

NANYANG TECHNOLOGICAL UNIVERSITY



# **Intrinsic Computation of Voronoi Diagrams on Surfaces and Its Application**

A thesis submitted  
for a degree of Doctor of Philosophy  
by

**Wang Xiaoning**

School of Computer Engineering  
Nanyang Technological University

Aug, 2015

# Abstract

The Voronoi diagram is a decomposition of a space, determined by distances to a given set of objects. As a fundamental geometric data structure, Voronoi diagrams have been widely applied in various computer graphics and visualization applications. While there is a popular Fortune's sweep line algorithm, there is no effective technique to parallelize it.

Centroidal Voronoi tessellation (CVT) is a special type of Voronoi diagram such that the generating point of each Voronoi cell is also its center of mass. The CVT has broad applications in computer graphics, such as meshing, stippling, sampling, etc. The existing methods for computing CVTs on meshes either require a global parameterization or compute it in the restricted sense (that is, intersecting a 3D CVT with the surface). Therefore, these approaches often fail on models with complicated geometry and/or topology. To get intrinsic CVT on meshes, we need to compute discrete geodesic efficiently. The existing exact discrete geodesic algorithms are relatively slow while the approximate geodesic methods suffer from numerical problems and are sensitive to mesh tessellation. Thus we need an efficient and accurate method to compute discrete geodesics.

To tackle these issues, we studied CVT and discrete geodesic problem systematically. We present a sweep circle algorithm for construct Voronoi diagrams in parallel and the Saddle Vertex Graph (SVG) to efficiently compute discrete geodesics on meshes. Combined discrete geodesic with CVT, we present two efficient algorithms for computing intrinsic CVT on meshes. Furthermore, we propose one application of CVT and discrete geodesics, anisotropic shape distribution on meshes. Our contributions are listed as follow:

We first present a new algorithm, called sweep circle, for constructing Voronoi diagram. Starting with a degenerate circle centered at an arbitrary location, our algorithm sweeps the circle by increasing its radius across the plane. Our sweep circle algorithm is flexible in allowing multiple circles at arbitrary locations to sweep the domain simultaneously, thus is parallel in nature. We demonstrate the efficacy of our parallel sweep circle algorithm using GPU.

Second, we propose the Saddle Vertex Graph (SVG) to efficiently compute discrete geodesics on triangle meshes. The SVG is a sparse undirected graph that encodes complete geodesic distance information, which can be solved efficiently using the shortest path algorithm (e.g., Dijkstra algorithm). It does not require any numerical solver, and is numerically stable and insensitive to the mesh resolution and tessellation. The experimental results on real-world models demonstrate significant improvement to the existing approximate geodesic methods in terms of both performance and accuracy.

Thirdly, we introduce two intrinsic algorithms for computing CVT on triangle meshes. The first algorithm adopts the Lloyd framework, which iteratively moves the generator of each geodesic Voronoi diagram to its mass center. The second algorithm uses the L-BFGS method to accelerate the intrinsic CVT computation. Thanks to the intrinsic nature, our methods work well for models with arbitrary topology and complicated geometry.

Finally, we show the shape distribution application. We demonstrate 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. Our method does not require global parameterization of the input 3D mesh. Instead, it computes local parameterization on the fly using geodesic polar coordinates. Thanks to the SVG for solving geodesic computation problem efficiently, the local parameterization can be computed at little cost.

# Acknowledgments

Many people have helped 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 is 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 helped me a lot. Without the collaboration and support, this thesis would not be in this form. 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, Mr. Tan Fuwen, Dr. Sun Qian, Dr. Wang Dayong, Ms. Wang Jin, Mr. Wu Jiajun, Dr. Wu Xiaoqun, Dr. Xin Shiqing, Ms. Yan Xiaoqi, Dr. Ying Xiang, Dr. Zhang Minqi.

I want to thank my parents for their support during the study. And 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.

## To be submitted

**Xiaoning Wang**, Le Tien Hung, Qian Sun, Xiang Ying, Wolfgang Mueller-witting, Ying He. User Controllable Anisotropic Shape Distribution on 3D Meshes.

# Contents

<b>Abstract</b> . . . . .	i
<b>Acknowledgments</b> . . . . .	iii
<b>Publications</b> . . . . .	iv
<b>List of Figures</b> . . . . .	viii
<b>List of Tables</b> . . . . .	xi
<b>1 Introduction</b>	<b>1</b>
1.1 Problem Statement . . . . .	1
1.1.1 Voronoi Diagrams . . . . .	1
1.1.2 Discrete Geodesics . . . . .	2
1.1.3 Intrinsic Centroidal Voronoi Tessellation . . . . .	3
1.2 Objectives . . . . .	5
1.3 Thesis Organization . . . . .	9
<b>2 Related Work</b>	<b>10</b>
2.1 Voronoi Diagrams . . . . .	10
2.2 Centroidal Voronoi Tessellation . . . . .	13
2.3 Discrete Geodesics . . . . .	17
2.3.1 Discrete Wavefront Propagation Methods . . . . .	17
2.3.2 PDE Methods . . . . .	18
2.3.3 Other Methods . . . . .	20
2.4 Exponential Map . . . . .	20
2.5 Sampling and Shape Distribution . . . . .	22

<b>3</b>	<b>Parallel Computing 2D Voronoi Diagrams Using Untransformed Sweep Circles</b>	<b>24</b>
3.1	Preliminaries . . . . .	25
3.2	Untransformed Sweep Circle Algorithm . . . . .	29
3.2.1	Algorithm Description . . . . .	29
3.2.2	Correctness . . . . .	34
3.2.3	Complexity . . . . .	41
3.3	Parallel Sweep Circles . . . . .	42
3.4	Experimental Results . . . . .	45
3.5	Discussion . . . . .	50
<b>4</b>	<b>Saddle Vertex Graph for Computing Discrete Geodesics</b>	<b>53</b>
4.1	Preliminary . . . . .	54
4.2	Saddle Vertex Graph . . . . .	57
4.2.1	Definition . . . . .	57
4.2.2	Data Structure . . . . .	59
4.3	SVG Construction . . . . .	60
4.3.1	Parameter . . . . .	61
4.3.2	Computing Direct Geodesic Paths . . . . .	62
4.3.3	Algorithm . . . . .	62
4.4	Computing Discrete Geodesics Using SVG . . . . .	64
4.4.1	Dijkstra’s Algorithm and Label Correcting Algorithm . . . . .	64
4.4.2	Single-source Single-destination (SSSD) . . . . .	66
4.4.3	Multiple-sources All-destinations (MSAD) . . . . .	67
4.4.4	Discrete Geodesic Distances Between Arbitrary Points . . . . .	69
4.5	Experimental Results . . . . .	72
<b>5</b>	<b>Intrinsic Computation of Centroidal Voronoi Tessellation (CVT) on Meshes</b>	<b>77</b>
5.1	The Lloyd Framework . . . . .	78
5.1.1	Overview . . . . .	78
5.1.2	Computing the Geodesic Voronoi Diagrams . . . . .	81
5.1.3	Computing the Riemannian Center . . . . .	81

5.1.4	Computing the Center of Mass . . . . .	83
5.2	The L-BFGS Framework . . . . .	85
5.3	Experimental Results . . . . .	86
5.3.1	Implementation . . . . .	86
5.3.2	Results & Comparison . . . . .	88
5.3.3	Limitation . . . . .	91
<b>6</b>	<b>Application</b>	<b>95</b>
6.1	Geodesic Polar Coordinates . . . . .	96
6.2	Shape Distribution Algorithm . . . . .	99
6.2.1	Single-class Shape Distribution . . . . .	102
6.2.2	Multi-Class Shape Distribution . . . . .	106
6.3	Experimental Results . . . . .	106
<b>7</b>	<b>Conclusions and Future Work</b>	<b>111</b>
7.1	Conclusions . . . . .	111
7.2	Future Work . . . . .	113
	<b>References</b>	<b>115</b>

# List of Figures

1.1	An ordinary Voronoi tessellation and a CVT of the same domain . . . . .	4
2.1	Bisector and geodesic Voronoi diagram. . . . .	14
2.2	Exponential map and geodesic polar coordinate system. . . . .	21
3.1	An example for the Voronoi diagram and its dual Delaunay triangulation and Fortune’s sweep line algorithm. . . . .	26
3.2	Using the sweep line algorithm needs a much larger domain. . . . .	28
3.3	Illustration of sweep circles for two sites. . . . .	31
3.4	Applying the sweep circle algorithm to a domain with five sites. . . . .	31
3.5	Out of date events. . . . .	36
3.6	The beach curve. . . . .	44
3.7	Comparison of the sequential sweep circle and sweep line algorithms. . . . .	46
3.8	The comparisons with CGAL [1] and Triangle [102]. . . . .	47
3.9	Parallel computing Voronoi diagram using GPU based sweep circle al- gorithm. . . . .	48
3.10	3D models and their parameterization. . . . .	48
3.11	Our method can be applied to 3D surfaces by using parameterization. . . . .	49
3.12	Computing the additively weighted Voronoi diagram. . . . .	50
4.1	Construction of an approximate SVG for the Dragon model. . . . .	54
4.2	Saddle vertex and geodesic path. . . . .	56
4.3	The SVG on the 9K-face Bimba. . . . .	59
4.4	The MSAD algorithm allows us to compute the geodesic Voronoi dia- gram and the geodesic distances to curve sources. . . . .	68

4.5	Computing the single-source all-destinations geodesic distances on a low-resolution mesh. . . . .	70
4.6	Visual comparison of the accuracy. . . . .	74
4.7	The parameter $K$ of the SVG controls the complexity and accuracy of the approximate geodesic distance. . . . .	74
4.8	The SVG method computes the geodesic path by unfolding a sequence of triangles. . . . .	75
4.9	Experimental results. . . . .	76
5.1	Our intrinsic method can compute a high-quality centroidal Voronoi tessellation on model with complicated geometry and topology. . . . .	78
5.2	Iteratively computing geodesic CVT on meshes. . . . .	80
5.3	The multiple-source geodesic distance field induces a geodesic Voronoi diagram. . . . .	82
5.4	Computing the center of mass for the Voronoi cell $\Omega_i$ . . . . .	84
5.5	Convergence rate comparison between the Lloyd method and the L-BFGS method. . . . .	87
5.6	Energy function and quality measures. . . . .	89
5.7	Intrinsic vs. extrinsic. . . . .	90
5.8	our method can produce consistent results on the various poses of the Lion model. . . . .	90
5.9	The intrinsic CVT and extrinsic RVD with very few sites. . . . .	91
5.10	Experimental results. . . . .	92
5.11	Experimental results on high-genus models. . . . .	93
5.12	Comparison to the RVD method [125] and the UCS method [89]. . . . .	94
6.1	Example of shape distribution on the Fertility model. . . . .	96
6.2	Geodesic polar coordinates. . . . .	97
6.3	Comparison with extended exponential map [107]. . . . .	99
6.4	Algorithmic pipeline of single-class shape distribution. . . . .	100
6.5	For long shapes, we divide it to several parts and test the intersection separately. . . . .	102

6.6	Comparison with Poisson disk sampling. . . . .	103
6.7	Collision illustration. . . . .	104
6.8	Single-class shape distribution on the Kitten model with different density and texture. . . . .	108
6.9	Multi-class shape distribution on a bumpy sphere with 50K faces. . . . .	108
6.10	Our method works well on the Bimba model with various resolutions. . . . .	109
6.11	Comparison to [65]. . . . .	110
6.12	More results. . . . .	110

# List of Tables

4.1	Time and space complexities. . . . .	65
4.2	Statistics of speed and accuracy. . . . .	73
5.1	Statistics of the mesh complexity and the timing. . . . .	88
6.1	Performance of geodesic algorithms. . . . .	107
6.2	Mesh complexity and running time performance. . . . .	107

# Chapter 1

## Introduction

### 1.1 Problem Statement

#### 1.1.1 Voronoi Diagrams

The 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 [106], meshing [124], vector field visualization [75], artistic effect, image segmentation [114], just to name a few. Among many efficient algorithms for the construction of Voronoi diagram, Fortune's sweep line algorithm [39] is popular due to its elegance 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. During the algorithm, on the left side of the sweep line, the input sites have already been inserted into the Voronoi diagram, and on the right side the sites have not been inserted yet. Each site to the left of the sweep line generates a parabola of points equidistant from the sweep line and that point. The wavefronts of all parabolas form a curved beach line, which follows the sweep line moving from left to right. As the sweep line moves, the beach line cuts out

the Voronoi diagram's edges. The sweep line algorithm is proven to be optimal with  $O(n)$  space complexity and  $O(n \log n)$  time complexity. In spite of its simplicity and popularity, there is no effective technique to parallelize the sweep line algorithm.

### 1.1.2 Discrete Geodesics

The discrete geodesic problem plays an important role in compute graphics. In 1987, Mitchell, Mount and Papadimitriou published the MMP algorithm [79], which computes the single-source geodesic distance in  $O(n^2 \log n)$  time, where  $n$  is the number of vertices. Then in 1990, Chen and Han [16] improved the time complexity to  $O(n^2)$ , which remains the best 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 ICH and MMP 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. [79] and Chen and Han [16] noted that the number of windows is  $O(n^2)$ , which means that none of the window propagation algorithms can break the theoretical  $O(n^2)$  time barrier. On the other hand, Surazhsky et al. [109] 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. [109] 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.

### 1.1.3 Intrinsic Centroidal Voronoi Tessellation

The centroidal Voronoi tessellation (CVT) is a particular Voronoi tessellation of a compact domain yielded by a set of samples such that each site locates at the same place with the centroid of its Voronoi cell. For example, Figure 1.1(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 Figure 1.1(b) shows a CVT of the same domain. CVT generates an evenly spaced distribution of sites in the domain 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. [30] and Okabe et al. [81].

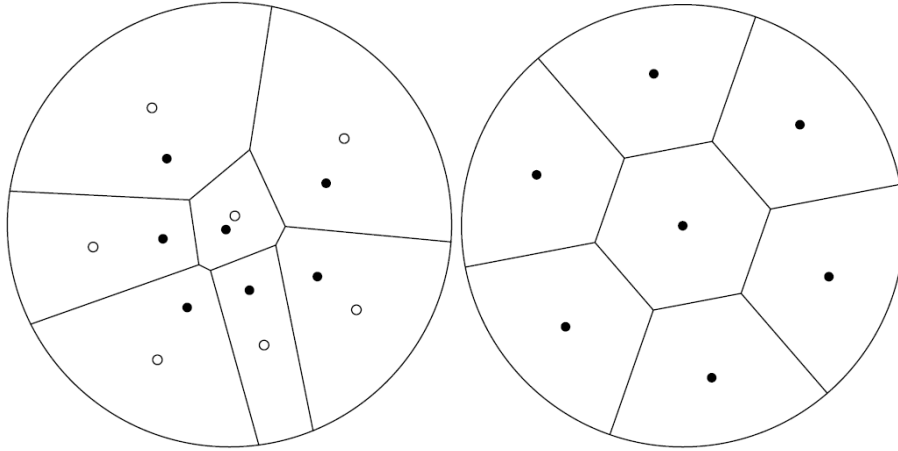


Figure 1.1: (a) An ordinary Voronoi tessellation of a circular domain with seven sites marked with black dots and the centroids of the Voronoi cells marked with small circles; (b) a CVT of the same domain with seven sites. Figures are taken from [68].

Although the CVT in Euclidean space has been extensively studied, relatively little progress has been reported towards computing the CVT on curved surfaces. A common step in computing the CVT is to construct the Voronoi diagrams in each iteration, while an alternative is to directly update the sites instead of really reconstruct the whole VD in each iteration [99], [100]. 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 this challenge by computing the restricted Voronoi diagrams [124], 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 [103]. 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.

## 1.2 Objectives

In this thesis we aim to find a fully parallel algorithm for calculating Voronoi Diagrams, and for efficiently computing intrinsic CVT on the surface, we want a fast and accurate algorithm for computing discrete geodesics. Also we demonstrate one applications of the CVT and discrete geodesics, anisotropic shape distribution.

**Fast Parallel Algorithm for Voronoi Diagrams.** We tackle the challenge of fast computing a Voronoi Diagram by proposing a new algorithm, called the *untransformed sweep circle*, for computing Voronoi diagrams. 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 [23], 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.

**Efficient Algorithm for the Discrete Geodesic Problem.** 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 thesis, 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 vertex. 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).

**Intrinsic Centroidal Voronoi Tessellation.** We present 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. 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.

**Application.** We demonstrate one application of CVT and discrete geodesics, shape distribution on the surface.

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. 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 compared to isotropic sampling on 3D models. We propose 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 [10, 118], 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 collision-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 SVG method for 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.

## 1.3 Thesis Organization

The remainder of this report is organized into following parts: Chapter 2 reviews the related work of Voronoi diagrams, discrete geodesic, CVT and some related techniques. Then in Chapter 3 we introduce the sweep circle algorithm. In Chapter 4 we present the Saddle Vertex Graph (SVG) to efficiently solve the discrete geodesic problem and in Chapter 5 we use discrete geodesics to compute intrinsic CVT on meshes. We demonstrate one application in Chapter 6, an automatic method for computing anisotropic 2D shape distribution on the mesh. Lastly, we conclude our work and provide some future directions in Chapter 7.

# Chapter 2

## Related Work

In this chapter, we mainly review the existing research work related to Voronoi diagram, centroidal Voronoi tessellation(CVT), and discrete geodesics. Discrete geodesics is closely related to intrinsic computation of Voronoi diagram and CVT on the surface. Also we review some literature in exponential map and shape distribution.

### 2.1 Voronoi Diagrams

Voronoi diagrams are a fundamental geometric structure, which can be dated back to 1644. It is named after Georgy Fedoseevich Voronoi in 1908. He is a Russian mathematician, who studied and defined the general  $n$ -dimensional case. 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. [4, 81], due to its nice geometric properties.

As the dual of Voronoi diagrams, Delaunay triangulations also have many useful properties. For example, in all triangulations of a planar point set the Delaunay triangulation maximizes the minimum angle, which is helpful for quality mesh generation [101, 40].

There are a large array of literature on the Delaunay triangulation problem. For example, Devillers [25] proposed a hierarchical 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 [60]. Among these algorithms, some are in a divide-and-conquer scheme [98, 49, 32], while others are incremental [45, 104]. The same optimal worst-case bound is also obtained by Fortune's elegant sweep line algorithm [39] and the traditional sweep circle algorithm [23]. Dinis and Mamede [27] extends the sweep line algorithm to spherical domain by sweeping the sphere with a circular line. Their approach uses sweep circle on the sphere during propagation based on closed form formula, while our sweep circle is more generalized and can be computed on arbitrary 3D surfaces.

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. [130] 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. [11] 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 [38] proposed an algorithm to compute the vertices of each Voronoi region, and then to compute the edges and facets of the region by identifying the sites that define each vertex. Tanemura et al. [73] 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. [125] presented an efficient algorithm for computing the 3D clipped Voronoi diagram.

There are also some algorithms [45, 116, 5] available for  $d$ -dimensional Voronoi diagrams ( $d \geq 3$ ). The combinatorial complexity of the Voronoi diagram is  $\Theta(n^{\lceil d/2 \rceil})$  [59] and it can be computed in  $O(n \log n + n^{\lceil d/2 \rceil})$  optimal time [14, 17, 96]. The duality of the Voronoi diagram and the Delaunay triangulation still holds in higher dimensions. Guibas et al. [48] 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 [88] and divide-and-conquer [19], 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. [51] 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. [106] computed the 2nd order discrete Voronoi diagram using GPU and applied it to perform N-body culling. Very recently, Rong et al. [90] 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. [13] 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 [67] 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 [69] 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 [35] 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. [70] 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 [33] for in-depth discussion on the properties of Delaunay triangulation and Voronoi diagram on a 2-manifold mesh.

## 2.2 Centroidal Voronoi Tessellation

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  $\Omega_i$ , and was first introduced by [30]. But long before similar concepts have been studied in various areas, e.g., k-means in computer vision, pattern recognition and optimal quantization in signal processing.

