks. Row 2: Our method allows a more dense distribution than Poisson disk sampling, as long as the stars are not overlapping.

search of $[0, 2r]$ in $O(D^2e^2 \log r)$ time and with $O(D^2)$ space.

If the object is a 2D image with b boundary pixels, we check each boundary pixel of shape B. If any boundary pixel locates inside shape A, the two shapes are conflicted. we can test whether two objects are conflicted in $O(b)$ time.

Putting it all together, the construction of MSDT takes $O(D^2 b \log r)$ time and occupies $O(D^2)$ space.

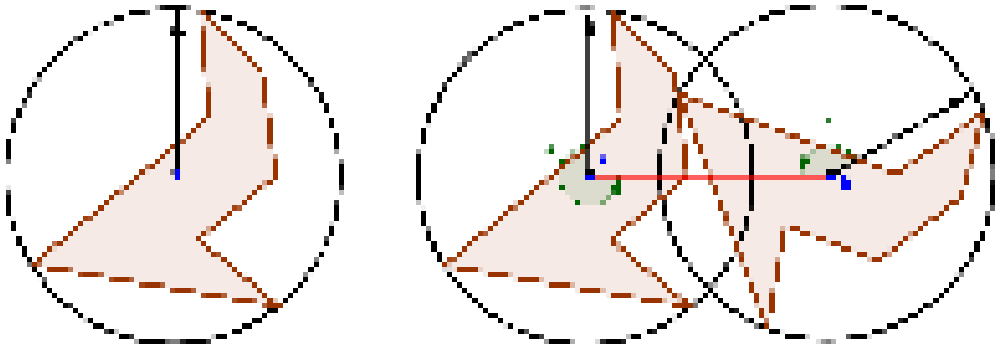


Figure 5.8: Left: circumcircle of shape. Right: The minimum safe distance between two shapes (red line).

Let C be the center-point of each shape, and vector $\overrightarrow{CO}$ be the orientation. During the shortest path computation using SVG, we can unfold point O together with the center C. Thus, the angles $\{\alpha, \beta\}$ could be computed.

Our algorithm can compute (α, β, d_g) for each point when computing the geodesic polar coordinates. Thus it's quite efficient and didn't requite additional computation cost.

The pseudo-code in algorithm 9 describes our distribution algorithm.

5.4.3.2 Multi-Class Shape Distribution

Our method is different from the multi-class blue noise sampling in [146], since we do not allow overlapped shapes.

In multi-class case, for a group of k shapes $\{c_1, c_2, \ldots, c_k\}$, we have to compute $\binom{k}{2}$ different classes MSD table and k same class MSD table. thus, the time complexity is $O(k^2D^2b\log r)$, takes $O(k^2D^2)$ memory space.

algorithm input: vector field for each class O_i, target number of each class N_i.

Similar like single shape distribution, we generate a dense point set. Each point are

Algorithm 9 Parallel Single Shape Distribution

$D \leftarrow$ Minimum Safe Distance table.
$S \leftarrow$ a dense random curve set
// The orientation of each curve is guided by user specified vector field.
random_shuffle(S);
// now, consider S as a queue of dart throwing curves.
foreach S_i **in** S **do** $S_i.myIndex \leftarrow +\infty$
$curveOrder \leftarrow 0$
$result \leftarrow \varnothing$

foreach parallel thread do:
while S is not empty **do**
  atomic $t \leftarrow curveOrder++$
  //t gets the value of $curveOrder$, and $curveOrder$ was increased by 1.
  atomic $p \leftarrow S.pop\_front()$
  $p.myIndex \leftarrow t$
  $\{w\} \leftarrow$ use *label correcting* algorithm to compute geodeisc region centered at curve c with radius $2r$
  $isAccepted \leftarrow true$
  foreach w_i **in** $\{w\}$
    if $w_i.d_g < D[w_i.\alpha][w_i.\beta]$ $w_i.myIndex < p.myIndex$ **then**
      $isAccepted \leftarrow false$
      break foreach loop
    end if
  end for
  if $isAccepted$ **then**
    atomic $result.push(c)$
    foreach w_i **in** $\{w\}$
      atomic remove w_i from S
    end for
  end if
end while
end thread

output $result$

specified to a shape. A point has a probability p_i to be i-th shape. p_i is defined as:

$$p_i \leftarrow \frac{N_i A_i}{\sum_{j=1}^{k} N_j A_j}$$

where A_i is the area of i-th shape.

let r_i be the circumradius of i-th shape, and $r_m \leftarrow \max r_i, i = 1 \ldots k$. for each point of shape i, we compute a geodesic disk with radius $r_i + r_m$, and collect all neighbor points. Then check the MSD by looking up the table $D_{i,\star}(\alpha, \beta)$

5.5 Experimental Results

Performance. We implement the algorithm in C++ and test it on an Intel Xeon quad-core CPU at 2.66GHz. Thanks to the nice parallel structure of our method, we also use OpenMP to parallelize our algorithm so that it can take the advantage of all CPU cores. We observe that our parallel implementation on a quad-core CPU is up to 3 times faster than that of a single-core and it allows interactive manipulation on 3D models with 100K faces. See Table 5.3 for running time performance and the accompanying video demonstration. Figures 5.9, 5.10 and 5.13 demonstrate our results on single-class and multi-class shape distribution.

Table 5.2: Performance of geodesic algorithms. The ICH algorithm computes the exact geodesic distances, whereas the SVG method and our LC-enhanced SVG+GTU method compute the approximate distances. We set $K = 50$ for constructing the SVG. ε: mean relative error of our method and the SVG method. Columns 3 to 5 report the running time (in seconds) of the geodesic algorithms.

Model	$\|V\|$	T_{ICH}	T_{SVG}	T_{our}	ε
Torus	$200K$	28.64	0.20	0.09	0.15%
Fertility	$400K$	49.47	0.37	0.21	0.14%
Kitten	$548K$	93.85	0.48	0.30	0.16%
Bimba	$800K$	125.03	0.70	0.38	0.13%
Dragon	$1.28M$	266.38	1.09	0.60	0.20%

Table 5.3: Mesh complexity and running time performance. T_1 and T_4: running time (in seconds) on a quad-core CPU using a single core and all cores, respectively.

Model	$\|F\|$	# of 2D shapes	T_1	T_4
Bump Sphere	50K	458	3.1	1.1
Bunny	80K	561	4.2	1.4
Fertility	400K	1,150	8.1	2.9
Kitten	250K	864	9.5	3.7

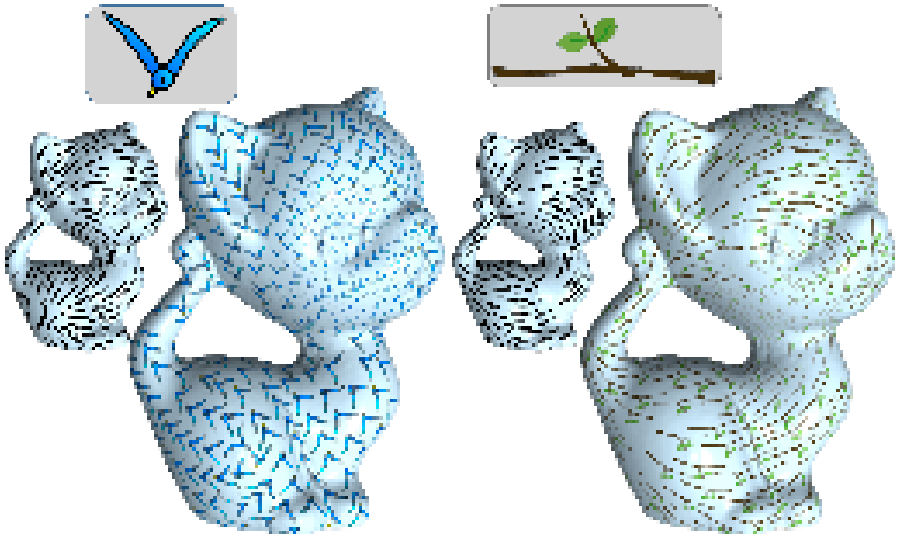


Figure 5.9: Single-class shape distribution on the Kitten model with different density and texture.

Robustness. Our method is intrinsic since both the collision detection and shape distribution depend on the metric only. Thanks to the intrinsic nature of the geodesic algorithms [156, 150], our method is insensitive to mesh resolution and tessellation. See Figure 5.11 for our results on the Bimba model with various resolutions.

Comparison to [80]. Our method and the anisotropic blue noise sampling method [80] differ in 4 aspects: First, their method is mainly designed for the 2D problem. Although it can be extended to 3D surfaces via global parameterization, computing a high quality parameterization for high-genus model is technically challenging. Our method is based on local parameterization, which can be computed efficiently on the fly. Therefore, our method can be applied to models of arbitrary geometry and topology. Second, their

