PARALLEL DISCRETE GEODESIC ALGORITHMS

DU JIE
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
2020

PARALLEL DISCRETE GEODESIC ALGORITHMS

DU JIE

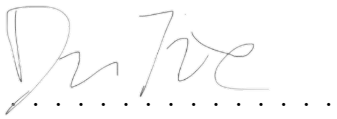
School of Computer Science and Engineering

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

2020

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research, is free of plagiarised materials, and has not been submitted for a higher degree to any other University or Institution.

10/09/2020 . . . . .  . . . . . .

Date Du Jie

Supervisor Declaration Statement

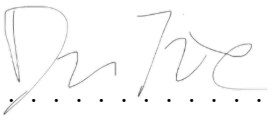
I have reviewed the content and presentation style of this thesis and declare it is free of plagiarism and of sufficient grammatical clarity to be examined. To the best of my knowledge, the research and writing are those of the candidate except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

10/09/2020 

Date He Ying

Authorship Attribution Statement

This thesis does not contain any materials from papers published in peer-reviewed journals or from papers accepted at conferences in which I am listed as an author.

10/09/2020 

Date Du Jie

Abstract

The computation of geodesic paths and distances on a 3D triangle mesh is an important and fundamental problem in computer graphics domain. Developing faster geodesic algorithms is the eternal pursuit of researchers. We focus on developing parallel exact and approximate discrete geodesic algorithms in this thesis. There are two main difficulties in parallelizing discrete geodesic algorithms. One is the strong dependence of data, the other one is the strict order of elements (windows or vertices) in sequential Dijkstra-like geodesic algorithms.

First of all, we propose the Parallel Mitchell-Mount-Papadimitriou (PMMP) algorithm, which extends the classical Mitchell-Mount-Papadimitriou (MMP) algorithm to the parallel framework. The classical MMP algorithm takes windows as primitives and maintains a priority queue to control the order of window propagation and window clipping to make sure that each window covers the shortest paths. To overcome the strong dependence of data, we divide our framework into five main steps: window selection, window propagation, window organization, window clipping, and window pushing. There is no data conflict in our concurrent steps: window propagation and window clipping. To concurrently propagate multi-windows and meanwhile maintain wavefront with great quality, we adopt the "buckets" structure to control windows in our implementation. Experimental results show that our parallel MMP algorithm is 1.6-1.7 times faster than FWP-MMP (the fastest

version of sequential MMP) on 4 CPU cores.

Considering that the MMP algorithm occupies too much memory and the major requirement of related applications is computing geodesic distance instead of geodesic paths, we move towards developing a parallel framework for vertex-oriented triangle propagation (VTP) algorithm, the fastest sequential exact geodesic algorithm up to now. The VTP algorithm takes vertices as primitives and maintains a priority queue to control the order of window list propagation around the vertex to update wavefront. Our parallel-VTP (PVTP) framework includes three main steps: K-window-list selection, parallel window list propagation, vertex distance updating and window list merging. Although there is only one concurrent step that window list propagation, the time complexity of other three steps is only $O(n)$ time complexity. Thus, our algorithm has an empirical $O(\frac{n^2}{T})$ time complexity, where n is the number of vertices and T is the number of threads. The experimental results show expected performance that our parallel VTP is 2.5-3 times faster than sequential VTP on a 4-core CPU and 4-5 times faster than sequential VTP algorithm on an 8-core CPU for most of regular triangle meshes with over 1 million vertices. Subsequently, we propose an approximate method based on VTP. Our approximate-VTP (AVTP) algorithm controls the global precision by parameter λ. Once the radius of the new wavefront achieves a multiple of λh, we delete all windows on wavefront and consider all vertices on the wavefront as new sources to reduce the number of windows and maintain geodesic properties. λ is a user-specified parameter for accuracy and speed control and h is average edge length. AVTP has a theoretical time complexity $O(n\lambda)$. The proposed parallelization and approximation techniques can be either used separately or combined together.

There are many applications requiring the query of geodesic distances between vertices on a given triangle mesh. Herein, we present an application

of discrete geodesic problems. Near repetitive patterns are ubiquitous in man-made and natural objects. Discovering their arrangement is helpful in understanding and analyzing of model geometry. Observing that patterns have consistent matching between the corresponding points, we propose a local feature matching algorithm. Our method is able to work for isotropic, anisotropic, and size varied patterns.

This thesis puts emphasis on developing faster discrete geodesic algorithms by parallelizing existing fast sequential geodesic algorithms. Our parallel-MMP algorithm is the fastest exact discrete geodesic algorithm that can trace the geodesic paths. Our parallel-VTP algorithm is also the fastest exact geodesic algorithm that can compute the geodesic distance. Our approximate-VTP algorithm has better performance than other homogeneous approximate geodesic algorithms so far.

Acknowledgements

Primarily, I'm really grateful to my supervisor, Prof. He Ying, for his considerable guidance, encouragement to me in my pursuit of Doctor of Philosophy Degree. Whenever I encounter the difficulties, he is always patient with me, and inspires me to think creatively.

I also appreciate Dr. Zhang Minqi, Dr. Sun Qian, Dr. Yuen Shan Leung, Dr. Le Tien Hung for their directions in my beginning years. Thanks to the labmates, Dr. Fang Zheng, Ms. Fu Qian, Ms. Yao Sidan, Mr. Zhang Dongbo, Dr. Ma Long, Dr. Lu Xuequan, Dr. Hou Fei for sharing their knowledge and ideas with me.

Thanks to my friends, Ms. Sun Yidan, Mr. Zhou Chengju and Ms. Chen Qian who always share happiness and consideration with me. Thanks to my parents and my brother for their constant care and support. At last, I sincerely express my gratitude to my boyfriend, Dr. Cai Weizheng, who always sustains, enlightens and accompanies me.

Publications

[1] **Jie Du**, Ying He, Zheng Fang, Wenlong Meng, and Shiqing Xin. On vertex-oriented triangle propagation algorithm: parallelization and approximation. *Computer-Aided Design, accepted*, 2020.

[2] **Jie Du** and Minqi Zhang. A novel method for repetitive pattern detection on 3d models. *under review*, 2020.

Contents

List of Figures

List of Tables

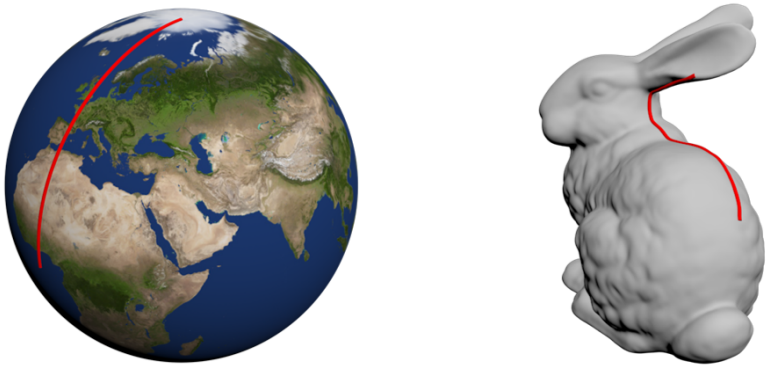
List of Algorithms

Chapter 1

Introduction

1.1 Motivations

1.1.1 Geodesics



(*a*) (*b*)

Figure 1.1: Illustrations of geodesic path. (a) A geodesic path on Earth; (b) A geodesic path on Bunny model.

Finding shortest paths and distance between given points is a primary and crucial problem in geometry and computer graphics. A shortest path between two given points is a straight segment in Euclidean space. Under non-Euclidean space, a shortest path from a source to a terminal point is

a geodesic curve. 2-manifold is an example of such a space that a curved surface in 3D Euclidean space. See Figure 1.1 (a). The surface of Earth is a curved surface of this kind in our familiar real world, and a part of one circle, as the shortest path between two locations on the Earth, is an example of a geodesic path. Figure 1.1 (b) exhibits a geodesic path between two vertices on a Bunny model constructed by a triangle mesh.

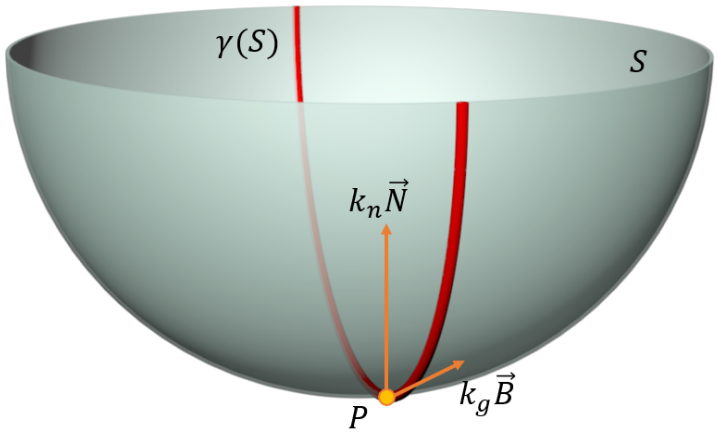


Figure 1.2: A geodesic curve on regular surface.

In differential geometry [5, 6], a geodesic can be considered as an extension of a straight segment from plane to the regular surface. See Figure 1.2. Given a regular surface S, let $\gamma(S)$ be a curve on S. If the curvature vector $\gamma"(S) = k_g \cdot \overrightarrow{B} + k_n \cdot \overrightarrow{N}$ of any point $P \in \gamma(S)$ coincides with the normal curvature vector $k_n \cdot \overrightarrow{N}$, namely the geodesic curvature k_g is equal to zero, $\gamma(S)$ is a geodesic curve. A geodesic curve is locally shortest, but may not globally shortest. For example, the arcs between two points on a spherical surface where one arc is relatively short and the other is relatively long. The shorter and the longer arc both are geodesic curves.

In computational geometry, a geodesic curve is generally regarded as the shortest path between two given points along a discrete surface, such as a triangle mesh. Most of discrete geodesic algorithms [3, 7, 8] computed

geodesics by unfolding 3D triangle mesh to 2D planes. [7] gave a definition of a geodesic path as "a path that passes through an alternating sequence of vertices and edge sequences such that the unfolded image of the path along any edge sequence is a straight segment". This thesis focuses on discrete geodesic algorithms.

1.1.2 The Discrete Geodesic Problem

The discrete geodesic problem can be divided into three types by the number of sources and destinations: (1) single source and single destination; (2) single source and all destinations; (3) all pairs. Most of existing methods [2, 3, 7–13] focus on solving the " single source and all destinations" problem, since the first type [9,14] is closely related to the second and all pairs problem [15, 16] is rarely studied due to high complexity. The solution of the "single source and all destinations" problem usually exhibits the geodesic paths or distances from one source to all vertices on a triangle mesh as Figure 1.3. This thesis works on studying the "single-source-all-destinations" problem as well.

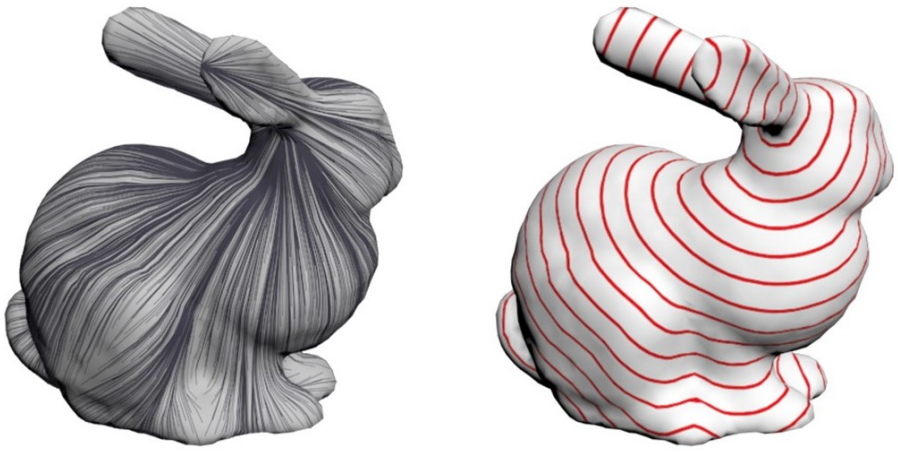


Figure 1.3: Geodesic paths from source to all directions and corresponding isolines.

The discrete geodesic algorithm also can be classified into exact geodesic algorithm and approximate geodesic algorithm. The representative exact geodesic algorithms includes Mitchell-Mount-Papadimitriou (MMP) algorithm [7], Chen-Han (CH) algorithm [8], improved Chen-Han (ICH) algorithm [2] and Vertex-oriented Triangle Propagation (VTP) algorithm [3]. The approximate geodesic algorithms, such as the fast marching method (FMM) [17], the heat method(HM) [18, 19], saddle vertex graph(SVG) [10] and discrete geodesic graph(DGG) [13, 20], aim at less practical time and lower error. This thesis works on developing more efficient exact and approximate geodesic algorithms.

Mitchell-Mount-Papadimitriou (MMP) algorithm [7] was first proposed as an efficient exact discrete geodesic algorithm with time complexity $O(n^2 logn)$ and space complexity $O(n^2)$ for an arbitrary polyhedron in 1987. MMP algorithm inherited the tenor of the Dijkstra algorithm [21] and transferred the windows on edges into the nodes of Dijkstra graph. Later, Chen-Han (CH) algorithm [8] was proposed with a "one angle one split" method to avoid generating too many windows in 1990. CH algorithm propagated windows in breadth-first-search (BFS) order and stored windows by a tree data structure, thus CH algorithm achieved $O(n^2)$ time complexity and $O(n)$ space complexity. However, [9] implemented the MMP algorithm and demonstrated that the practical time complexity of the MMP algorithm was sub-quadratic in 2005. MMP algorithm is much more efficient than the CH algorithm since it propagates fewer windows. In 2009, [2] proposed improved Chen-Han (ICH) algorithm by creating "checking with vertices" filter to remove 99% redundant windows and adopting a priority queue to store windows as MMP algorithm doing. The performance of the ICH algorithm is comparable with the MMP algorithm. In 2015, [12] proposed the fast wavefront propagation (FWP) framework with the help of "buckets" data structure. They organized windows with "buckets" container instead

of priority queue so that reducing the time complexity of adding or deleting operations to $O(1)$. Their FWP-CH and FWP-MMP algorithms are 3-10 times faster than original ICH and MMP algorithms. In 2016, [3] proposed the vertex-oriented triangle propagation (VTP) algorithm which considers vertices instead of windows as primitives in the process of geodesic wave-front expansion. Their experiment showed that the VTP algorithm is 2-3 times faster than FWP-CH and FWP-MMP algorithms. So far, the VTP algorithm is the fastest sequential exact geodesic algorithm with $O(n^2)$ time and $O(n)$ space complexity.

In recent years, the performance of many algorithms can be remarkably improved by developing a parallel framework. There are some parallel geodesic algorithm as well, such as [11, 22] based on ICH algorithm, [22, 23] based on fast marching method (FMM) and [24] based on Heat Method (HM). This thesis focuses on developing parallel framework based on MMP and VTP geodesic algorithms.

1.1.3 Applications

Geodesic is widely used in computational geometry and graphics because its good properties [25]. Computing geodesic paths and distance is applied to many fields, such as robot motion [26], terrain navigation [7], geographic information systems [27], industrial design [28], image processing [29], shape segmentation [30,31], remeshing [32,33], shape descriptor [34], sampling [35], non-rigid registration [36] and so on. For more information, please see [6].

Herein, we exhibit two examples, Geodesic Voronoi Diagrams, and Geodesic Surface Correspondence.

1.1.3.1 Geodesic Voronoi Diagram

Voronoi Diagram is a partition from the whole to local regions based on the distances between all points in a plane as follows. S denotes a set of all points in a plane. Given a set of points $P = \{p_0, p_1, p_2, ..., p_n\} \subset S$ and suppose that $p_i \neq p_j$ for any $i \neq j$. The Voronoi Cell VCp_i of p_i consists of each point whose distance to p_i is less than or equal to other p_j, i.e.,

$$VC(p_i) = \{q | D_{p_i}(q) < D_{p_j}(q), i \neq j, q \in S\},$$

where $D_{p_i}(q)$ denotes the distance between point p_i and point q. Figure 1.4 shows an example of a Voronoi Diagram in Euclidean space.

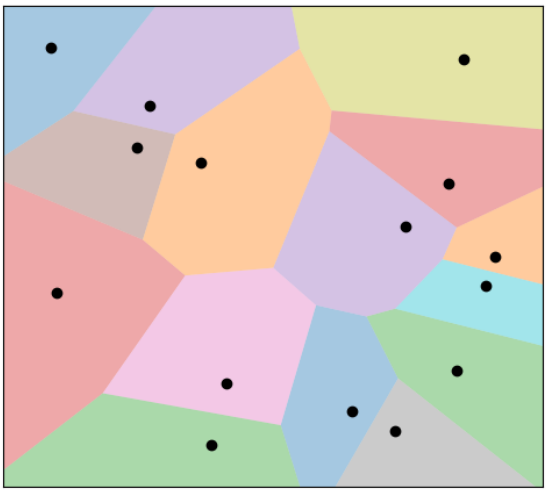


Figure 1.4: 15 points and their Voronoi Cells in a plane.