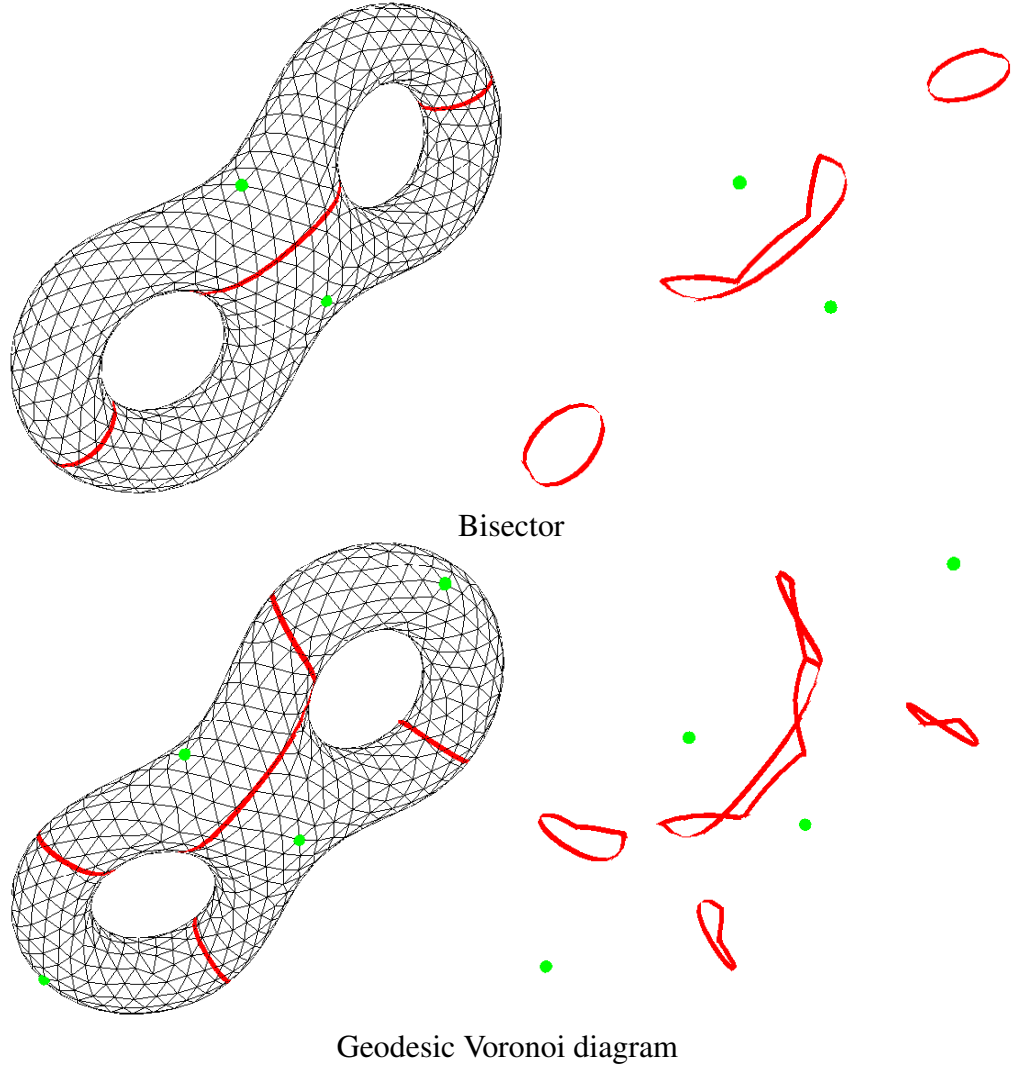


Figure 2.1: Bisector and geodesic Voronoi diagram. Row 1: The bisector (red) of two sites (green) on the double-torus has three separated components. Besides line segments, a bisector on triangle mesh may also contain hyperbola segments. Row 2: The geodesic Voronoi diagram of four sites. Each Voronoi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

To compute the CVT, one of the earliest algorithms is introduced by [74], which is a probabilistic algorithm. Although it will surely converge, its convergence is quite slow. Lloyd presented a deterministic method, which is published officially in [72]. Du et al. [29] proved the convergence of Lloyds algorithm later. Because of its robustness and simplicity, Lloyds algorithm is used widely. Ju et al. [53] combined Lloyds algorithm with MacQueens method, and presented a parallel probabilistic Lloyd algorithm. Liu et al. [68] 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 [66]), which has super linear convergence rate, thus, significantly outperforms the Lloyd algorithm. Lévy and Liu [64] 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. [89] extended CVT from Euclidean space to hyperbolic and spherical space by using global parameterization.

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. [3] 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. [89] 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 opposite side 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. [90] 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 [46], which poses a great numerical challenge to compute the CVT on  $\mathbb{S}^2$ .

Yan et al. [124] 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 [63] 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 Discrete Geodesics

The discrete geodesic problem has four variations, namely, single-source single-destination (SSSD), single-source all-destinations (SSAD), multi-sources all-destinations (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 review 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. Section 2.4 discusses other types of discrete geodesics and the methods for computing geodesics on discrete domains other than meshes.

### 2.3.1 Discrete Wavefront Propagation Methods

Two representative methods are the MMP algorithm [79] and the CH algorithm [16], 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 windows. 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 [109],[121],[69],[71],[123], parallelization [127], and handling degeneracy [71]. It is worth noting that there are at most  $O(n^2)$  windows on an  $n$ -face mesh [79], 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 [109], the MMP algorithm and the improved CH algorithm [121] run in  $O(n^{1.5} \log n)$  time on real-world meshes.

Schreiber and Sharir [95] 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 [94] 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 [55] 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.

### 2.3.2 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 [58] 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 [111]. Later, the FMM was generalized to arbitrary triangulated surfaces [58], unstructured meshes [97], implicit surfaces [77], parametric surfaces [105] and even broken meshes [12]. 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. [117] developed a parallel FMM on geometry images, which runs in  $O(n)$  time. Other parallel methods for solving the Eikonal equation include [129], [42], and [24].

The heat method [22] 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 in linear time. Wang et al. [113] use the concept of heat flow to compute the geodesic distances and paths from sources to the boundary of a mesh surface, which can be considered as a special case of the multiple sources geodesic path problem.

Recently, Belyaev and Fayolle [7] 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.

### 2.3.3 Other Methods

Kanai and Suzuki [54] 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. [120] 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.

## 2.4 Exponential Map

With the above-mentioned discrete geodesic algorithms, we can compute the geodesic path on the surface. 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 [28]. 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.

After we obtain the geodesic path, we 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  $\exp_p : T_pM \rightarrow M$  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  $\gamma$  originating at  $p$  such that  $\gamma'(0) = \mathbf{v}$  and  $\|\gamma(t)\| = 1$ . Conversely, given an arc-length parameterized geodesic  $\gamma$  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 [93], texture mapping [107], Poisson disk sampling [128], etc.

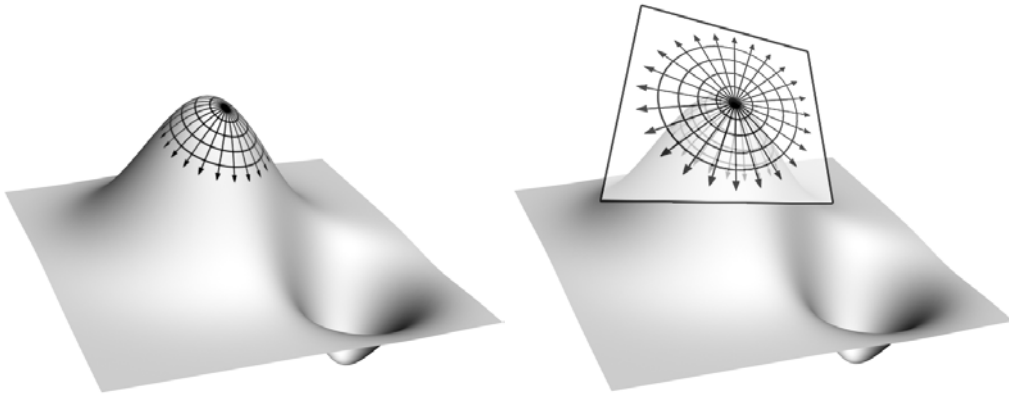


Figure 2.2: Exponential map naturally defines a geodesic polar coordinate system on curved surfaces.

## 2.5 Sampling and Shape Distribution

The widely used sampling techniques include blue noise [20], point repulsion [112], centroid Voronoi tessellation (CVT) [30], stratified sampling [80], spectral sampling [84], and polyominoes [83], just name a few.

Among them, Poisson disk sampling is popular due to its nice spatial and spectral properties. Dart throwing [20] 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 [78, 31, 43, 34], kernel density model [36], and tile-based approaches [18, 62, 61, 83]. Wei [118] 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. [65] 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. [10] extended the phase group algorithm for Poisson disk sampling on 3D surfaces. Wei [119] introduced multi-class blue noise sampling where each individual class and their union exhibit blue noise properties. Chen et al. [15] presented bilateral blue noise sampling for handling problems with non-spatial features.

Besides 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 [108], and rectangle distributions are used to simulate decorative mosaics [50, 6]. Feng [37] generated non-overlapping ellipses with blue noise characteristic. Sun et al. [108] 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. [10] computed Poisson disk distributions on 3D surfaces by space partition. With a global parameterization, Li et al. [65] 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 Sweep Circles

This chapter presents a new algorithm, called *untransformed sweep circle*, 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 algorithm is called untransformed because it's different with Dehne & Klein's sweep circle algorithm, which requires to transform the Voronoi diagrams on the surface

of a cone. Our algorithm directly computes the Voronoi diagrams on the original domain without the transformation. Our untransformed sweep circle 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 3.1 briefly reviews some math preliminaries. Then we introduce the key concepts and properties of the untransformed sweep circle algorithm in Section 3.2 and then present the parallel implementation on modern GPUs in Section 3.3. We provide the detailed experimental results in Section 3.4, then we discuss the fundamental differences between the untransformed sweep circle algorithm and Dehne & Klein’s sweep circle method [23] in Section 3.5.

### 3.1 Preliminaries

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

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

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

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.

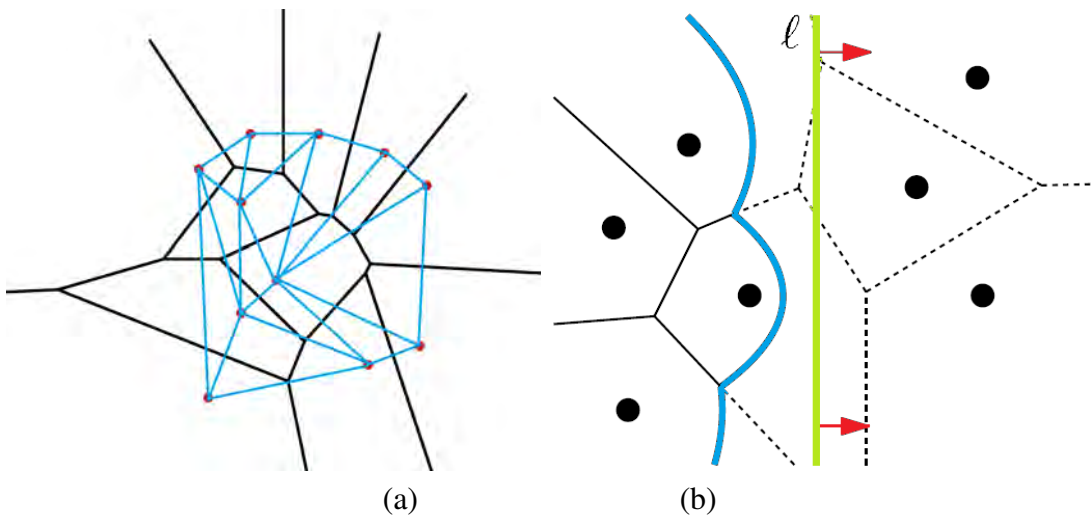


Figure 3.1: (a) An example for the Voronoi diagram (black) and its dual Delaunay triangulation (blue). (b) Fortune's sweep line algorithm [39] maintains a sweep line (green) and a beach line (blue) moving from left to right. The region left to the beach line contains the partial Voronoi diagram with 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 [39] 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 cells have been 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  $\tau$ , 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 rounded 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 rounded 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

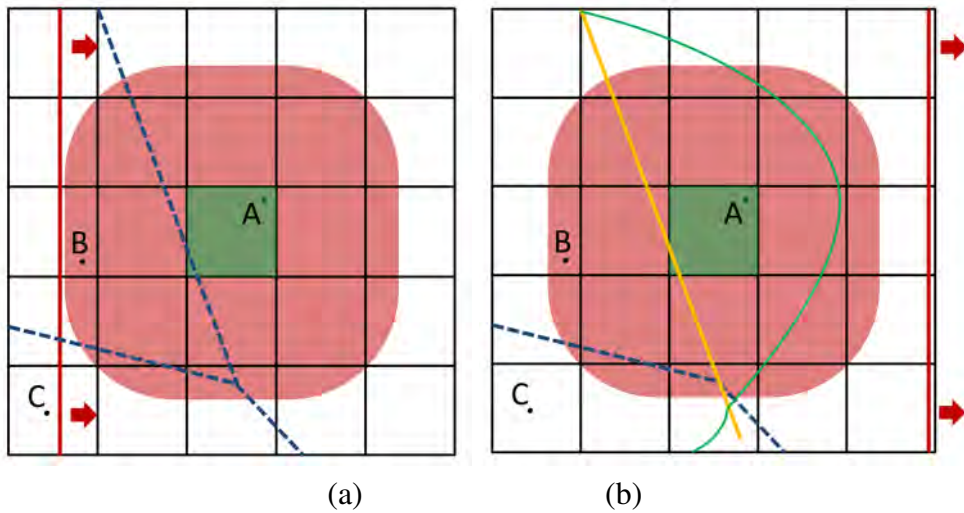


Figure 3.2: (a) To compute the Voronoi diagram for the green region, one has to apply the sweep line algorithm to a much larger domain (in red). (b) At the conclusion of the sweep line algorithm, one has to trim the computed Voronoi diagram with the centered green region. The dotted blue lines are the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

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  $\tau^2$  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 sweep circle algorithm overcomes the drawbacks of the sweep line algorithm, and can be naturally extended to parallel setting.

## 3.2 Untransformed Sweep Circle Algorithm

### 3.2.1 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 sweep circle algorithm contains four fundamental elements:

- 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 site. 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 sweep circle 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

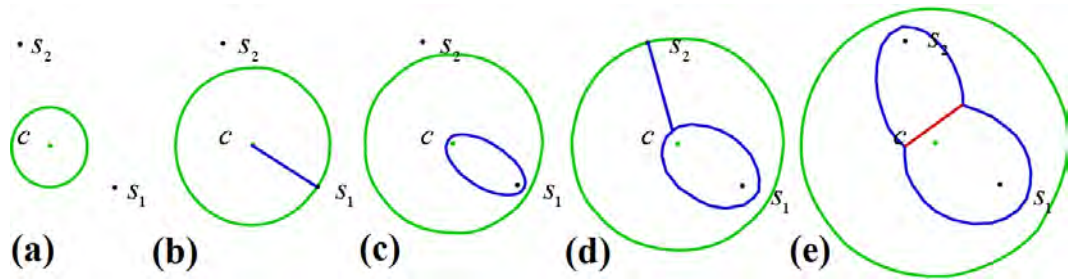


Figure 3.3: Illustration of sweep circles for two sites. The beach curve consists of elliptic segments, each of which corresponds to a site. The elliptic segment is the bisector of the corresponding site and the sweep circle. The vertices of the beach curve trace out the bisector between two sites. The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

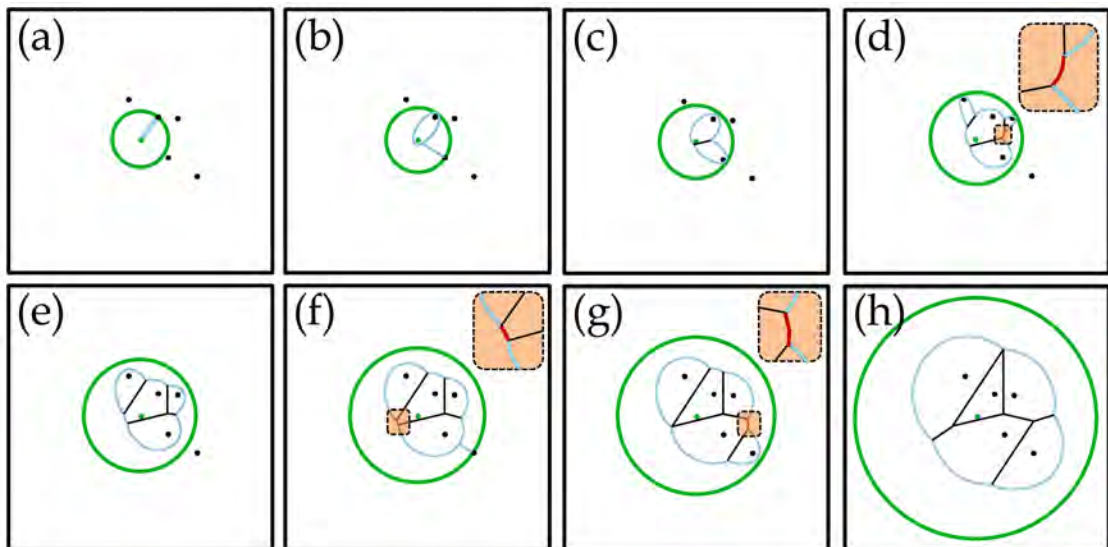


Figure 3.4: Applying the sweep circle algorithm to a domain with five sites. (a) The sweep circle touches the first site. (b) The sweep circle touches the second site. (c) The vertices of the beach curve, at which two ellipses cross, trace out the bisector of the two corresponding sites. (d) An elliptic 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).

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 sweep circle 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 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 sweep circle algorithm is shown in Algorithm 1.

### 3.2.2 Correctness

This subsection proves the correctness of our untransformed sweep circle algorithm. Consider a set of 2D sites  $S = \{s_1, s_2, \dots, s_n\}$ . 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.,  $e_i = \{x \mid \|x - c\| + \|x - s_i\| = R\}$ . For any point  $x \in e_i$ , the shortest distance between  $x$  and the sweep circle is  $d(x, \odot(c, R)) = R - \|x - c\| = \|x - s_i\|$ . 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.

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

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

*and*

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

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

---

**Algorithm 1** Untransformed sweep circle algorithm for computing 2D Voronoi diagram

---

```

1: Input: A set of 2D sites  $S = \{s_1, \dots, s_n\}$  and the center of the sweep circle  $c$ ;
2: Output: The Voronoi diagram of  $S$ 
3: //  $\mathcal{T}$  : a binary tree to maintain the beach curve;
4: //  $\mathcal{V}$  : the Voronoi vertex and edge structure;
5: //  $\mathcal{Q}$  : the priority queue of discrete events
6: Push  $n$  touching events into  $\mathcal{Q}$  sorted by  $\|c - s_i\|$ ,  $1 \leq i \leq n$ .
7: while  $\mathcal{Q}.isEmpty() == false$  do
8:   Pop up the top event  $q$  from  $\mathcal{Q}$ ;
9:   if  $q$  is a touching event then
10:     Insert a new ellipse  $E$  into  $\mathcal{T}$ ;
11:     //Suppose  $E$  splits the existing elliptic segment
12:     // $E'$  into  $E'_1$  and  $E'_2$ , and let  $p$  be the intersection
13:     //point.
14:     Build two radial edges rooted at  $p$ , one shared by  $E'_1$  and  $E$ , the other shared
    by  $E$  and  $E'_2$ ;
15:     Compute the time when  $E$ ,  $E'_1$ , and  $E'_2$  disappear;
16:     Push the vanishing event into  $\mathcal{Q}$ ;
17:   else if  $q$  is not out of date then
18:     // $q$  is a vanishing event.
19:     //Let  $E$  denote the vanishing
20:     //elliptic segment,  $E_1$  and  $E_2$  the two neighboring
21:     //elliptic segments. If one of them is out of data,
22:     //so is the event. see Figure 3.5.
23:     Create a Voronoi vertex  $v$ ;
24:     Terminate the two active edges, joined by  $E$ , at  $v$ ;
25:     Update the Voronoi structure  $\mathcal{V}$ ;
26:     Create an active Voronoi edge  $e$ ;
27:     Push a vanishing event according to  $e$  and its neighboring active edges into
    the queue  $\mathcal{Q}$ ;
28:   end if
29: end while

```

---

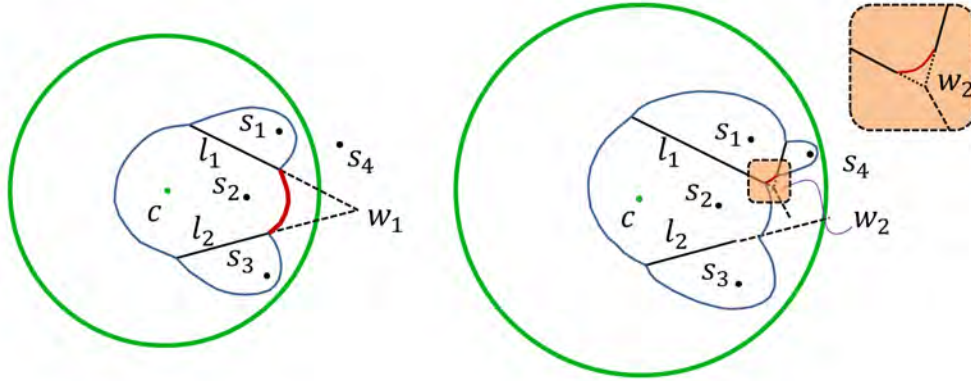


Figure 3.5: Out of date events. The red elliptic segment is sandwiched by two Voronoi edges, say  $l_1$  and  $l_2$ , and will vanish when the sweep circle touches the point  $w_1$ . As a result, a vanishing event  $q$  has been pushed into the priority queue  $\mathcal{Q}$ . However, when the sweep 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  $\mathcal{Q}$ , which is simply ignored by the sweep circle algorithm.