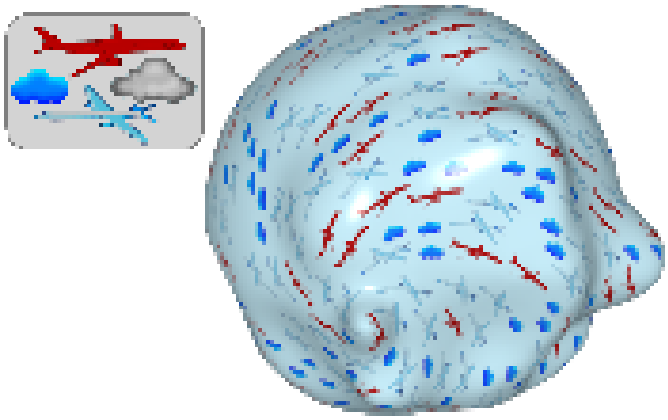
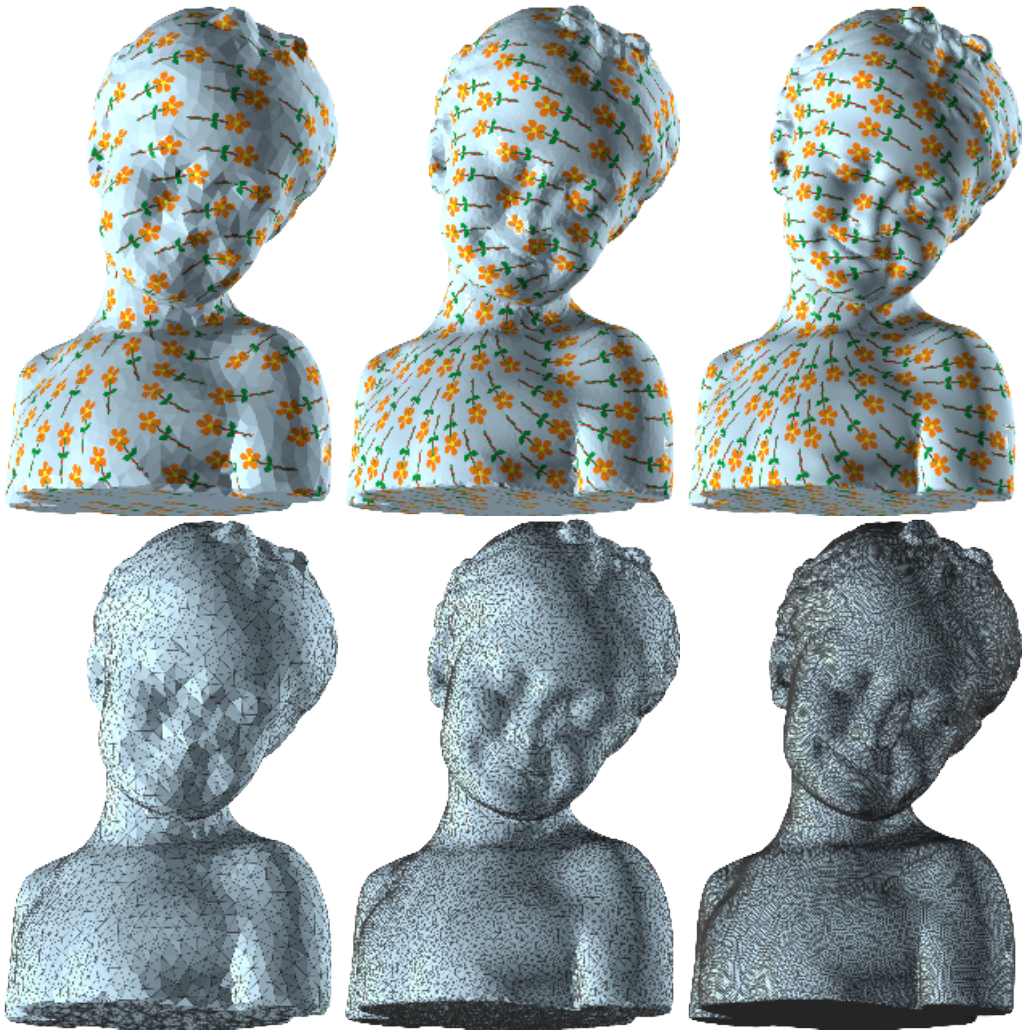


Figure 5.10: Multi-class shape distribution on a bumpy sphere with 50K faces.

method is based on the dart throwing technique, which is a sequential process. It is hard to extend the existing parallelization techniques (such as the phase group method [145]) to anisotropic setting, since anisotropy poses a challenge in domain partition. Third, their method abstracts the anisotropic shapes by bounding ellipses so that collision detection and quality evaluation can be done in an analytical manner. However, ellipse is not an effective representation if the 2D shape has a highly concave boundary. As a result, their method may produce a distribution which is far from maximal. Thanks to the minimal safe distance table, our method can faithfully keep the geometry of the 2D shapes hereby allow a much denser distribution. See Figure 5.12. Last but not the least, their method supports single-class sampling only, whereas our method can compute multi-class shape distribution.

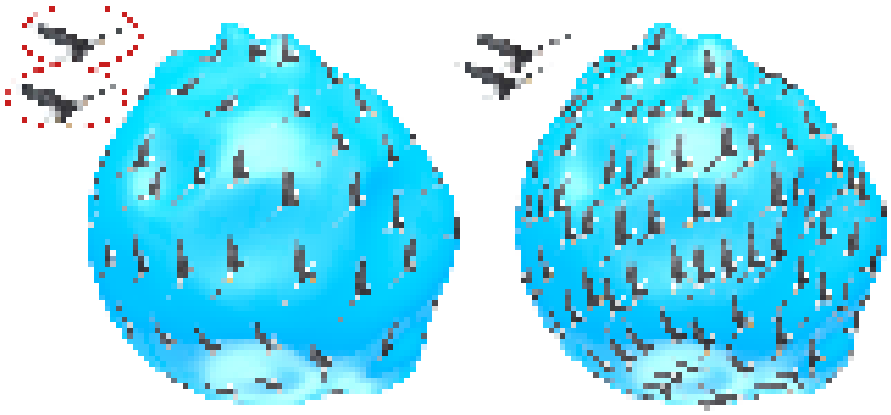
5.6 Conclusion

This paper presents a practical method for computing anisotropic 2D shape distribution on arbitrary 2-manifold meshes with user control. Our method has several advantages: First, unlike the existing sampling approaches, which usually assume the 2D objects are isotropic and of simple geometry, our method applies for complex 2D objects and can



7K faces 30K faces 150K faces

Figure 5.11: Our method works well on the Bimba model with various resolutions.



(a) Li et al.'s result (b) Our result

Figure 5.12: Comparison to [80]. The 2D Eagle shape is highly concave, so a simple bounding ellipse cannot capture its geometric features. As a result, requiring non-overlapping ellipses is very pessimistic. Li et al.'s method [80] can distribute only 115 shapes. Obviously, the distribution is not maximal. Our method produces a much more dense distribution, containing 224 eagles, since it allows overlapping ellipses as long as the adjacent eagles do not collide.

guarantee the distribution is conflict-free, which is a critical constraint in many applications. Second, it is able to compute multi-class shape distribution and can also support maximal distribution so that no additional shapes can be inserted. Third, our method does not require global parameterization of the input 3D mesh. Instead, it compute local parameterization on the fly using geodesic polar coordinates, which has little computational cost. Therefore, our method can be applied to models of complicated geometry and topology. Last but not the least, our method has a natural parallel structure, and can be implemented on multi-core CPUs. Experimental results on a wide range of 3D models and 2D anisotropic shapes demonstrate the good performance as well as the effectiveness of our method.

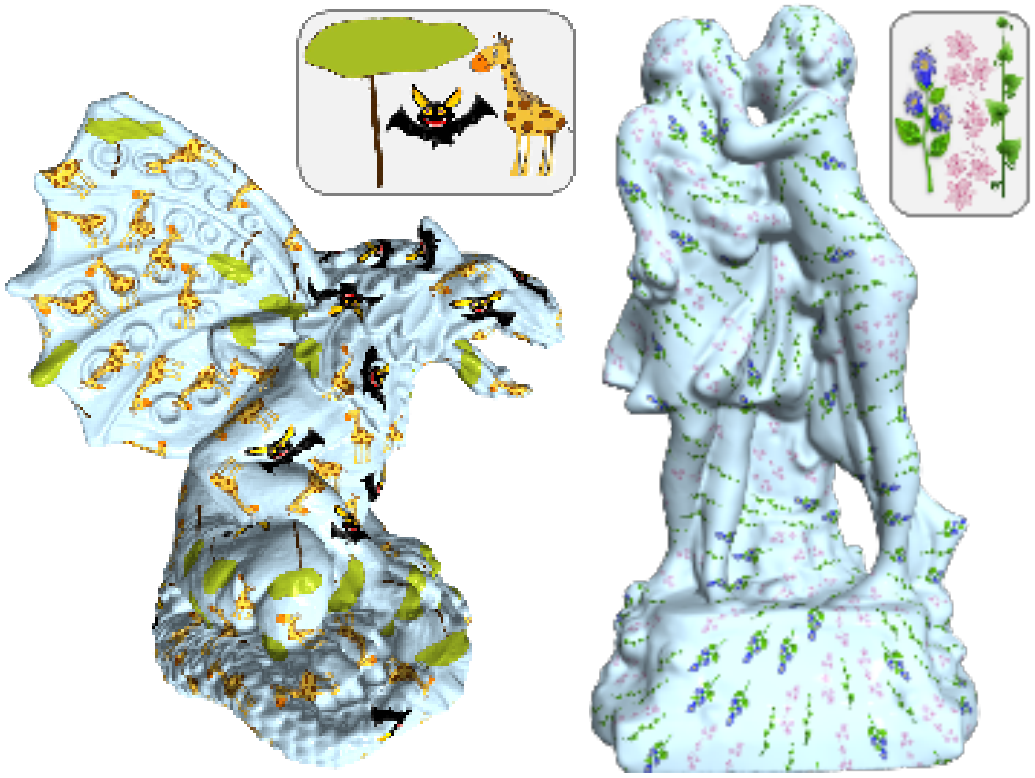


Figure 5.13: More results.

Chapter 6

Robust and GPU-friendly Isotropic Meshing Based on Narrow-banded Euclidean Distance Transformation

6.1 Overview

In this paper, we propose a simple-yet-effective method for isotropic meshing via Euclidean distance transformation based Centroidal Voronoi Tessellation (CVT). The proposed approach aims at improving the performance as well as robustness of computing CVT on curved domains while simultaneously maintaining the high-quality of the output meshes. In contrast to the conventional extrinsic methods which compute CVTs in the entire volume bounded by the input model, our idea is to restrict the computation in a 3D shell space with user-controlled thickness. Taking the voxels which contain the surface samples as the sites, we compute the exact Euclidean distance transform on the GPU. Our algorithm is fully parallel and memory-efficient, and it can construct the shell space with resolution up to 2048^3 at interactive speed. Since the shell space is able to bridge holes and gaps up to a certain tolerance, and tolerate non-manifold edges and degenerate triangles, our algorithm works well on models with such defects, whereas the conventional remeshing methods often fail.

6.2 Introduction

Triangle meshes have found widespread acceptance in computer graphics as a simple, convenient, and versatile representation of surfaces. However, raw meshes obtained from 3D scanners are often not ready for subsequent geometric processing, since they may contain holes, gaps, noise, degenerate triangles and non-manifold edges.

A popular approach to improve the mesh quality is via centroidal Voronoi tessellation (CVT), which can generate a highly regular distribution of sites with respect to a given density function. A typical CVT-based remeshing method iteratively updates the generator of each Voronoi cell until it coincides with its center of mass. Then the isotropic mesh is obtained by the dual graph of the computed CVT. A key step in CVT computation is to construct Voronoi diagrams (VD) in each iteration. Although it is fairly simple to construct VD in Euclidean spaces, computing VD on curved domains is expensive due to lack of closed-form formula of geodesic distance. A practical way is to compute the restricted Voronoi diagrams (RVD) [152], which is the intersection between the given model and a CVT defined in $\mathbb{R}^3$.

In this paper, we propose a new RVD-based computational framework for isotropic meshing, aiming at improving the performance as well as robustness of computing RVD while simultaneously maintaining the high-quality of output meshes. Rather than computing CVTs in the entire volume bounded by the input model, our idea is to restrict the computation in a 3D shell space with user-controlled thickness. Since the shell space is able to bridge holes and gaps up to a certain tolerance, and also tolerate non-manifold edges and degeneracies, our algorithm works well on imperfect meshes with such defects, whereas the conventional *remeshing* methods often fail. See Figure 6.1.