[33] describes a definition of Geodesic Voronoi Diagram (GVD) on triangle mesh, which extends the domain to manifold. The Geodesic Voronoi Diagram of a given set of points $P = \{p_0, p_1, p_2, ..., p_n\}$ on triangle mesh M is a set $VD(P) = \{VC(p_1), VC(p_2), ..., VC(p_n)\}$, where $VC(p_i) = \{q | D_{p_i}(q) < D_{p_j}(q), i \neq j, q \in M\}$. Herein, $D_{p_i}(q)$ represents the geodesic distance between point q to point p_i. Figure 1.5 shows an example of the Geodesic Voronoi Diagram on a 2-manifold sphere model with ten sources.

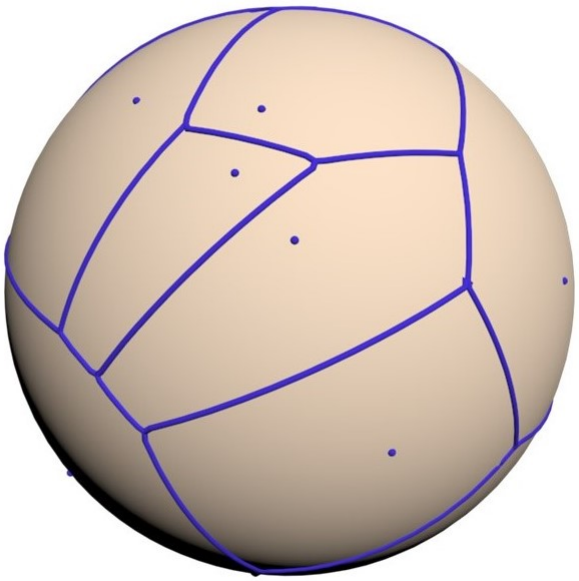


Figure 1.5: 10 points and their Geodesic Voronoi Cells on a sphere model.

There are two main challenges in the process of extending the Voronoi Diagram to 2-manifold. The first challenge is the performance of computing geodesic distance. Existing exact geodesic algorithms including MMP [7,9], CH [8], VTP [3]. Another challenge lies in the inherent structure in the Voronoi Diagram on triangle mesh. Any three lines which intersect with one another form a triangle in 2D space. However, not all three intersecting geodesics could form a geodesic triangle on triangle mesh instead. Any three points that are not on a line uniquely determine a circumcircle in a 2D plane, whereas such kind of geodesic circumcircle may not exist or many geodesic circumcircles exist on 2-manifolds [37]. [33] fully analyzed the structure of the Geodesic Voronoi Diagram and presented an efficient algorithm to compute the Geodesic Voronoi Diagram on triangle mesh.

1.1.3.2 Geodesic Surface Correspondence

Shape and surface comparisons are required to shape recognition, shape retrieval, registration, pattern detection, biological morphology, etc. [38] gives a detailed description of the geodesic shape and surface correspondence in these fields. A corresponding map is a bijective mapping function between the manifolds Ω_0 and Ω_1 as follow

$$\varphi : \Omega_0 \longrightarrow \Omega_1.$$

This function means that for any sampled point $x \in \Omega_0$, it will exist corresponding point $\varphi(x) \in \Omega_1$, vice versa.

To find robust correspondence between manifolds in practice, not only sampled points' signature but also geodesic consistency are considered [36] as follows

$$\begin{gathered} d_{\Omega_0}(x, y) \approx d_{\Omega_1}(\varphi(x), \varphi(y)), \\ f(x) \approx f(\varphi(x)), \\ f(y) \approx f(\varphi(y)), \\ x, y \in \Omega_0, \varphi(x), \varphi(y) \in \Omega_1, \end{gathered}$$

where $d(x, y)$ is defined as the geodesic distance between sampled points x and y, $f(x)$ is defined as the feature vector at sampled point x [39]. This thesis proposes a novel method for detecting repetitive patterns on 3D models, which also adopts the key idea of Geodesic Voronoi Diagram and geodesic surface correspondence.

1.2 Objectives and Contributions

This thesis aims at developing more effective discrete geodesic algorithms from one source to all destinations on a triangle mesh. First, we intend to develop a faster exact geodesic algorithm by parallelizing the classical MMP algorithm, an efficient geodesic algorithm preserving geodesic paths. And then, we move to design more efficient geodesic algorithm by parallelizing and approximating VTP, the fastest exact geodesic algorithm at present. Besides, we present a novel method for repetitive pattern detection on 3D models as an application of geodesic distance.

1.2.1 Parallel Mitchell-Mount-Papadimitriou Algorithm

The classical Mitchell-Mount-Papadimitriou (MMP) algorithm [7] was first proposed as an efficient exact discrete geodesic algorithm from one source to all destinations that adopted Dijkstra-like [21] manner. [9] first implemented the MMP algorithm by dividing each triangle edge into intervals called "windows" in the process of geodesic paths extending. Herein, windows replace nodes in the graph of Dijkstra's algorithm and are propagated as a wavefront. Observing the MMP algorithm is the most efficient algorithm to preserve geodesic paths, we aim at developing a faster algorithm by parallelizing the MMP algorithm.

There are two main difficulties in designing a parallel framework for the MMP algorithm, global priority queue who stores windows and pruning overlapped windows during propagation. The classical MMP algorithm maintains windows in a priority queue. Herein, we adopt "buckets" structure in FWP-MMP for efficiency. There exists data conflict when windows are added, deleted or modified concurrently in the same bucket container on multi-threads. To overcome this problem, we pre-allocate memory for pos-

sible window operations as PCH algorithm [11]. Unlike the independent checking for each window in the PCH algorithm, MMP algorithm prunes overlapped part for every two windows. To parallelize window pruning, we classify all new-generated windows by the index of half-edge and then clip overlapped windows in different edges synchronously.

Our parallel-MMP framework includes five main steps: window selection, window propagation, window organization, window clipping, and window pushing. Window propagation and window clipping are concurrent steps and there is no data conflict. Our experimental results show that our parallel framework is about 1.6-1.7 times faster than the sequential FWP-MMP algorithm based on 4 threads and has a comparable performance with the VTP algorithm.

1.2.2 Parallel Vertex-oriented Triangle Propagation Algorithm

Since our parallel-MMP algorithm's performance based on 4 threads is just equivalent to the sequential Vertex-oriented Triangle Propagation (VTP) algorithm and many applications only need geodesic distances instead of geodesic paths, we shift our attention to developing parallel VTP algorithm.

1.2.2.1 Parallel Exact Geodesic Algorithm based on VTP

The VTP algorithm takes vertices as primitives and aggregates multi-windows operation to a window list by a half-edge structure. In each iteration, VTP pops a vertex with the nearest distance and propagates all windows inside this vertex's one-ring area to update the wavefront. Since there would be data conflict on window lists selection if we design parallel operations for vertices and window operations costs most of time, we directly design the parallel framework for window lists. Our parallel-VTP framework

includes three main steps: K-nearest window list selection, parallel window list propagation, vertex distance updating and window list merging. The parallel window list propagation inherits all window pruning strategies in the sequential VTP algorithm. Although there is only one concurrent step, window list propagation, the time complexity of the other three steps is only $O(n)$. Our algorithm has an empirical $O(\frac{n^2}{T})$ time complexity, where n is the number of vertices and T is the number of threads. Experimental results show expected performance that our parallel VTP algorithm is over 2.5-3 times faster than sequential VTP algorithm based on 4 threads and 4-5 times based on 8 threads for most of regular triangle meshes with over 1 million vertices.

1.2.2.2 Parallel Approximate Geodesic Algorithm based on VTP

Subsequently, we also propose an approximate geodesic algorithm based on VTP. Observing that wavefront propagation of exact algorithms slows down when the wavefront has a long circumference, hereby containing a large number of pending windows at a certain moment, we propose an approximate version for VTP (AVTP) by resetting the windows on the current wavefront regularly. Once the radius of the geodesic wavefront achieves the pre-determined value which is controlled by λ, we delete all windows on wavefront and consider all vertices on the wavefront as new sources to propagate so that our approximate algorithm reduces the number of propagated windows and meanwhile maintains geodesic properties. AVTP has a theoretical time complexity $O(n\lambda)$, which is also confirmed by computational results. The proposed parallelization and approximation techniques can be either used separately or combined together.

1.2.3 Repetitive Pattern Detection on 3D Models

There are many applications of geodesic algorithms in computer graphics, such as remeshing [32, 33], non-rigid registration [36], shape analysis [38] and shape descriptor [34], etc, which require the query of geodesic distance on a given polygonal mesh. Herein, we propose a novel method for detecting repetitive patterns on 3D models with the aid of computing geodesic distance.

Observing that repetitive patterns have consistent matching between the corresponding feature points, we tackle the problem by an efficient local feature matching algorithm. Taking a 2D-manifold triangle mesh M and the user-specified to-be-detected pattern P as input, our algorithm solves the pattern detection problem in two main steps as follows. First, with the extracted signature from sample points on pattern P and the area around P, we match the sample points in pairs. Then we cluster the matches to classify several reasonable patches around the original pattern P. Second, we achieve the optimal correspondence between patterns via a continuous optimization, which not only matches the corresponding features between patterns but also considers the feature consistency within each patch. Finally, we build arrangement graph by computing the Voronoi diagram of detected patterns. Our method doesn't assume potential patterns are regular. As a result, it works for both isotropic and anisotropic, even non-regularly distributed patterns. Moreover, our method can be accelerated by a parallel implementation.

1.3 Report Organization

This thesis is organized as the following: Chapter 2 presents the related work of geodesic algorithms and pattern detection; Chapter 3 exhibits our parallel exact geodesic algorithm based on MMP; Chapter 4 exhibits our

parallelization and approximation method based on VTP; Chapter 5 proposes a novel framework for detecting repetitive patterns on 3D models; Chapter 6 concludes this thesis and describe feasible future work.

Chapter 2

Related Work

This chapter reviews related work in the discrete geodesic problem and pattern detection problem.

2.1 Discrete Geodesic Algorithms

This section studies the discrete geodesic algorithm by its sequential or parallel manner. The comprehensive surveys are given in [26, 38, 40].

2.1.1 Sequential Algorithms

The discrete geodesic problem could be classified into three types by the number of sources and destinations: one-source-all-destinations, one-source-one-destination, all pairs. Herein, we review these three types of geodesic problem and especially studies the algorithms compute geodesic distances from one source to all destinations since our work focuses on this class as well.

2.1.1.1 Single Source and All destinations

Mitchell-Mount-Papadimitriou (MMP) Algorithm The shortest path problem on a polyhedral surface was first proposed by [41] in 1984. They presented an $O(n^3 logn)$ time complexity and $O(n^2)$ space complexity algorithm to compute geodesic distances from one source to all destinations on a convex polyhedron. In 1985, [42] gave an improved algorithm that runs in $O(n^2 logn)$ time and stores in $O(nlogn)$ space for a convex polyhedron. [43] extended [41] to non-convex polyhedra with $O(n^5)$ time complexity in 1985, which established that the problem can be solved in polynomial time. In 1987, Mitchell, Mount, and Papadimitriou (MMP) [7] first proposed an efficient exact algorithm for the single-source-all-destinations discrete geodesic problem on arbitrary polyhedron which has $O(n^2 logn)$ time complexity, where n represents the number of vertices on a triangle mesh. [9] presented an implementation of the MMP algorithm in 2005.

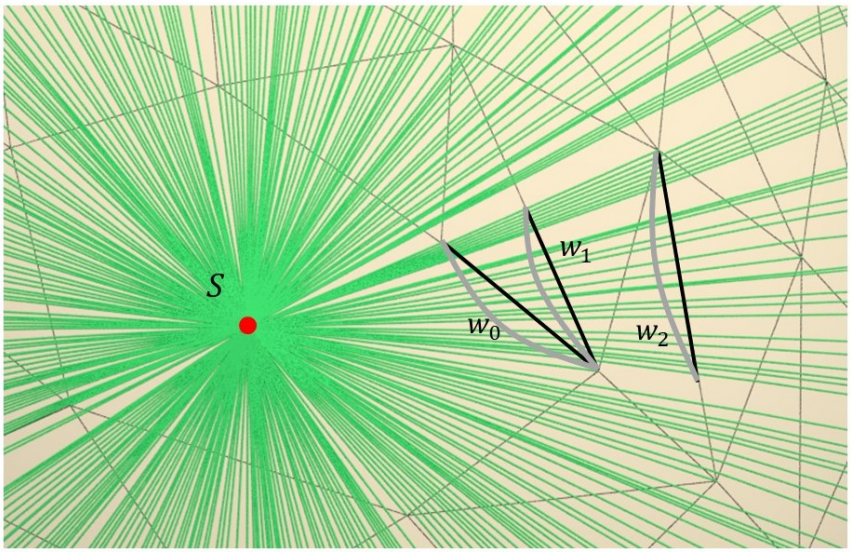


Figure 2.1: Geodesic paths from the source S.

Imagining that there is a light source on the location of source vertex S. See Figure 2.1. Geodesic paths from one source vertex to all other vertices on a

triangle mesh can be visualized as rays emitting from the light source in all directions. Intuitively, the shortest path must be a straight segment when all passed faces are unfolded into a coincide plane. The basic idea of the MMP algorithm is to track bunches of shortest paths that can be parameterized as windows in a "continuous Dijkstra" manner. Figure 2.1 presents some examples of window as w_0, w_1, w_2. Finally, all vertices' geodesic distance can be updated by their adjacent windows. There are $O(n)$ windows that would be generated at most for one edge, and all upcoming updated windows are stored in a priority queue. Therefore, the time complexity of the MMP algorithm is $O(n^2 logn)$ in the worst case and the space complexity is $O(n^2)$. In practice, [9] observed that each edge has an average $O(\sqrt{n})$ windows for regular models since the number of edges can be considered as being proportional to surface area and the number of edges passed by the shortest path is proportional to the diameter. Their experimental results also were corresponded with their observation that the empirical time complexity is sub-quadratic. The newly generated windows may intersect the old generated windows on the same edge during window propagation. In order to ensure all windows cover the shortest paths, the overlapped windows have to be pruned. The clipping can be operated by solving a quadratic equation. Since our work of parallel-MMP is based on the classical MMP algorithm, the more detailed techniques of the MMP algorithm are shown in Chapter 3.

There are two typical published variances of the MMP algorithm for improving the algorithm performance. One is an implementation using edge-based data structures presented by [4] which optimizes the algorithm by decreasing the number of generated windows. Their results showed that the edge-based implementation reduces 44% running time and 29% storage on average when compared with the classical half-edge structure. The other is Fast Wavefront Propagation(FWP) proposed by [12], which uses "buckets" data structure

to organize windows during propagation so that it can process a number of windows at each iteration. Their experiment resulted that the implementation of MMP based on the FWP framework ran 3-10 times faster than the classical MMP algorithm. Besides, Hung [1] presented an improving MMP algorithm by optimizing the techniques of solving the quadratic equation during window clipping. Surazhsky et al. [9] also proposed an approximate geodesic algorithm with bounded error, which added a merging operation into classical MMP.

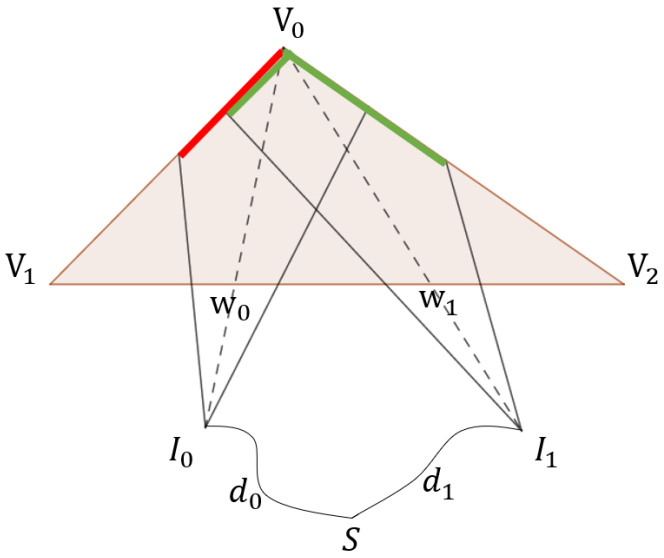


Figure 2.2: "One angle one split" filter rule. When two windows w_0 and w_1 cover the same vertex V_0, only w_1 who provides shorter distance $d_0 + \| \overline{I_0V_0} \| > d_1 + \| \overline{I_1V_0} \|$ to V_0 has two children (green), where d_i denotes the shortest distance from source S to pseudo-source I_i.

Chen-Han (CH) Algorithm Chen and Han (CH) [8] proposed an $O(n^2)$ time complexity and $O(n)$ space complexity algorithm in 1990, which has better theoretical time complexity and space complexity than MMP algorithm. CH algorithm also organizes distance information by "window" and propagates them like "continuous Dijkstra". Unlike the MMP algorithm, this algorithm stores windows in a tree data structure and propagates windows in breadth-first-search order. To avoid an exponential explosion, they

proposed a "One angle one split" filter that if two windows covered the same vertex, at most one of them could have two children windows. See Figure 2.2. Chen and Han [8] proved that the total number of generated nodes was $O(n^2)$ rather than an exponential size. Their CH algorithm lowered the space complexity to $O(n)$ by only storing leaf nodes and those nodes which occupied an angle or a saddle vertex. [44] gave an implementation of the CH algorithm in 2000.