*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 a common 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  $e_i \cap e_j$  must be a subset of  $L \cap e_i$  (or  $L \cap e_j$ ). 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.,  $\|x - c\| + \|x - s_i\| = R$ . 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  $\Omega$ . 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  $S_O = \{s_j \mid s_j \notin \odot(c, R)\}$  and the *inside* group  $S_I = \{s_j \mid s_j \in \odot(c, R)\}$ .

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.,  $e_j = \{x \mid \|x - s_j\| + \|x - c\| = R\}$ . 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  $\odot(c, \frac{R-d}{2})$  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, 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.

Furthermore, with sufficiently large radius  $R$ , all the Voronoi vertices can be enclosed in 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 sweep circle algorithm.

**Theorem 3.1** *The Fortune's sweep line algorithm [39] is a degenerate form of our sweep circle algorithm if the center  $c$  is at infinite distance to the sites.*

*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 sweep circle algorithm.

### 3.2.3 Complexity

**Theorem 3.2** *The untransformed sweep circle 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 3.5).

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  $\mathcal{T}$ . The insertion takes  $O(\log n)$  time. For the vanishing event, the priority queue  $\mathcal{Q}$  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.

### 3.3 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 sweep circle 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 sweep circle 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  $\Omega$  containing sites  $S$ , we can partition the domain  $\Omega$  into disjoint sub-regions,  $\Omega = \bigcup_{i=1}^m \Omega_i$ ,  $\Omega_i \cap \Omega_j = \emptyset$ ,  $\forall i \neq j$ . Then each sub-region  $\Omega_i$  is assigned a sweep circle thread, which places a sweep circle at arbitrary location inside  $\Omega_i$ . In practice, one can set the center  $c_i$  the barycenter of  $\Omega_i$ . 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  $\Omega_i$ , 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  $\Omega$  containing  $n$  sites  $S = \{s_1, \dots, s_n\}$  and a partition of  $\Omega$  into  $m$  disjoint sub-regions  $\Omega_i$ ,  $\Omega_i \cap \Omega_j = \emptyset \forall i \neq j$ .

**Output:** The Voronoi diagram of  $S$

**parallel for** each sub-region  $\Omega_i$  **do**

Compute the barycenter  $c_i$  of  $\Omega_i$

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  $\Omega_i$ . If so, we can stop the sweep circle algorithm for  $\Omega_i$ . In practical application, the sub-region  $\Omega_i$  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  $\Omega_i$ , 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  $\Omega_i$  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  $\Omega_i$  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

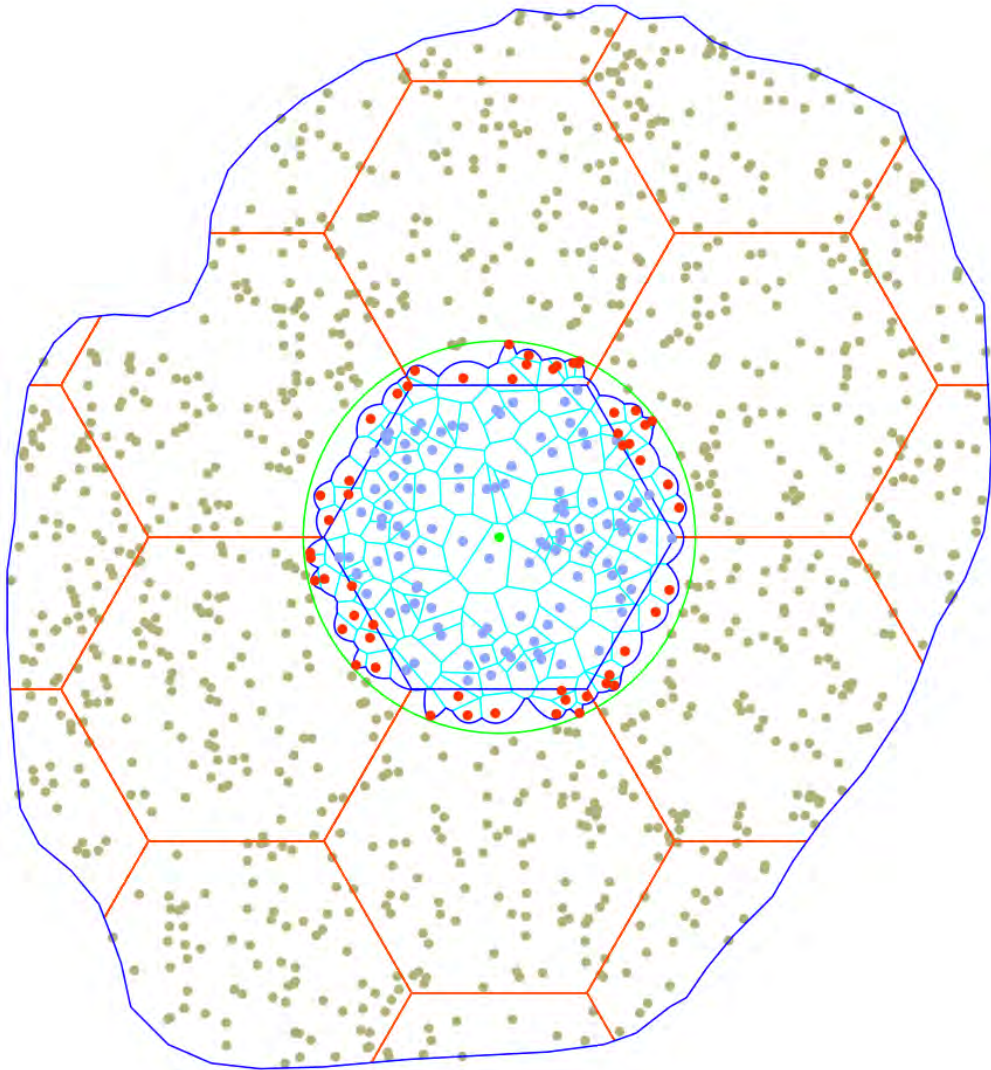


Figure 3.6: In practice, the beach curve follows the sweep circle very closely, which leads to a small stopping radius  $R_s$ . For a domain  $\Omega$  with 1000 randomly distributed sites, we partition  $\Omega$  into equally sized regular hexagons and compute the ratio of the number of sites inside 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.

### 3.4 Experimental Results

We implemented our untransformed sweep circle algorithm in C++ and tested it on a PC with an Intel Xeon 2.50GHz CPU and 8GB memory. The graphics card is an NVIDIA 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 3.7. 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 sweep circle algorithm, the average number of events can be significantly reduced by increasing the number of sweep circles.

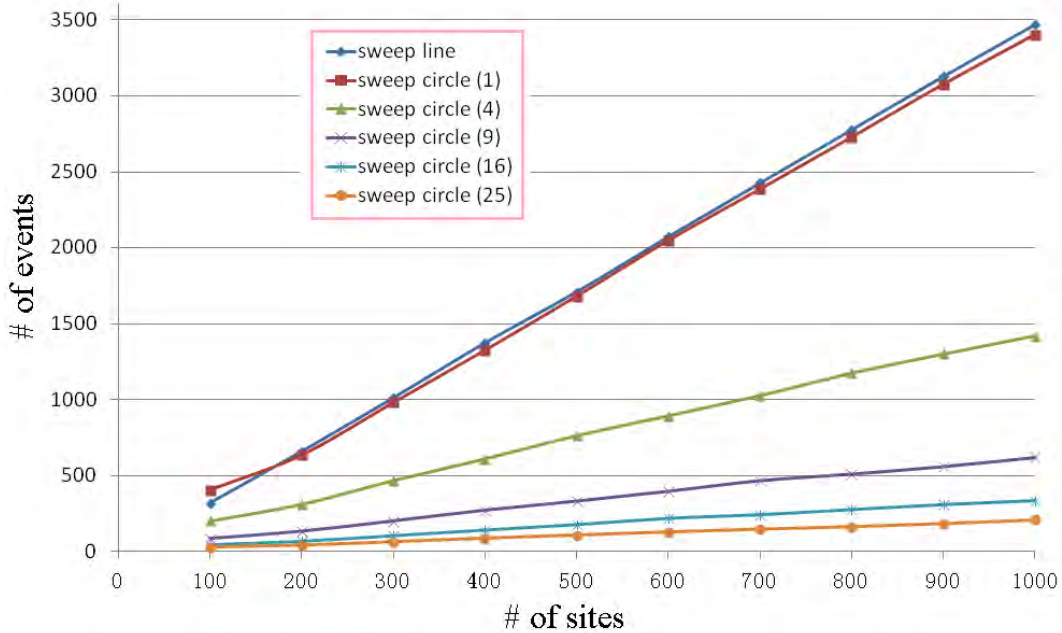


Figure 3.7: Both the sequential sweep circle and sweep line algorithms have very similar number of discrete events. With the parallel sweep circle algorithm, the average number of events for each GPU thread is significantly reduced. The horizontal axis and the vertical axis show the number of sites and the average number of discrete events respectively. The number in the bracket represents the number of sweep circles.

Figure 3.9 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  $\Omega_i$  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 3.8 demonstrates a comparison of our sweep circle algorithm with CGAL [1] and Triangle [102]. 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.

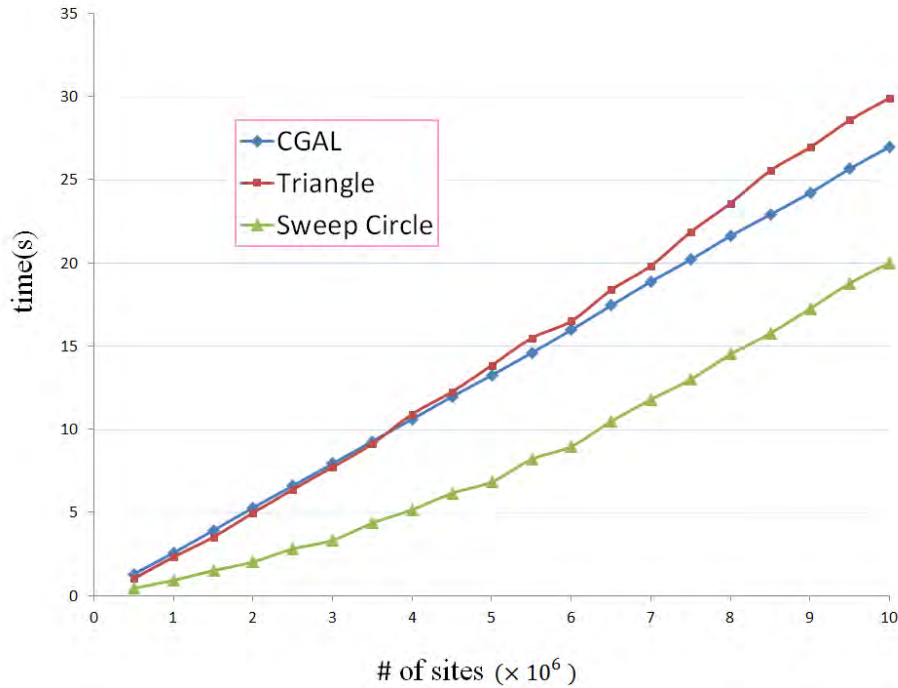


Figure 3.8: The comparisons with CGAL [1] and Triangle [102] show that our sweep circle algorithm has parallel speedup ratio of 1.63 and 1.73 respectively. The horizontal axis and the vertical axis shows the number of sites and computing time respectively.

Our algorithm can be applied to 3D surface by using parameterization. Figure 3.10 shows two genus-0 3D models which are conformally parameterized to disc and square using the harmonic map [46] and the holomorphic 1-form method [47]. 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 3.11. 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

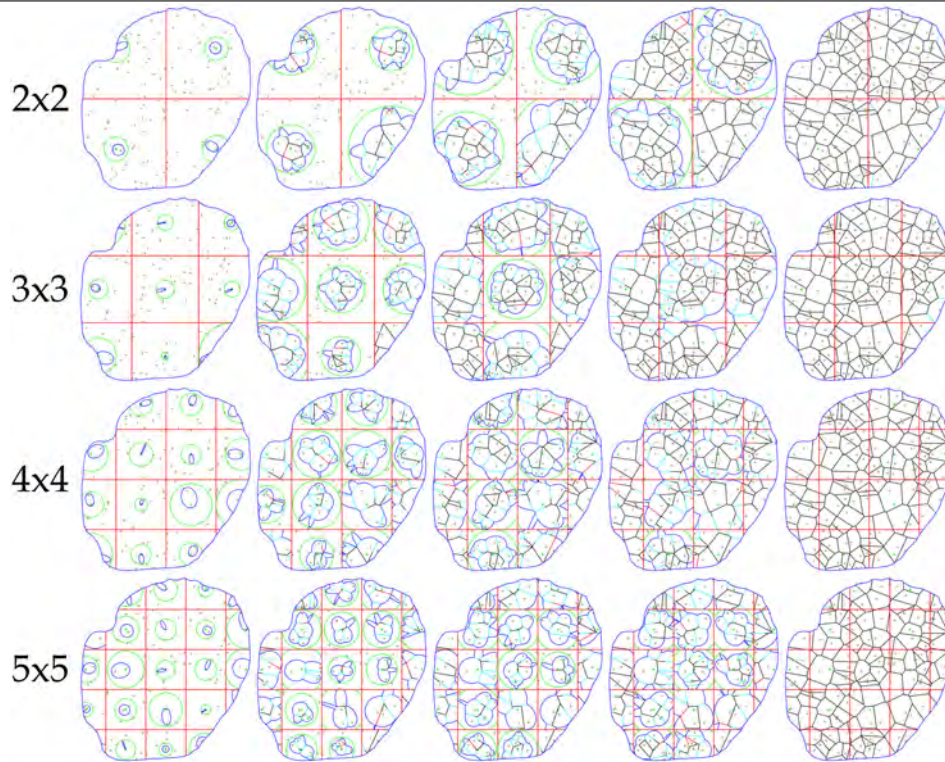


Figure 3.9: Parallel computing Voronoi diagram using GPU based sweep circle algorithm. The entire domain is uniformly divided into  $m^2$  sub-regions (where  $m = 2, 3, 4, 5$  for this experiment), and a sweep circle thread is run for each sub-region independently. The intermediate results are shown 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.

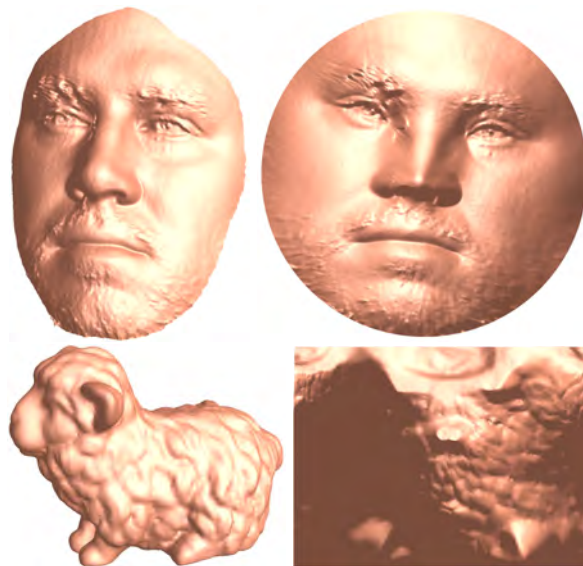


Figure 3.10: 3D models and their parameterization.

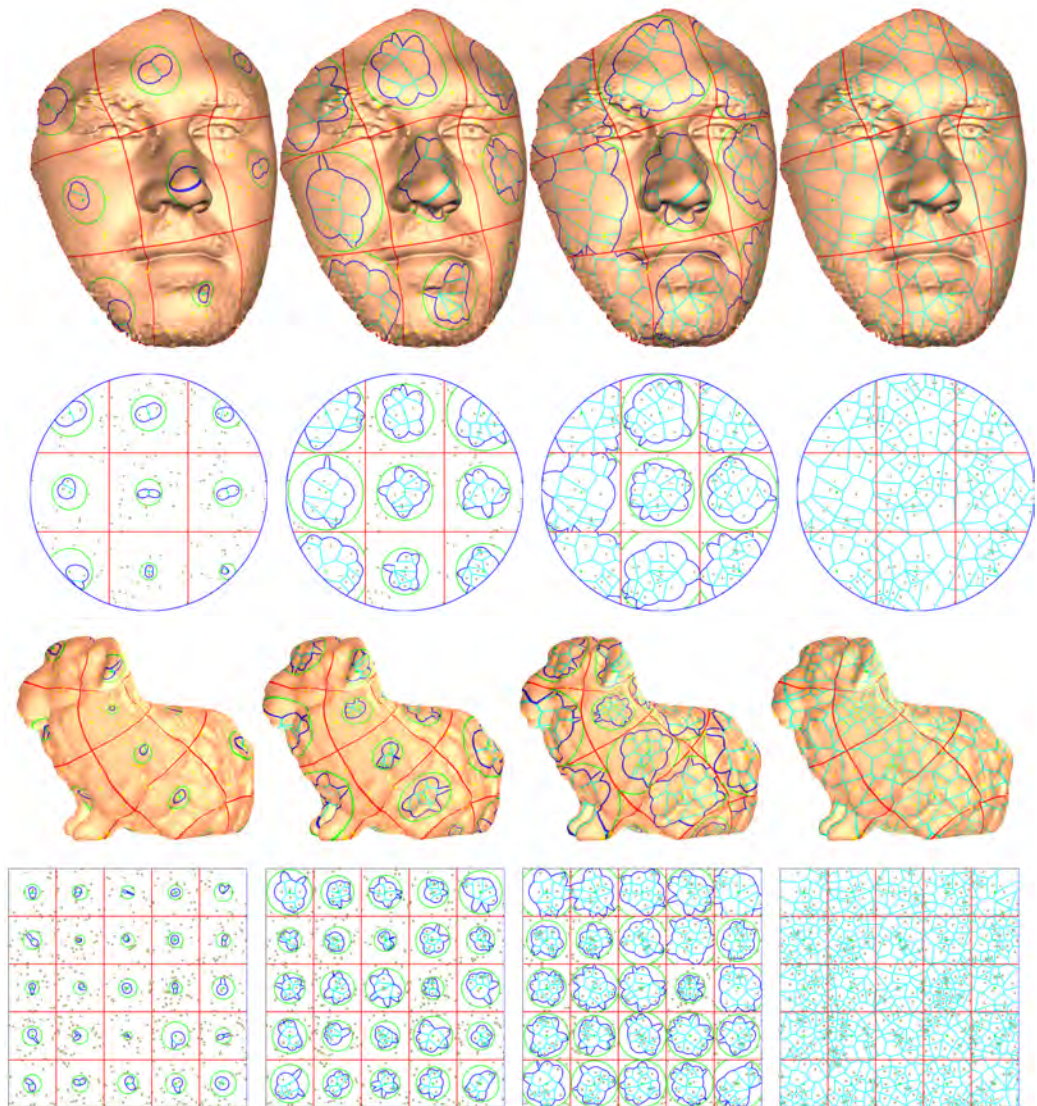


Figure 3.11: Our method can be applied to 3D surfaces by using parameterization.

$s_i$ , 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 3.12.

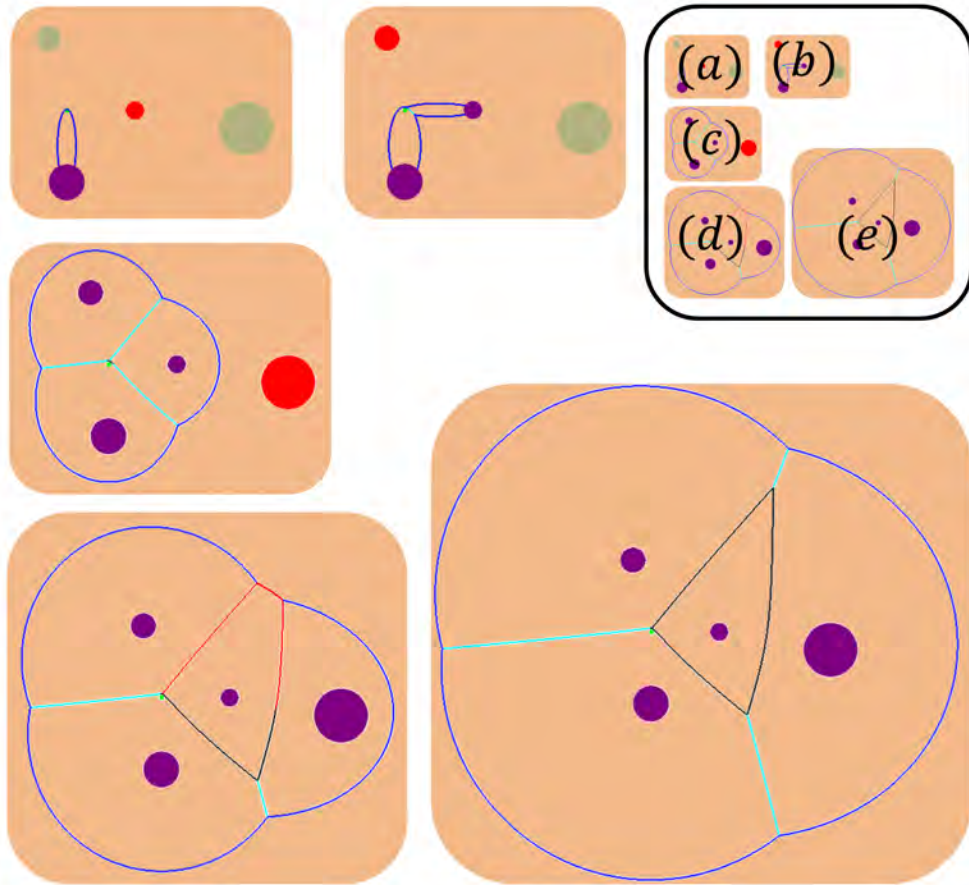


Figure 3.12: Computing the additively weighted Voronoi diagram, where the conventional Euclidean distance is modified by the weights assigned to the generator sites. The weights are illustrated by the size of the site. The resulting Voronoi diagram contains both hyperbolic segments and line segments. The animation from (a) to (e) illustrates the sweeping process.