This paper makes the following contributions:

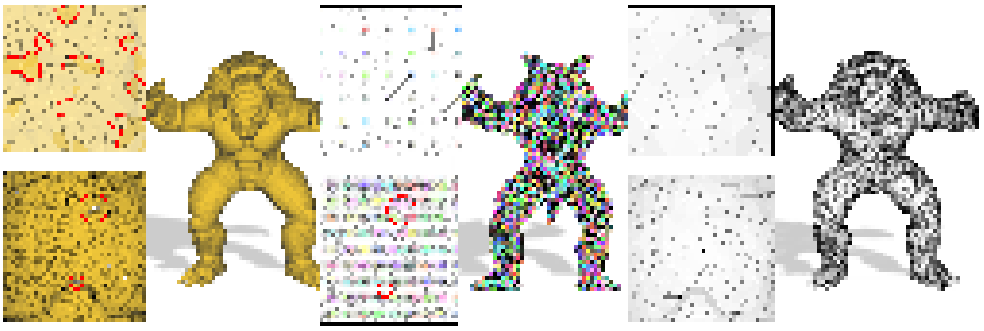


Figure 6.1: Isotropic meshing on an imperfect mesh with non-manifold edges, degenerate triangles and holes.

- We propose an efficient framework for constructing isotropic meshes via voxel representation. It can completely avoid the computationally expensive components, such as implicit function fitting, isosurface extraction, and geodesic distance computation, which are often used in the existing methods.
- Our framework can produce topologically consistent shell space with user control so that it can bridge holes and gaps and tolerate noise to some certain extent. It also works well for models with non-manifold edges and degenerate triangles.
- We present a fast and memory-efficient algorithm for computing narrow-banded distance fields on GPUs. The CVT, RVD and the dual Delaunay triangulations are also computed in parallel on the GPUs.

6.3 Algorithm

Let O denote the input 3D mesh. We first construct a voxel representation M of O at a given resolution *res*. Then we construct a shell space $\bar{P}$ consisting of off-surface points, where each point in $\bar{P}$ has a distance $d_p \leq d$ to its closest point on M. The threshold d is specified by the user and model-dependent.

Our isotropic meshing algorithm adopts Lloyd's framework. Starting with k randomly

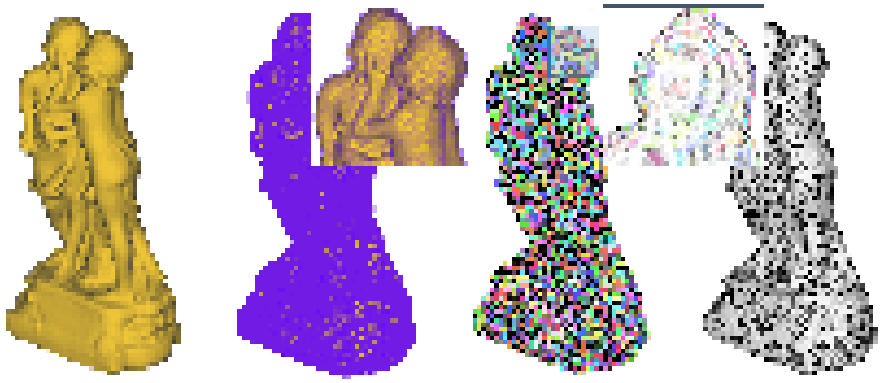


Figure 6.2: Overview of our approach on the Sculpture model. (a) Input mesh; (b) Shell space with $d = 3$; (c) CVT with 3K seeds; (d) The output isotropic mesh.

generated seeds, it minimizes the CVT energy by iteratively updating the seed positions. In each iteration, it computes the Voronoi diagrams confined in the shell space and moves the seeds toward the corresponding mass centers. The algorithm projects the seeds back to $\bar{P}$ if they are outside the shell space. Upon convergence, it propagates the seed information in the shell space to look for connected Voronoi cells and extracts the dual Delaunay triangulation. Our method is described in detail in the next subsections, and furthermore outlined in Algorithm 10.

Algorithm 10 Isotropic Meshing based on EDT

Input: 3D surface O, voxel resolution res, shell space thickness d, convergence threshold ε, and number of seeds k

Output: Isotropic mesh with k vertices

1: $S \leftarrow k$ random seeds
2: $M \leftarrow$ Voxelization(O, res)
3: $\bar{P} \leftarrow$ ShellSpaceConstruction(M, res, d)
4: **while** convergence not reached **do**
5: $\quad V_k \leftarrow$ SearchClosestSeedInShell($\bar{P}$, S)
6: $\quad C_k \leftarrow$ CenterMass(V_k)
7: $\quad \bar{C}_k \leftarrow$ UpdateSeed($\bar{P}$, C_k)
8: $\quad S \leftarrow \bar{C}$
9: **end while**
10: $V_k \leftarrow$ ShellFlooding($\bar{P}$, S) **return** DualTriangulation(V_k)

6.3.1 Memory-efficient Shell Space Construction

We introduce a memory-efficient way to construct shell spaces in real-time. We extend the Parallel Banding Algorithm (PBA) of Cao et al. [18] to compute Euclidean distance transform (EDT) in the narrow-band manner. Their algorithm partitioned the input domain into small chunks of equal size, which can be processed in parallel. The results are then merged concurrently. Although their method is exact and efficient, it is not practical for large-scale models due to rapidly growing memory consumption. Our method addresses this memory issue by on-the-fly computation and integrating fast bitmap indexing technology on GPUs. We explain the principle in two dimensions for simplicity; the idea can be easily extended to three dimensions.

The input image is divided into a virtual grid made up of occupied and non-occupied pixels, where the former pixels, denoted as sites $\in S$, are run-length encoded. Every pixel goes through a two-step process : (1) finding the nearest site S_{ij}, among all sites in row j; (2) determining the closest site, among all the nearest sites in the current column i. For the first step we assign one thread to process a row because it's more efficient to do more computation in a single thread than repeatedly accessing global memory with multiple threads. The second step extends the dividing-and-merging approach of PBA, with employing warp[1]-vote and warp-shuffle functions in CUDA to exchange the nearest sites information within a chunk. We make every thread in the same warp doing the same calculation, hence greatly reducing the warp variation that often compromises performance.

Let's take Fig. 6.3 as an example. When the threads in a warp come to column i, each row will compute their corresponding nearest sites S_{ij}. The nearest site of the current pixel is colored in blue. Note that only some sites (Blue-blank dots) satisfy the the

[1]A warp is a pool of threads that executes physically in parallel.

```
// To ease representation, we show the 2D version here
function ShellSpaceConstruction(M, res, d)
for all thread j = 0 to res in parallel do
    for i = 0 to res do
        S_ij ← GetNearestSite(M,j,d)
        discard S_ij if || d_ij || > d
        set barrier // Ensure every thread gets S_ij
        // Compare with other threads in same warp
        // warp size = h, current warp ID = k
        C_k ← S_ij
        for x = 0 to h do
            if IsCloser(C_k, C_x) then
                C_k ← C_x
                id ← x
            end if
        end for
        // Mark the closest site of pixel (i, j)
        Bitmap[i][id] = true
    end for
end for
Collect and merge the closest sites if Bitmap[i][j] = true
return P̄ // return pixels located inside the shell
```

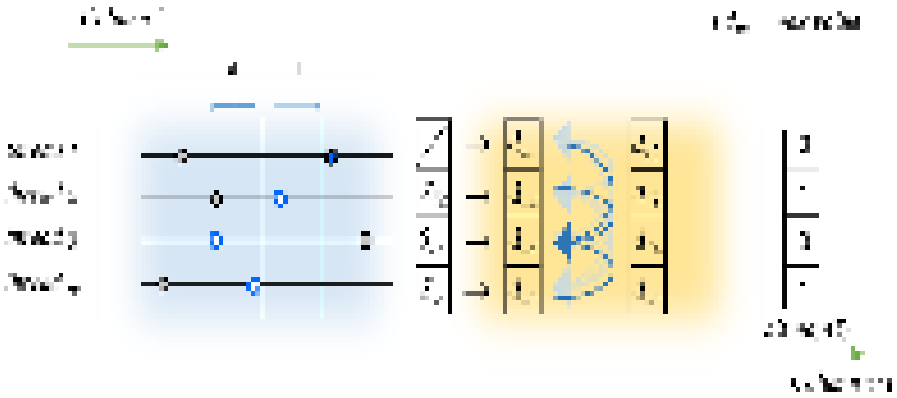


Figure 6.3: An illustrative example on distance field computation in a narrow band. See the text for the description and function *ShellSpaceConstruction* for the pseudo code.

distance constraint. Therefore, the sites S_{i1} of thread 1 can be safely discarded. We set a barrier to ensure that every thread obtains some sites before exchanging information. After synchronization, we sweep each S_{ij} to other threads in the same warp to update the closest site of the current pixel (i,j), based on the distance function d_{ij}. A bitmap stores a key (boolean) value for every pixel. The '0's indicate that the corresponding nearest sites are **not** possible to be the closest sites of the current column. Then the threads repeat the same procedure for the next column $i+1$ until they reach the last column. In the final step we collect the closest sites with flag '1' in different chunks and merge them to get the pixels that form the shell region. Since the nearest sites are computed on-the-fly and the temporary result is indexed by bitmap only, our algorithm requires less memory than the PBA method.

6.3.2 Constructing 3D Voronoi Diagrams in Shell Space

As mentioned above, the shell space represented by $\bar{P}$ is used as constraints to construct Voronoi diagrams and update the positions of seeds. Initially, the seeds $S = \{s_1, .., s_k\}$ are located on the input mesh. We collect points $\in \bar{P}$ that share the same closest seeds to build Voronoi diagrams.

We perform a proximity search to find the closest seed for all query points from $\bar{P}$. To speed up the process, the seeds are projected onto a uniform grid G with smaller resolution of *res* (e.g. 32^3), such that, for a query point q, it just looks up the seeds in the grid cell G_q where q falls into. In case any border of the grid cell is closer to q than the seed found in G_q, query point q looks up the neighbor cell of G_q.

6.3.3 Computing CVTs

Updating the seeds' position towards uniform distribution is crucial for constructing CVT. Based on the following energy function, optimal positions can be reached by minimizing $E(S)$.