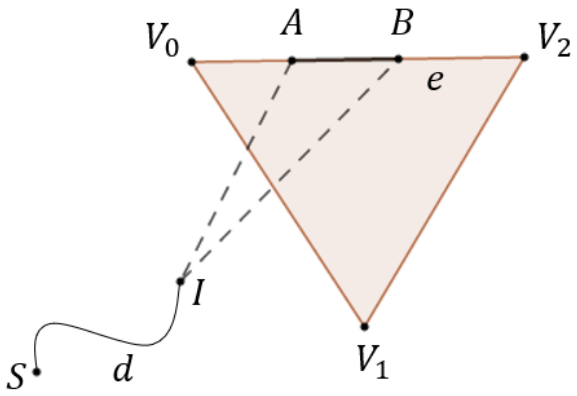


Figure 2.3: "Checking with vertices" rule.

Although the CH algorithm has lower time complexity than the MMP algorithm, in practice, MMP algorithm [9] is much faster than CH algorithm [44], since there are too many useless redundant windows generated in CH algorithm. In 2009, [2] proposed a simple method "checking with vertices" to prune most of redundant windows of CH. See Figure 2.3. The window $W = (d, I, e, \overline{AB})$ represents a bunch of shortest paths entering the triangle $\Delta V_0V_1V_2$ by edge $\overline{V_0V_1}$ and arriving the segment $\overline{AB}$ on edge $\overline{V_0V_2}$. [2] presented the theorem as Equation 2.1, where d_i is the current shortest distance from source vertex S to vertex V_i, $i = 0, 1, 2$. If the Equation 2.1 is satisfied,

the window W should be removed.

$$\begin{aligned} d+ \parallel \overline{IB} \parallel &> d_0+ \parallel \overline{V_0B} \parallel \\ d+ \parallel \overline{IA} \parallel &> d_1+ \parallel \overline{V_1A} \parallel \\ d+ \parallel \overline{IA} \parallel &> d_2+ \parallel \overline{V_2A} \parallel \end{aligned} \tag{2.1}$$

[2] also adopted a priority queue to control the order of window propagation, which could generate fewer windows and accelerate the practical time. Their improved Chen-Han algorithm(ICH) has comparable performance with MMP algorithm [2,3]. Furthermore, [12] also improved the CH algorithm by their FWP framework. The FWP-CH algorithm is 2-5 times faster than the ICH algorithm.

[14] also proposed an approximate method to compute geodesic distance, which changes the parameter λ so that over-filtering out windows as Equation 2.2.

$$\begin{aligned} d+ \parallel \overline{IB} \parallel +\lambda &> d_0+ \parallel \overline{V_0B} \parallel \\ d+ \parallel \overline{IA} \parallel +\lambda &> d_1+ \parallel \overline{V_1A} \parallel \\ d+ \parallel \overline{IA} \parallel +\lambda &> d_2+ \parallel \overline{V_2A} \parallel \end{aligned} \tag{2.2}$$

Vertex-oriented Triangle Propagation (VTP) Algorithm In 2016, [3] proposed an exact geodesic algorithm based on vertex-oriented instead of window-oriented in MMP and CH. Their Vertex-oriented Triangle Propagation (VTP) algorithm gathers window clipping operations within one same triangle with the aid of the window list structure.

At each iteration, the nearest vertex is popped from the priority queue and the window lists within the one-ring area of this vertex would be propagated. A new wavefront is formed after the propagation of window lists. Once the vertices' distance on the new wavefront has been updated, they are pushed into the priority queue. All vertices' geodesic distance would be figured out

when the priority queue is empty.

Besides inheriting the window clipping strategies in the ICH algorithm, VTP also gives exhaustive filtering methods during window lists propagation. Since the VTP algorithm organizes vertices instead of windows in the priority queue and only stores the windows on wavefront, it achieves $O(n^2)$ time complexity and $O(n)$ space complexity. Their experimental results showed that their VTP algorithm was 6 times faster than both MMP and ICH, more than 3 times faster than FWP-CH, and 2 times faster than FWP-MMP. VTP algorithm is the fastest sequential exact geodesic algorithm so far. Since our work of parallel-VTP is based on this sequential VTP algorithm, more detailed techniques of the VTP algorithm are shown in Chapter 4.

Other Algorithms Except for the three mentioned classical geodesic algorithms and their variance, there are also approximate algorithms that aim at reducing the computing time dramatically and promising acceptable error.

The fast marching method (FMM) [17,45] and heat method (HM) [18,19] are PDE approach, which computes geodesic distance by solving discrete partial differential equations. FMM [45] is a popular numerical algorithm for solving the Eikonal equation on a regular grid in $O(MlogM)$ time complexity, where M is the number of grid points. [17] extended the fast marching method to triangular domain. They computed geodesic distances with same time complexity as the original FMM method by solving a discrete version of the Eikonal equation. [18, 19] proposed the heat method to compute geodesics, which places heat at the source vertex and diffuses it to all vertices. It's simple to implement heat method with the aid of solving a pair of standard linear elliptic problems [18]. The experiment showed that the HM method could solve the geodesic problem in near-linear time apart from the pre-

processing steps.

The saddle vertex graph (SVG) [10] method and the discrete distance graph (DGG) [13] method are based on graph and include pre-process steps. They encoded geodesic information into a sparse graph in advance and then used Dijkstra's algorithm to compute geodesic distances. There is a trade-off process between time and accuracy which is controlled by their parameters in SVG and DGG.

2.1.1.2 Single Source and Single Destination

Except for the algorithms from one source to all destinations, there is some research work focusing on computing geodesic path for the single-source-single-destination problem.

In 1985, [46] firstly posed an approximate efficient algorithm with polynomial time complexity to compute the shortest path between two given points in three-dimensional space including polyhedral obstacles that the motion planning problem. There are some $1+\epsilon$ approximate algorithms proposed with a trade-off between accuracy and time cost, such as [47–51].

In 2005, [9] proposed an exact algorithm by utilizing pruned searches with narrowing distance bounds like a continuous A* search [52]. In 2007, [53] proposed an efficient locally exact shortest path algorithm on triangle mesh, which evolves an initial approximately shortest path into a local exact shortest path by finite iterations. In 2010, [14] applied their ICH algorithm to compute geodesic distance between two points in a heuristic method.

2.1.1.3 All Pairs

Besides, there are some published algorithms in computing geodesic distance between all pairs of vertices. In 1997, [54] proposed a $O((\sqrt{n}/m^{1/4})logn)$ time complexity algorithm for querying geodesic distance between any given

two points on a convex polyhedron with $O(n^6 m^{1+\delta})$ preprocess step. Then, [55] speedups the preprocess [54] time complexity a linear factor in 2009.

In 2009, [16] proposed an exact algorithm to solve the all-pairs geodesic problem for convex and non-convex polyhedrons, which firstly constructs a vertex-to-vertex graph with minimal geodesic distance and then computes the shortest distance between any given two points by searching the graph. Their algorithm has exponential time complexity in theory and runs less than $O(n^3)$ time complexity in practice.

In 2012, [15] presented an approximate algorithm to compute all-pairs' geodesic distance, which includes a $O(mn^2 logn)$ time preprocess step and a $O(1)$ time query step for any pair of points, where $m(<< n)$ is specified by user.

There exists some research work on computing the shortest paths in a graph for all pairs of nodes as well. [56,57] proposed algorithms for computing the shortest paths between all pairs in a weighted graph. [58,59] committed to compute shortest paths in an unweighted graph.

2.1.2 Parallel Algorithms

2.1.2.1 Parallel Chen-Han (PCH) Algorithm

In recent decades, a number of sequential geodesic algorithms have been parallelized to remarkably speed up the performance for large models. [11] firstly developed a parallelized version of the classical Chen-Han(PCH) algorithm in 2013. At the beginning of their algorithm, windows for each edge facing source vertex are created. And then, it iteratively operates these windows in four steps. First, the algorithm selects K windows which are nearest to the source vertex. Second, to distribute these K windows to T threads and then propagate them in parallel. Herein, if a window w can

provide a shorter distance for a vertex v, the algorithm will not immediately update its distance information but create an update event for processing later. Third, to organize the newly-generated windows in the memory pool such that there are no memory gaps between adjacent windows. Finally, it handles the update events and updates the corresponding distance information. These four steps are repeated until all new windows in the memory pool have been processed. The experimental results showed that their PCH algorithm improves the efficiency an order of magnitude than ICH algorithm on GPUs (Nvidia GTX 580) and the performance improvement is consistent with GPU double-precision performance.

2.1.2.2 Autonomous Wavefront Propagation

Although PCH algorithm shows great efficiency, [22] mentioned its drawback that PCH algorithm is not fully parallel and its sequential part becomes dominant when the mesh complexity increases. For overcome this difficulty, [22] proposes a fully parallel framework called Autonomous Wavefront Propagation(AWP) in 2019. They constructed the propagation dependency graph that consists of half-edges and vertices. Then, they define the half-edge or vertex node as active state when the half-edge contains to-be-generated windows or the vertex to-be-generate new windows. Since there is no data conflict during generating new windows and updating information within their own memory space, all the read and write operations can be fully paralleled. Their experimental results showed that AWP-CH runs in n^p empirical time complexity, where $p \in [1.25, 1.35]$. There is a disadvantage of AWP framework that it produce too many windows during window propagation because they adopt first-in-first-out queue instead of priority queue to organize windows. And the performance of one GPU core is much lower than a CPU core, as a result, their AWP algorithm (GPU: GTX Titan XP with 3840 CUDA cores) is only 2 times faster than the sequential VTP

(CPU: Intel i7-7700k Quad Core 4.2 GHz). [22] also implemented the parallel version of approximate ICH [14] (AICH), the approximate AWP-CH outperforms approximate ICH 4-9 times for all $\lambda \in [0.01, 0.2]$.

2.1.2.3 Parallelization On Partial Differential Equation Method

There also exist several parallel approximate geodesic algorithms based on the Partial Differential Equation (PDE) approaches.

Parallelization of the Fast Marching Method [23] proposed a paralleled fast marching algorithm(PMM) that a raster scan-based version of the fast marching algorithm in 2008. Although the PMM algorithm is several orders of magnitude faster than the sequential FMM based on GPU, its paralleled structure doesn't apply to the complicated surface. There are many parallel implementations based on FMM [60] [61] [62], but they do not apply to the surface. [22] also implemented the parallel version of FMM which applies to a triangle mesh. They considered each half-edge as a window, and then operate their AWP framework to parallel the fast marching algorithm. Their experimental results show that AWP-FMM is 3-8 times faster than the sequential FMM based.

Parallel Heat Method [24] proposed a parallel and scalable implementation of Heat Method (HM) based on CPU in 2019. The classical Heat method adopts the Cholesky factorization consuming too much time and occupying memory. As a result, it doesn't work for large models. [24] improved the Heat method with four main steps: heat diffusion from source vertex, gradient normalization, optimizing an integrable gradient field, and recovering geodesic distance. Different from the classical heat method, their Gauss-Seidel solver and the geodesic distance integration can be easily parallelized and scalable. Their experimental results showed that their improved Heat method significantly outperformed the sequential HM under the same

accuracy. Moreover, its memory consumption is much lower than the original Heat method. Therefore, it can work on large models with 200 million vertices on a PC with 128GB memory. It should be noted that the accuracy of their method is also sensitive to triangulation quality. This problem could be improved by Delaunay triangulation. However, it costs much time as a preprocessing step.

2.2 The Repetitive Pattern Detection Problem

2.2.1 Pattern Detection for 2D Domain in Images

The pattern detection problem for images is a long-standing problem in computer vision. The related published work could be classified into lattice and non-lattice by their detective objects. We exhibit a few representative papers as follows.

2.2.1.1 Lattice

[63] pointed out the importance of detecting repetitive elements in an image in 1996. They proposed an algorithm to detect repeated scene elements in an image, which utilizes window matching and region growing. Given the interesting windows in the image, they computed the feature of each pixel and matched neighbor elements with affine transformation. Their method used the brute-force searching for detecting the elements.

[64] detected the repeated elements and constructing their geometric grouping from a perspective image in 1999. First of all, it identifies the interesting element and hypothesizes the grouping region by Harris corners detection [65] and cross-correlation between elements. Then, the hypothesized grouping is verified by local affine transformations. At last, it extends the

grouping region by conjectural parameterized transformation and greedy scheme.

In 2006, [66] proposed a higher-order feature matching algorithm to discover the lattices of near-regular textures in real images. It finds a plausible lattice by iterative executing four steps as follows. First, to propose texels by correlating the region around the user-specified patch and selecting the peaks in the correlation score map [67]. And then, based on these potential texels, it finds out the lattice assignment with the aid of high-order assignment. This stage computes a higher-order feature affinity matrix between all potential texels and utilizes the spectral techniques to clustering the high-order matches. The third step refines the lattice by bringing in a few topological constraints to discard false-positive texels. Finally, their method parameterizes a regularized thin-plate spline warp in case of geometric distortion.

In 2009, [68] proposed a method for automatically detecting deformed wallpaper patterns. First, it clusters the interest points and votes for consistent lattice unit proposals. The 2D lattice detection problem could be transferred into a multi-target tracking problem. Therefore, based on the lattice unit result, then it utilizes an efficient Mean-Shift Belief Propagation method to solve this problem in the Markov Random Field. Moreover, in the process of the iteration, it also brings in the thin-plate spline warping technique to rectify the deformation of the original lattice so that make sure the stability of Markov Random Field.

In 2017, [69] first proposed a CNN based method for detecting repeated patterns on a grid. The deep CNN filters encode features at conceptual levels (from low-level patches to high-level semantics) as well as scales (from local to global). Thus, the detection of repeated patterns on the lattice is more robust on the situation of appearance variations in contrast to traditional

methods [68, 70]. [71] proposed an end-to-end pipeline that could find the minimal repeating pattern in the texture which simplified [69] and ensured the precise.

2.2.1.2 Non-Lattice

In 2003, [72] presented a system to detect regular repetitive planar patterns (not necessarily coplanar) under perspective skew. The geometric relation between such repeated patterns is characterized by a planar homology. Therefore, the main task is transferred into finding out the planar homology. It first detects repeated patches by affinely invariant neighborhoods, which are robust to oblique viewpoints and nonuniform illumination. Then, it considers a cluster of these neighborhoods in feature space, following by inputting the cluster to the Cascaded Hough Transform to produce fixed structure candidates. Finally, based on these fixed candidates, it hypothesizes and validates the planar homologies by comparing the original image with its "warped" version.

In 2013, [73] proposed a new framework for repetitive pattern detection and grouping in a 2D image, called higher-level segmentation. Their method follows similar steps as the classical region growing segmentation [74] algorithm, but results in a set of repetitive patches. Given a user-specified area, it extracts a texton template initially. First, to detect new candidate patches matching the texton template by region growing. Second, to refine the subspace grouping by a mean-shift-like clustering. All repetitive patterns could be detected by iteratively operate these steps.

In 2014, [75] proposed a novel method of detection, rectification and segmentation of coplanar repeated patterns. There are five stages in their method: feature appearance matching, projective distortion removal, motif construction, affine distortion correction, and non-linear optimization. First, it uses

Maximally Stable Extremal Regions (MSER) features [76] with the Local Affine Frame (LAF) extension [77] to extract Scale Invariant Feature Transform (SIFT) descriptors [78], and computes the spectral clustering of the descriptor matches. Second, to estimate a rectifying homography so as to delete the matching features after rectification. Third, the pattern is constructed by the largest clusters' SIFT which defines local coordinate systems at the origin of SIFT. Then, it upgrades an affine-rectified pattern by utilizing a length constraint to reduce the affine ambiguity. Finally, it adopts a non-linear least squares optimizer to minimize the possible re-projection error. As a result, their method can detect reflection, rotation and translation symmetry.

2.2.2 Pattern Detection for 3D Domain in Meshes

There is some research work detecting intrinsic symmetry of 3D models [79–83]. Since our work focuses on detecting repetitive patterns and explaining the corresponding regularities beyond the symmetry detection requirement on an input model, we list detailed techniques of existing repeated patches detection as follows.

In 2007, [84] segmented periodic relief from a triangle mesh, and then extracted a single repeat unit from the relief. It utilizes a snake-based method to detect the relief boundaries from a pair of user-drawn contours and segments away from the background. Taken the two points on one belief boundary specified by the user, it computes the third point on the same boundary and gets the coarse repeat unit by correspondences. Next, to refine the matches by a surface registration strategy based on iterative closest point (ICP) algorithm [85, 86].

In 2008, [87] proposed a computational framework for discovering regular repeated 3D geometry. The detected shape is rigorously limited to consist

of repeated elements in accord with the combination of Euclidean transformations which includes rotation, translation and scaling. Their framework extracts regular patterns by three main steps as follows. First, to compose the input model into small local patches and estimate similarity transformations between them. Then, this stage clusters the similarity transformations to generate characteristic lattice patterns for shapes. The second stage estimates the parameters of the generative model by a global optimization procedure in transformation space and outputs the regular structures at the scale of the initial surface patches. Finally, it aggregates adjacent patches and builds their larger repetitive structures by a simultaneous registration method.