### 3.5 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

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 sweep circle to determine the priority in handling sites. Suppose  $s_1$  and  $s_2$  are in a nearly equal distance to the sweep circle 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 sweep circle 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, we use a tolerance  $1e-7$  to identify if the vanishing point is a high-order Voronoi vertex.

Thurston [110] observed the sweep circle technique allows to compute the Voronoi diagram locally, which shows the Voronoi diagram can be parallelized in nature. Dehne and Klein [23] applied the sweep circle to compute a type of transformed Voronoi diagram. Although their algorithm also supports multiple sweep circles, it is fundamentally different with our untransformed sweep circle algorithm.

- (i) In the traditional sweep circle algorithm, the edges are very complicated in the transformed representation, which is computed on the cone. 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 sweep circle algorithm, while the beach curve is rather complicated data structure on the cone in Dehne and Klein's sweep circle algorithm (they didn't mention the representation form of the beach curve in [23]).
- (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 sweep circle algorithm to support 2D weighted Voronoi diagrams.
- (iv) Dehne and Klein's sweep circle algorithm can be easily extended to the Voronoi diagram on the surface of a cone, while it seems not easy for our algorithm.

## Chapter 4

# Saddle Vertex Graph for Computing Discrete Geodesics

This chapter presents the Saddle Vertex Graph (SVG) for efficiently computing discrete geodesics on triangle meshes. 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.



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.

## 4.1 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 chapter, 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  $2\pi$ , *Euclidean* if the total vertex angle equals  $2\pi$ , and *saddle* if the total vertex angle is greater than  $2\pi$ . 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. [79] 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  $\pi$ .
- 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  $\gamma$  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 [87]. Due to this fundamental difference between the smooth

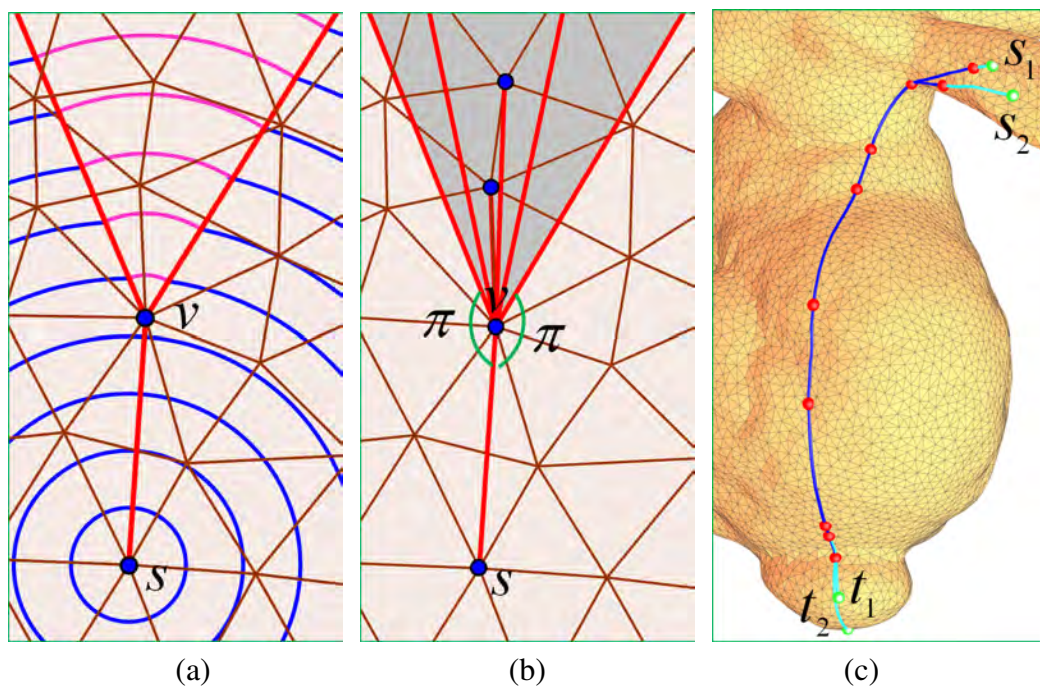


Figure 4.2: Saddle vertex and geodesic path. (a) A saddle vertex  $v$  has the total vertex angle  $\sum \theta_i > 2\pi$ . When passing through  $v$ , the geodesic wavefront, a circle centered at the source  $s$ , splits into three arcs. The middle arc (in pink) is centered at  $v$ , and the left and right arcs remain centered at  $s$ . (b) The incoming geodesic path  $\gamma$  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  $\gamma$  usually passes through one or more saddle vertices (the red points), which partition  $\gamma$  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.

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 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.

## 4.2 Saddle Vertex Graph

### 4.2.1 Definition

**Definition 1.** Consider  $\gamma(p, q)$  a globally shortest geodesic path between  $p$  and  $q$ .  $\gamma$  is called *direct* if it does not pass through any saddle vertex, 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  $\mathcal{S}(v)$ , are the saddle vertices which can be reached from  $v$  via a direct geodesic path. Similarly, the *non-saddle neighbors* of  $v$ , denoted by  $\mathcal{N}(v)$ , 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  $S = S_1 \cup S_2 \cup S_3$  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  $S_2 = (V_S \cup V_N, E_{NS})$  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.

Throughout this chapter, 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  $q$ .

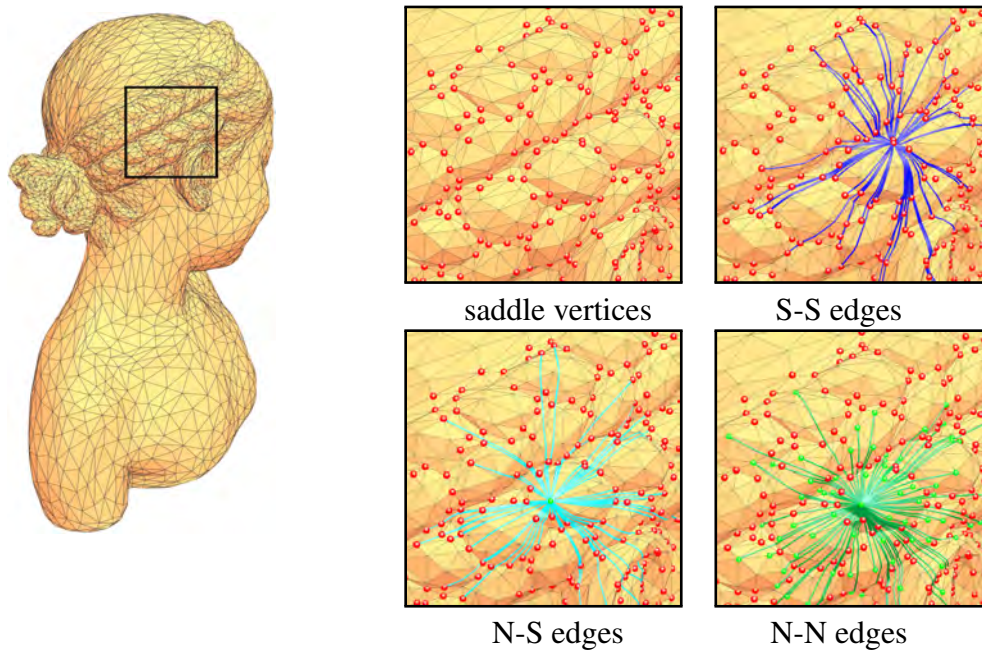


Figure 4.3: The SVG on the 9K-face Bimba. The saddle vertices are drawn in red. For illustration purpose, we show only the S-S, N-S and N-N edges incident to a vertex.

## 4.2.2 Data Structure

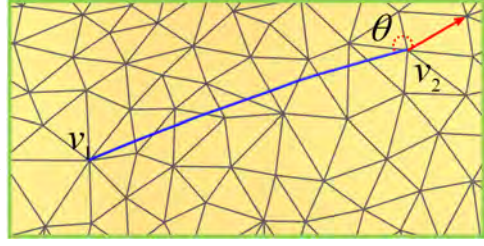
We use the conventional incidence list to store the SVG. Each vertex stores its incident edges (i.e., the direct geodesic paths) and each edge stores one incident vertex and its length (i.e., the geodesic distance).

```

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

```

In order to find the geodesic path, we also associate each SVG edge with an angle  $\theta$ , which measures the direction of the geodesic path with respect to its endpoint  $v_2$ . The reference direc-



tion (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  $\theta$ , we can obtain the first segment of  $\gamma$ , 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  $\text{Backtrace}(e)$  is defined to compute the direct geodesic path for an SVG edge  $e$ .

### 4.3 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<sup>1</sup> 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.

<sup>1</sup>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  $q$ .

### 4.3.1 Parameter

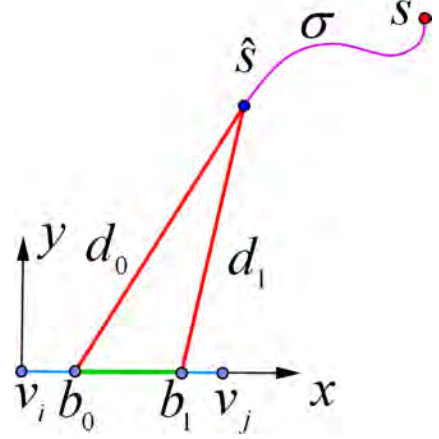
Our goal is to find a model and resolution insensitive parameter that can intuitively and effectively control the SVG complexity. Therefore, the maximal SVG edge length is not suitable, since it is resolution-dependent. The relative SVG edge length cannot be used either, since it is model dependent and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

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 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.

### 4.3.2 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  $\sigma$  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  $\sigma$  of  $w$ 's child windows cannot decrease. Therefore, the geodesic algorithm can simply stop when all direct windows have been processed.

### 4.3.3 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  $S_i \leftarrow \text{ComputeDirectGeodesicPaths}(M, v_i, K)$ ;
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
```

---

## 4.4 Computing Discrete Geodesics Using SVG

### 4.4.1 Dijkstra's Algorithm and Label Correcting 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 [26] to compute the shortest path on the SVG.

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 [26] 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  $(d_1, d_2, \dots, d_{|N|})$  and a set of nodes  $\mathcal{C}$ , called the candidate list, starting with  $d_1 = 0$ ,  $d_i = \infty$  for  $i \neq 1$ ,  $\mathcal{C} = \{1\}$ . The Dijkstra algorithm iteratively processes the nodes from the candidate list  $\mathcal{C}$  and terminates when  $\mathcal{C}$  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  $\mathcal{C}$ . Since each node enters and exits  $\mathcal{C}$  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  $\mathcal{C}$ . 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  $\mathcal{C}$  and the nodes can enter and exit the queue in constant time  $O(1)$ . Therefore, selection of the node to be removed from  $\mathcal{C}$  is faster than Dijkstra’s algorithm, at the expense of multiple entrances of nodes in  $\mathcal{C}$ .

We adopt the Small Label to the Front (SLF) scheme [8] for node insertion and the Large Label Last (LLL) scheme [9] for extraction: When a node  $n_j$  enters the queue  $\mathcal{C}$ , its labels  $d_j$  is compared with the label  $d_i$  of the top node  $n_i$  of  $\mathcal{C}$ . If  $d_j \leq d_i$ , node  $n_j$  is entered at the top of  $\mathcal{C}$ ; otherwise  $n_j$  is entered at the bottom of  $\mathcal{C}$ . The top node  $n_i$  exits the queue  $\mathcal{C}$  if its label is less than a threshold; otherwise send  $n_i$  to the bottom of  $\mathcal{C}$ . The threshold is usually set as the mean value of the labels of all the nodes in  $\mathcal{C}$ .

Method	Pre-computing time	Space	SSAD	Query points
GTU [122]	$O(mn^2 \log n)$	$O(m^2)$	$O(n)$	arbitrary
SVG [126]	$O(nK^2 \log K)$	$O(Dn)$	$O(Dn \log n)$	mesh vertices only
LC-SVG+GUT	$O(nK^2 \log K)$	$O(Dn)$	empirical $O(Dn)$	arbitrary

Table 4.1: 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-destinations.

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 6.1 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 em-

pirical  $O(Dn)$  time complexity to compute the single-source all-destinations geodesic distances, since the graph has  $O(Dn)$  edges and the overhead per iteration is  $O(1)$ .

#### 4.4.2 Single-source Single-destination (SSSD)

To compute the single-source single destination geodesic distance  $d(s,t)$ , we adopt the widely used A\* search [86] [109], 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  $S_1 \cup S_2 \cup S_3$ . 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  $\{e_1, e_2, \dots, e_k\}$  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

**Algorithm 4** Computing the SSSD Geodesic Distance

**Require:** A triangle mesh  $M = (V, E, F)$ , the associated saddle vertex graph  $S = S_1 \cup S_2 \cup S_3$  and two vertices  $p_1, p_2 \in V$ .

**Ensure:** The geodesic distance  $d(p_1, p_2)$ .

```

1: if  $p_1, p_2$  are direct neighbor then
2:   return  $\|\{p_1, p_2\}\|$ ;
3: end if
4:  $U \leftarrow \text{BidirectionalDijkstra}(\{V, E\}, p_1, p_2)$ ;
5: if  $S$  is exact then
6:    $S'_1 \leftarrow S_1$ ;
7:   for  $i=1, 2$  do
8:     if  $p_i \notin V_S$  then
9:        $S'_1 \leftarrow S'_1 \cup \{p_i \text{ and its N-S edges}\}$ ;
10:    end if
11:  end for
12:   $A * \text{Dijkstra}(S'_1, p_1, p_2, U)$ ;
13: else
14:   $A * \text{Dijkstra}(S_1 \cup S_2 \cup S_3, p_1, p_2, U)$ ;
15: end if
16: return  $d_{p_1}(p_2)$ ;

```

time complexity to obtain  $\gamma(p, q)$  is  $O(k)$ , where  $k$  is the number of triangles crossed by the path  $\gamma(p, q)$ .

### 4.4.3 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  $S_1 \cup S_2 \cup S_3$ , 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  $S = S_1 \cup S_2 \cup S_3$  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  $s_i \in V$ .