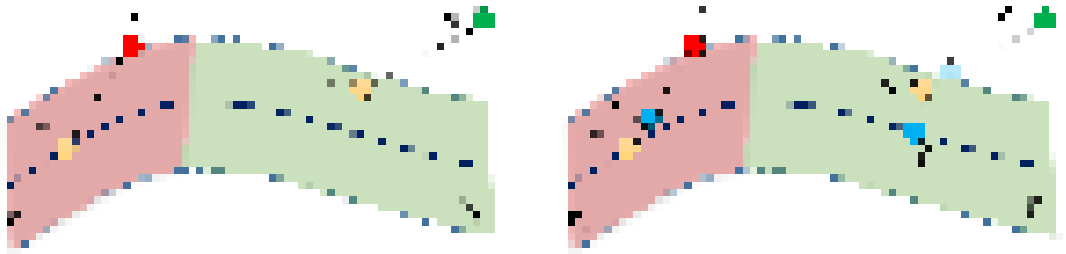
$$E(S) = \sum_{i=1}^{m} \int_{V_i} \rho(p)\|p - s_i\|^2 dp,$$

where V_i is the Voronoi cell of seed s_i, $p \in \bar{P}$ and ρ is a non-negative user-defined density function.

According to Lloyd's algorithm, a seed s_i moves iteratively toward the corresponding mass center c_i of Voronoi Diagram V_i until convergence. However, the mass center could be located far from the surface, as is constructed in the shell space. We consider the following new position to replace mass center for each iteration.

$$\bar{c}_i = s_i + u\frac{\overrightarrow{s_i c_i}}{\|\overrightarrow{s_i c_i}\|},$$

where $u \in \mathbb{R}^+$ is the magnitude of movement of seeds. We observed that if the seeds move in different magnitude, the area of CVTs will largely vary depending on surface curvature. In addition, in order to guarantee the topology consistence, the new center will be projected back to M if it exceeds the shell space, as shown in Fig. 6.4.



(a) (b)

Figure 6.4: Illustration of the update process with two seeds in an iteration. (a) Yellow dots are the seeds of Voronoi cells V_i (in red) and V_j (in green). Red and green dots are their mass centers respectively. (b) The seeds move along vector $\overrightarrow{sc}$ to the new centers (blue dots). Project $\bar{c}_j$ (light blue dot) to the surface since it is outside the shell region.

6.3.4 Computing Dual Triangulations

Upon convergence we have all the generators uniformly distributed. The remain part describes how to extract the dual Delaunay triangulation.

First, we find the direct neighbors of all seeds, where the direct means if there exists two voxels from their Voronoi cells that are connected. The adjacency neighbors can be found by flooding all seeds information to all voxels in the shell $\in \bar{P}$. Each voxel associates a hash table to hold the location of neighbors (26 voxels). Each propagation updates the current seed information to neighbor voxels until all voxels are reached. This approach avoids producing wrong network for seeds that are geometrically close, but topologically far from each other. After that we organize their direct neighbors in clockwise order and finally extract the triangle mesh.

6.4 Experimental Results

All tests were performed on a PC with an Intel Xeon E5 2.5GHz CPU and an nVidia Quadro K5000 with 4GB RAM.

6.4.1 Narrow-banded Distance Fields

Table 6.1 lists the computational time and the peak memory under varying parameter d and *res* (Fig.6.5). Clearly, when d increases, the computational time increases insignificantly with the increased amount of nearest sites in the shell. This is due to the low cost of intra-warp communication and the reduction of warp divergence in our algorithm. Also, the memory consumption is remarkably small, considering the scene is in high resolution uniform grid. Traditional algorithms (e.g., [18]) usually require memory at least 10 times more than ours.

Table 6.1: Performance (time in seconds) of our algorithm on different d in resolution 1024^3 and 2048^3.

Model	d	Memory	Time	Memory	Time
Dinosaur	1	149MB	1.186	1.18GB	12.3
	3	174MB	1.239	1.23GB	13.1
	6	206MB	1.313	1.30GB	13.3
	9	235MB	1.383	1.36GB	13.5

Table 6.2 compares our algorithm with PBA on the Sculpture model in resolution 512^3. Since PBA computes a full distance map, their performance is independent of distance d. The result shows that our algorithm consumes significantly less memory and runs much faster than PBA with a reasonably small d.

Table 6.2: Comparison of our algorithm with PBA in resolution 512^3 (time in seconds).

Model	Memory	d	Time
Sculpture (PBA)	1073MB	N/A	0.310
Sculpture (Ours)	26.6MB	1	0.147($\times$2.1)
	33.7MB	3	0.173 ($\times$1.8)
	49.7MB	6	0.200 ($\times$1.6)
	66.1MB	15	0.286 ($\times$1.1)
	76.2MB	20	0.339($\times$0.9)

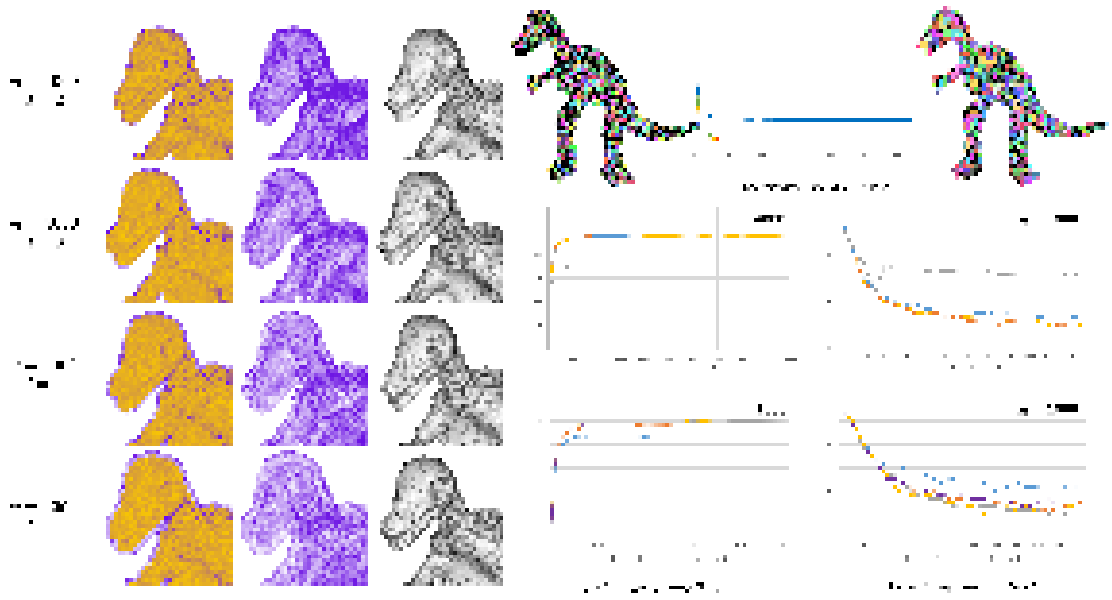


Figure 6.5: Evaluating the mesh quality under various shell space parameter d. (a) and (b) show the triangulation quality measure and the singularity ratio. (c) We also observe that our GPU-based Lloyd algorithm converges in usually 100-200 iterations and d has little impact on the convergence rate. The horizontal axis shows the iteration number and the vertical axis is the normalized CVT energy function.

6.4.2 CVT Computation

Similar to [141], we adopted the following criteria to measure the triangle mesh quality. (1) Triangle quality $Q(t)$ defined by $6P_t/\sqrt{3}H_t$, where P_t and H_t are the inradius and the length of the longest edge of triangle t. (2) The smallest angle θ_{min} and the average θ_{avg} of minimal angles of all triangles. (3) The ratio of singularities, defined by v_s/k, where v_s is the number of non 6-valent vertices and k is the number of vertices.

We allow the user to balance accuracy and efficiency in the choice of offset d. Figure 6.5 describes the relationship between the distance d, the number of generator and the quality of remeshed surface. As the offset increases to 9, with 4k generators, the mesh quality of dinosaur model dramatically drops. This also happens when the offset decreases to 2 with 1k generators. Figure 6.5(b) illustrates the quality difference between different d clearly. Along with other examples in Table 6.3, we can show that the mesh is at best

quality with offset distance in specified range (2 to 6). Figure 6.6 compares our method with two parameterization-free isotropic meshing methods, the *intrinsic* CVT method by Wang et al [141] and the *extrinsic* RVD method by Yan et al. [152]. Thanks to the GPU-friendly structure and the computational power of modern GPUs, our method runs significantly faster than their CPU-based implementations.

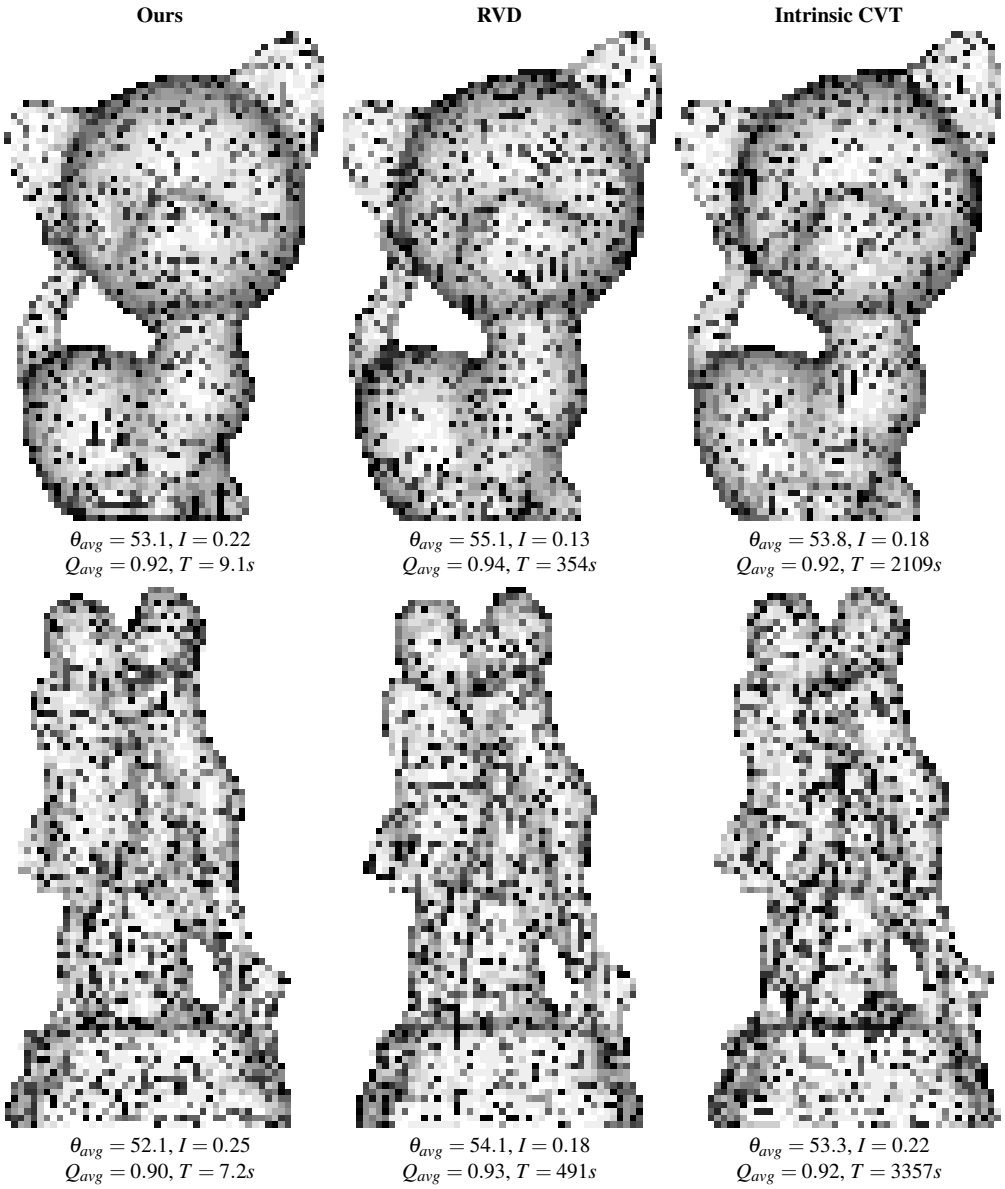


Figure 6.6: Comparison with the RVD method [152] and the intrinsic CVT method [141].