In 2010, [88] first attempted to detect intrinsic repeated geometries under isometric deformations by generalizing [87]. It utilizes multidimensional scaling (MDS) to map an intrinsic surface into a Euclidean one. Therefore, the problem is transferred to the case of Euclidean regularities again.

In 2014, [89] proposed a novel method for discovering near-regular structures on 3D models, which formulated the repetitive pattern detection problem as a constrained optimization with both geometric and topological aspects. Taking a triangle mesh as input, it initially obtains the set of sample points on the surface and constructs near-regular geodesic subdivision as follows. First, to connect geodesic curves whose length is smaller than d_{max} between pairwise sample points so as to form a candidate curve set. Moreover, it needs to assemble valid closed loops based on the graph constructed by geodesic curves and perturb the loops so as to move all geodesic curves to the triangle edges. Second, to generate the candidate patches inside the nonempty valid closed loops. At last, their method extracts the near-regular geodesic subdivision by selecting a subset of patches to minimize combination constrained optimization by integer program formulation

and linear programming relaxation. They showed that this kind of mixed discrete-continuous problem can be efficiently solved using linear programming techniques. However, their method assumed the repeated patterns are isotropic, of similar sizes, and regularly arranged on the surface. As a result, it does not work for patterns with high degree of anisotropy or non-uniformly distributed patterns with varying sizes.

Chapter 3

Parallel Mitchell-Mount-Papadimitriou Algorithm

This chapter focuses on developing a parallel framework based on the classical Mitchell-Mount-Papadimitriou(MMP) algorithm. The MMP algorithm takes windows as primitives and maintains a priority queue to control the order of window propagation and window clipping to propagate geodesic paths. The difficulty of parallelizing the MMP algorithm is to handle the strong data dependency of window clipping during window propagation. To overcome this problem, we set up five main steps: window selection, window propagation, window organization, window clipping, and window pushing in our parallel MMP framework. There is no data conflict in our concurrent steps that window propagation and window clipping.

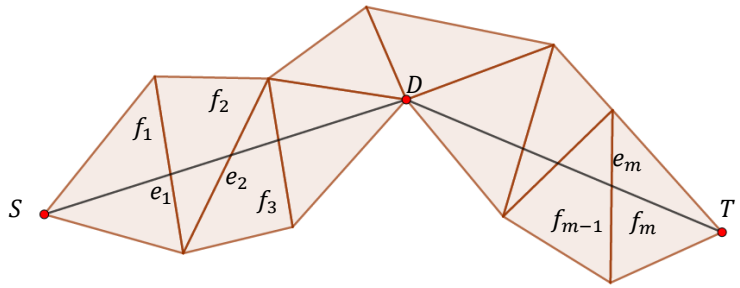


Figure 3.1: A shortest path that the black polyline from the source S to the destination T on the face sequence F.

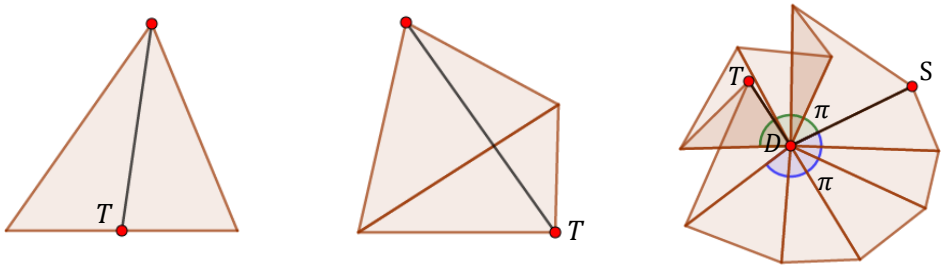
3.1 Preliminaries

3.1.1 Geodesic Paths

The shortest path in the 2D plane is a straight segment between two points. A geodesic path on a triangle mesh is also a locally straight polyline along with the unfolding image. See Figure 3.1. Herein, we cite the complete definition of a geodesic path from [7] as Definition 3.1. A *face sequence* F is defined as a list of adjacent faces $f_1, f_2, ..., f_m$, which f_i and f_{i+1} share a common triangle edge e_i as shown in Figure 3.1. The list of edges $E = (e_1, e_2, ..., e_{m-1})$ is an *edge sequence* E. [7] also gave a definition of *planar unfolding* that is often used in the research of geodesic path on a triangle mesh. An *edge sequence* $E = (e_1, e_2, ..., e_n)$ is unfolded as follows: rotate f_1 around e_1 until its plane is coplanar to that of f_2, rotate f_1 and f_2 around e_2 until their plane is coplanar to that of f_3, and repeat this way until all the faces $f_1, f_2, ..., f_{m-1}$ locate on the plane of f_{m-1}.

Definition 3.1. *"The general form of a geodesic path is a path which goes through an alternating sequence of vertices and (possibly empty) edge sequences such that the unfolded image of the path along any edge sequence is a straight line segment and the angle of the path passing through a vertex is greater than or equal to π."*

There are three possible situations of the local geodesic path between two endpoints as shown in Figure 3.2. Figure 3.2 (a) (b) exhibit the shortest path as a black segment within a (unfolding) 2D plane between S and T. Figure 3.2 (c) represents an shortest path through the saddle vertex D as a black polyline from source S to termination T. A vertex is called *saddle* if its total angle is bigger than 2π.



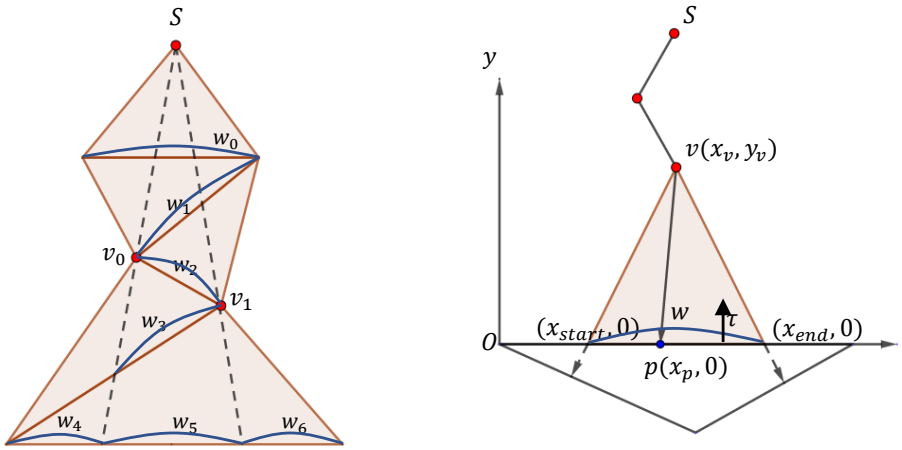
(a) (b) (c)

Figure 3.2: Three types of local geodesic path. (a) The shortest path within one triangle; (b) The shortest path along unfolding triangles; (c) The shortest path through a saddle vertex.

3.1.2 Windows

Let $M = (V, E, F)$ be a triangle mesh, where V, E, F represents the set of vertices, edges, and faces respectively. Given a source vertex S, our task is to find the geodesic paths from S to all vertices $v_i \in V$ on mesh M. One example of the geodesic paths in local triangle mesh is illustrated in Figure 3.3 (a). The geodesic paths emit from source vertex S to all directions, MMP algorithm adopts the "window" structure to track groups of shortest paths which share the same face-edge sequence. Suppose v_0 and v_1 are saddle vertices, the geodesic path from source S to any point $p \in w_4$ and the geodesic path from source S to any point $q \in w_6$ must pass through v_0 and v_1 respectively as Figure 3.3 (a). v_0 and v_1 are defined as $pseudo - source$ in this situation, namely, the geodesic path emits from a

pseudo-source when it meets a saddle vertex.



(a) (b)

Figure 3.3: The illustration of windows. (a) Windows on local triangle mesh; (b) The window structure.

The window structure is denoted by a 6-tuple $(x_{start}, x_{end}, x_v, y_v, d_v, \tau)$ in MMP algorithm. Suppose that geodesic path starting from the source vertex S passes through one or more saddle vertices on its way, the pseudo-source v is the nearest saddle vertex to the window w, p is any point in window w. Let d_v represents the length of geodesic path from source S to pseudo-source v, $d(v, p)$ represents the geodesic distance between pseudo-source v and point p, and d_p represents the length of geodesic path from source S to point p. Obviously, the value of d_p equals to the sum of d_v and $d(v, p)$. In order to compute conveniently, the window is transformed into a 2D coordinate system as Figure 3.3 (b). The origin locates in $(0, 0)$, $(x_{start}, 0)$ and $(x_{end}, 0)$ are the coordinates of window endpoints, the pseudo-source v is represented by (x_v, y_v) and the point p is represented by $(x_p, 0)$. The parameter τ is a binary direction that identifies the pseudo-source v lies in which side of the window w. Then the geodesic distance d_p can be computed as follow:

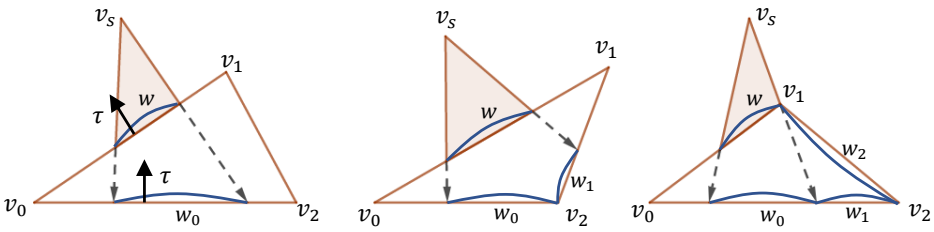
$$d_p = d_v + \sqrt{(x_v - x_p)^2 + {y_v}^2} \tag{3.1}$$

3.1.3 Window Propagation

In the process of window propagation, new windows would be generated on the corresponding opposite edges. Suppose a window w lies on edge $\overline{v_0v_1}$ of $\triangle v_0v_1v_2$, new windows will be found on edge $\overline{v_1v_2}$ and edge $\overline{v_2v_0}$. Broadly speaking, new potential windows are generated in three ways. As shown in Figure 3.4 (a), the rays from pseudo-source v_s extend through old window w towards one edge $\overline{v_2v_0}$ and arrive on new window w_0. The new window w_0 can be represented by its 6-tuple, as vertex v_0 will represent its origin on 2D coordinates. The pseudo-source v_s and the pseudo-source distance d_v are unchanged, the parameter τ is assigned to point into face $\triangle v_0v_1v_2$. Likewise, as shown in Figure 3.4 (b), two new windows w_0 and w_1 are generated when the rays field covers the new intervals on edge $\overline{v_2v_0}$ and edge $\overline{v_1v_2}$. Figure 3.4 (c) shows a special case of window propagation when one of the endpoints of old window w is a saddle vertex v_1 and meanwhile a part of triangle $\triangle v_0v_1v_2$ lies to the right of ray(v_s, v_1). Since the shortest paths to the unilluminated area must pass through vertex v_1, v_1 will be a new pseudo-source. In this case, new windows w_1 and w_2 are generated except for new window w_0 which is illuminated by old pseudo-source at vertex v_s. The definition of the new window w_0 is similar to the two former cases. For the definition of new windows w_1 and w_2, the pseudo-source v_s will be updated by new pseudo-source v_1, the pseudo-source distance d_v will be updated by the geodesic distance d_{v_1} between vertex v_1 and source.

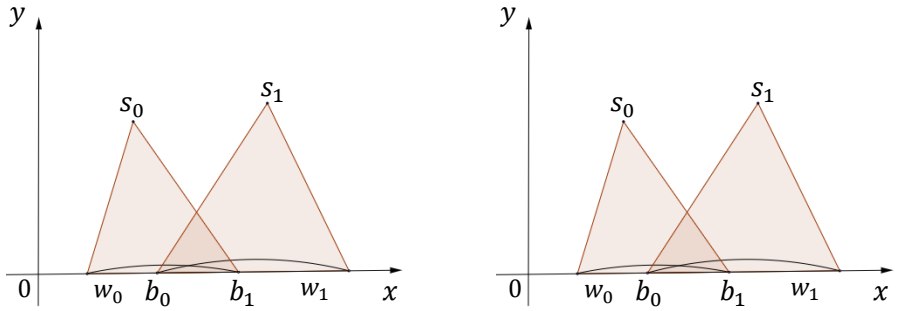
3.1.4 Window Clipping

The new potential window may intersect the old window on the same edge during the propagation. To ensure that all windows cover the shortest paths, the overlapped windows have to be clipped. As shown in Figure 3.5 (a), windows w_0 and w_1 intersect at interval $[b_0, b_1]$, vertex s_0



(a) (b) (c)

Figure 3.4: Window propagation on the opposing edges. (a) Only one new window w_0 is generated on the edge $\overline{v_0v_2}$; (b) Two new windows w_0 and w_1 are generated on the edge $\overline{v_0v_2}$ and $\overline{v_2v_1}$ respectively; (c) Special case as v_1 is a new pseudo-source, w_0, w_1 and w_2 are generated.



(a) (b)

Figure 3.5: (a) Windows w_0 and w_1 intersect at interval $[b_0, b_1]$; (b) One clipping case at point p. It's equal that the length of paths which pass through different pseudo-source vertex s_0 and vertex s_1, i.e., $d_{s_0} + \|s_0 - p\| = d_{s_1} + \|s_1 - p\|$.

and vertex s_1 are pseudo-sources, d_{s_0} and d_{s_1} represent the shortest distance from source to pseudo-sources s_0 and s_1 respectively. To clip the overlapped interval $[b_0, b_1]$, it's the key to find splitting points with equal distance defined by different windows. As shown in Figure 3.5 (b), the length of path which passes through window w_0, w_1 can be described as: $\Phi_0(p) = d_{s_0} + \sqrt{(x_{s_0} - x_p)^2 + {y_{s_0}}^2}$ and $\Phi_1(p) = d_{s_1} + \sqrt{(x_{s_1} - x_p)^2 + {y_{s_1}}^2}$ respectively in the 2D coordinates.

Define $\Phi(x)$ as the difference:

$$\Phi(x) = \Phi_1(x) - \Phi_0(x) = d_{s_1} - d_{s_0} + \sqrt{(x_{s_1} - x)^2 + {y_{s_1}}^2} - \sqrt{(x_{s_0} - x)^2 + {y_{s_0}}^2} \tag{3.2}$$

The x-value of possible clipping point p must be the solution of $\Phi(x) = 0$ as:

$$d_{s_1} - d_{s_0} + \sqrt{(x_{s_1} - x)^2 + {y_{s_1}}^2} - \sqrt{(x_{s_0} - x)^2 + {y_{s_0}}^2} = 0 \tag{3.3}$$

[9] showed that Equation 3.3 is able to be simplified to a quadratic equation with one unknown in the required range, i.e., $x \in [b_0, b_1]$ such that

$$Ax^2 + Bx + C = 0, \tag{3.4}$$

where:

$$A = \alpha^2 - \beta^2, \qquad B = \gamma\alpha + 2x_{s_1}\beta^2, \qquad C = \frac{1}{4}\gamma^2 - \|s_1\|^2\beta^2,$$

with:

$$\alpha = x_{s_1} - x_{s_0}, \qquad \beta = d_{s_1} - d_{s_0}, \qquad \gamma = \|s_0\|^2 - \|s_1\|^2 - \beta^2,$$
$$\|s_0\| = \sqrt{{x_{s_0}}^2 + {y_{s_0}}^2}, \qquad \|s_1\| = \sqrt{{x_{s_1}}^2 + {y_{s_1}}^2}.$$

Finally, they find the reasonable x-value by judging the solutions whether they exist in the intersection $[b_0, b_1]$. Although the manner of solving the quadratic Equation 3.4 is well-understood, the unnecessary square-root calculations happen during computing $delta\Delta$ and unreasonable solutions. To avoid these disadvantages, [1] presented a method which checks whether the solutions of Equation 3.3 exists inside the intersection $[b_0, b_1]$ before solving its quadratic Equation 3.4 as below.