```

1: if  $S$  is exact then
2:    $S'_1 \leftarrow S_1$ ;
3:   for each  $s_i \in \mathfrak{S}$  do
4:     if  $s_i \notin V_S$  then
5:        $S'_1 \leftarrow S'_1 \cup \{s_i \text{ and its N-S edges}\}$ ;
6:     end if
7:   end for
8:   Dijkstra( $S'_1, \mathfrak{S}$ );
9:   for each  $q \in V_N$  do
10:     $d(q) = \min_{t \in \mathcal{S}(q)} \{d(t) + \|\{t, q\}\|\}$ ;
11:   end for
12: else
13:   Dijkstra( $S_1 \cup S_2 \cup S_3, \mathfrak{S}$ );
14: end if

```

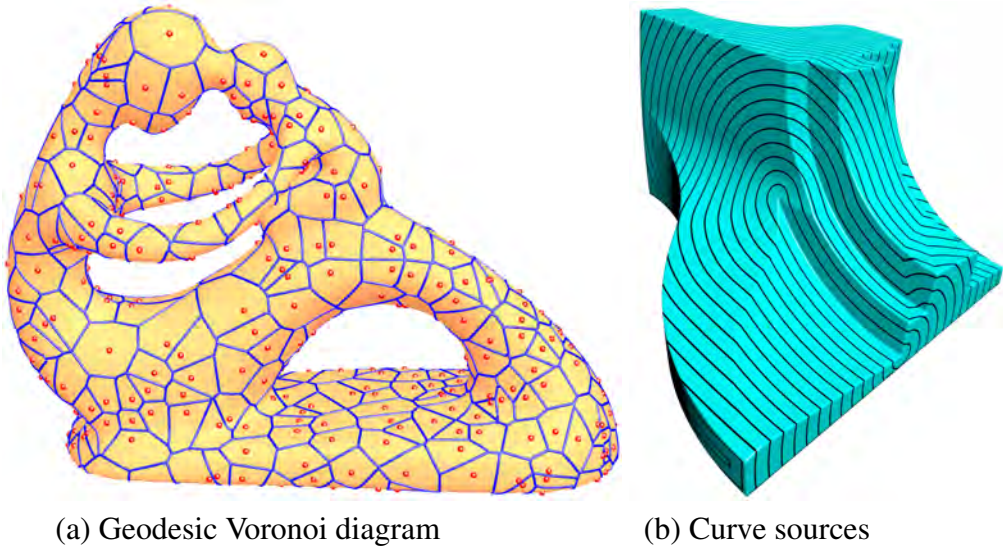


Figure 4.4: The MSAD algorithm allows us to compute the geodesic Voronoi diagram and the geodesic distances to curve sources. (a) 500 random selected vertices are used as the seeds for the Voronoi diagrams. (b) The vertices on the curves are used as the sources.

#### 4.4.4 Discrete Geodesic Distances Between Arbitrary Points

We notice that both the GTU method [122] 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 4.5(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 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.

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  $2\pi$ . Mitchell et al. [79] 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 vertex. 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,

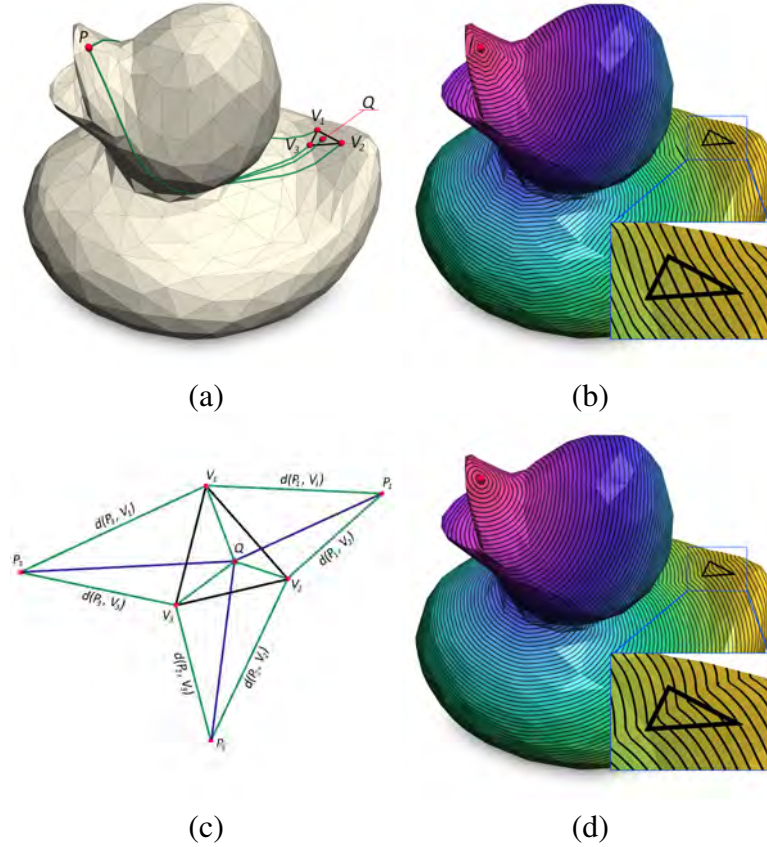


Figure 4.5: Computing the single-source all-destination geodesic distances on a low-resolution mesh. (a) Let  $p$  be the source point (also a mesh vertex) and  $q$  a point inside a triangle  $\triangle v_1 v_2 v_3$ . (b) Using the saddle vertex graph, we can accurately compute the geodesic distances 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,  $\triangle p v_1 v_2$ ,  $\triangle v_1 v_2 p$  and  $\triangle v_1 v_2 v_3$ , 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.

such as the MMP algorithm [79], the ICH algorithm [121], and most recently, the FWP-enhanced algorithm [123], can be used to construct SVG. Ying et al. [126] observed that it is *not* necessary to compute *all* direct geodesic paths, in fact, a small subset of  $\Gamma$  can lead to a pretty good result. Therefore, they suggested a simple-yet-effective heuristic to 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  $(V, \Gamma)$  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 do not 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. [122], 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 4.5(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  $v_1 v_2$ ,  $v_2 v_3$ ,  $v_3 v_1$ , 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 4.5(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 [122] for the details.

## 4.5 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. 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 5.3.

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;  $\epsilon$ : 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)	$\epsilon$	$T_p^c$ (s)	$T$ (s)	$\epsilon$	$T_p^c$ (s)	$T$ (s)	$\epsilon$	$S$ (Mb)	$T_p^c$ (s)	$T_p^g$ (s)	$T$ (s)	$\epsilon$	$S$ (Mb)	$T_p^c$ (s)	$T_p^g$ (s)	$T$ (s)	$\epsilon$
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 [79] [121] and measure the relative difference  $|d(x, y) - \tilde{d}(x, y)|/d(x, y)$ , where  $d(x, y)$  and  $\tilde{d}(x, y)$  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. 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 4.9 shows the results on the Bunny and the Armadillo. See more results in the Supplementary Material.

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 4.1.

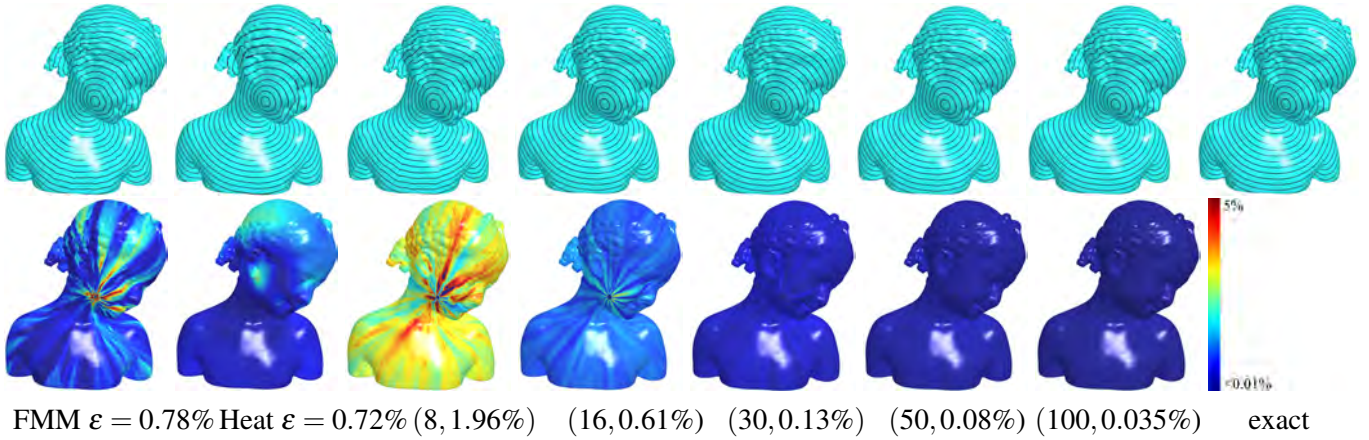


Figure 4.6: Visual comparison of the accuracy. The 2-tuple under the SVG result is  $(K, \varepsilon)$ , where  $\varepsilon$  is the mean relative error.  $K = 50$  leads to high quality result, where the difference to the exact result is hardly visible.

**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.

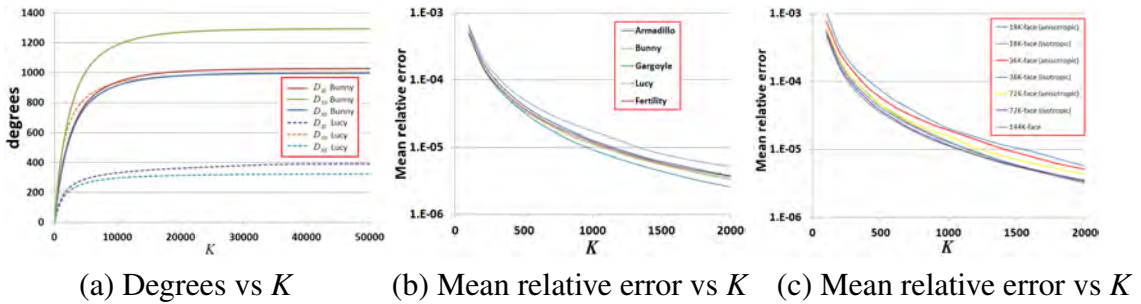


Figure 4.7: The parameter  $K$  effectively controls the SVG complexity and the accuracy of the approximate geodesic distance. (a) A sufficiently large  $K$  produces the exact SVG. (b) The mean relative error as a function of  $K$  on various models. (c) The mean relative error as a function of  $K$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

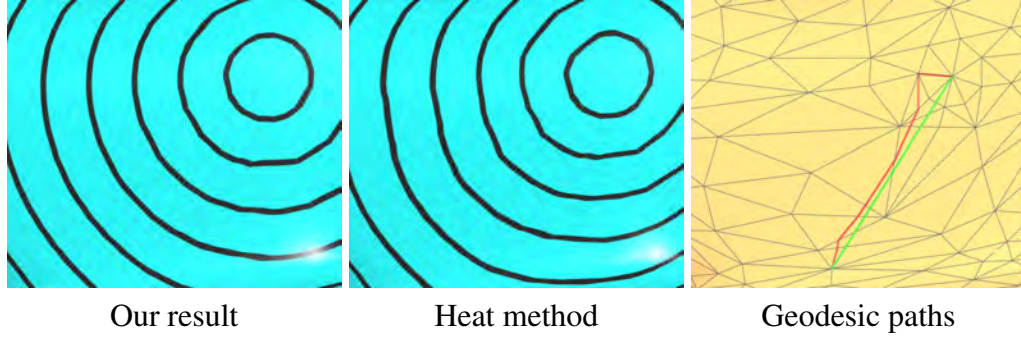
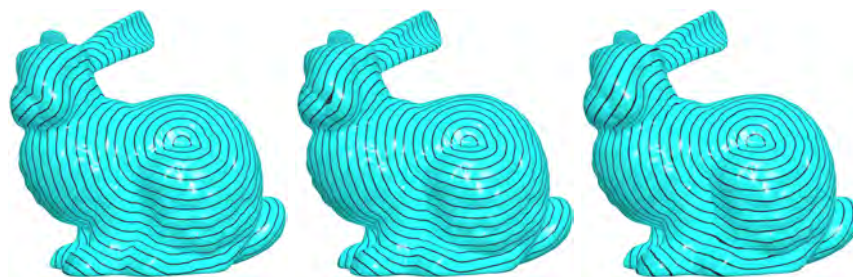


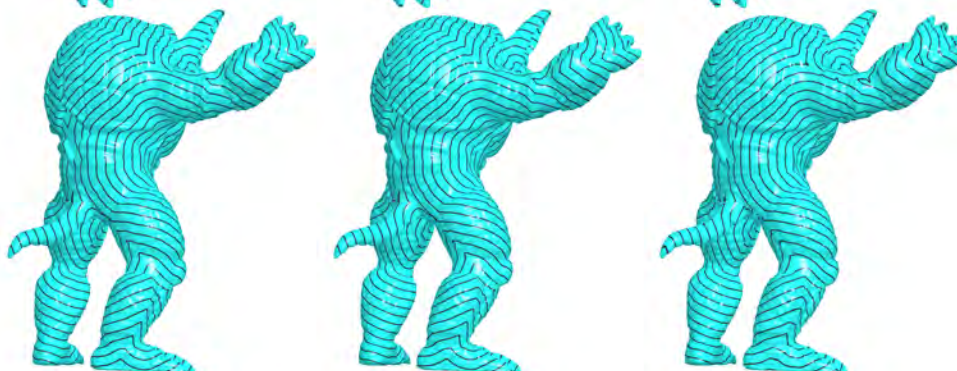
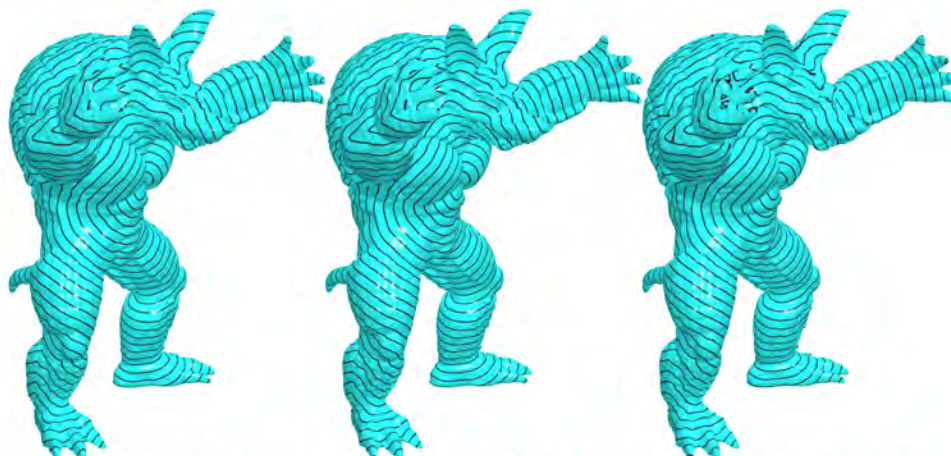
Figure 4.8: The SVG method computes the geodesic path by unfolding a sequence of triangles, which is stable and accurate. The other approximate algorithms compute the geodesic path by tracing the gradient of the distance, which is sensitive to the triangulation. The geodesic paths obtained by our method 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 results 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  $\{p_0, p_1\}, \{p_1, p_2\}, \dots, \{p_k, p_{k+1}\}$  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.

**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.



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 4.9: 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.

## Chapter 5

# Intrinsic Computation of Centroidal Voronoi Tessellation (CVT) on Meshes

This chapter presents two intrinsic algorithms for computing CVT on triangle meshes. The 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 Voronoi diagram. The second algorithm uses the L-BFGS method to accelerate the intrinsic CVT computation. 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. The promising experimental results show the advantages of our method.

The rest of the chapter is organized as follows. Section 5.1 and Section 5.2 present our intrinsic Lloyd and L-BFGS CVT algorithms in details. Section 5.3 shows the experimental results, compares our method to the existing techniques and discusses its advantages and limitations.



Figure 5.1: Our intrinsic method can compute a high-quality centroidal Voronoi tessellation on model with complicated geometry and topology. The CVT on the Pegasus model was created by 3000 sites.

## 5.1 The Lloyd Framework

### 5.1.1 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  $S = \{s_i | s_i \in M, i = 1, \dots, m\}$  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-sources all-destinations geodesic distance with the sites  $s_i, i = 1, \dots, m$ , as the sources. This geodesic distance 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

**Algorithm 6** Intrinsic computation of centroidal Voronoi tessellation on meshes

---

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

**Ensure:** The centroidal Voronoi tessellation on  $M$ ;

```

1: do
2:   Compute geodesic distance field with  $\{s_i\}_{i=1}^m$  as sources;
3:   Form the geodesic Voronoi diagram;
4:   for  $i = 1$  to  $m$  do
5:     Compute the Riemannian center  $r_i$  for Voronoi cell  $\Omega_i$ ;
6:     Compute the center of mass  $c_i$  for  $\Omega_i$ ;
7:      $d_i \leftarrow d(s_i, c_i)$ ;
8:      $s_i \leftarrow c_i$ ;
9:   end for
10: while  $\frac{\sum_{i=1}^m d_i}{m} > \varepsilon$ 

```

---

$r_j$ , and map the Voronoi cell  $\Omega_j$  to the tangent plane  $T_{r_j}$ , on which we can compute the center of mass  $c_j$ . The tangent plane  $T_{r_j}$  contacts with the mesh at the Riemannian center  $r_j$ . Finally, we map the mass center from the tangent plane  $T_{r_j}$  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.

Our algorithm uses the tangent plane to facilitate the exponential map computation. In Riemannian geometry, the exponential map takes a given tangent vector on the tangent plane to the manifold, runs along the geodesic starting at that point. On the discrete triangle mesh, we still takes a given tangent vector and runs along the discrete exact geodesic algorithm like ICH or MMP. So it's an exact method on the discrete mesh, but if the discrete mesh approximates to the underlying continuous surface, there is some approximation.

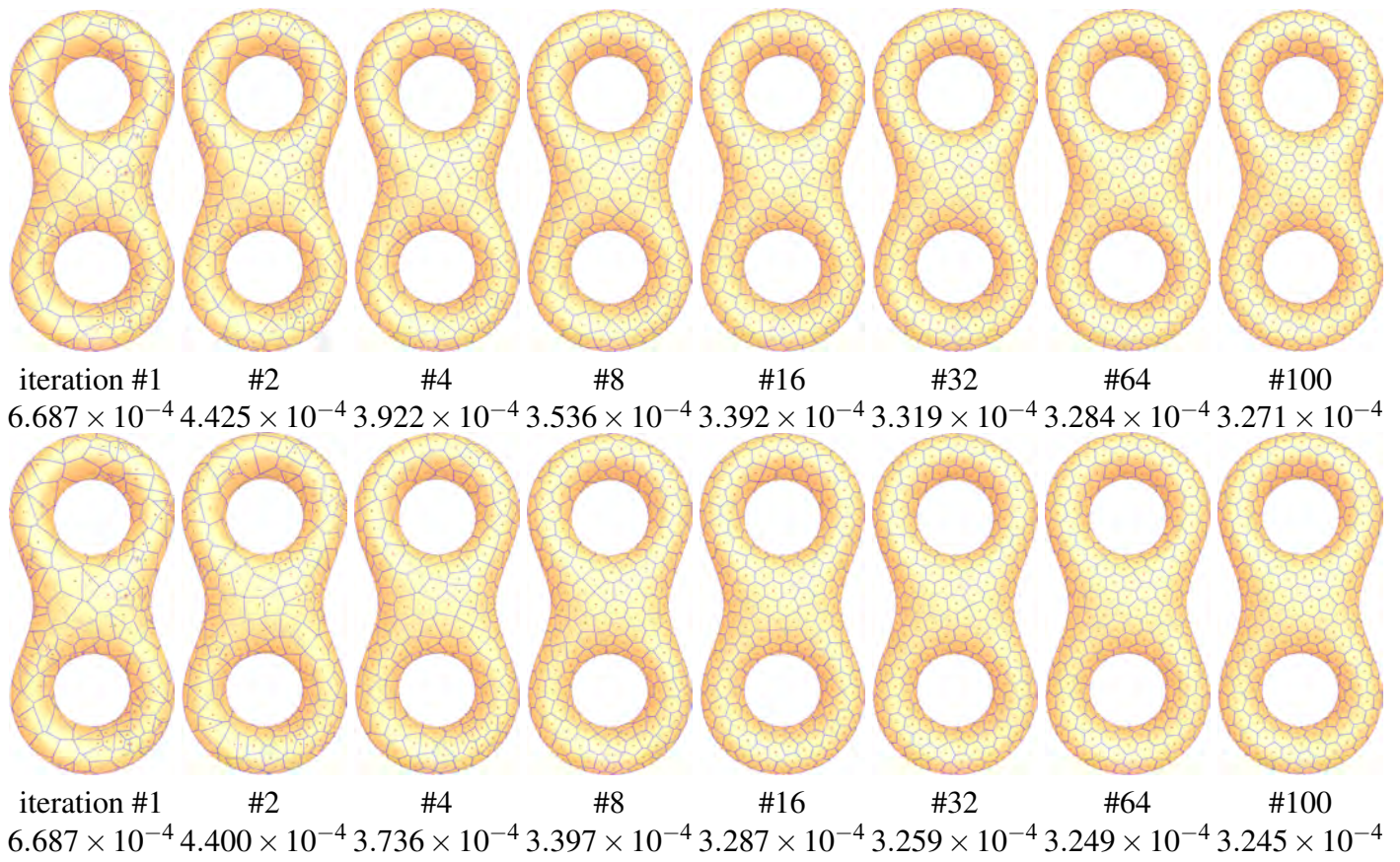


Figure 5.2: Iteratively computing geodesic CVT on meshes. Row 1: the Lloyd algorithm; Row 2: the L-BFGS algorithm.

## 5.1.2 Computing the Geodesic Voronoi Diagrams

Taking  $\{s_i\}_{i=1}^m$  as sources, we apply the SVG method in chapter 4 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 a list  $LT$  all the triangles in  $M$  that are incident to any  $LE$  edge. As [67] 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 [67] to extract 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 5.2). The SVG method 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  $k$  is the number of triangles in  $LT$ . Figure 5.3 shows the geodesic distance field and its induced geodesic Voronoi diagram on the double-torus model.

## 5.1.3 Computing the Riemannian Center

Let  $v_1, v_2, \dots, v_k$  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

$$U(x) = \sum_{i=1}^k d^2(x, v_i), \quad (5.1)$$

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

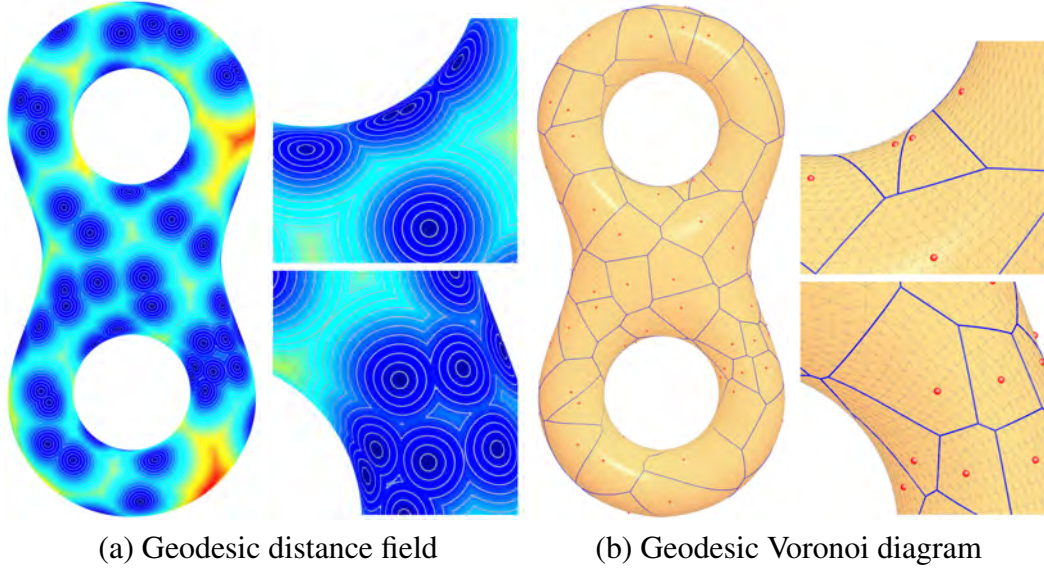


Figure 5.3: The multiple-source geodesic distance field induces a geodesic Voronoi diagram. The cold (resp. warm) color in (a) indicates the small (resp. large) geodesic distance.

general, the function  $U(x)$  is not convex, and the minimizer may not be unique. Kendall [57] and Karcher [56] 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 [85][91] for the rigorous results.

Let  $x^* \in M$  be the local minimal of Equation (5.1). Then  $x^*$  satisfies

$$\vec{0} = \nabla U(x^*) = \sum_{i=1}^k \nabla d^2(x^*, v_i) = 2 \sum_{i=1}^k d(x^*, v_i) \nabla d(x^*, v_i). \quad (5.2)$$

Since  $d(\cdot)$  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 (5.2) 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

---

**Algorithm 7** Computing the Riemannian center

---

**Require:** A set of points  $v_1, v_2, \dots, v_k$  on  $M$ ; the convergence threshold  $\delta$ ;

**Ensure:** The Riemannian center  $x$ ;

- 1:  $x \leftarrow v_1$ ;
  - 2: **do**
  - 3:    $x_0 \leftarrow x$ ;
  - 4:   Compute the exponential map  $\exp_x$  at  $x$
  - 5:   The inverse map  $\exp_x^{-1}$  brings the points  $v_i, i = 1, \dots, k$ , back to the tangent plane  $T_x M$ ;
  - 6:    $\hat{x} \leftarrow \frac{\sum_{i=1}^k \exp_x^{-1}(v_i)}{k}$ ;
  - 7:    $x \leftarrow \exp_x(\hat{x})$ ;
  - 8: **while**  $d(x, x_0) > \delta$
- 

map  $\exp_x$  at  $x$ . The exponential map  $\exp_x : T_x M \rightarrow M$  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  $T_x M$ . Let  $\hat{x} \in T_x M$  be the average of the points  $\exp_x^{-1}(v_1), \dots, \exp_x^{-1}(v_k)$ . 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 repeat the above procedure 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[85], 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 this chapter.

### 5.1.4 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-

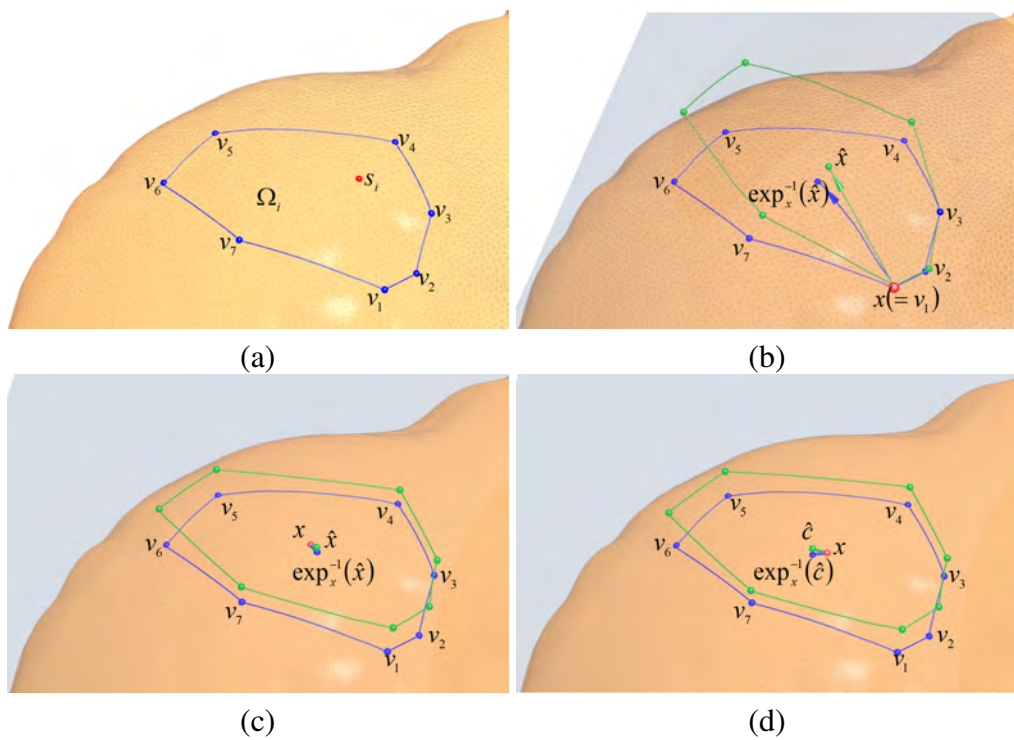


Figure 5.4: Computing the center of mass for the Voronoi cell  $\Omega_i$ . It takes two iterations (b) and (c) to obtain the Riemannian center  $c$ . The blue dot  $\exp_x^{-1}(\hat{c})$  in (d) is the center of mass. The green dots are the Voronoi cell  $\Omega_i$ 's projected images on the tangent plane.

image of  $v_i$ , which lies on the tangent plane  $T_rM$ . Since the points  $\hat{v}_i, i = 1, \dots, k$ , form a polygon on the tangent plane, its center of mass  $\hat{c} \in T_rM$  is given by

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

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

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  $\Omega$  is given by  $c = \exp_r(\hat{c})$ .

## 5.2 The L-BFGS Framework

Liu et al. [68] 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 use the algorithm in Section 4.4.4 to compute the geodesic distance between any point (not necessarily a vertex) to the source point. We use [44] to compute numerical integration on each triangle. 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 [52][30]:

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

where  $m_i = \int_{\Omega_i} \rho(x) d\sigma$ ,  $c_i$  is the center of mass of the Voronoi cell  $\Omega_i$ . We use the methods in Section 5.1.3 and Section 5.1.4 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  $exp_x : T_x M \rightarrow M$ , 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 5.5 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  $S = \{s_i | s_i \in M, i = 1, \dots, m\}$  and the convergence threshold  $\varepsilon$ ;

**Ensure:** The centroidal Voronoi tessellation on  $M$ ;