Table 6.3: Model complexity and runtime performance. *SS*: time (in seconds) for shell construction; m: the number of seeds; T: average time for each Lloyd iteration; n: the number of iterations; I: singularity ratio.

Model	d	SS(s)	# of Sites	m	T(s)	n	I	Q_{min}	Q_{avg}	θ_{min}	θ_{avg}	Time(s)
Sculpture	3	0.966	1.04×10^6	3K	0.064	100	0.25	0.639	0.907	36.1	52.4	7.2
Heptoroid	2	1.429	2.63×10^6	9K	0.138	120	0.24	0.624	0.902	35.3	51.9	19.5
Helix	2	1.212	4.4×10^5	4K	0.036	100	0.29	0.589	0.889	33.4	50.9	5.14
Pegaso	2	0.948	1.33×10^6	8K	0.063	150	0.26	0.600	0.894	31.5	51.3	10.6
Dinosaur	2	0.880	5.5×10^5	4K	0.025	160	0.28	0.636	0.894	30.2	51.2	4.8
	4	0.913	5.5×10^5	1K	0.046	170	0.31	0.602	0.900	30.6	51.8	9.2
Armadillo	2	0.944	1.26×10^6	4K	0.051	200	0.26	0.613	0.902	30.3	51.9	11.3

6.5 Conclusion and Future Work

This paper presents a robust and efficient method for constructing isotropic meshes using Euclidean Distance Transform. Our algorithm constructs a narrow band space enclosing the input surface, in which 3D centroidal Voronoi tessellations and restricted Voronoi diagrams are computed. Our algorithm is fully parallel and memory-efficient, and it can construct the shell space with resolution up to 2048^3 at interactive speed. Moreover, our method can process implicit surfaces, polyhedral surfaces and point clouds in a unified framework. Computational results show that our GPU-friendly isotropic meshing algorithm produces results comparable to state-of-the-art techniques, but runs significantly faster than the conventional CPU-based implementations.

Our current implementation adopts a constant resolution to construct the shell space. This, however, is not optimal, since it is over-pessimistic for the regions with fairly flat geometry, and it may not be enough for the highly-curved. In the future, we will develop a geometry-aware algorithm for parallel constructing the shell space with adaptive resolution.

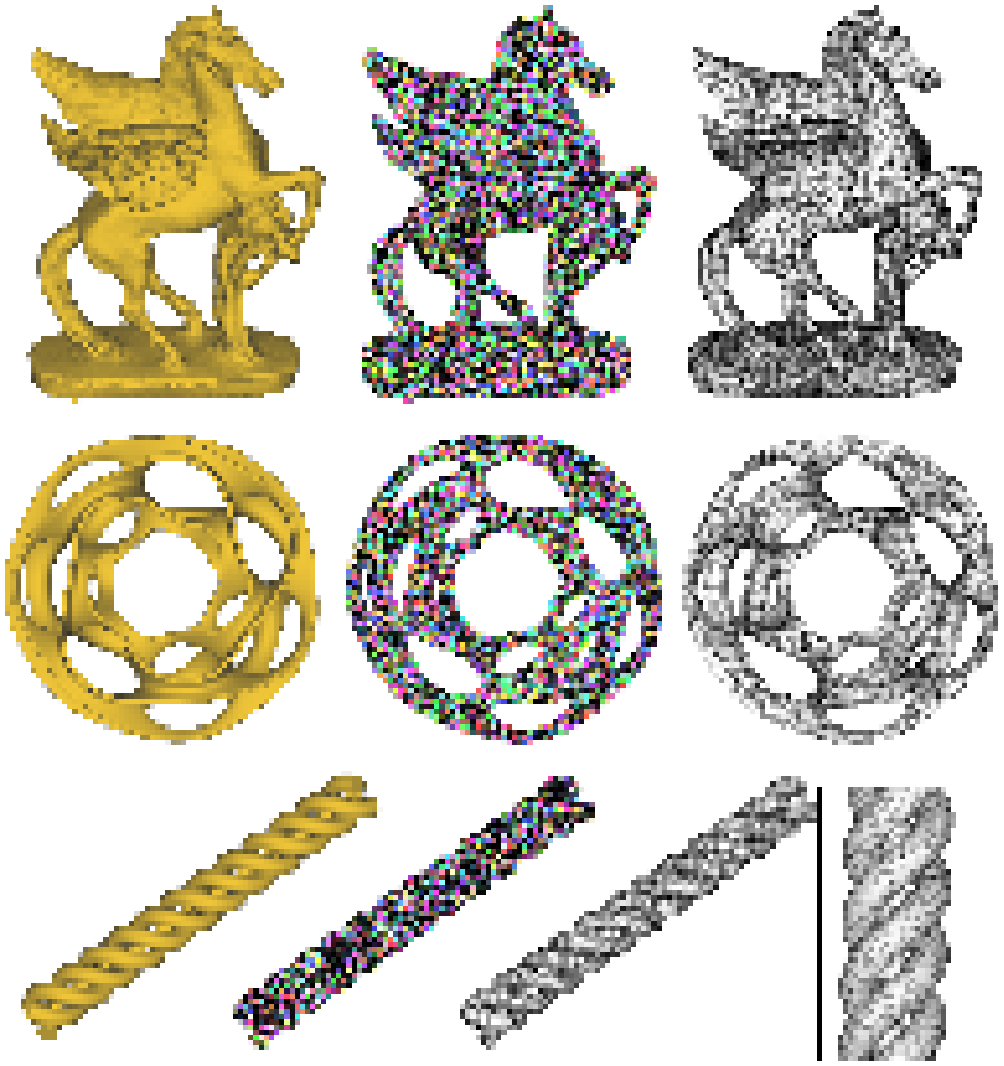


Figure 6.7: Experimental results. Images are rendered in high-resolution, allowing zooming in examination.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

7.1.1 Parallel Computing 2D Voronoi Diagrams Using Sweepcircles

This chapter presents the untransformed sweepcircle algorithm for 2D Voronoi diagram. Our algorithm has the optimal $O(n \log n)$ time complexity and $O(n)$ space complexity. The classical sweep line algorithm is the degenerate form of our algorithm when the circle center is at infinity. Our algorithm is flexible in that it allows multiple circles at arbitrary locations to sweep the domain simultaneously, which naturally leads to a parallel implementation. It is easy to implement, without complicated numerical calculation. We demonstrate the efficacy of our parallel sweep circle algorithm using GPU.

7.1.2 Centroidal Voronoi Tessellation on Arbitrary Triangle Meshes

Although Voronoi diagrams in Euclidean space have been well studied and understood, very little progress has been reported on computing Voronoi diagrams on curved surfaces. In this report, we developed an intrinsic algorithm to compute the geodesic

Voronoi diagram (GVD) on a surface, and used GVD to iteratively compute a geodesic centroid Voronoi diagram on a surface.

7.2 Future Work

Computing intrinsic surface Voronoi Diagram and Centroidal Voronoi Tessellation requires efficient method to obtain geodesic distance. We developed an efficient and parallel method for computing discrete geodesic distance and plan to use it in calculating surface Voronoi, see 7.2.1 for reference. Also, we can compute a Voronoi Diagram in R^3, and calculate the intersection of a 3D Voronoi Diagram and a surface. The result is also a surface CVT. Also using Euclidean distance in 3D is not appropriate because it ignores the boundary condition of the surface. Thus, we need an efficient way to calculate intrinsic distance in R^3, see 7.2.2 for reference.

7.2.1 Fast and Parallel Method for Computing Discrete Geodesic Distance

To compute an intrinsic CVT on surfaces, it is necessary to compute geodesic distances. However, the existing methods for calculating geodesic distance are not fast enough. As a fundamental problem in geometric modeling, computing geodesics distance on triangle meshes has been widely studied. Many elegant algorithms have been presented to compute geodesic distance, e.g.,the CH method [21], the MMP method [97] and the ICH method [148], the fast marching method [72] and the approximate MMP algorithm [133]. While the state-of-the-art approaches [133] [148] work quite well for models of moderate size, they are not practical for large-scale models or time-critical applications due to their high computational cost.

In order to compute geodesic distance field in real time, we need a faster algorithm and a more efficient method. The past decade has witnessed an increasing trend for the traditionally CPU-handled computation to be performed using graphics processing unit (GPU), which uses large numbers of graphics chips to parallelize the computation. However, it is technically challenging to develop parallel algorithms for discrete geodesics due to the lack of parallel structure, i.e., the geodesic distance is propagated from the source to all destination in a sequential order. So far, the only parallel geodesic algorithm is that created by Weber et al. [143], who developed a raster scan-based version of the fast marching algorithm. Although being highly efficient, their method computes only the first-order approximation of geodesic distance and also requires parameterizing the surface into a regular domain, which is usually difficult for surfaces with complicated geometry and/or topology.

We present a new data structure, called the *Saddle Vertex Graph* or SVG, which can be used to compute all types of discrete geodesics. The SVG is an undirected graph that has the same vertex set as the input triangle mesh. Each edge of the SVG corresponds to a geodesic path that does not pass through any saddle vertices. We call such a geodesic path *direct*. The weight of the edge is the length of the corresponding direct geodesic path. The distinctive feature of the SVG is that it elegantly links the geodesic problem on meshes with the shortest path problem on graphs, i.e., any geodesic path in the input mesh is the shortest path in the SVG. As a result, we can apply the classical Dijkstra's shortest path algorithm to compute all three types of discrete geodesics in a unified framework, and avoids complex geometry calculations. Thus we can use this method to compute intrinsic CVT in real time.