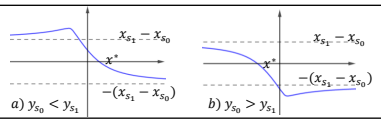
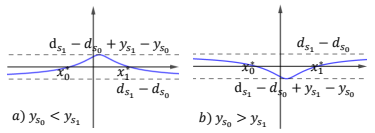
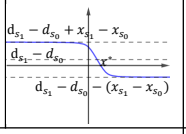
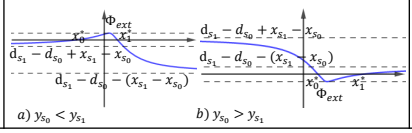
Situations	$\Phi(x)$	Clipping Strategies
1. $x_{s_0} = x_{s_1}, y_{s_0} = y_{s_1}, d_{s_0} \neq d_{s_1}$	$\Phi(x) = d_{s_1} - d_{s_0}$	1) If $d_{s_0} < d_{s_1}$, del$\langle s_1 b_0 b_1\rangle$; 2) If $d_{s_0} > d_{s_1}$, del$\langle s_0 b_0 b_1\rangle$.
2. $x_{s_0} = x_{s_1}, y_{s_0} \neq y_{s_1}, d_{s_0} = d_{s_1}$	$\Phi(x) = \frac{(y_{s_1} - y_{s_0})(y_{s_1} + y_{s_0})}{\Gamma_1(x) + \Gamma_0(x)}$	1) If $y_{s_0} < y_{s_1}$, del$\langle s_1 b_0 b_1\rangle$; 2) If $y_{s_0} > y_{s_1}$, del$\langle s_0 b_0 b_1\rangle$.
3. $x_{s_0} \neq x_{s_1}, y_{s_0} = y_{s_1}, d_{s_0} = d_{s_1}$	Based on Equation 3.5 and 3.6, $\Phi'(x) < 0$ is a monotonic decreasing function. Assume that $\Phi(x^*) = 0$.	1) If $x^* < b_0$, del$\langle s_0 b_0 b_1\rangle$; 2) If $x^* > b_1$, del$\langle s_1 b_0 b_1\rangle$; 3) If $x^* \epsilon (b_0, b_1)$, del$\langle s_1 b_0 x^*\rangle\langle s_0 x^* b_1\rangle$
4. $x_{s_0} \neq x_{s_1}, y_{s_0} \neq y_{s_1}, d_{s_0} = d_{s_1}$		For case a) and case b): 1) If $x^* < b_0$, del$\langle s_0 b_0 b_1\rangle$; 2) If $x^* > b_1$, del$\langle s_1 b_0 b_1\rangle$; 3) If $x^* \epsilon (b_0, b_1)$, del$\langle s_1 b_0 x^*\rangle\langle s_0 x^* b_1\rangle$.
5. $x_{s_0} = x_{s_1}, y_{s_0} \neq y_{s_1}, d_{s_0} \neq d_{s_1}$		For case a): (case b) could be analysed similarly.) 1) If $d_{s_1} - d_{s_0} > 0$, del$\langle s_1 b_0 b_1\rangle$; 2) If $d_{s_1} - d_{s_0} + y_{s_1} - y_{s_0} < 0$, del$\langle s_0 b_0 b_1\rangle$; 3) If $d_{s_1} - d_{s_0} < 0$ and $d_{s_1} - d_{s_0} + y_{s_1} - y_{s_0} > 0$, compute x_0^* and x_1^* by Equation 3.3. 3.1) If $x_0^* > b_1$ or $x_1^* < b_0$, del$\langle s_0 b_0 b_1\rangle$; 3.2) If $x_0^* < b_0$ and $x_1^* > b_1$, del$\langle s_1 b_0 b_1\rangle$; 3.3) If $x_0^* > b_0$ and $b_1 \epsilon (x_0^*, x_1^*)$, del$\langle s_0 b_0 x_0^*\rangle\langle s_1 x_0^* b_1\rangle$; 3.4) If $x_1^* < b_1$ and $b_0 \epsilon (x_0^*, x_1^*)$, del$\langle s_1 b_0 x_1^*\rangle\langle s_0 x_1^* b_1\rangle$; 3.5) If $x_0^* > b_0$ and $x_1^* < b_1$, del$\langle s_0 b_0 x_0^*\rangle\langle s_1 x_0^* x_1^*\rangle\langle s_0 x_1^* b_1\rangle$
6. $x_{s_0} \neq x_{s_1}, y_{s_0} = y_{s_1}, d_{s_0} \neq d_{s_1}$	$\Phi(x)$ has a local extremum $d_{s_1} - d_{s_0}$ at $x_{ext} = \frac{x_{s_0} + x_{s_1}}{2}$, Equation 3.3 has two potential solutions $x_0^* = x_{ext} - \frac{\sqrt{\Delta}}{2A}, x_1^* = x_{ext} + \frac{\sqrt{\Delta}}{2A}$ If $d_{s_1} - d_{s_0} > 0$, $x^* = x_1^*$; If $d_{s_1} - d_{s_0} < 0$, $x^* = x_0^*$ 	1) If $d_{s_1} - d_{s_0} + x_{s_1} - x_{s_0} < 0$, del$\langle s_0 b_0 b_1\rangle$; 2) If $d_{s_1} - d_{s_0} - (x_{s_1} - x_{s_0}) > 0$, del$\langle s_1 b_0 b_1\rangle$; 3) If $d_{s_1} - d_{s_0} + x_{s_1} - x_{s_0} > 0$ and $d_{s_1} - d_{s_0} - (x_{s_1} - x_{s_0}) < 0$, compute x^* and clip w_0, w_1 as Situation 4.
7. $x_{s_0} \neq x_{s_1}, y_{s_0} \neq y_{s_1}, d_{s_0} \neq d_{s_1}$		For case a): (case b) could be analysed similarly.) 1) If $d_{s_1} - d_{s_0} - (x_{s_1} - x_{s_0}) > 0$, del$\langle s_1 b_0 b_1\rangle$; 2) If $\Phi_{ext} < 0$, del$\langle s_0 b_0 b_1\rangle$; 3) If $d_{s_1} - d_{s_0} + (x_{s_1} - x_{s_0}) < 0$ and $\Phi_{ext} > 0$, compute x_0^* and x_1^* by Equation 3.3 and clip w_0, w_1 as Situation 5. case a).3). 4) If $d_{s_1} - d_{s_0} + (x_{s_1} - x_{s_0}) > 0$ and $d_{s_1} - d_{s_0} - (x_{s_1} - x_{s_0}) < 0$, compute x^* by Equation 3.3 and clip w_0, w_1 as Situation 4.

Figure 3.6: Window clipping strategies [1]. Assume that $x_{s_0} < x_{s_1}$ and $y_{s_0}, y_{s_1} > 0$ in all situations. $del\langle sb_0b_1\rangle$ represents deleting the window or sub-window defined by this three points.

[1] is based on the first order derivative of $\Phi(x)$ and it's transformation:

$$\begin{aligned}\Phi'(x) &= \Gamma_1'(x) - \Gamma_0'(x), \\ &= \frac{\Gamma_0(x)(x - x_1) - \Gamma_1(x)(x - x_0)}{\Gamma_0(x) * \Gamma_1(x)}, \qquad (3.5) \\ &= \frac{F(x) * G(x)}{\Gamma_0(x) * \Gamma_1(x) * H(x)}. \qquad (3.6)\end{aligned}$$

where:

$$\Gamma_0(x) = \sqrt{(x_{s_0} - x)^2 + {y_{s_0}}^2}, \qquad \Gamma_1(x) = \sqrt{(x_{s_1} - x)^2 + {y_{s_1}}^2},$$
$$F(x) = (x - x_{s_1}) * y_{s_0} + (x - x_{s_0}) * y_{s_1},$$
$$G(x) = (x - x_{s_1}) * y_{s_0} - (x - x_{s_0}) * y_{s_1},$$
$$H(x) = (x - x_{s_1}) * \Gamma_0(x) + (x - x_{s_0}) * \Gamma_1(x).$$

Assume that $x_{s_0} < x_{s_1}, y_{s_0} > 0, y_{s_1} > 0$, the monotonic of $\Phi(x)$ can be identified by judging whether the value of $\Phi'(x)$ is positive or negative according Equation 3.5 or Equation 3.6 under different situations. And then the clipping strategies are generated as shown in Figure 3.6.

3.1.5 The MMP Algorithm

Algorithm 1: Mitchell-Mount-Papadimitriou(MMP) algorithm

Input: A triangle mesh $M = (V, E, F)$, source vertex s.
Output: The geodesic distance d_i for each vertex $v_i \in V$.

```
1 Assign geodesic distance d_s of source vertex with zero and all other
    vertices' d_i with +∞.
2 Create a priority queue WindowQueue.
3 Create windows for each edge opposite to the source points s and push
    these windows into priority queue WindowQueue.
4 while WindowQueue is not empty do
5 |   Pop a window w_min from WindowQueue.
6 |   w_new = Propagate(w_min).
7 |   w_res = Clipping(w_new, w_old).
8 |   Update priority queue WindowQueue with the result windows w_res.
9 Update the geodesic distance d_i for each vertex v_i with shortest
    distance information of adjacent windows.
```

The main steps of the MMP algorithm are given as Algorithm 1, please refer to [7] [9] for more details. MMP algorithm first assigns the distance value of source s with zero and other vertices' with infinity. To create the windows for each edge opposite to source s which cover the whole edge, and to push

them into the priority queue $WindowQueue$ so that initializing the following window propagation, where $WindowQueue$ is sorted with window distance from source vertex.

Firstly, to pop the window w_{min} with the shortest distance from priority queue $WindowQueue$. Secondly, to propagate window w_{min} and generate new windows as Section 3.1.3. Thirdly, to clip overlapped part between new-generated window and old window as Section 3.1.4. Fourthly, to push the result window after window clipping into priority queue $WindowQueue$. These four steps are repeated until $WindowQueue$ is empty. Finally, to update the geodesic distance d_i for each vertex v_i with shortest distance information of adjacent windows. The basic idea of the MMP algorithm is to propagate the groups of shortest paths, namely windows, through triangle faces in Dijkstra-like [21] manner.

Let n be the number of vertices on the triangle mesh, the number of generated windows is $O(n^2)$ in worst case [7]. It consumes $O(logn)$ time to pop or push a window from or into a priority queue. Therefore, the time complexity of the MMP is n^2logn. MMP algorithm needs to store all windows, so it has $O(n^2)$ space complexity in the worst case. [9] observed that each edge has $O(\sqrt{n})$ windows on average for uniform distributed triangle meshes, therefore MMP algorithm has $O(n^{1.5})$ empirical time complexity.

3.2 Parallelization

3.2.1 Overview

Our parallel-MMP algorithm includes five main steps: window selection, window propagation, window organization, window clipping, and window pushing as Algorithm 2. The two main time-consumed steps that window propagation and window clipping are concurrent, others are sequential. Dif-

ferent from the classical MMP algorithm, we adopt "buckets" structure [12] instead of priority queue to control the order of window propagation in parallel-MMP.

Algorithm 2: Parallel-MMP algorithm

Input: A triangle mesh $M = (V, E, F)$, source vertex s, the selection parameter K and the number of CPU threads T

Output: The geodesic distance d_i for each vertex v_i.

```
1 Initialization: Create windows for edges opposite to source s and push
    them into windows container WindowQueue.
2 while WindowQueue is not empty do
3     Select K windows from WindowQueue.
4     Concurrently Propagate K selected windows in T threads.
5     Organize the newly-generated windows by the index of half-edge.
6     Concurrently Clip overlapped windows in T threads.
7     Push the new and updated windows into WindowQueue.
8 Update the geodesic distance d_i for each vertex v_i.
```

At the beginning of our parallel-MMP algorithm, we create windows for edges opposite to source s and push them into the buckets $WindowQueue$ so that initializing the following windows operations. Firstly, to select K approximate nearest windows from the buckets $WindowQueue$. Secondly, to concurrently propagate these K windows and generate new windows in T threads. Thirdly, to classify the new-generated windows so that the windows on the same half-edge could be aggregated. Fourthly, to concurrently clip the overlapped windows in T threads. There is no data conflict since the intersected windows on the same half-edge are clipped in the same thread. Fifthly, to push the new-generated and updated windows after clipping into the buckets $WindowQueue$. These five steps are repeated until the buckets $WindowQueue$ is empty. Finally, to update the geodesic distance d_i for each vertex v_i with shortest distance information of adjacent windows. Figure 3.7 shows the pipeline of the parallel-MMP framework, the implementation details as following subsections.

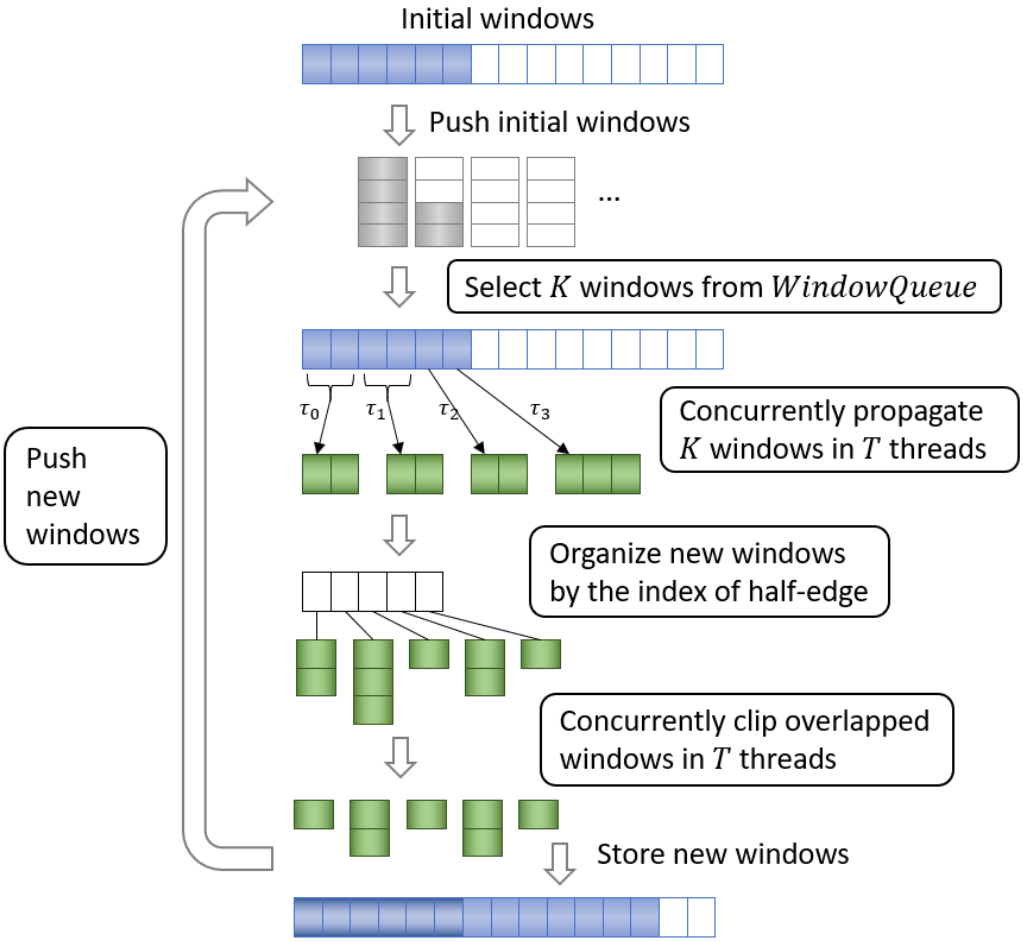


Figure 3.7: Pipeline of parallel-MMP algorithm.

3.2.2 Data Structure

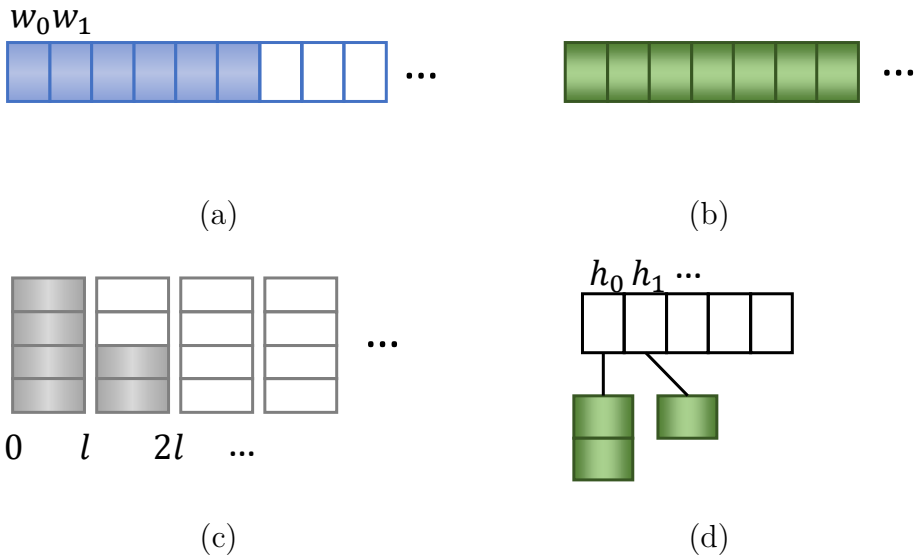
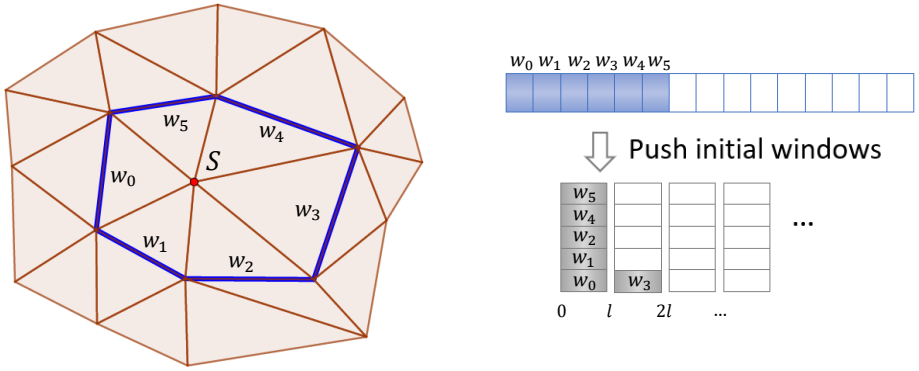


Figure 3.8: (a) Global memory to store windows on edges; (b) Buffer to store temporary generated windows or active half-edges; (c) Buckets to control the order of window propagation; (d) Global halfedge-window mapping container.