```

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

```

---

## 5.3 Experimental Results

### 5.3.1 Implementation

The combination of GTU and SVG method methentioned in Section 4.4.4 can compute the geodesic distance between arbitrary points. It can also compute the discrete exponential map on triangle meshes. Thanks to the parallel structure of the Lloyd iteration, our algorithm can be easily parallelized. Each thread takes a point (not necessarily a mesh vertex) as the source, the Dijkstra algorithm used in the SVG method encodes both the geodesic distance and the direction of the geodesic path emanating from the source. The

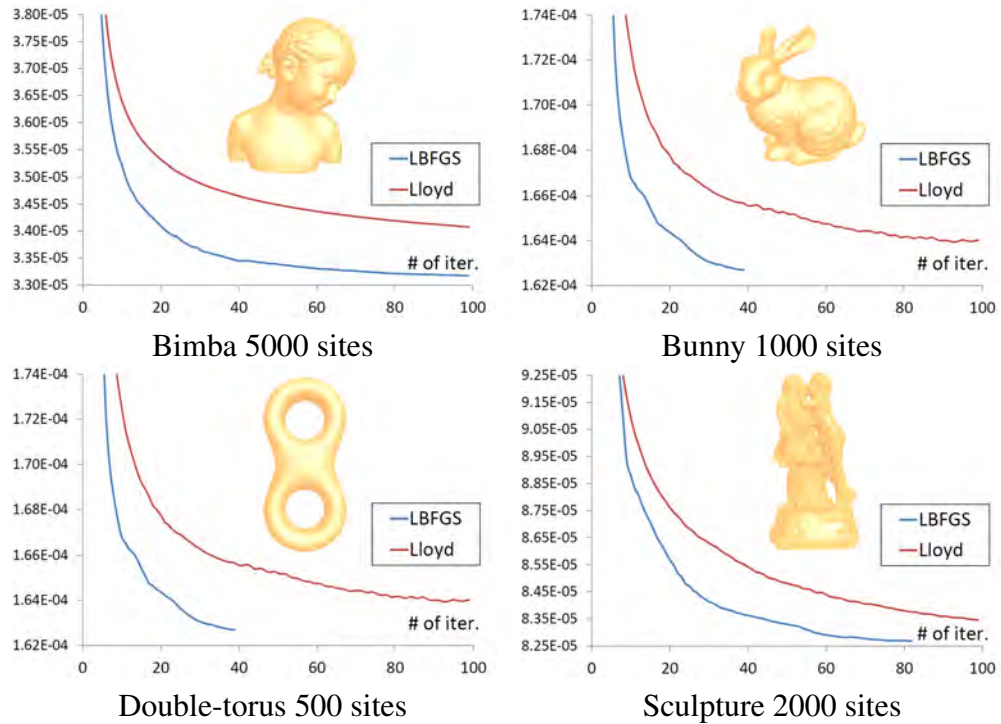


Figure 5.5: Convergence rate comparison between the Lloyd method and the L-BFGS method.

vertices are maintained in a priority queue according to the distance from the source and are propagated across the mesh faces: pops a vertex from the queue and then computes its neighbor vertices in the SVG which can add, modify, or remove existing vertices, and updates the queue accordingly. When a vertex  $v$  enters the queue, it updates  $v$ 's distance and direction, which are used for the polar coordinates. The Dijkstra 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 Dijkstra threads may compute on the overlapped regions, they do not have any data and control conflicts, so each thread can proceed independently.

Model	$g$	$ V $	$m$	$T$	$Q_{min}$	$Q_{avg}$	$\theta_{min}$	$\theta_{avg}$
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 5.1: 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.

### 5.3.2 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 5.1 lists the model complexity and the performance of our algorithm and Figure 5.10 shows the computed CVT on some 3D models. Figure 5.11 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 [41] [124]:

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

Figure 5.6 shows the quality improvement via the Lloyd iteration.

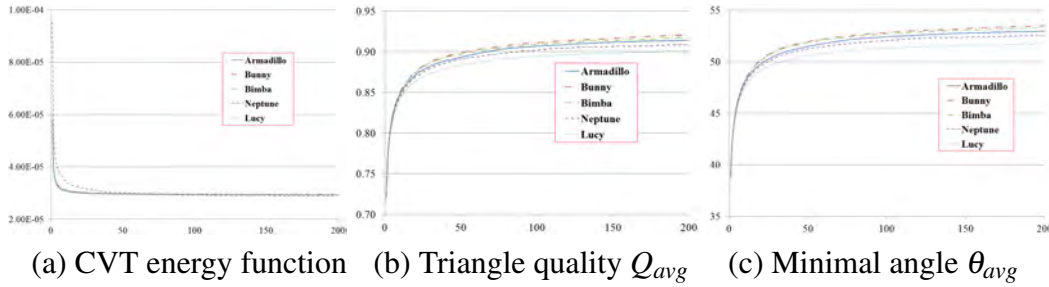


Figure 5.6: Energy function and quality measures. The horizontal axis in the plots shows the iteration number. The vertical axis in (a) is the *normalized* CVT energy function, that is,  $\frac{F(S)}{A^2}$ , where  $A$  is the area of the model.

Compared to the parameterization-based methods [3], [89] [90], 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 5.12 shows, our method outperforms the UCS method [89] and the RVD method [124] in terms of quality (higher angle measure  $Q_{ave}$  and lower number of singularities).

The restricted Voronoi diagram methods [124] [125] 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 5.7 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 “ambiguities”. To further demonstrate the

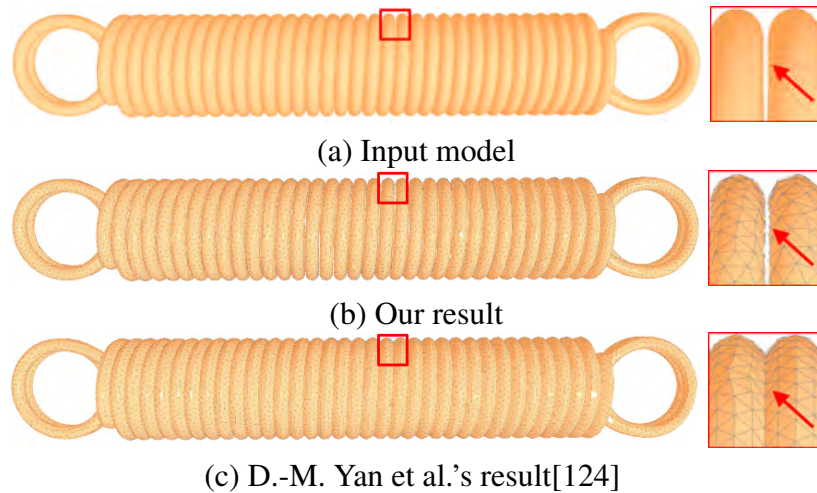


Figure 5.7: Intrinsic vs. extrinsic. Consider the Coil Spring model, where the pitch of the helix equals the diameter of the coil. Therefore, the coil almost touches itself and leaves very small gap. See the closeup view. As an intrinsic method, our method is independent of the embedding space and it can correctly separate the coil. The extrinsic method [124] 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.

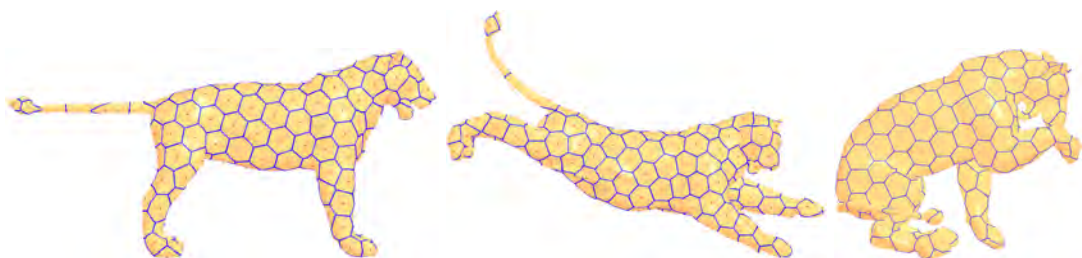


Figure 5.8: Due to its intrinsic property, our method can produce consistent results on the various poses of the Lion model.

efficacy of our intrinsic method, we apply it to the Lion model in various poses. As Figure 5.8 shows, the computed CVTs are consistently among the near-isometric poses. Figure 5.9 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.

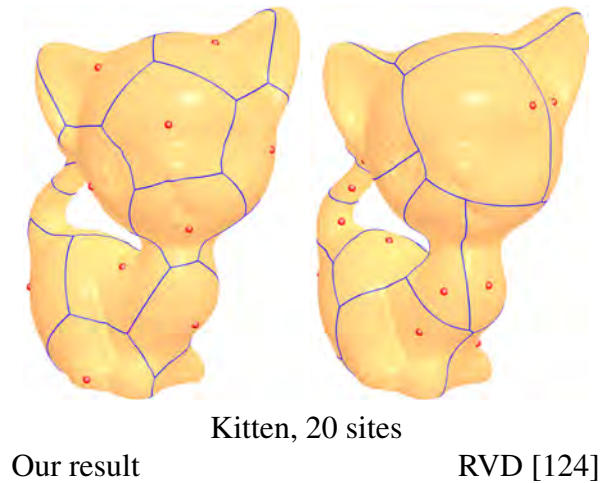


Figure 5.9: The intrinsic CVT and extrinsic RVD with very few sites.

### 5.3.3 Limitation

The limitation of our intrinsic CVT method is that it requires the triangle mesh as the computation domain and cannot be computed directly on point clouds. If the input data is point clouds we need to reconstruct the surface first and then to compute intrinsic CVT on the surface. In the future we hope to develop ways to directly compute the intrinsic CVT on point clouds.

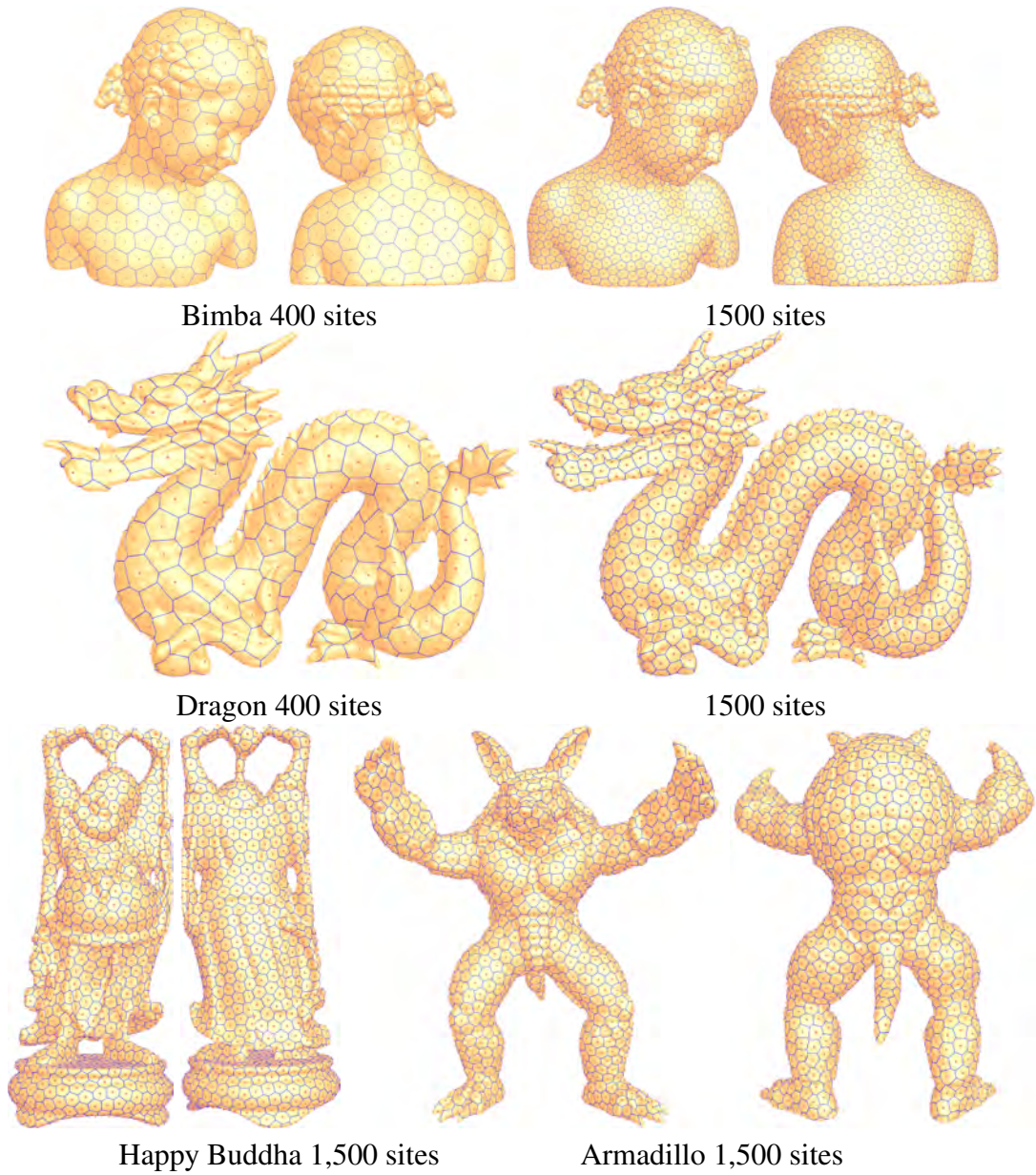


Figure 5.10: Experimental results. Images are rendered in high resolution that allows close-up examination.

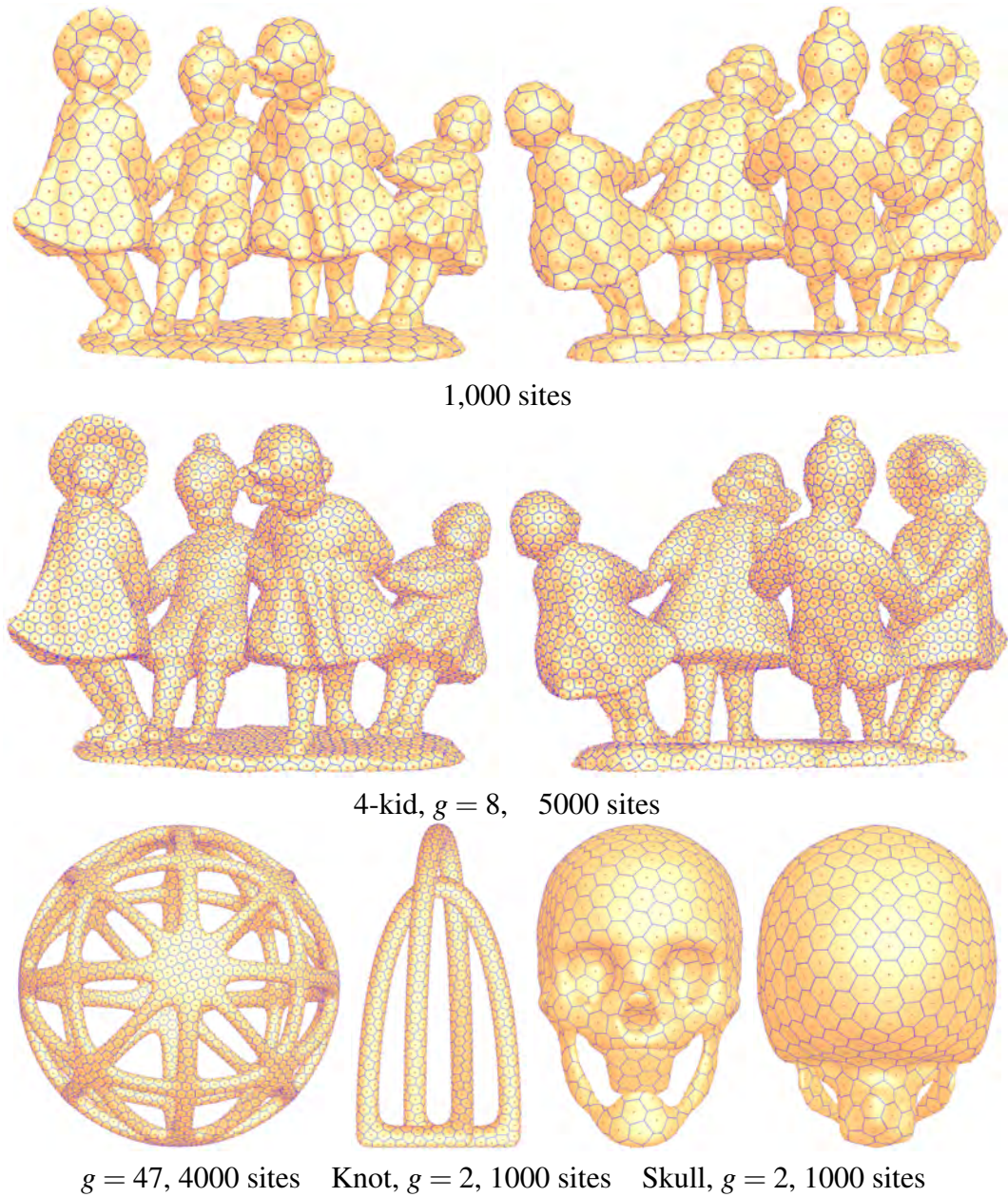


Figure 5.11: Experimental results on high-genus models.

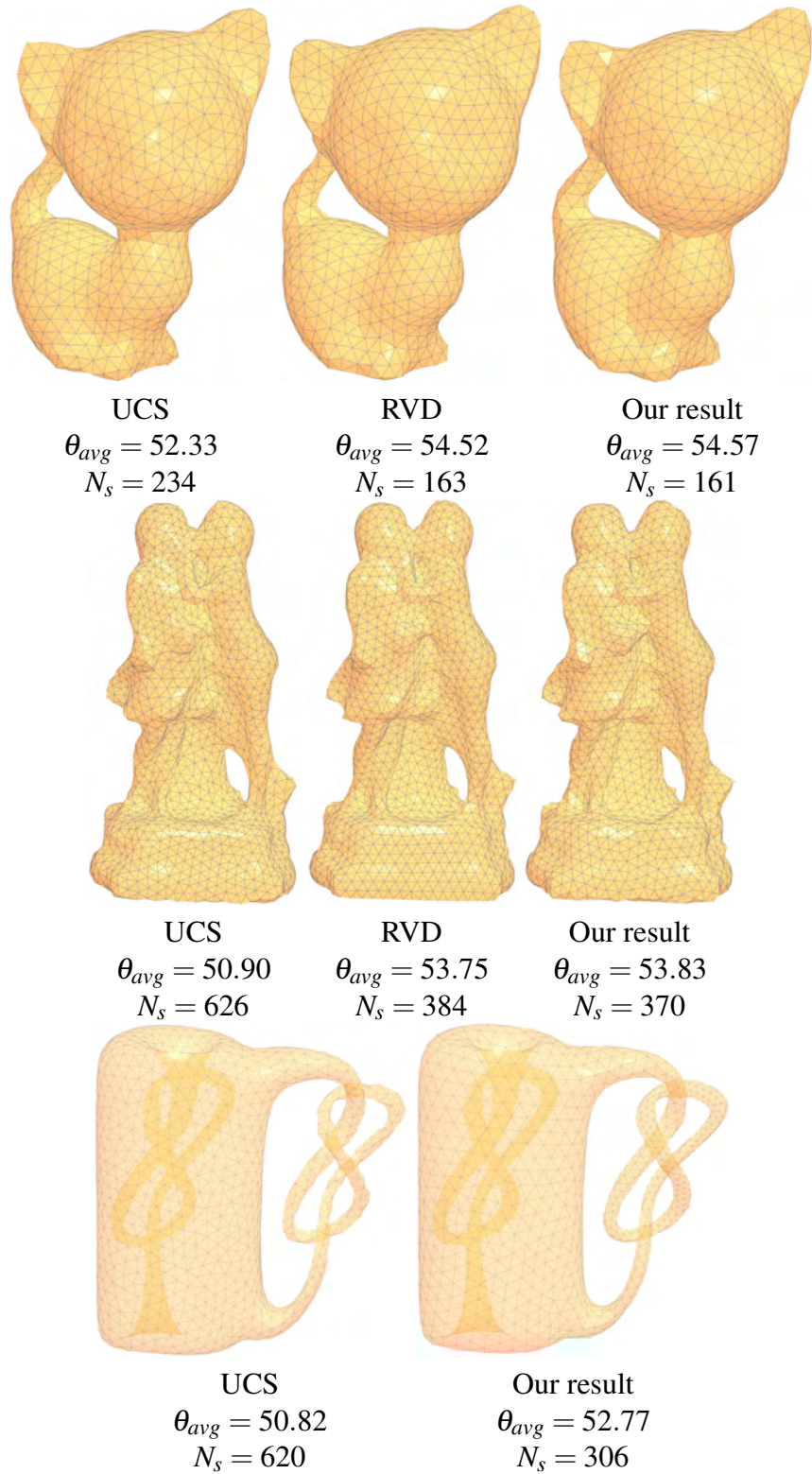


Figure 5.12: Comparison to the RVD method [125] and the UCS method [89].  $N_s$  denotes the number of singularities (i.e., vertices whose valence is not six).

# Chapter 6

## Application

In this chapter we present one application of CVT and discrete geodesics, an automatic method for computing anisotropic 2D shape distribution on the mesh. 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 collision-free, which is a critical constraint in many applications. It is able to compute multi-class shape distribution. 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 meshes of complicated topology and geometry . 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. We first introduce a key technique called geodesic polar coordinates, then we present the anisotropic shape distribution algorithm and the experimental results.

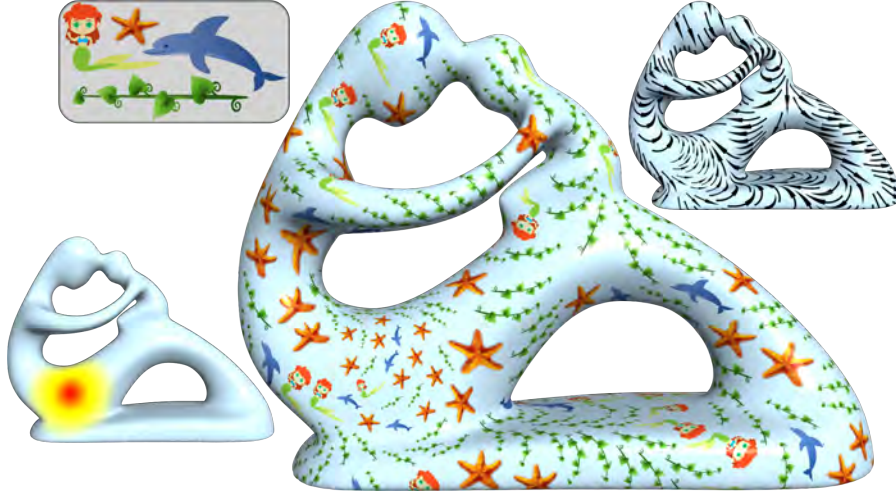


Figure 6.1: Given a 3D triangle mesh  $M$  and a set of 2D anisotropic shapes, the user first specifies a vector field for directional control (see the top-right inset) and a scalar field for density control (see the bottom-left inset). Then our method automatically distributes the given 2D shapes onto  $M$ , 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.

## 6.1 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 [65], is not a good choice, since global parameterization is computationally expensive 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  $(\rho, \theta)$ , where  $\rho$  is the geodesic distance between  $p$  and  $q$ , and  $\theta$ , the polar angle, corresponds to the tangent direction  $\mathbf{v}$ . Differential



(a) Geodesic offsets



(b) Parameterization result

Figure 6.2: 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  $\delta$  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.

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.

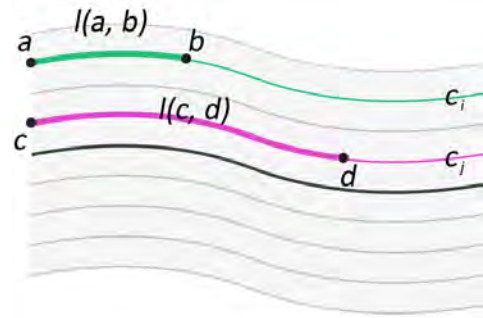
*Discrete* exponential map has many applications in computer graphics and digital geometry processing, for example, surface decaling [93], Poisson disk sampling [128], intrinsic CVT computation [115], just name a few. Moreover, exponential map can be extended to a general setting, where the source is a curve [107]. In spite of its popularity, 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  $\gamma_{v_1}$ ,  $\gamma_{v_2}$  of equal length  $\rho$  meet at a point  $q$ , then both  $(\rho, \theta_1)$  and  $(\rho, \theta_2)$  refer to the same point  $q$ . Second, as shown in [126], when a geodesic path passes through a saddle vertex (whose cone angle is more than  $2\pi$ ), 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. [107]

<sup>1</sup>When  $\rho$  is less than the injective radius of  $p$ , the minimizing geodesic is unique, so that the geodesic distance  $\rho$  and the polar angle can *uniquely* determine a point in the neighborhood.