7.2.2 Computing Discrete Geodesics Distance in R^3 for a Surface CVT

We can compute a Voronoi Diagram in R^3, and calculate the intersection of a 3D Voronoi Diagram and surface [83]. The result is surface CVT. Using Euclidean distance in 3D is not quite suitable since it does not take the boundary condition into account. Thus, we need an efficient way to calculate intrinsic geodesic distance in R^3, we plan to calculate the geodesic distance in tetrahedral meshes as a robust distance measurement. The distance field in tetrahedral meshes is very attractive in computer graphics and related fields. It is used for multi-body dynamics [59], deformable objects [48], mesh generation [98], motion planning [64], and sculpting [10]. We will use it to calculate a 3D Voronoi Diagram, and use the 3D Voronoi Diagram to intersect with a surface to get a surface CVT.

The conventional fast marching method is tessellation dependent and very sensitive to the topological noise. To tackle these challenges, we present an intrinsic method for computing geodesics on tetrahedral meshes with noises.

We prompt this method by the following intuition: Let $\Omega \subset \mathbb{R}^3$ be a planar domain with noises. Given two points $q, p \in \Omega$, we can calculate the Euclidean distance $d(q,p) = \sqrt{(p_x-q_x)^2+(p_y-q_y)^2+(p_z-q_z)^2}$ easily, but the geodesic path pq may be hindered by the gaps. Put differently, if Ω is planar, the shortest path inside Ω can pass *through* the holes. We can change the classical wavefront algorithm based on this observation. Fast marching method keeps a wave increasing from the origin point to the target point. one can flatten the *spatial* boundary surfaces to the Euclidean space, when the wavefront gets the boundary surfaces. Then we can let the wavefront pass the boundaries by using the Euclidean distance in the tetrahedron mesh. The wavefront keeps increasing up the time when it gets all destinations.

References

[1] CGAL, Computational Geometry Algorithms Library. http://www.cgal.org. xi, 48, 49

[2] G. Albers, L. J. Guibas, J. S. B. Mitchell, and T. Roos. Voronoi diagrams of moving points. *Int. J. Comput. Geometry Appl.*, 8(3):365–380, 1998. 10

[3] P. Alliez, M. Attene, C. Gotsman, and G. Ucelli. Recent advances in remeshing of surfaces. In *Shape Analysis and Structuring*, pages 53–82. Springer, 2008. 6, 14

[4] P. Alliez, É. C. de Verdière, O. Devillers, and M. Isenburg. Isotropic surface remeshing. In *Shape Modeling International*, pages 49–58, 2003. 16

[5] P. Alliez, É. C. de Verdière, O. Devillers, and M. Isenburg. Centroidal voronoi diagrams for isotropic surface remeshing. *Graphical Models*, 67(3):204–231, 2005. 16, 100

[6] P. Alliez, M. Meyer, and M. Desbrun. Interactive geometry remeshing. In *ACM Transactions on Graphics (TOG)*, volume 21, pages 347–354. ACM, 2002. 15

[7] P. Alliez, É. Verdière, O. Devillers, and M. Isenburg. Centroidal voronoi diagrams for isotropic surface remeshing. *Graphical Models*, 67(3):204–231, 2005. 15

[8] F. Aurenhammer. Voronoi diagrams - a survey of a fundamental geometric data structure. *ACM Comput. Surv.*, 23(3):345–405, 1991. 9

[9] D. Avis and B. Bhattacharya. Algorithms for computing d-dimensional Voronoi diagrams and their duals. *Adv. Comput. Res.*, 1:159–180, 1983. 10

[10] J. Bærentzen. *Manipulation of Volumetric Solids with applications to sculpting*. Citeseer, 2002. 149

[11] S. Battiato, A. Milone, and G. Puglisi. Artificial mosaic generation with gradient vector flow and tile cutting. *Journal of Electrical and Computer Engineering*, 2013:8, 2013. 24

[12] A. Belyaev and P.-A. Fayolle. On variational and PDE-based distance function approximations. *Computer Graphics Forum*, 2015. 22

[13] D. P. Bertsekas. A simple and fast label correcting algorithm for shortest paths. *Networks*, 23(8):703–709, 1993. 115

[14] D. P. Bertsekas, F. Guerriero, and R. Musmanno. Parallel asynchronous label-correcting methods for shortest paths. *Journal of Optimization Theory and Applications*, 88(2):297–320, 1996. 115

[15] J. Bowers, R. Wang, L.-Y. Wei, and D. Maletz. Parallel poisson disk sampling with spectrum analysis on surfaces. *ACM Trans. Graph.*, 29:166:1–166:10, 2010. 24, 108

[16] W. Browstow, J.-P. Dussault, and B. L. Fox. Construction of Voronoi polyhedra. *J. Comput. Phys.*, 29:81–97, 1978. 10

[17] M. Campen and L. Kobbelt. Walking on broken mesh: Defect-tolerant geodesic distances and parameterizations. *Comput. Graph. Forum*, 30(2):623–632, 2011. 22

[18] T.-T. Cao, K. Tang, A. Mohamed, and T.-S. Tan. Parallel banding algorithm to compute exact distance transform with the GPU. In *Proceedings of ACM I3D '10*, pages 83–90, 2010. 11, 136, 141

[19] B. Chazelle. An optimal convex hull algorithm and new results on cuttings (extended abstract). In *Proc. 32nd annual symposium on Foundations of computer science*, pages 29–38, 1991. 11

[20] J. Chen, X. Ge, L.-Y. Wei, B. Wang, Y. Wang, H. Wang, Y. Fei, K.-L. Qian, J.-H. Yong, and W. Wang. Bilateral blue noise sampling. *ACM Trans. Graph.*, 32(6):216:1–216:11, 2013. 24, 108

[21] J. Chen and Y. Han. Shortest paths on a polyhedron. In *SCG '90*, pages 360–369, 1990. 18, 57, 58, 147

[22] J. Chen and Y. Han. Shortest paths on a polyhedron. In *Proceedings of the Sixth Annual Symposium on Computational Geometry*, pages 360–369. ACM, 1990. 20

[23] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry II. *Discrete Comput. Geom*, 4:387–421, 1995. 11

[24] M. F. Cohen, J. Shade, S. Hiller, and O. Deussen. Wang tiles for image and texture generation. In *ACM SIGGRAPH 2003 Papers*, pages 287–294, 2003. 23

[25] R. Cole, M. Goodrich, and C. Dunlaing. Merging free trees in parallel for efficient Voronoi diagram construction. Technical report, Johns Hopkins University, 1989. 11

[26] R. L. Cook. Stochastic sampling in computer graphics. *ACM Trans. Graph.*, 5:51–72, 1986. 23

[27] K. Crane, M. Desbrun, and P. Schröder. Trivial connections on discrete surfaces. *Computer Graphics Forum*, 29(5):1525–1533, 2010. xvi, 120, 121

[28] K. Crane, C. Weischedel, and M. Wardetzky. Geodesics in heat: A new approach to computing distance based on heat flow. *ACM Transactions on Graphics*, 32(5):152, 2013. 22

[29] K. Crane, C. Weischedel, and M. Wardetzky. Geodesics in heat: A new approach to computing distance based on heat flow. *ACM Transactions on Graphics*, 2013. 80

[30] T. A. Davis and W. W. Hager. Dynamic supernodes in sparse cholesky update/downdate and triangular solves. *ACM Trans. Math. Softw.*, 35(4):27:1–27:23, Feb. 2009. 83

[31] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, second edition, 2000. 9

[32] F. Dehne and R. Klein. A sweepcircle algorithm for voronoi diagrams. In *Proc. International Workshop on Graphtheoretic Concepts in Computer Science (WG'87)*, volume 314 of *Lecture Notes in Computer Science*, pages 59–70. Springer Verlag, 1987. 2, 5, 10, 26, 54

[33] M. Detrixhe, F. Gibou, and C. Min. A parallel fast sweeping method for the eikonal equation. *J. Comput. Phys.*, 237:46–55, 2013. 22

[34] O. Devillers. Improved incremental randomized delaunay triangulation. In *ACM SoCG*, pages 106–115. ACM, 1998. 10

[35] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959. 75, 114

[36] M. P. do Carmo. *Differential Geometry of Curves and Surfaces*. Prentice-Hall, Inc., 1976. 18

[37] Q. Du, M. Emelianenko, and L. Ju. Convergence of the lloyd algorithm for computing centroidal voronoi tessellations. *SIAM J. Numer. Anal.*, 44(1):102–119, Jan. 2006. 14

[38] Q. Du, V. Faber, and M. Gunzburger. Centroidal Voronoi tessellations: applications and algorithms. *SIAM Rev.*, 41(4):637–676, 1999. 3

[39] Q. Du, V. Faber, and M. Gunzburger. Centroidal voronoi tessellations: Applications and algorithms. *SIAM Rev.*, 41(4):637–676, Dec. 1999. 12, 14, 16, 23, 84, 96

[40] D. Dunbar and G. Humphreys. A spatial data structure for fast Poisson-disk sample generation. *ACM Transactions on Graphics*, 25:503–508, 2006. 23

[41] R. Dwyer. A faster divide-and-conquer algorithm for constructing Delaunay triangulations. *Algorithmica*, 2(1):137–151, 1987. 10

[42] R. Dyer, H. Zhang, and T. Moller. Surface sampling and the intrinsic Voronoi diagram. In *Proceedings of the Symposium on Geometry Processing*, pages 1393–1402, 2008. 12

[43] M. S. Ebeida, A. A. Davidson, A. Patney, P. M. Knupp, S. A. Mitchell, and J. D. Owens. Efficient maximal poisson-disk sampling. *ACM Trans. Graph.*, 30(4):49:1–49:12, Aug. 2011. 23, 108

[44] H. Edelsbrunner and N. Shah. Triangulating topological spaces. *International Journal of Computational Geometry and Applications*, 7(4):365–378, 1997. 12

[45] R. Fattal. Blue-noise point sampling using kernel density model. *ACM Transactions on Graphics*, 30(4):48:1–48:12, 2011. 23

[46] L. Feng, I. Hotz, B. Hamann, and K. Joy. Anisotropic noise samples. *IEEE Transactions on Visualization and Computer Graphics*, 14(2):342–354, Mar. 2008. 24

[47] J. L. Finney. A procedure for the construction of Voronoi polyhedra. *J. Comput. Phys.*, 32:137–143, 1979. 10

[48] S. Fisher and M. Lin. Deformed distance fields for simulation of non-penetrating flexible bodies. *Computer Animation and Simulation 2001*, pages 99–111, 2001. 149

[49] S. Fortune. A sweepline algorithm for Voronoi diagrams. In *SCG '86: Proc. 2nd annual symposium on Computational geometry*, pages 313–322, 1986. viii, 1, 10, 16, 27, 41