Let $M = (V, E, F)$ denote the input triangle mesh, where V, E and F are the set of vertices, edges and faces respectively. We adopt *half-edge* data structure to represent mesh M. Besides the *window* structure as Section 3.1.2, there are four kinds of structures in our algorithm as shown in Figure 3.8. Figure 3.8(a) is a global container to store all generated windows' information which each thread can read and write. Figure 3.8(b) represents the buffer during our algorithm. We utilize buffer queue to store temporarily generated windows' information or the index of active half-edges. Figure 3.8(c) is the "buckets" which consists of multi double linked lists. The *i-th* double linked list stores the pointers of windows whose geodesic distance between $[i*l, (i+1)*l]$, where we assign the l as the average length of edges on a triangle mesh. Our parallel-MMP algorithm adopts "buckets" to store the windows which are about to propagate. Figure 3.8(d) is a global container to store the mapping relationship between half-edge and their win-

dows' information so that we can classify all new windows. h_i represents the *i-th* half-edge and points to the queue of storing the windows that lie on this half-edge. The details of utilizing these four structures are exhibited in the following description of our five main steps.

3.2.3 Initialization



(a) (b)

Figure 3.9: Create initial windows and push them into the buckets according to geodesic distance. (a) The bold blue lines denote the initial windows $w_0, w_1, w_2, w_3, w_4, w_5$ which cover edges opposite to source s; (b) The blue container stores windows' information as global memory, the grey buckets stores the pointer of windows.

---

1 **Function** `Initialization(source` s`):`

  `/* Create windows for covering each half-edge` h_i
  `opposite to source` s `and push them into buckets`
  $WindowQueue$`.` `*/`

  `/*` N `is the number of edges opposite to source` s`.` `*/`

2 **for** $i \leftarrow 0$ **to** N **do**

3 Create a window 6-tuple $w_i = (x_{start}, x_{end}, x_s, y_s, d_s, \tau)$ for half-edge h_i.

4 Store w_i into global window container W.

  `/*` d_i `denotes the geodesic distance between source` s
  `and window` w_i`,` l `denotes width of each bin.` `*/`

5 Push the pointer of w_i into the buckets $WindowQueue.bin[\lfloor \frac{d_i}{l} \rfloor]$.

At the beginning of our algorithm, as Algorithm 2 line 1 shows, we plan

to create windows for each edge opposite to source vertex s and push these windows into the buckets $WindowQueue$ to initial the whole algorithm. See Function *Initialization*. As an example as Figure 3.9(a) that a local triangle mesh, we create initial windows $w_0, w_1, w_2, w_3, w_4, w_5$ and store them in global container W. Then we push these initial windows into $WindowQueue$ according to their geodesic distance. The buckets $WindowQueue$ consists of multi bins which are implemented by a double linked list. The *i-th* bin contains the windows whose geodesic distance between $[i * l, (i + 1)l]$, where l represents the average length of edges on the input triangle mesh. As shown in Figure 3.9(b), we push the pointers of w_0, w_1, w_2, w_4, w_5 into $WindowQueue.bin[0]$ and push the pointer of w_3 into $WindowQueue.bin[1]$ since its geodesic distance is bigger than the average length l.

3.2.4 Window Selection

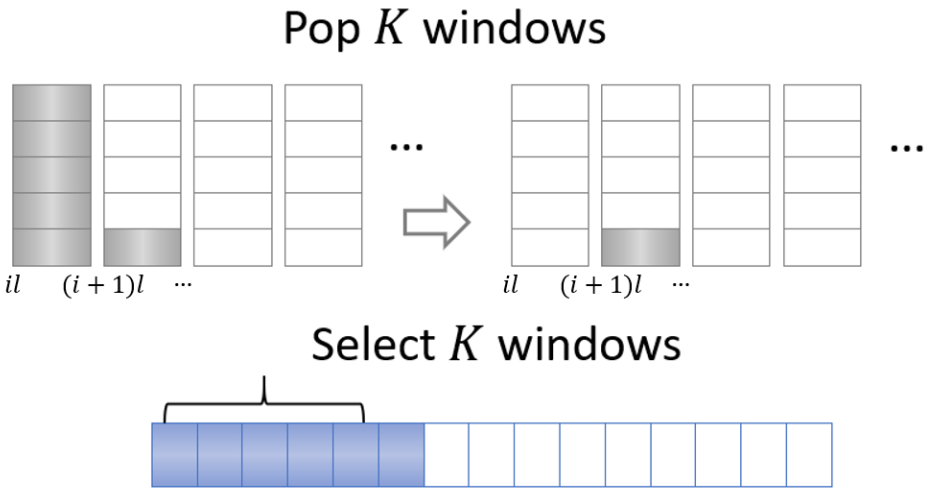


Figure 3.10: Selection K windows from buckets.

In each iteration, we select K approximate nearest windows from $WindowQueue$ as shown in Figure 3.10 at first in order to parallelize window propagation next step. See Function *Window Selection*. As an example of Figure 3.10, we pop 5 nearest windows except for w_3 and store these

```
1 Function Window Selection(parameter K):
      /* k_current denotes the number of selected windows.      */
2     Assign k_current with zero.
      /* WindowQueue.bin[first] is the first nonempty bin in
         the buckets.                                           */
3     while k_current < K do
4         while WindowQueue.bin[first] is empty do
5             first ← first + 1
6         Pop a window from the buckets WindowQueue.bin[first] and
           store it into buffer ToPropagateQueue.
7         k_current ← k_current + 1
```

windows' pointer into buffer queue *ToPropagateQueue*. In this step, we also bring in the property of Fast Wavefront Propagation [12] that selecting multi popped windows and meanwhile maintaining high-quality wavefront.

3.2.5 Parallel Window Propagation

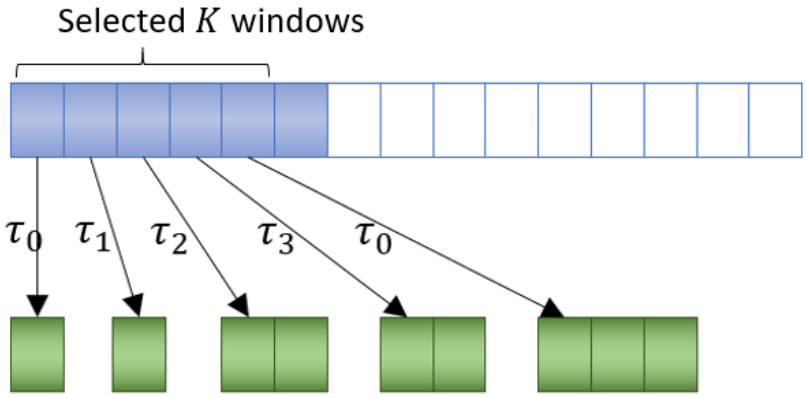


Figure 3.11: Concurrently propagate selected windows on multi threads.

We regard the windows' pointer container *ToPropagateQueue* as the input data in the *Parallel Window Propagation Step*. See Figure 3.11, the K selected windows could be propagated concurrently as Section 3.1.3 on T threads evenly and the newly-generated windows are stored into the buffer queue $NewWindowBuffer$ for linking next step.

```
1 Function Parallel Window Propagation(Selected windows
  ToPropagateQueue):
      /* N_win is the size of ToPropagateQueue that the number
         of selected windows.                                  */
2     Parallel For i ← 0 to N_win do
          /* ToPropagateQueue[i] denotes the pointer of
             corresponding window.                             */
3         Propagate ToPropagateQueue[i] as Section 3.1.3.
4         Store newly-generated windows into the buffer queue
           NewWindowBuffer.
```

3.2.6 Window Organization

In the sequential MMP algorithm, there is a window clipping operation following the window propagation to make sure all windows cover the shortest paths. In our parallel framework, we add *Window Organization* step for preparing to parallelize window clipping later. The newly-generated windows are classified by the index of half-edge as shown in Figure 3.12. The half-edge which includes newly-generated windows is called *active*. We store the index of active half-edges into buffer queue $ActiveHalfedgeBuffer$. By accessing the new windows in $NewWindowBuffer$, we store the pointer of the newly-generated windows into the corresponding half-edge queue in global mapping container $Halfedge$.

```
1 Function Window Organization(Newly-generated windows
  NewWindowBuffer):
      /* N_win is the size of NewWindowBuffer that the
         number of newly-generated windows.                    */
2     for i ← 0 to N_win do
          /* h_i is the index of active half-edge where window
             NewWindowBuffer[i] lies on.                       */
3         Store h_i into buffer queue ActiveHalfedgeBuffer.
          /* w_i is the pointer of window NewWindowBuffer[i].
             */
4         Push w_i into global mapping container Halfedge[h_i].
```

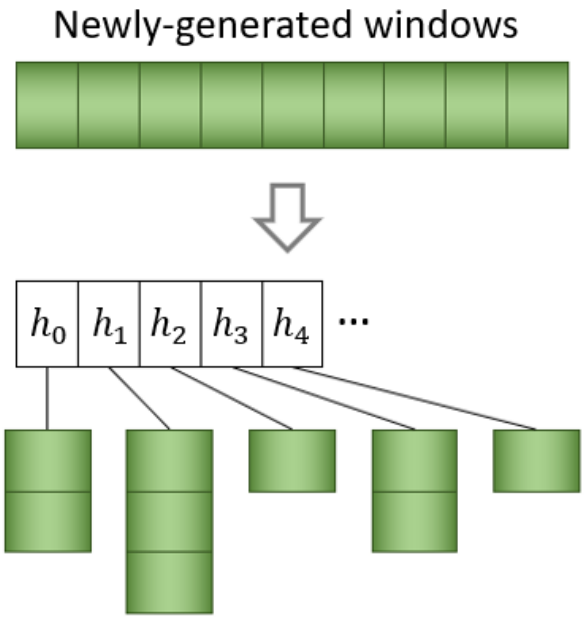


Figure 3.12: Organize newly-generated windows by the index of half-edge.

3.2.7 Parallel Window Clipping

After window organization step, the clipping operation for newly-generated windows could be parallelized as Figure 3.13 and Function *Parallel Window Clipping*. There is no data conflict since the overlapped windows within the same half-edge are clipped within the same thread. Note that there exist possible window adding, deleting, modifying operations during parallelizing window clipping. For preserving the parallelization of window clipping operation, we pre-allocate sufficient global memory in advance to write new windows without data conflict. As a result, the modifying, adding, deleting operations both can be concurrently in global memory container since the deleting operation is equivalent to re-allocate memory for new windows in advance. In addition, we utilize buffer queues $ActiveWindows[thread.id]$ and $RemoveWindows[thread.id]$ to record the added, updated and deleted windows so that updating the buckets $WindowQueue$ in next step.

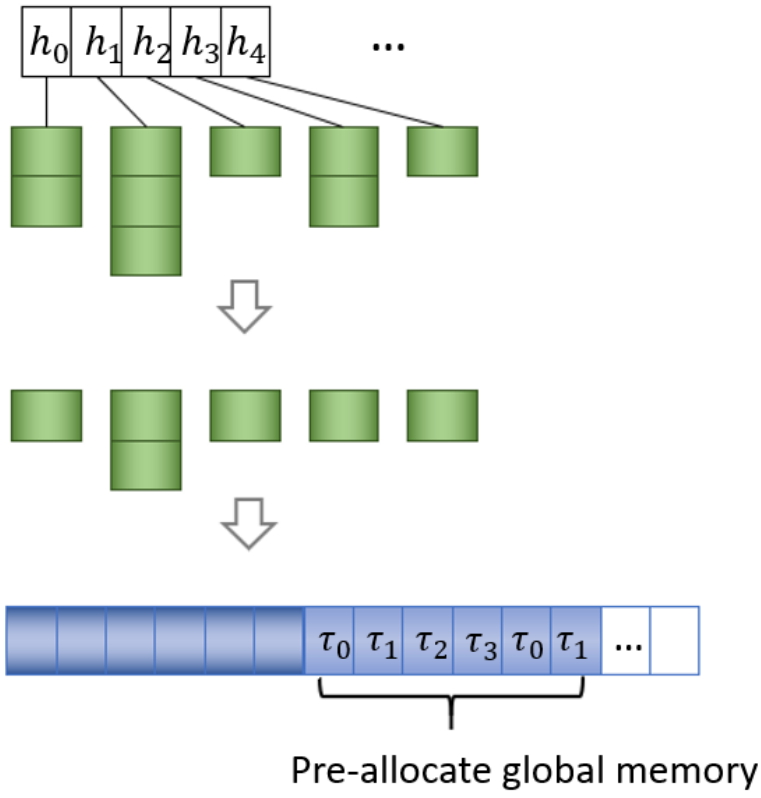


Figure 3.13: Concurrently clip overlapped windows on multi threads.

```
1  Function Parallel Window Clipping(Active half-edges
     ActiveHalfedgeBuffer):
       /* N_h is the size of ActiveHalfedgeBuffer that the
          number of active half-edges.                    */
2      Parallel For i ← 0 to N_h do
         /* ActiveHalfedgeBuffer[i] denotes the pointer of
            corresponding window.                         */
3        h_i ← ActiveHalfedgeBuffer[i].
4        while Halfedge[h_i] is not empty do
5          Pop a newly-generated window w_new from Halfedge[h_i].
           /* N_list is the number of existing windows on this
              half-edge in global memory.                 */
6          for j ← 0 to N_list do
7            if w_j ∩ w_new ≠ ∅ then
8              Clipping(w_j, w_new).
               /* thread.id is the index of thread currently.
                  */
9              Record possible added and updated windows as
                ActiveWindows[thread.id].
10             Record possible deleted windows as
                RemoveWindows[thread.id].
```

3.2.8 Window Pushing

Figure 3.14: Push and update windows for *WindowQueue*.

```
1 Function Window Pushing(Buffer queue ActiveWindows and
  RemoveWindows):
      /* T is the number of threads.                          */
2     for i ← 0 to T do
          /* N_r is the size of RemoveWindows[i].             */
3         for j ← 0 to N_r do
4             w_delete ← RemoveWindows[i][j].
5             Delete the pointer of window w_delete from WindowQueue.
          /* N_a is the size of ActiveWindows[i].             */
6         for j ← 0 to N_a do
7             w_add ← ActiveWindows[i][j].
              /* d_add is the geodesic distance from w_add to
                 source, l denotes the width of each bin.     */
8             Push the pointer of window w_add into the buckets
               WindowQueue.bin[⌊d_add / l⌋].
```

At last, we delete the pointers of $RemoveWindows$ from $WindowQueue$ and push the pointers of $ActiveWindows$ into the buckets $WindowQueue$. See Function *Window Pushing* and Figure 3.14, which it runs $O(1)$ time for per deleting or adding operation. The five steps are repeated until $WindowQueue$ is empty. Finally, we update the geodesic distance for each vertex by its adjacent window with the shortest geodesic distance.

3.2.9 Correctness and Performance Analysis

Our parallel-MMP algorithm has the same results as the sequential MMP algorithm in theory and practice because our parallel-MMP algorithm only modifies the order of window propagation and avoids all possible existing data conflict.

The classical MMP algorithm runs in $O(n^2 logn)$ time complexity, for it adopts the priority queue to store windows and control the order of windows propagation. In fact, our parallel-MMP algorithm is a parallel version based on the FWP-MMP algorithm with $O(n^2)$ time complexity, since we

adopt the "buckets" data structure instead of priority queue to organize windows. Since the number of windows is $O(n^2)$ [9], the parallel window propagation and parallel window clipping have $O(n^2/T)$ time complexity, where T denotes the number of threads based on CPU. Other steps' time complexity is $O(n^2)$. In total, the complexity of our parallel MMP algorithm is $O(n^2/T + n^2)$.

3.3 Experimental Results

3.3.1 Environment Setting

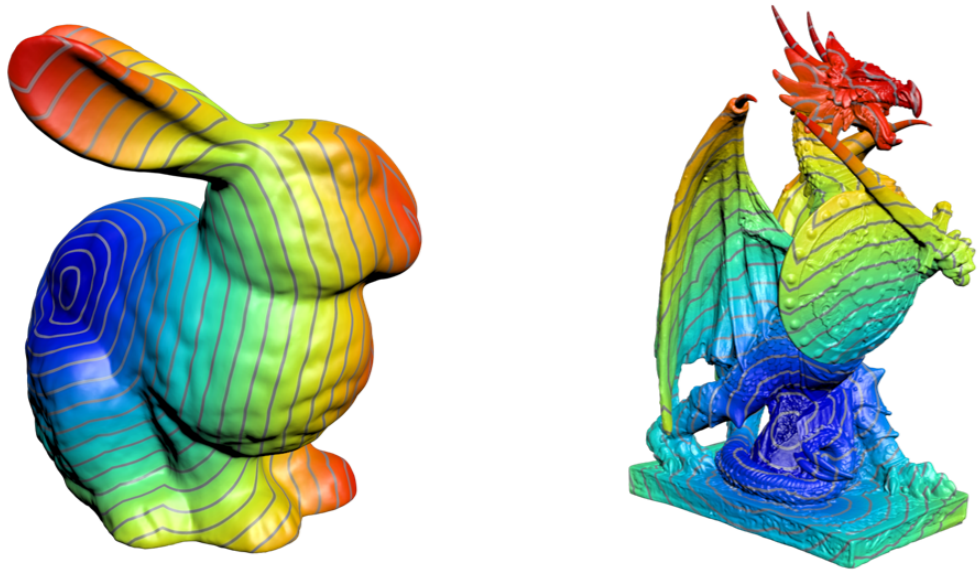


Figure 3.15: Two results of our parallel-MMP algorithm. The left image represents a Bunny model with 500 thousand vertices; the right image represents a FantasyDragon model with 2 million vertices.