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 [92] 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 when 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,  $c_i$  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  $u$  value of each point is only related to the arc-length on the corresponding iso-curve, meaning that the polar coordinate is guaranteed to be bijective. In contrast to [107], 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 [107]. See Figure 6.3.

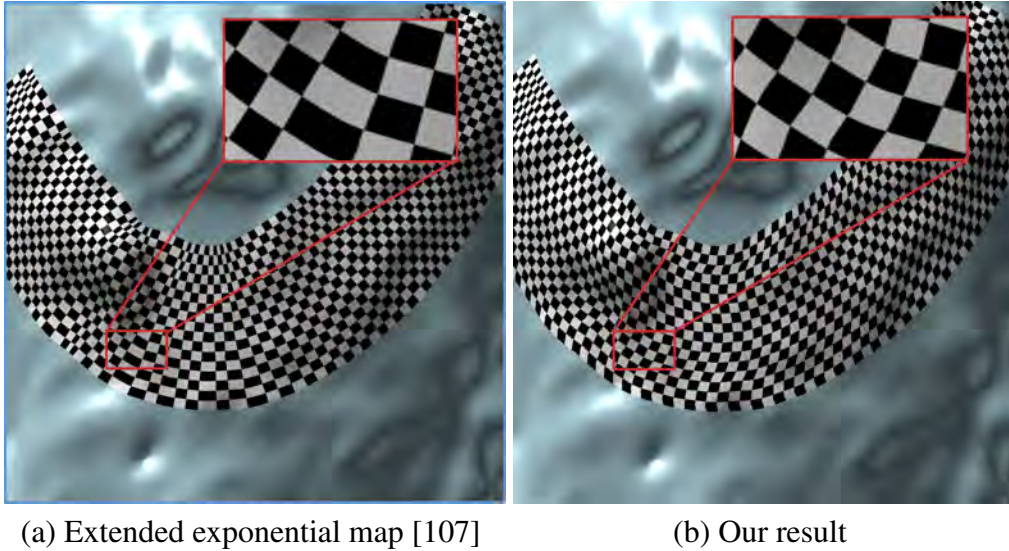


Figure 6.3: Comparison with extended exponential map [107]. Our result has comparable quality as Sun et al.’s method [107]. Since their method computes a harmonic map to fix the non-bijective issue of the exponential map, it is more computationally expensive than ours. It takes their method 1.7 seconds to parameterize the 10K-vertex patch, whereas our method spends only 0.18 seconds.

## 6.2 Shape Distribution Algorithm

A simple algorithm for distributing shapes is the traditional dart throwing method. Dart throwing method can generate uniformly and randomly distributed samples but is not efficient. It throws a large number of darts and only accepts a small percentage of them. And it is not easy to be parallelized. Thus we present an improved algorithm inspired from Ying et al. [128]. We first randomly generate a large number of points on the surface following Osada et al. [82]’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 serves as the center line of geodesic

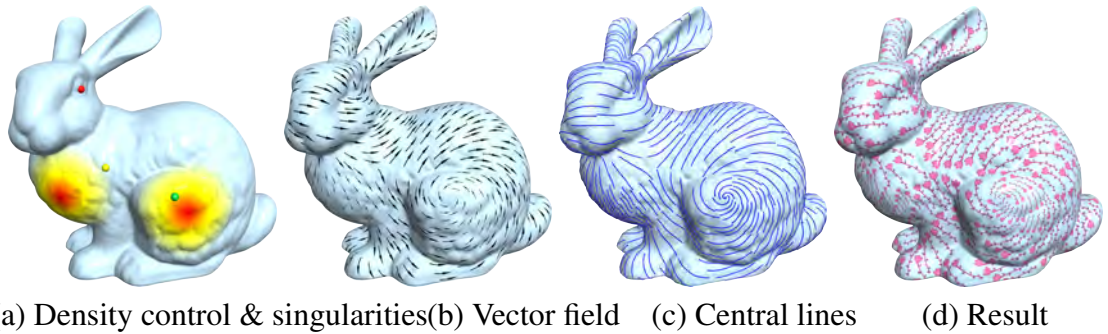


Figure 6.4: Algorithmic pipeline of single-class shape distribution. (a) User specifies a scalar field to control the shape density and the locations of singularities and their indices. (b) We adopt Crane et al.’s method [21] to compute the vector field which controls the shape orientation. (c) Our method 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.

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 have conflicts. Then we use a technique introduced in Section 6.2.1 called *Minimum Safe Distance Table* to check conflicts and reject all the shapes, which have conflicts. The algorithm terminates when all predefined curves have been processed. To parallelize the algorithm, we set a *unique random* value to each shape. The value represents the shape’s priority. 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.

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 field by choosing singularities. Then we will generate a vector field using Crane et al.’s method [21]. We distribute shapes along the vector field adaptively and anisotropically. Please refer to Figure 6.4 for the algorithm overview.

After the density field is generated, we can get a user specified spatial variety function  $g(\star) \rightarrow [1, M]$ , then an adaptive distribution is defined. Here  $M$  is the max value of the density field, and min density is set to 1. We need to compute a geodesic region with radius  $(1 + M)r$  for each point. Here  $r$  is the radius for point with minimum density. For point  $p$  and  $q$ , we use the new min safe distance (the detail of min safe distance will explained in Section 6.2.1):

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

.

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 of 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. [21]. Our system allows the user to set singularities and direction constraints on the surface directly, as shown in Figure 6.4(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 [21]. The user also can sketch strokes on the surface to specify the desired vector field direction. According to the strokes' 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.

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  $M$ .

### 6.2.1 Single-class Shape Distribution

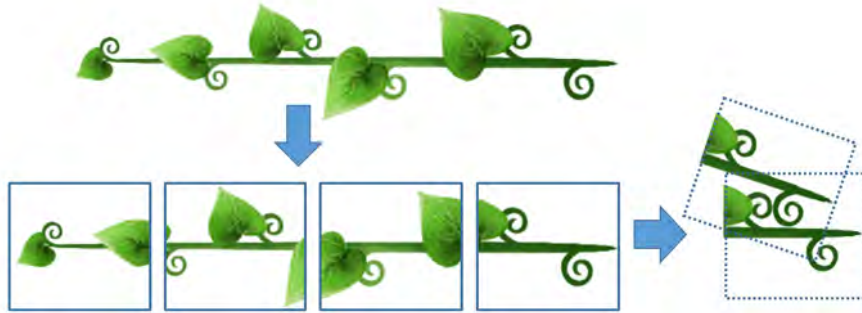


Figure 6.5: For long shapes, we divide it to several parts and test the intersection separately.

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 split the long shape into equal size parts and check each part using precomputed distance table. Figure 6.5 gives a brief illustration.

Now we introduce the details of distance table technique. Given a 2D object  $S$ , we compute the smallest covering circumcircle  $\odot(c, r)$ , here  $r$  is the radius and  $c$  is the center (see Figure 6.7). We can compute the circle in linear time using [76]. 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.

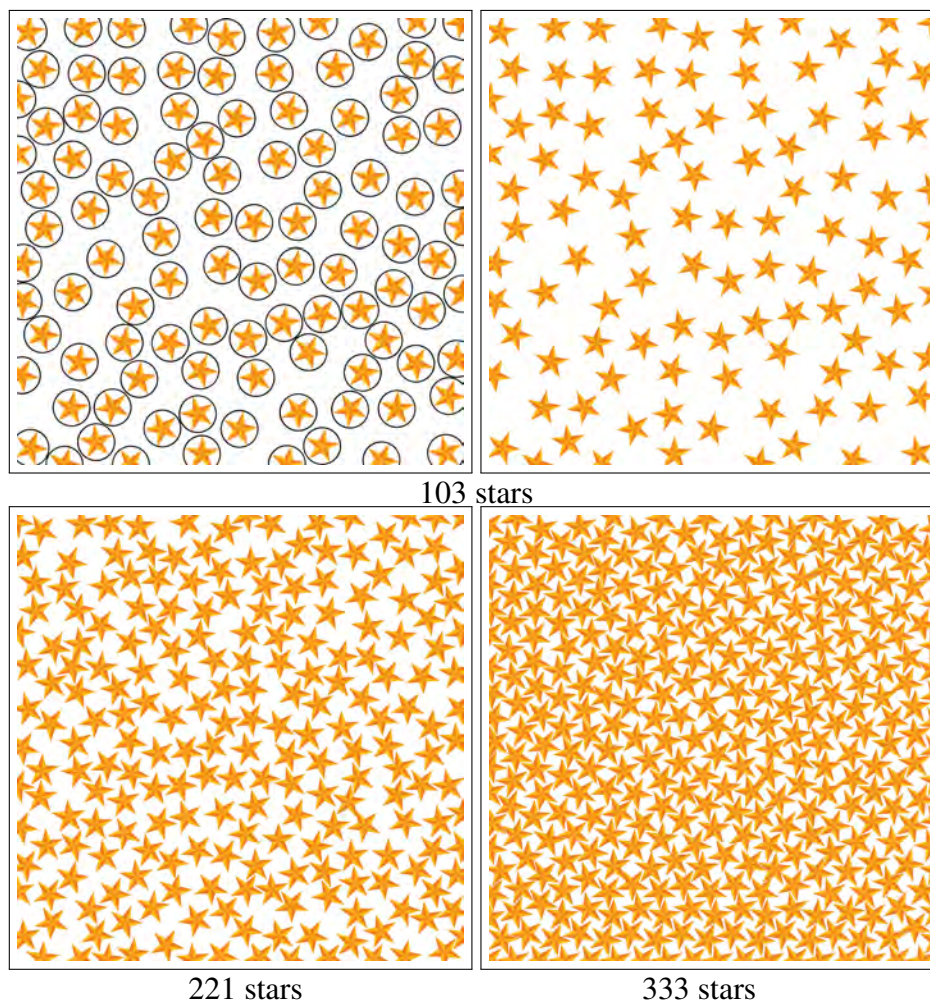


Figure 6.6: Comparison with Poisson disk sampling. Row 1: Poisson disk sampling considers each sample as a disk and does not allow overlapping disks. Row 2: Our method allows a more dense distribution than Poisson disk sampling, as long as the stars are not overlapping.

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 6.7(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  $c_A c_B$ , denoted by  $\alpha$  and  $\beta$ , 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 search of  $[0, 2r]$  in  $O(D^2 e^2 \log r)$  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.

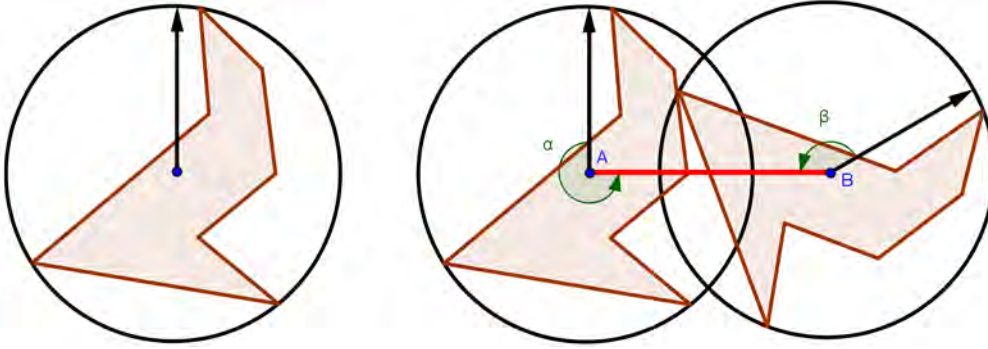


Figure 6.7: Left: circumcircle of shape. Right: The minimum safe distance between two shapes (red line).

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

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  $(\alpha, \beta, d_g)$  for each point when computing the geodesic polar coordinates. Thus it is quite efficient and does not require additional computation cost.

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

Algorithm input: vector field for each class  $O_i$ , and target number of each class  $N_i$ .

---

**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  $S_i.myIndex \leftarrow +\infty$   
 $curveOrder \leftarrow 0$   
 $result \leftarrow \emptyset$   
  
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  $result$ 
```

---

## 6.2.2 Multi-Class Shape Distribution

Our method is different from the multi-class blue noise sampling in [119], since we do not allow overlapped shapes.

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

Similar to single shape distribution, we generate a dense point set. Each point is 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  $r_m \leftarrow \max r_i, i = 1 \dots k$ . For each point of shape  $i$ , we compute a geodesic disk with radius  $r_i + r_m$ , and collect all neighbor points. Then we check the MSD by looking up the table  $D_{i,*}(\alpha, \beta)$ .

## 6.3 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 6.2 for running time performance. Figures 6.8, 6.9 and 6.12 demonstrate our results on single-class and multi-class shape distribution.

Table 6.1: 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.  $\epsilon$ : 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 $	$T_{ICH}$	$T_{SVG}$	$T_{our}$	$\epsilon$
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 6.2: 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

**Robustness.** Our method is intrinsic since both the collision detection and shape distribution depend on the metric only. Taking advantage of the intrinsic nature of the geodesic algorithms [126, 122], our method is insensitive to model resolution and tessellation. See Figure 6.10 for our results on the Bimba mesh with various tessellations.

**Comparison to [65].** Our method and the anisotropic blue noise sampling method [65] 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 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,

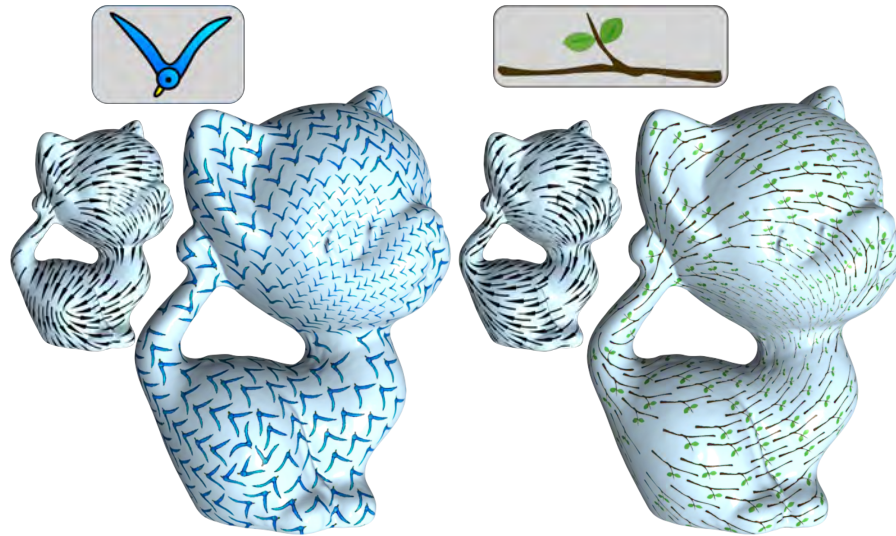


Figure 6.8: Single-class shape distribution on the Kitten model with different density and texture.