[50] P. Foteinos and N. Chrisochoides. Dynamic parallel 3d delaunay triangulation. In *International Meshing Roundtable*, pages 9–26, Paris, France, October 2011. SANDIA. 9

[51] P. Frey and H. Borouchaki. Surface mesh evaluation. In *6th International Meshing Roundtable*, pages 363–374, 1997. 99

[52] Z. Fu, W.-K. Jeong, Y. Pan, R. M. Kirby, and R. T. Whitaker. A fast iterative method for solving the eikonal equation on triangulated surfaces. *SIAM J. Sci. Comput.*, 33(5):2468–2488, 2011. 22

[53] M. N. Gamito and S. C. Maddock. Accurate multidimensional Poisson-disk sampling. *TOG*, 29:8:1–8:19, 2009. 23

[54] A. Genz and R. Cools. An adaptive numerical cubature algorithm for simplices. *ACM Trans. Math. Softw.*, 29(3):297–308, Sept. 2003. 96

[55] A. V. Goldberg and C. Harrelson. Computing the shortest path: A search meets graph theory. In *Proceedings of SODA '05*, pages 156–165, 2005. 76

[56] P. J. Green and R. Sibson. Computing dirichlet tessellations in the plane. *The Computer Journal*, 21(2):168–173, 1978. 10

[57] X. Gu, Y. Wang, T. F. Chan, P. M. Thompson, and S.-T. Yau. Genus zero surface conformal mapping and its application to brain surface mapping. *IEEE Trans. Med. Imaging*, 23(8):949–958, 2004. 17, 49

[58] X. Gu and S.-T. Yau. Global conformal parameterization. In *Symposium on Geometry Processing*, pages 127–137, 2003. 49

[59] E. Guendelman, R. Bridson, and R. Fedkiw. Nonconvex rigid bodies with stacking. *ACM Transactions on Graphics (TOG)*, 22(3):871–878, 2003. 149

[60] L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. In *Proc. 7th international colloquium on Automata, languages and programming*, pages 414–431, 1990. 11

[61] L. J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. In *Proc. 15th annual ACM symposium on Theory of computing*, pages 221–234, 1983. 10

[62] A. Hausner. Simulating decorative mosaics. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 573–580, 2001. 24

[63] K. E. Hoff, III, J. Keyser, M. Lin, D. Manocha, and T. Culver. Fast computation of generalized Voronoi diagrams using graphics hardware. In *Proc. 26th annual conference on Computer graphics and interactive techniques*, pages 277–286, 1999. 11

[64] K. Hoff III, J. Keyser, M. Lin, D. Manocha, and T. Culver. Fast computation of generalized voronoi diagrams using graphics hardware. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 277–286. ACM Press/Addison-Wesley Publishing Co., 1999. 149

[65] M. Iri, K. Murota, and T. Ohya. A fast voronoi-diagram algorithm with applications to geographical optimization problems. In *System Modelling and Optimization*, pages 273–288. Springer, 1984. 96

[66] L. Ju, Q. Du, and M. Gunzburger. Probabilistic methods for centroidal voronoi tessellations and their parallel implementations. *Parallel Computing*, 28(10):1477–1500, 2002. 14

[67] T. Kanai and H. Suzuki. Approximate shortest path on a polyhedral surface and its applications. *Computer-Aided Design*, 33(11):801–811, 2001. 22

[68] S. Kapoor. Efficient computation of geodesic shortest paths. In *Proceedings of the Thirty-first Annual ACM Symposium on Theory of Computing*, pages 770–779, 1999. 21

[69] H. Karcher. Riemannian center of mass and mollifier smoothing. *Comm. Pure Appl. Math*, 30:509–541, 1977. 93

[70] W. Kendall. Probability, convexity, and harmonic maps with small image i: uniqueness and fine existence. *Proc. London Math. Soc.*, 61(2):371–406, 1990. 93

[71] R. Kimmel and J. Sethian. Computing geodesic paths on manifolds. *Proceedings of National Academy of Sciences*, 95:8431–8435, 1998. 22

[72] R. Kimmel and J. A. Sethian. Computing geodesic paths on manifolds. In *Proc. Natl. Acad. Sci. USA*, pages 8431–8435, 1998. 18, 147

[73] V. Klee. On the complexity of d-dimensional Voronoi diagrams. *Archiv. Math.*, 34:75–80, 1980. 10

[74] R. Klein and D. Wood. Voronoi diagrams based on general metrics in the plane. In *Proc. 5th Annual Symposium on Theoretical Aspects of Computer Science*, STACS '88, pages 281–291, 1988. 2, 10

[75] J. Kopf, D. Cohen-Or, O. Deussen, and D. Lischinski. Recursive Wang tiles for real-time blue noise. In *SIGGRAPH '06*, pages 509–518, 2006. 23

[76] R. Kunze, F. Wolter, and T. Rausch. Geodesic voronoi diagrams on parametric surfaces. In *Computer Graphics International, 1997. Proceedings*, pages 230–237. IEEE, 1997. 14

[77] A. Lagae and P. Dutré. A procedural object distribution function. *ACM Trans. Graph.*, 24:1442–1461, 2005. 23

[78] B. Lévy and N. Bonneel. Variational anisotropic surface meshing with Voronoi parallel linear enumeration. In *Proceedings of the 21st International Meshing Roundtable*, pages 349–366, 2013. 18

[79] B. Lévy and Y. Liu. Lp centroidal voronoi tessellation and its applications. *ACM Trans. Graph.*, 29(4):119:1–119:11, July 2010. 14

[80] H. Li, L.-Y. Wei, P. V. Sander, and C.-W. Fu. Anisotropic blue noise sampling. *ACM Transactions on Graphics*, 29(6):167, 2010. xvii, 23, 24, 109, 115, 127, 130

[81] D. Liu and J. Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical programming*, 45(1):503–528, 1989. 14

[82] Y. Liu, Z. Chen, and K. Tang. Construction of iso-contours, bisectors and voronoi diagrams on triangulated surfaces. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(8):1502–1517, 2011. 12, 92

[83] Y. Liu, W. Wang, B. Lévy, F. Sun, D.-M. Yan, L. Lu, and C. Yang. On centroidal Voronoi tessellation - energy smoothness and fast computation. *ACM Trans. Graph.*, 28(4):101:1–101:17, 2009. viii, 3, 4, 7, 14, 16, 96, 149

[84] Y.-J. Liu. Exact geodesic metric in 2-manifold triangle meshes using edge-based data structures. *Computer-Aided Design*, 45(3):695–704, 2013. 12, 20

[85] Y.-J. Liu. Exact geodesic metric in 2-manifold triangle meshes using edge-based data structures. *Computer-Aided Design*, 45(3):695–704, 2013. 18

[86] Y.-J. Liu, C.-X. Xu, Y. He, and D.-S. Kim. The duality of geodesic Voronoi/Delaunay diagrams for an intrinsic discrete Laplace-Beltrami operator on simplicial surfaces. In *Proceedings of the 26th Canadian Conference on Computational Geometry (CCCG '14)*, 2014. 12

[87] Y.-J. Liu, Q.-Y. Zhou, and S.-M. Hu. Handling degenerate cases in exact geodesic computation on triangle meshes. *The Visual Computer*, 23(9-11):661–668, 2007. 18

[88] Y.-J. Liu, Q.-Y. Zhou, and S.-M. Hu. Handling degenerate cases in exact geodesic computation on triangle meshes. *The Visual Computer*, 23(9-11):661–668, 2007. 20

[89] S. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129 – 137, 1982. 4, 14, 16

[90] T. O. M. Tanemura and N. Ogita. A new algorithm for three-dimensional Voronoi tessellation. *J. Comput. Phys.*, 32:137–143, 1983. 10

[91] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In L. M. L. Cam and J. Neyman, editors, *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. University of California Press, 1967. 12

[92] A. McKenzie, S. V. Lombeyda, and M. Desbrun. Vector field analysis and visualization through variational clustering. In *Eurographics / IEEE VGTC Symposium on Visualization*, pages 29–35, 2005. 1

[93] N. Megiddo. Linear-time algorithms for linear programming in R3 and related problems. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, pages 329–338, 1982. 122

[94] F. Mémoli and G. Sapiro. Fast computation of weighted distance functions and geodesics on implicit hyper-surfaces. *Journal of Computational Physics*, 173(2):730–764, 2001. 22

[95] D. P. Mitchell. Spectrally optimal sampling for distribution ray tracing. In *SIGGRAPH '91*, pages 157–164, 1991. 23

[96] J. S. Mitchell, D. M. Mount, and C. H. Papadimitriou. The discrete geodesic problem. *SIAM Journal on Computing*, 16(4):647–668, 1987. 20

[97] J. S. B. Mitchell, D. M. Mount, and C. H. Papadimitriou. The discrete geodesic problem. *SIAM J. Comput.*, 16(4):647–668, 1987. 18, 57, 58, 61, 79, 111, 147

[98] N. Molino, R. Bridson, J. Teran, and R. Fedkiw. A crystalline, red green strategy for meshing highly deformable objects with tetrahedra. 149

[99] D. Nehab and P. Shilane. Stratified point sampling of 3D models. In *PBG '04*, pages 49–56, 2004. 23

[100] M. Novotni and R. Klein. Computing geodesic distances on triangular meshes. In *Proc. WSCG '02*, 2002. 82

[101] A. Okabe, B. Boots, K. Sugihara, and S. N. Chiu. *Spatial tessellations: Concepts and applications of Voronoi diagrams*. Wiley, 2nd edition, 2000. 3, 9

[102] R. Osada, T. Funkhouser, B. Chazelle, and D. Dobkin. Shape distributions. *ACM Trans. Graph.*, 21:807–832, 2002. 119

[103] V. Ostromoukhov. Sampling with polyominoes. *ACM Trans. Graph.*, 26, 2007. 23

[104] A. C. Öztireli, M. Alexa, and M. H. Gross. Spectral sampling of manifolds. *ACM Trans. Graph.*, 29(6):168, 2010. 23

[105] X. Pennec. Intrinsic statistics on riemannian manifolds: Basic tools for geometric measurements. *Journal of Mathematical Imaging and Vision*, 25(1):127–154, 2006. 93, 94

[106] G. Peyré and L. Cohen. Geodesic remeshing using front propagation. *International Journal of Computer Vision*, 69(1):145–156, 2006. 14

[107] I. Pohl. Bi-directional search. *Machine Intelligence*, 6:124–140, 1971. 76

[108] K. Polthier and M. Schmies. chapter Straightest Geodesics on Polyhedral Surfaces. 61

[109] H. P.S. and G. M. Survey of surface simplification algorithms. Technical report, Carnegie Mellon University - Dept. of Computer Science, 1997. 6, 14