We test our parallel-MMP algorithm on various models. Figure 3.15 shows the geodesic isoline result of two models, where the blue vertices are nearer than the read vertices to the source. To evaluate our parallel-MMP further, we compare our algorithm with the state-of-the-art algorithms, MMP, FWP-MMP, VTP. See Table 3.1. The whole parallel-MMP algorithm is 1.6-1.7

Model	MMP	FWP-MMP	VTP	Parallel-MMP		
				T = 1	T = 2	T = 4
Fandisk(215k)	79.882	26.891	21.973	28.01	21.121	16.014
Block(315k)	128.893	37.973	27.328	40.521	29.803	23.15
Impeller(317k)	113.494	37.492	27.198	39.717	29.748	23.428
Bunny(500k)	374.026	89.894	58.433	94.017	69.936	53.543
Bimba(500k)	281.686	78.042	41.164	82.616	60.757	45.571
Hand(575k)	752.561	184.103	101.597	193.325	143.431	109.43
Rockarm(600k)	690.677	165.623	94.986	173.17	129.116	99.172
FantasyDragon(1m)	166.802	59.902	41.834	65.518	47.963	36.611
FantasyDragon(2m)	592.696	175.734	95.296	200.61	139.998	100.199
Tricep(1m)	160.062	59.32	39.397	65.030	47.220	36.077
Tricep(2m)	525.891	167.058	89.961	191.334	136.711	102.977

Table 3.1: Performance comparison with MMP, FWP-MMP, VTP algorithm, where the unit of time is second, T is the number of threads. The contents in parentheses means the number of vertices on a model.

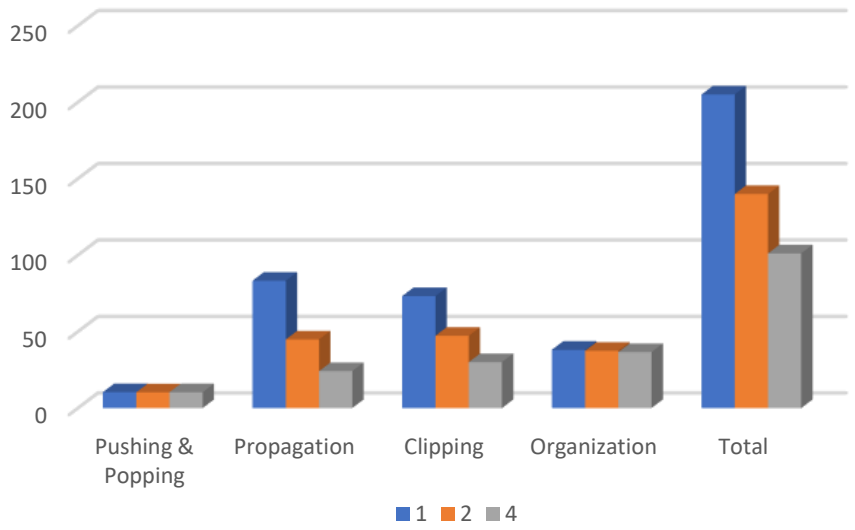
times faster than the FWP-MMP algorithm and 5-7 times faster than the MMP algorithm.

3.3.2 Profiling

To observe the performance of our parallel-MMP algorithm under multi-threads, we measure the execution time of each step in Figure 3.16. Taking FantasyDragon with 2 million vertices, for example, we divide the whole algorithm into four kinds of operations, where *Pushing&Poping* includes window selection and window pushing. The two parallel steps, window propagation, and window clipping obviously speed up as the number of threads increase.

3.3.3 Workload and Imbalance

We test our parallel-MMP algorithm on a Hand model and Fantasy model with different K parameter. K denotes the number of selected to-be-propagated windows in the beginning for each iteration. Figure 3.17 illustrates that when K is bigger than 1000, the performance of our parallel-MMP becomes stable.



T	Pushing & Popping	Propagation	Clipping	Organization	Total
1	10.373	83.242	73.224	38.059	204.898
2	10.238	44.848	47.264	37.444	139.794
4	10.320	24.129	30.027	36.569	101.045

Figure 3.16: The profiling of our parallel MMP algorithm under different number of threads on FantasyDragon model with 2 million triangles. The unit of time is second.

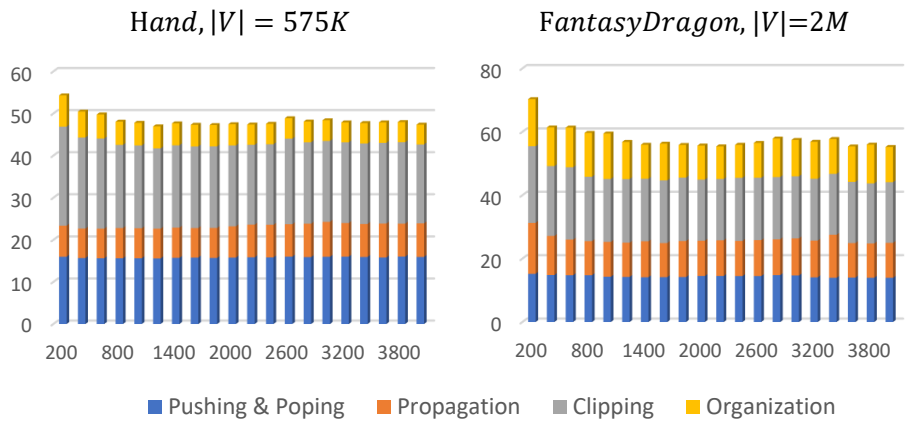


Figure 3.17: The performance curve under different K parameter in 4 threads. (a) Hand model with 575 thousand vertices; (b) Fantasy model with 2 million vertices.

The window propagation tasks for each thread are balanced, for the to-be-propagated windows are distributed into all threads evenly. However, the window clipping tasks is not as balanced as window propagation in our parallel framework, since the number of possible pruned windows for each half-edge is different. Thus, we adopt a dynamic scheduling type of OpenMP [90] in parallel window clipping and slightly improve the effective physical core utilization.

3.4 Discussions

The classical MMP algorithm runs as $O(n^2 logn)$ time complexity since it adopts the priority queue to store windows and control the order of windows propagation. In fact, our parallel-MMP algorithm is a parallel version based on the FWP-MMP algorithm with $O(n^2)$ time complexity since we adopt the "buckets" data structure instead of priority queue to organize windows. Our parallel-MMP preserves the same number of generated windows with FWP-MMP and the 1-thread parallel-MMP has approximate performance with the FWP-MMP algorithm. The main time-consumed steps of

our parallel-MMP, window propagation, and window clipping could be parallelized fully. Therefore, our parallel-MMP algorithm speedup as the number of threads increases in general. Our parallel-MMP algorithm is the fastest geodesic algorithm to preserve the geodesic paths from source to all destinations. However, our parallel-MMP algorithm under 4 threads only has comparable performance with Vertex-oriented Triangle Propagation(VTP) algorithm that is the state-of-art sequential exact algorithm. Many applications only require the geodesic distance instead of the geodesic path, so we move to develop a much faster geodesic algorithm based on VTP.

Chapter 4

Parallel Vertex-oriented Triangle Propagation Algorithm

In this chapter, we develop a parallel version of VTP, called Parallel-VTP or PVTP, that can propagate multiple window lists from multiple vertices simultaneously. To avoid data conflicts, PVTP proceeds with 3 steps in each iteration, which are K-window-list selection, parallel window list propagation, and vertex distance updating and window list merging. Our algorithm has an empirical $O(\frac{n^2}{T})$ time complexity, where n is the number of vertices and T is the number of threads. Extensive evaluation shows that PVTP improves the speed of the sequential VTP by a factor of 2.5$\sim$3 for $T = 4$ and 4$\sim$5 for $T = 8$ for triangular meshes with regular tessellation and over 1 million vertices.

We observe that wavefront propagation of exact algorithms slows down when the wavefront has a long circumference, hereby containing a large number of windows pending processing. To improve the efficiency of window prop-

agation, we develop an approximate variant of VTP, called Approximate VTP or AVTP, which trades speed for accuracy by resetting window on the wavefront when its radius is a multiple of λh, where λ is a user-specified parameter and h is average edge length. AVTP becomes Dijkstra's algorithm when λh is less than the minimal edge length and becomes the exact VTP when λh is greater than the longest geodesic distance on the model. We show that AVTP has a theoretical time complexity $O(n\lambda)$, which is also confirmed by computational results. The proposed parallelization and approximation techniques can be either used separately or combined together. Our source code is available at https://github.com/djie-0329/PVTP.

4.1 Preliminaries

Our parallelization and approximation techniques are built upon the sequential VTP algorithm [3]. To make the paper self-contained, we briefly introduce the core of VTP in this section. We refer the readers to [3] [91] for details.

4.1.1 Window Pruning

There would be a lot of useless windows created during propagation, which do not contribute the shortest distance. VTP detects redundant windows by a set of distance-based filters.

Consider a window $w = [a_0, b_0]$ with pseudo-source s_0 on the triangle edge AB. Using the ICH filter [2], w is redundant if one of the following inequalities is satisfied

$$d(s, s_0) + \| \overline{s_0 b_0} \| > d(s, A) + \| \overline{A b_0} \|, \tag{4.1}$$

or

$$d(s, s_0) + \| \overline{s_0 a_0} \| > d(s, B) + \| \overline{B a_0} \|, \tag{4.2}$$

where $\|PQ\|$ is the Euclidean distance between P and Q. See Figure 4.1(a).

There are 3 pairwise window pruning strategies used in VTP [3] [4]. Consider two windows $w_0 = [a_0, b_0]$ and $w_1 = [a_1, b_1]$ on the triangle edge AB. s_0 and s_1 are the pseudo-sources of windows w_0 and w_1 respectively. In Figure 4.1(b), let I be the intersection of $\overline{s_0 a_0}$ and $\overline{s_1 b_1}$. If $d(s, s_0) + \| \overline{s_0 I} \| > d(s, s_1) + \| \overline{s_1 I} \|$, delete window w_0; otherwise, delete window w_1. In Figure 4.1(c), if $d(s, s_0) + \| \overline{s_0 a_0} \| > d(s, s_1) + \| \overline{s_1 a_0} \|$, delete w_0; otherwise, the part $[a_1, a_0]$ of window w_1 is redundant. In Figure 4.1(d), if $d(s, s_0) + \| \overline{s_0 b_0} \| > d(s, s_1) + \| \overline{s_1 b_0} \|$, delete window w_0; otherwise, the part $[b_0, b_1]$ of window w_1 is redundant.

4.1.2 Synchronized Window Propagation in a Triangle

In order to thoroughly remove abundant windows at early stage, VTP [3] groups the windows on a half-edge and propagates *window lists* in a synchronized manner across triangles. See Figure 4.2(a). VTP adopts three rules for window list propagation and window pruning in a triangle.

Rule 1, called "one angle two sides" rule, removes unnecessary propagation during window list splitting. Consider a window list $wl_{\mathrm{AB}} = \{w_i | i = 0, 1, 2....\}$ on half-edge AB, which will propagate across ΔABC. For each window $w_i \in wl$, VTP computes a *separating point sp* by minimizing

$$\arg\min_{sp \in w_i} \{\sigma_{s_i} + \|s_i - sp\| + \|sp - C\|\}, \tag{4.3}$$

where s_i is the pseudo-source of w_i, σ_{s_i} is the geodesic distance between source s and pseudo-source s_i. Denote by w_s the window which supports shortest distance to vertex C. $wl.sp$ is the separating point of w_s. Fig-

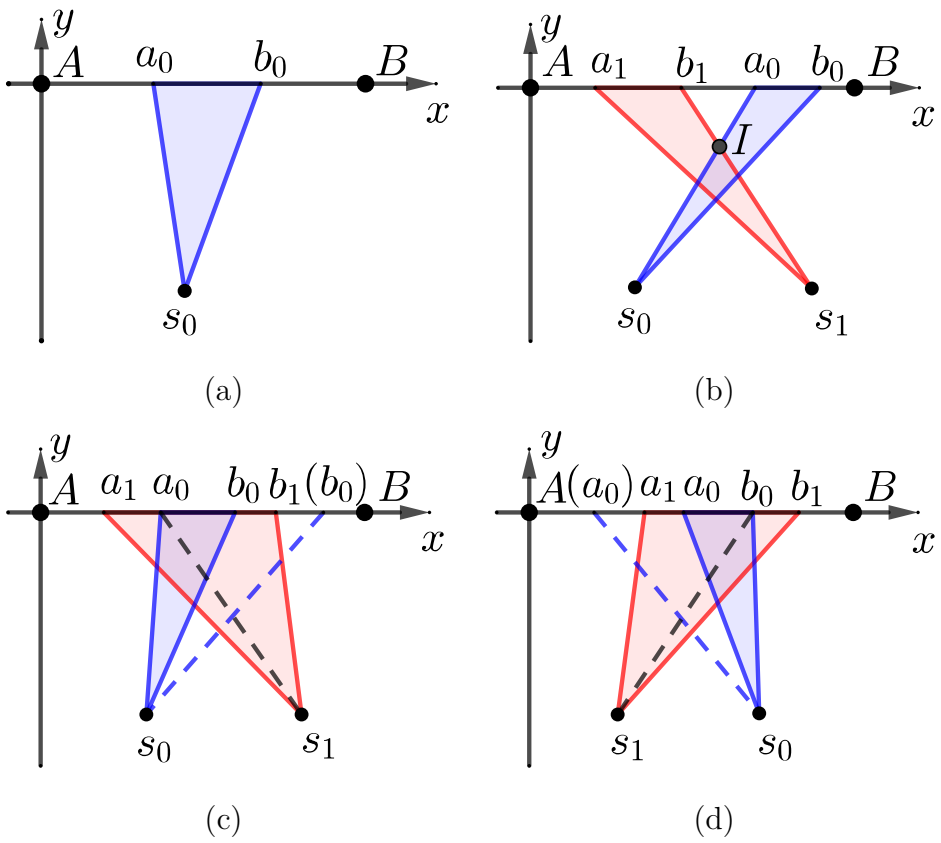


Figure 4.1: Window pruning rules. (a) The ICH window filter [2]. (b), (c) and (d) show three situations of pairwise pruning between two overlapped windows [3] [4].

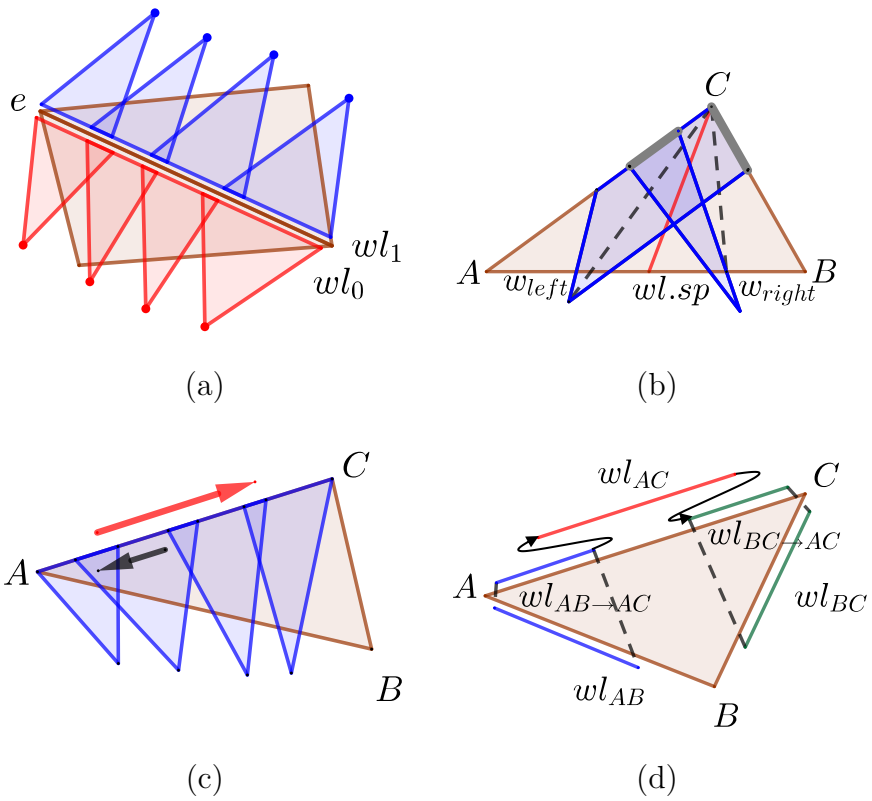


Figure 4.2: Window propagation rules in a triangle [3]. (a) Each side of the edge e contains a window list. (b) The propagation of window w_{left} to edge BC is redundant when $w_{\text{left}}.sp < wl.sp$, and the propagation of window w_{right} to edge AC is redundant when $w_{\text{right}}.sp > wl.sp$. The redundant windows are drawn in grey. (c) Pairwise windows pruning during window list propagation. (d) Propagating window lists wl_{AB} and wl_{BC} produces $wl_{\text{AB}\to\text{AC}}$ and $wl_{\text{BC}\to\text{AC}}$ respectively. Update window list wl_{AC} by merging $wl_{\text{AB}\to\text{AC}}$ and $wl_{\text{BC}\to\text{AC}}$.

ure 4.2(b) shows an example of the "one angle two sides" rule. For each window $w_i \in wl$ and $w_i \neq w_s$, the propagation of w_i to edge BC is redundant if $w_i.sp < wl.sp$, and the propagation of w_i to edge AC is redundant if $w_i.sp > wl.sp$ [3].