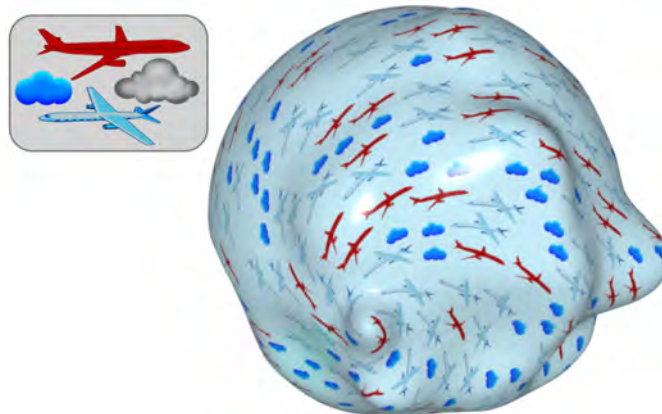


Figure 6.9: Multi-class shape distribution on a bumpy sphere with 50K faces.

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 6.11. Last but not the least, results from [65] demonstrate single-class sampling only, whereas our method can compute multi-class shape distribution.



Figure 6.10: Our method works well on the Bimba model with various resolutions.

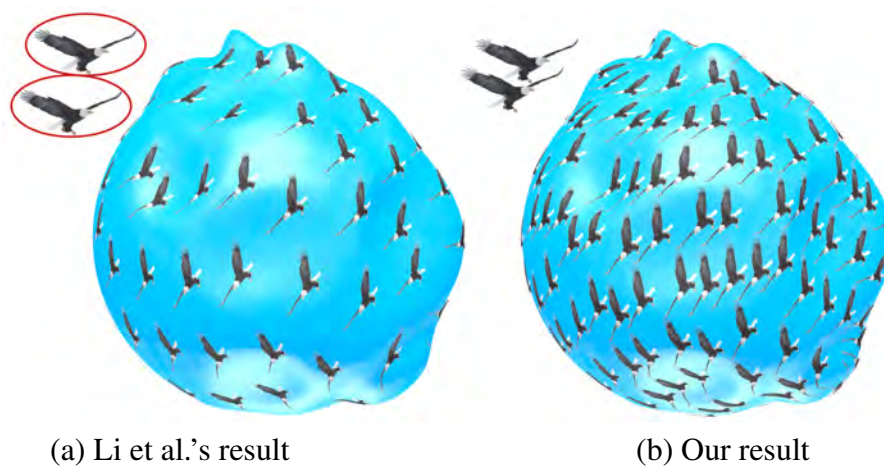
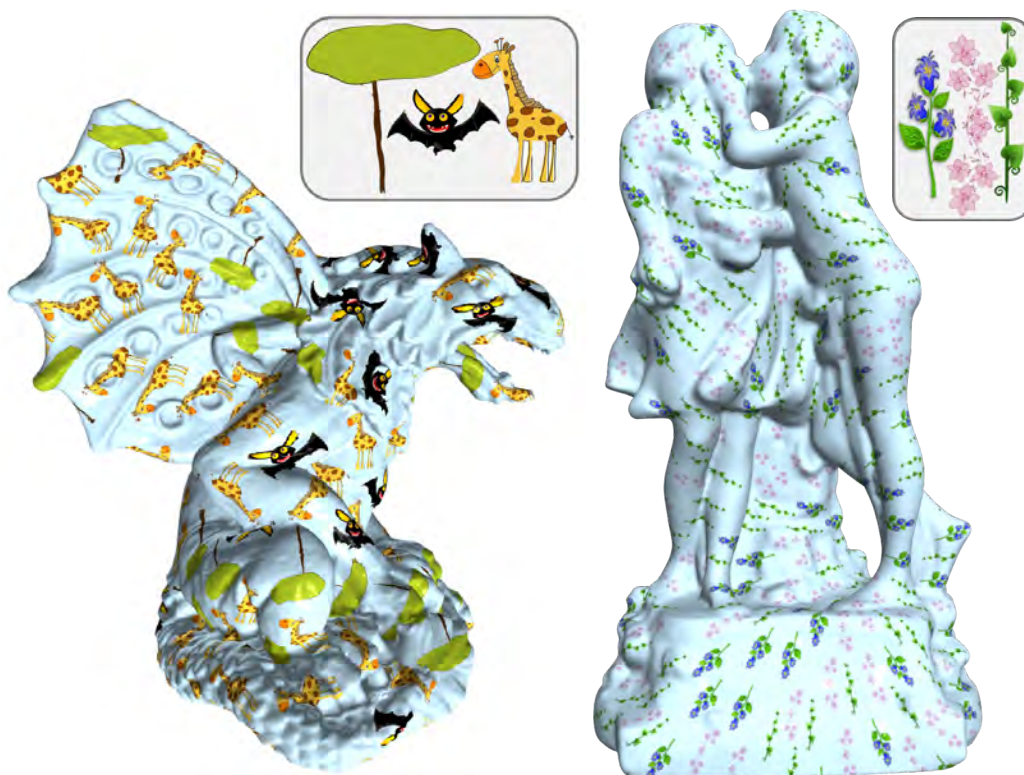


Figure 6.11: Comparison to [65]. The 2D Eagle shape is highly concave, so a simple bounding ellipse cannot capture its geometric features. As a result, requiring non-overlapping ellipses is very pessimistic. Li et al.'s method [65] can distribute only 115 shapes. Obviously, the distribution is not maximal. 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.



# Chapter 7

## Conclusions and Future Work

### 7.1 Conclusions

This thesis focuses on computing Voronoi Diagram and Centroidal Voronoi Tessellation on surfaces. We also propose an efficient algorithm based on SVG to solve the discrete geodesic problem. With SVG, we can compute intrinsic CVT on surfaces and avoid the defects of the existing extrinsic methods. We also demonstrate one applications of CVT and discrete geodesics, anisotropic shape distribution on meshes.

In Chapter 3 we present the untransformed sweep circle 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. 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.

In Chapter 4 we propose the SVG data structure to tackle the discrete geodesics problem. It 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 large number of real-world models demonstrate that our method significantly outperforms the existing approximate algorithms in both accuracy and speed.

In Chapter 5 we present 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. 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.

In Chapter 6 we present one application of CVT and discrete geodesics. We present 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 guarantee the

distribution is collision-free, which is a critical constraint in many applications. Second, it is able to compute multi-class shape distribution. 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.

## 7.2 Future Work

Computing intrinsic surface Voronoi Diagram and Centroidal Voronoi Tessellation requires efficient method to obtain geodesic distance. The SVG is an efficient method compared to the existing discrete geodesic like ICH or MMP, but it has several disadvantages. The SVG method requires a user input parameter  $K$ , and it's not intuitive how to adjust the parameter  $K$  to obtain the desired accuracy. We are developing a more efficient graph-theoretic frame work for computing discrete geodesics. The new framework allows user to directly control the approximation error, while the existing approaches such as the SVG requires a fine tuned parameter  $K$ . Furthermore, the new frame work is for arbitrary triangle meshes, whereas SVG applies to triangle meshes with sufficient number of saddle vertices.

We can use intrinsic Centroidal Voronoi Tessellation to segment the input mesh to super-patches. The super-patches can greatly reduce the number of faces of the input mesh, while maintaining the geometries boundaries. The existing method for computing super-

patches is extrinsic and computationally inefficient. We plan to use our intrinsic CVT method to solve the super-patch problem efficiently.

For the shape distribution project, we found it hard to theoretically guarantee the distribution is maximal. In the future we hope to provide a theoretical proof that the method is maximal, and if it's not, we plan to find the modified algorithm to guarantee the maximal property.

# References

- [1] CGAL, Computational Geometry Algorithms Library. <http://www.cgal.org>. viii, 46, 47
- [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. 11
- [3] 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. 15, 89
- [4] F. Aurenhammer. Voronoi diagrams - a survey of a fundamental geometric data structure. *ACM Comput. Surv.*, 23(3):345–405, 1991. 10
- [5] D. Avis and B. Bhattacharya. Algorithms for computing d-dimensional Voronoi diagrams and their duals. *Adv. Comput. Res.*, 1:159–180, 1983. 12
- [6] 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. 22
- [7] A. Belyaev and P.-A. Fayolle. On variational and PDE-based distance function approximations. *Computer Graphics Forum*, 2015. 19
- [8] D. P. Bertsekas. A simple and fast label correcting algorithm for shortest paths. *Networks*, 23(8):703–709, 1993. 65
- [9] 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. 65
- [10] 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. 8, 22, 23
- [11] W. Browstow, J.-P. Dussault, and B. L. Fox. Construction of Voronoi polyhedra. *J. Comput. Phys.*, 29:81–97, 1978. 11
- [12] M. Campen and L. Kobbelt. Walking on broken mesh: Defect-tolerant geodesic distances and parameterizations. *Comput. Graph. Forum*, 30(2):623–632, 2011. 19
- [13] 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. 12

## REFERENCES

---

- [14] 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. 12
- [15] 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. 22
- [16] J. Chen and Y. Han. Shortest paths on a polyhedron. In *SCG '90*, pages 360–369, 1990. 2, 3, 17
- [17] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry II. *Discrete Comput. Geom.*, 4:387–421, 1995. 12
- [18] 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. 22
- [19] R. Cole, M. Goodrich, and C. Dunlaing. Merging free trees in parallel for efficient Voronoi diagram construction. Technical report, Johns Hopkins University, 1989. 12
- [20] R. L. Cook. Stochastic sampling in computer graphics. *ACM Trans. Graph.*, 5:51–72, 1986. 22
- [21] K. Crane, M. Desbrun, and P. Schröder. Trivial connections on discrete surfaces. *Computer Graphics Forum*, 29(5):1525–1533, 2010. 100, 101
- [22] 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. 19
- [23] 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. 5, 11, 25, 51, 52
- [24] M. Detrixhe, F. Gibou, and C. Min. A parallel fast sweeping method for the eikonal equation. *J. Comput. Phys.*, 237:46–55, 2013. 19
- [25] O. Devillers. Improved incremental randomized delaunay triangulation. In *ACM SoCG*, pages 106–115. ACM, 1998. 11
- [26] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959. 64
- [27] J. a. Dinis and M. Mamede. Sweeping the sphere. In *Proceedings of the 2010 International Symposium on Voronoi Diagrams in Science and Engineering, ISVD '10*, pages 151–160, Washington, DC, USA, 2010. IEEE Computer Society. 11
- [28] M. P. do Carmo. *Differential Geometry of Curves and Surfaces*. Prentice-Hall, Inc., 1976. 20

## REFERENCES

---

- [29] 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. 15
- [30] Q. Du, V. Faber, and M. Gunzburger. Centroidal voronoi tessellations: Applications and algorithms. *SIAM Rev.*, 41(4):637–676, Dec. 1999. 3, 13, 22, 85
- [31] D. Dunbar and G. Humphreys. A spatial data structure for fast Poisson-disk sample generation. *ACM Transactions on Graphics*, 25:503–508, 2006. 22
- [32] R. Dwyer. A faster divide-and-conquer algorithm for constructing Delaunay triangulations. *Algorithmica*, 2(1):137–151, 1987. 11
- [33] 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. 13
- [34] 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. 22
- [35] H. Edelsbrunner and N. Shah. Triangulating topological spaces. *International Journal of Computational Geometry and Applications*, 7(4):365–378, 1997. 13
- [36] R. Fattal. Blue-noise point sampling using kernel density model. *ACM Transactions on Graphics*, 30(4):48:1–48:12, 2011. 22
- [37] 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. 22
- [38] J. L. Finney. A procedure for the construction of Voronoi polyhedra. *J. Comput. Phys.*, 32:137–143, 1979. 11
- [39] S. Fortune. A sweepline algorithm for Voronoi diagrams. In *SCG '86: Proc. 2nd annual symposium on Computational geometry*, pages 313–322, 1986. 1, 11, 26, 40
- [40] P. Foteinos and N. Chrisochoides. Dynamic parallel 3d delaunay triangulation. In *International Meshing Roundtable*, pages 9–26, Paris, France, October 2011. SANDIA. 10
- [41] P. Frey and H. Borouchaki. Surface mesh evaluation. In *6th International Meshing Roundtable*, pages 363–374, 1997. 88
- [42] 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. 19
- [43] M. N. Gamito and S. C. Maddock. Accurate multidimensional Poisson-disk sampling. *TOG*, 29:8:1–8:19, 2009. 22
- [44] A. Genz and R. Cools. An adaptive numerical cubature algorithm for simplices. *ACM Trans. Math. Softw.*, 29(3):297–308, Sept. 2003. 85

## REFERENCES

---

- [45] P. J. Green and R. Sibson. Computing dirichlet tessellations in the plane. *The Computer Journal*, 21(2):168–173, 1978. 11, 12
- [46] 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. 16, 47
- [47] X. Gu and S.-T. Yau. Global conformal parameterization. In *Symposium on Geometry Processing*, pages 127–137, 2003. 47
- [48] 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. 12
- [49] 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. 11
- [50] A. Hausner. Simulating decorative mosaics. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 573–580, 2001. 22
- [51] 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. 12
- [52] 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. 85
- [53] L. Ju, Q. Du, and M. Gunzburger. Probabilistic methods for centroidal voronoi tessellations and their parallel implementations. *Parallel Computing*, 28(10):1477–1500, 2002. 15
- [54] T. Kanai and H. Suzuki. Approximate shortest path on a polyhedral surface and its applications. *Computer-Aided Design*, 33(11):801–811, 2001. 20
- [55] 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. 18
- [56] H. Karcher. Riemannian center of mass and mollifier smoothing. *Comm. Pure Appl. Math*, 30:509–541, 1977. 82
- [57] W. Kendall. Probability, convexity, and harmonic maps with small image i: uniqueness and fine existence. *Proc. London Math. Soc.*, 61(2):371–406, 1990. 82
- [58] R. Kimmel and J. A. Sethian. Computing geodesic paths on manifolds. In *Proc. Natl. Acad. Sci. USA*, pages 8431–8435, 1998. 19

## REFERENCES

---

- [59] V. Klee. On the complexity of d-dimensional Voronoi diagrams. *Archiv. Math.*, 34:75–80, 1980. 12
- [60] 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. 11
- [61] 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. 22
- [62] A. Lagae and P. Dutré. A procedural object distribution function. *ACM Trans. Graph.*, 24:1442–1461, 2005. 22
- [63] 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. 16
- [64] B. Lévy and Y. Liu. Lp centroidal voronoi tessellation and its applications. *ACM Trans. Graph.*, 29(4):119:1–119:11, July 2010. 15
- [65] H. Li, L.-Y. Wei, P. V. Sander, and C.-W. Fu. Anisotropic blue noise sampling. *ACM Transactions on Graphics*, 29(6):167, 2010. x, 22, 23, 96, 107, 108, 110
- [66] D. Liu and J. Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical programming*, 45(1):503–528, 1989. 15
- [67] 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. 13, 81
- [68] 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. 4, 15, 85
- [69] Y.-J. Liu. Exact geodesic metric in 2-manifold triangle meshes using edge-based data structures. *Computer-Aided Design*, 45(3):695–704, 2013. 13, 18
- [70] 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. 13
- [71] 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
- [72] S. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129 – 137, 1982. 15
- [73] T. O. M. Tanemura and N. Ogita. A new algorithm for three-dimensional Voronoi tessellation. *J. Comput. Phys.*, 32:137–143, 1983. 11

## REFERENCES

---

- [74] 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. 15
- [75] 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
- [76] 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. 102
- [77] 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. 19
- [78] D. P. Mitchell. Spectrally optimal sampling for distribution ray tracing. In *SIGGRAPH '91*, pages 157–164, 1991. 22
- [79] J. S. B. Mitchell, D. M. Mount, and C. H. Papadimitriou. The discrete geodesic problem. *SIAM J. Comput.*, 16(4):647–668, 1987. 2, 3, 17, 18, 55, 69, 71, 73
- [80] D. Nehab and P. Shilane. Stratified point sampling of 3D models. In *PBG '04*, pages 49–56, 2004. 22
- [81] A. Okabe, B. Boots, K. Sugihara, and S. N. Chiu. *Spatial tessellations: Concepts and applications of Voronoi diagrams*. Wiley, 2nd edition, 2000. 3, 10
- [82] R. Osada, T. Funkhouser, B. Chazelle, and D. Dobkin. Shape distributions. *ACM Trans. Graph.*, 21:807–832, 2002. 99
- [83] V. Ostromoukhov. Sampling with polyominoes. *ACM Trans. Graph.*, 26, 2007. 22
- [84] A. C. Öztireli, M. Alexa, and M. H. Gross. Spectral sampling of manifolds. *ACM Trans. Graph.*, 29(6):168, 2010. 22
- [85] X. Pennec. Intrinsic statistics on riemannian manifolds: Basic tools for geometric measurements. *Journal of Mathematical Imaging and Vision*, 25(1):127–154, 2006. 82, 83
- [86] I. Pohl. Bi-directional search. *Machine Intelligence*, 6:124–140, 1971. 66
- [87] K. Polthier and M. Schmies. chapter Straightest Geodesics on Polyhedral Surfaces. 55
- [88] 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. 12
- [89] 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. ix, 15, 89, 94

## REFERENCES

---

- [90] 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. 12, 16, 89
- [91] R. M. Rustamov. Barycentric coordinates on surfaces. *Comput. Graph. Forum*, 29(5):1507–1516, 2010. 82
- [92] R. Schmidt. Stroke parameterization. *Computer Graphics Forum*, 32(2pt2):255–263, 2013. 98
- [93] R. Schmidt, C. Grimm, and B. Wyvill. Interactive decal compositing with discrete exponential maps. *ACM Trans. Graph.*, 25(3):605–613, July 2006. 21, 97
- [94] Y. Schreiber. An optimal-time algorithm for shortest paths on realistic polyhedra. *Discrete & Computational Geometry*, 43(1):21–53, 2010. 18
- [95] 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. 18
- [96] R. Seidel. Small-dimensional linear programming and convex hulls made easy. *Discrete Comput. Geom.*, 6:423–434, 1991. 12
- [97] 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. 19
- [98] M. I. Shamos and D. Hoey. Closest-point problems. In *FOCS*, pages 151–162, 1975. 11
- [99] H. Sheung, S.-P. Mok, and C. C. Wang. A highly parallel approach to meshing scattered point data. In *ASME 2009 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, pages 559–569. American Society of Mechanical Engineers, 2009. 4
- [100] H. Sheung and C. C. L. Wang. Robust mesh reconstruction from unoriented noisy points. In *2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling, SPM '09*, pages 13–24, New York, NY, USA, 2009. ACM. 4
- [101] J. Shewchuk. Delaunay refinement algorithms for triangular mesh generation. *Computational geometry*, 22(1):21–74, 2002. 10
- [102] J. R. Shewchuk. Triangle: Engineering a 2d quality mesh generator and delaunay triangulator. *FCRC '96/WACG '96*, pages 203–222, 1996. viii, 46, 47
- [103] 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. 4
- [104] S. W. Sloan. A fast algorithm for constructing Delaunay triangulations in the plane. *Adv. Eng. Softw.*, 9:34–55, 1987. 11

## REFERENCES

---

- [105] A. Spira and R. Kimmel. An efficient solution to the Eikonal equation on parametric manifolds. *Interfaces and Free Boundaries*, 6(4):315–327, 2004. 19
- [106] 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, 12
- [107] 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. ix, 21, 97, 98, 99
- [108] 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. 22
- [109] 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. 3, 18, 66
- [110] 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. 51
- [111] J. Tsitsiklis. Efficient algorithms for globally optimal trajectories. *IEEE Transactions on Automatic Control*, 40(9):1528–1538, 1995. 19
- [112] G. Turk. Re-tiling polygonal surfaces. In *SIGGRAPH '92*, pages 55–64, 1992. 22
- [113] C. C. Wang, Y. Wang, K. Tang, and M. M. Yuen. Reduce the stretch in surface flattening by finding cutting paths to the surface boundary. *Computer-Aided Design*, 36(8):665–677, 2004. 19
- [114] 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
- [115] 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. 97
- [116] D. F. Watson. Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes. *The Computer Journal*, 24(2):167–172, 1981. 12
- [117] 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. 19
- [118] L.-Y. Wei. Parallel poisson disk sampling. In *ACM SIGGRAPH 2008 papers*, SIGGRAPH '08, pages 20:1–20:9, 2008. 8, 22
- [119] L.-Y. Wei. Multi-class blue noise sampling. *ACM Trans. Graph.*, 29, 2010. 22, 106

## REFERENCES

---

- [120] 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. 20
- [121] 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, 71, 73
- [122] 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. 65, 69, 71, 72, 107
- [123] 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. 18, 71
- [124] 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. 1, 4, 16, 88, 89, 90, 91
- [125] 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. ix, 12, 89, 94
- [126] 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. 65, 71, 97, 107
- [127] 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
- [128] 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. 21, 97, 99
- [129] H. Zhao. Parallel implementations of the fast sweeping method. *Journal of Computational Mathematics*, 25(4):421–429, 2007. 19
- [130] 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. 11