[110] S. Rajasekaran and S. Ramaswami. Optimal parallel randomized algorithms for the Voronoi diagram of line segments in the plane and related problems. In *Symposium on Computational Geometry*, pages 57–66, 1994. 11

[111] G. Rong, M. Jin, L. Shuai, and X. Guo. Centroidal voronoi tessellation in universal covering space of manifold surfaces. *Comput. Aided Geom. Des.*, 28(8):475–496, Nov. 2011. xiv, 7, 14, 17, 100, 104

[112] G. Rong, Y. Liu, W. Wang, X. Yin, X. Gu, and X. Guo. GPU-assisted computation of centroidal Voronoi tessellation. *IEEE Transactions on Visualization and Computer Graphics*, 17(3):345–356, 2011. 11, 17, 100

[113] R. M. Rustamov. Barycentric coordinates on surfaces. *Comput. Graph. Forum*, 29(5):1507–1516, 2010. 93

[114] R. Schmidt. Stroke parameterization. *Computer Graphics Forum*, 32(2pt2):255–263, 2013. 117

[115] R. Schmidt, C. Grimm, and B. Wyvill. Interactive decal compositing with discrete exponential maps. In *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, pages 605–613, 2006. 7

[116] R. Schmidt, C. Grimm, and B. Wyvill. Interactive decal compositing with discrete exponential maps. *ACM Trans. Graph.*, 25(3):605–613, July 2006. 19, 116

[117] Y. Schreiber. An optimal-time algorithm for shortest paths on realistic polyhedra. *Discrete & Computational Geometry*, 43(1):21–53, 2010. 21

[118] Y. Schreiber and M. Sharir. An optimal-time algorithm for shortest paths on a convex polytope in three dimensions. In *Proc. SCG '06*, pages 30–39, 2006. 82

[119] Y. Schreiber and M. Sharir. An optimal-time algorithm for shortest paths on a convex polytope in three dimensions. *Discrete & Computational Geometry*, 39(1-3):500–579, 2008. 21

[120] R. Seidel. Small-dimensional linear programming and convex hulls made easy. *Discrete Comput. Geom.*, 6:423–434, 1991. 11

[121] J. Sethian. A fast marching level set method for monotonically advancing fronts. *Proceedings of National Academy of Sciences*, 93:1591–1595, 1996. 21

[122] J. Sethian and A. Vladimirsky. Fast methods for the Eikonal and related Hamilton-Jacobi equations on unstructured meshes. *Proceedings of National Academy of Sciences*, 97:5699–5703, 2000. 22

[123] M. I. Shamos and D. Hoey. Closest-point problems. In *FOCS*, pages 151–162, 1975. 10, 16

[124] J. Shewchuk. Delaunay refinement algorithms for triangular mesh generation. *Computational geometry*, 22(1):21–74, 2002. 9

[125] J. R. Shewchuk. Triangle: Engineering a 2d quality mesh generator and delaunay triangulator. FCRC '96/WACG '96, pages 203–222, 1996. xi, 49

[126] J. R. Shewchuk. Triangle: Engineering a 2d quality mesh generator and delaunay triangulator. In *Applied COMPUTATIONAL Geormetry, Towards Geometric Engineering, FCRC'96 Workshop, WACG'96, Philadelphia, PA, May 27-28, 1996, Selected Papers*, pages 203–222, 1996. 48

[127] L. Shuai, X. Guo, and M. Jin. Gpu-based computation of discrete periodic centroidal voronoi tessellation in hyperbolic space. *Computer-Aided Design*, 45(2):463–472, 2013. 88

[128] S. W. Sloan. A fast algorithm for constructing Delaunay triangulations in the plane. *Adv. Eng. Softw.*, 9:34–55, 1987. 10

[129] A. Spira and R. Kimmel. An efficient solution to the Eikonal equation on parametric manifolds. *Interfaces and Free Boundaries*, 6(4):315–327, 2004. 22

[130] A. Sud, N. Govindaraju, R. Gayle, I. Kabul, and D. Manocha. Fast proximity computation among deformable models using discrete Voronoi diagrams. *ACM Trans. Graph.*, 25:1144–1153, 2006. 1, 11

[131] Q. Sun, L. Zhang, M. Zhang, X. Ying, S.-Q. Xin, J. Xia, and Y. He. Texture brush: An interactive surface texturing interface. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '13, pages 153–160, New York, NY, USA, 2013. ACM. xvi, 19, 116, 117, 118

[132] X. Sun, K. Zhou, J. Guo, G. Xie, J. Pan, W. Wang, and B. Guo. Line segment sampling with blue-noise properties. *ACM Trans. Graph.*, 32(4):127:1–127:14, July 2013. 24

[133] V. Surazhsky, T. Surazhsky, D. Kirsanov, S. J. Gortler, and H. Hoppe. Fast exact and approximate geodesics on meshes. *ACM Trans. Graph.*, 24(3):553–560, 2005. 18, 58, 76, 147

[134] V. Surazhsky, T. Surazhsky, D. Kirsanov, S. J. Gortler, and H. Hoppe. Fast exact and approximate geodesics on meshes. *ACM Trans. Graph.*, 24(3):553–560, 2005. 20, 21

[135] J. O. Talton. A short survey of mesh simplification algorithm. Technical report, University of Illinois at Urbana-Champaign, 2004. 6, 14

[136] Thurston. The geometry of circles: Voronoi diagrams, moebius transformations, convex hulls, fortune's algorithm, the cut locus and parametrization of shapes. Unpublished notes, Princeton, 1986. 2, 54

[137] J. Tsitsiklis. Efficient algorithms for globally optimal trajectories. *IEEE Transactions on Automatic Control*, 40(9):1528–1538, 1995. 21

[138] G. Turk. Re-tiling polygonal surfaces. In *SIGGRAPH '92*, pages 55–64, 1992. 23

[139] S. Valette, J. Chassery, and R. Prost. Generic remeshing of 3d triangular meshes with metric-dependent discrete voronoi diagrams. *Visualization and Computer Graphics, IEEE Transactions on*, 14(2):369–381, 2008. 15

[140] J. Wang, L. Ju, and X. Wang. Image segmentation using local variation and edge-weighted centroidal Voronoi tessellations. *IEEE Transactions on Image Processing*, 20(11):3242–3256, 2011. 1

[141] X. Wang, X. Ying, Y.-J. Liu, S.-Q. Xin, W. Wang, X. Gu, W. Mueller-Wittig, and Y. He. Intrinsic computation of centroidal Voronoi tessellation (CVT) on meshes. *Computer-Aided Design*, 58:51–61, 2015. xvii, 116, 142, 143

[142] D. F. Watson. Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes. *The Computer Journal*, 24(2):167–172, 1981. 10

[143] O. Weber, Y. Devir, A. Bronstein, M. Bronstein, and R. Kimmel. Parallel algorithms for approximation of distance maps on parametric surfaces. *ACM Transactions on Graphics (TOG)*, 27(4):104, 2008. 148

[144] O. Weber, Y. S. Devir, A. M. Bronstein, M. M. Bronstein, and R. Kimmel. Parallel algorithms for approximation of distance maps on parametric surfaces. *ACM Trans. Graph.*, 27(4):104:1–104:16, 2008. 22

[145] L.-Y. Wei. Parallel poisson disk sampling. In *ACM SIGGRAPH 2008 papers*, SIGGRAPH '08, pages 20:1–20:9, 2008. 23, 108, 109, 128

[146] L.-Y. Wei. Multi-class blue noise sampling. *ACM Trans. Graph.*, 29, 2010. 24, 108, 124

[147] S.-Q. Xin, Y. He, and C.-W. Fu. Efficiently computing exact geodesic loops within finite steps. *IEEE Trans. Vis. Comput. Graph.*, 18(6):879–889, 2012. 23

[148] S.-Q. Xin and G.-J. Wang. Improving Chen and Han's algorithm on the discrete geodesic problem. *ACM Trans. Graph.*, 28(4):104:1–104:8, 2009. 18, 79, 90, 96, 97, 111, 147

[149] S.-Q. Xin and G.-J. Wang. Improving Chen and Han's algorithm on the discrete geodesic problem. *ACM Transactions on Graphics*, 28(4):104, 2009. 20, 21

[150] S.-Q. Xin, X. Ying, and Y. He. Constant-time all-pairs geodesic distance query on triangle meshes. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 31–38, 2012. 113, 115, 127

[151] C.-X. Xu, T. Y. Wang, Y.-J. Liu, L. Liu, and Y. He. Fast wavefront propagation (FWP) for computing exact geodesic distances on meshes. *IEEE Transactions on Visualization and Computer Graphics*, 2015. 20, 111

[152] D.-M. Yan, B. Lévy, Y. Liu, F. Sun, and W. Wang. Isotropic remeshing with fast and exact computation of restricted Voronoi diagram. *Comput. Graph. Forum*, 28(5):1445–1454, 2009. xiv, xvii, 1, 17, 88, 99, 100, 103, 105, 133, 143

[153] D.-M. Yan, B. Lévy, Y. Liu, F. Sun, and W. Wang. Isotropic remeshing with fast and exact computation of restricted voronoi diagram. In *Proceedings of the Symposium on Geometry Processing*, pages 1445–1454, 2009. 4

[154] D.-M. Yan, W. Wang, B. Lévy, and Y. Liu. Efficient computation of 3d clipped Voronoi diagram. In *Geometric Modeling and Processing (GMP)*, pages 269–282, 2010. xiv, 10, 100, 104

[155] D.-M. Yan and P. Wonka. Gap processing for adaptive maximal Poisson-disk sampling. *ACM Trans. Graph.*, 32(5):148:1–148:15, 2013. 108

[156] X. Ying, X. Wang, and Y. He. Saddle vertex graph (SVG): A novel solution to the discrete geodesic problem. *ACM Trans. Graph.*, 32(6):170:1–170:12, 2013. 19, 111, 115, 117, 127

[157] X. Ying, S.-Q. Xin, and Y. He. Parallel Chen-Han (PCH) algorithm for discrete geodesics. *ACM Transactions on Graphics*, 33(1):9:1–9:11, 2014. 18, 20

[158] X. Ying, S.-Q. Xin, Q. Sun, and Y. He. An intrinsic algorithm for parallel Poisson disk sampling on arbitrary surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 19(9):1425–1437, 2013. 19, 108, 116, 119

[159] H. Zhao. Parallel implementations of the fast sweeping method. *Journal of Computational Mathematics*, 25(4):421–429, 2007. 22

[160] Y. Zhou, F. Sun, W. Wang, J. Wang, and C. Zhang. Fast updating of Delaunay triangulation of moving points by bi-cell filtering. *Comput. Graph. Forum*, 29(7):2233–2242, 2010. 10