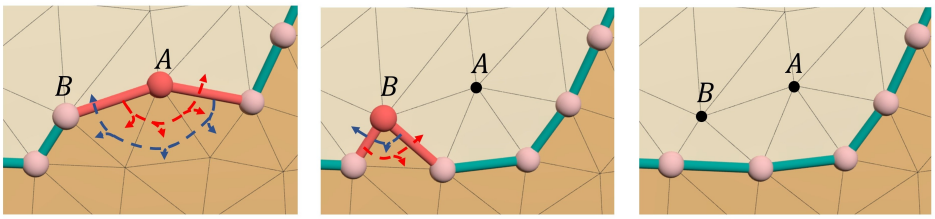
Rule 2 performs the pairwise window pruning for window list wl. It traverses all windows in the outer loop (red arrow) and checks each window against its preceding windows in the inner loop (black arrow). See Figure 3(c). During window list propagation, VTP utilizes the ICH filter and Rule 2 to prune or remove redundant windows.

The newly generated window lists associated to a half-edge are merged by Rule 3. As Figure 4.2(d) shows, for the half-edge (A, C), $wl_{\mathrm{BC}\to\mathrm{AC}}$ is inserted in front of wl_{AC} and $wl_{\mathrm{AB}\to\mathrm{AC}}$ is appended after it.

4.1.3 Vertex-oriented Triangle Propagation

One of the key features of VTP is that it takes *vertices* as primitives and propagates geodesic path across triangles around the vertex popped from a priority queue iteratively. Algorithm 3 shows the pseudo code of VTP. Given a source vertex s, VTP initializes the distances $d(s) = 0$ and $d(v) = \infty$ for $v \neq s$. It creates a window for each half-edge opposite to s and stores it in the half-edge's window list. It also pushes the adjacent vertices into a priority queue $\mathcal{Q}$ according to their distances to s.

Denote by wf the wavefront and R the interior region it borders. At each iteration, the VTP algorithm pops up a vertex v_i with shortest geodesic distance value from $\mathcal{Q}$ and extends its one-ring region as follows. First, the wavefront wf and its enclosed area R are extended to the one-ring region of v_i. It pushes the window lists adjacent or opposite to v_i into the FIFO queue $\mathcal{W}$ if they are inside the swept region R. While the window list queue $\mathcal{W}$ is not empty, VTP pops a window list wl_i from $\mathcal{W}$. Then it prunes



(a) (b) (c)

Figure 4.3: Vertex-oriented triangle propagation. From (a) to (b): window propagation across triangles around vertex A; From (b) to (c): window propagation across triangles around vertex B. The swept region is light-colored and the un-swept region is dark-colored.

the redundant windows of wl_i using the pruning rules in Section 4.1.1 and Rule 2. Next it propagates the windows of wl_i in a synchronized manner and merge the newly-generated windows into the corresponding window lists (Rule 1 and Rule 3 in Section 4.1.2). If the newly-generated windows are inside the area R, push them into $\mathcal{W}$. Finally, if there exist vertices whose geodesic distances are updated during one-ring region growing of v_i, push the updated vertices into the priority queue $\mathcal{Q}$. Repeat these steps until the priority queue $\mathcal{Q}$ is empty.

In summary, the VTP algorithm iteratively extends the swept area R in a Dijkstra-like manner with the aid of vertices in priority queue. Figure 4.3 shows examples of VTP iterations. Suppose A is the nearest vertex to source s at the moment. The triangles incident to A are propagated in the current iteration. Let B be next vertex popped up from the priority queue $\mathcal{Q}$. The triangles adjacent to B will be propagated in the next iteration. After window list propagation, the updated vertices on the wavefront are pushed into $\mathcal{Q}$.

Algorithm 3: Vertex-oriented Triangle Propagation Algorithm [3]

Input: A triangle mesh $M = (V, E, F)$, source vertex id $s \in \{1, \cdots, |V|\}$.

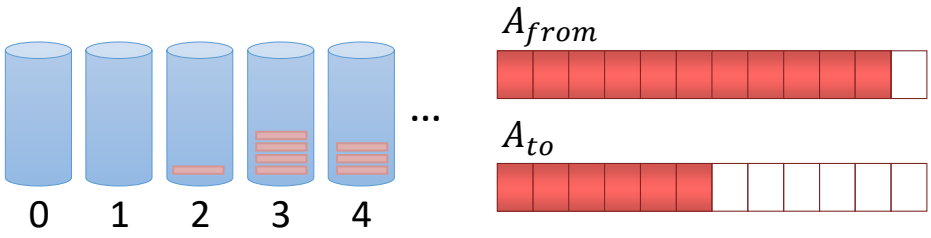
Output: The geodesic distance d_i for each vertex v_i.

```
Initialize distances d_S ← 0 and d_i ← ∞, ∀i ≠ s.
Create a priority queue Q for vertices.
Create a FIFO queue W for window lists.
Create a window for each edge opposite to s and store them into the
  corresponding window lists.
Compute the geodesic distance for each vertex adjacent to s and push
  the vertices into Q.
Update the wavefront wf and its swept area R using the one-ring
  triangles of s.
while Q ≠ ∅ do
    Pop a vertex v_i from Q.
    if v_i is a saddle vertex then
        Create a window for each edge opposite to v_i and store the
          windows into the corresponding window lists.
        Update the geodesic distances of vertices adjacent to v_i and
          push the updated vertices into Q.
    Update the wavefront wf and its enclosed area R using the one-ring
      triangles of v_i.
    Push the window lists into W if they are inside the swept area R
      and adjacent or opposite to v_i.
    while W ≠ ∅ do
        Pop a window list wl_i from W.
        Prune the redundant windows of wl_i (Rule 2).
        Propagate wl_i (Rule 1).
        Push updated vertex into Q.
        Merge newly-generated window lists (Rule 3).
        Push the newly-generated window lists into W if they are inside
          the swept area R.
```

4.2 Parallel VTP

4.2.1 Overview

The original VTP propagates geodesic distances across triangles around the vertex with the highest priority at a time, organizes windows located on the same half-edge into a window list, and propagates one window list at a time. PVTP aims at processing multiple vertices in each iteration and propagating multiple window lists simultaneously. There are main technical challenges including (a) dealing with data conflicts during concurrent propagation across triangles; and (b) replacing the priority queue by a low-cost alternative without compromising the wavefront quality too much. We use a delayed-writing strategy to avoid data conflicts and borrow the bucket data structure, which is used in the sequential FWP algorithm [12], to address the second challenge.



(a) (b)

Figure 4.4: Data structures of PVTP. (a) Buckets $\mathcal{B}$ store the soon-to-be propagated vertices. (b) Dual arrays $\mathcal{A}_{\text{from}}$ and $\mathcal{A}_{\text{to}}$ control concurrent window list propagation alternately.

We list the core data structures to facilitate the description of our algorithm:

- A **window list** wl, as achieved in the original VTP, to organize the windows on a half-edge. See Figure 4.2(a).
- A list of **buckets** $\mathcal{B}$ to maintain the priority of soon-to-be propagated vertices. See Figure 4.4(a).

- A pair of **arrays** $\mathcal{A}_{\text{from}}$ and $\mathcal{A}_{\text{to}}$ to facilitate parallel propagation of window lists. See Figure 4.4(b).
- A **buffer** buf to temporarily store the generated window lists and updated vertices. This delayed writing strategy can avoid data conflicts.

The pseudo code of PVTP is shown in Algorithm 4. Given a source vertex s, we first create a window for each half-edge opposite to s and update the geodesic distances of s's neighboring vertices. We then insert the newly generated windows into the corresponding window lists and push the updated vertices into buckets $\mathcal{B}$. Denote by wf the wavefront and R the area it encloses. At each iteration, PVTP selects top k_v vertices from buckets $\mathcal{B}$ and extends the wavefront by a range of one-ring: (a) selecting K window lists around the k_v vertices and storing them in $\mathcal{A}_{\text{from}}$; (b) propagating the selected K window lists in a parallel manner, and saving their child window lists and the swept vertices[1] into buffer buf; (c) handling the data in buf which includes merging window lists in buf by Rule 3, updating the distances of the recently-swept vertices, and pushing the merged window lists into A_{to} and the updated vertices into buckets $\mathcal{B}$; and (d) swapping $\mathcal{A}_{\text{from}}$ and $\mathcal{A}_{\text{to}}$. If array $\mathcal{A}_{\text{from}}$ is not empty, go to (b); otherwise go to (a). The algorithm terminates when the buckets $\mathcal{B}$ become empty - at this moment, the geodesic distances to all destination vertices have been computed. Figure 4.5 visualizes a typical PVTP iteration on the Bunny model with 72K vertices.

4.2.2 Buckets for Maintaining Priorities of Vertices

In contrast to the original VTP algorithm, PVTP handles multiple vertices at a time. Since the quality of the wavefront highly depends on the order of vertex processing, we use a bucket data structure [12] to maintain the

[1]The swept vertices include both the vertices inside the wavefront and the ones on the wavefront.

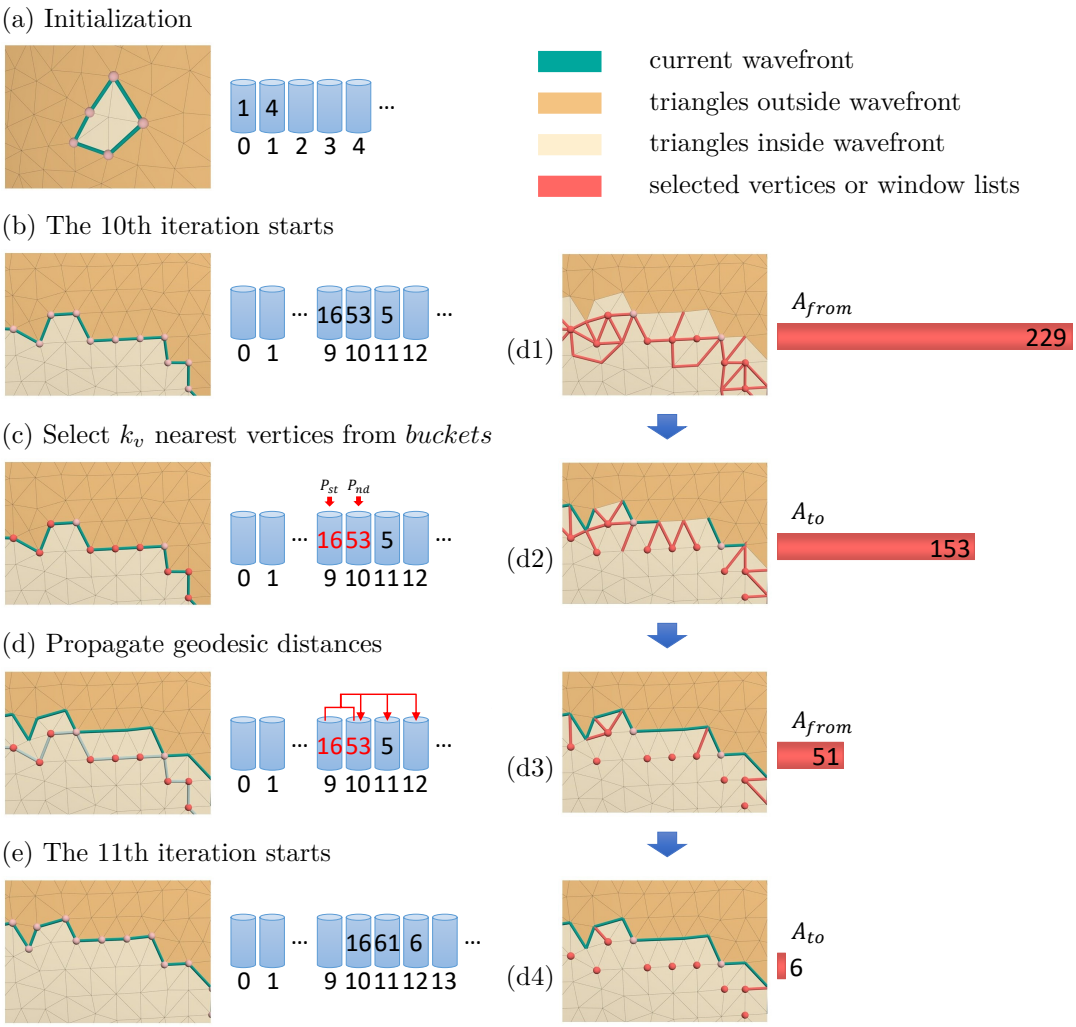


Figure 4.5: A typical iteration of PVTP. The group of blue cylinders denotes the buckets $\mathcal{B}$ and the red horizontal bars denote the dual arrays $\mathcal{A}_{\text{from}}$ and $\mathcal{A}_{\text{to}}$ to store the window lists alternately. (a) Create a window for each edge opposite to the source s and store the windows into their corresponding window lists. Then the updated vertices are pushed into the buckets $\mathcal{B}$. Green segments describe the current wavefront. (b) At the beginning of 10th iteration, all to-be-propagated vertices are stored in the buckets $\mathcal{B}$ according to their geodesic distances. (c) Select k_v-nearest vertices from buckets $\mathcal{B}$ so that K window lists can be handled at the same time; See Section 4.2.3 for the choices of k_v and K. (d) Propagating and merging window lists: (d1) 229 window lists are selected and stored into one of the dual arrays $\mathcal{A}_{\text{from}}$; (d2) After parallel propagation, there are 153 updated window lists inside the wavefront and they are pushed into the other array $\mathcal{A}_{\text{to}}$ for the next parallel propagation; (d3,d4) Operate the data alternatively between $\mathcal{A}_{\text{from}}$ and $\mathcal{A}_{\text{to}}$ until both the dual arrays are empty. At the moment, the new wavefront are reshaped as the green segments. During performing (d1-d4), the vertices with updated geodesic distances would be pushed into the buckets $\mathcal{B}$. (e) The 11th iteration.

Algorithm 4: Parallel VTP

Input: A triangle mesh $M = (V, E, F)$, source vertex $s \in V$, the number of threads T

Output: Geodesic distance $d(v_i)$ for each vertex v_i

```
d(s) ← 0 and d(v_i) ← ∞, ∀i ≠ s.
Create buckets B for vertices.
Create dual arrays A_from and A_to for window lists.
Create windows for each edge opposite to s and store them into the corresponding window lists.
Compute the geodesic distance for each vertex adjacent to s and push the vertices into B.
Update the wavefront wf and its enclosed area R using the one-ring triangles of s.
while B ≠ ∅ do
    Select k_v vertices from B.
    Update the wavefront wf and its enclosed area R using the one-ring triangles of the k_v selected vertices.
    Push K window lists into A_from if they are inside R and are adjacent or opposite to the k_v selected vertices.
    while A_from ≠ ∅ do
        Parallel propagate window lists in T threads and save the generated window lists and updated vertices in buf.
        Merge newly-generated window lists and update vertices' distance.
        Push the newly-generated window lists into A_to if they are inside the swept area R.
        Push updated vertices into B.
        Swap A_from and A_to.
```

priority in which the vertices are processed. Let L be the maximum geodesic distance, which shall be estimated later. We create $\lceil L\tau/h \rceil$ buckets, where h is the average edge length and τ (≥ 1) is the anisotropy degree [92], which measures how far the input triangulation is from isotropic. A complete isotropic triangular mesh has $\tau = 1$. The i-th bucket records the distance ranging from ih/τ to $(i+1)h/\tau$. When the distance of a vertex v_i is updated during the wavefront propagation, we push the vertex's id into the $\lfloor d_i\tau/h \rfloor$-th bucket, where d_i is the updated distance of v_i.

If the wavefront is close to a geodesic isocontour, it has good quality so that many vertices can be processed simultaneously without producing too many redundant windows. On the other hand, if the vertices on the wavefront have distances with large variations, only a few vertices (i.e., the ones with shortest distances to the source) can be processed. Therefore, the throughput of parallel window propagation depends closely on the quality of wavefronts.

Let $P_{\text{st}}^{(i)}$ and $P_{\text{nd}}^{(i)}$ be the distances of the first and last buckets where vertices are popped from in the current iteration. Denote by $c_{\text{large}}^{(i)}$ (resp. $c_{\text{small}}^{(i)}$) the number of updated vertices whose distances are greater (resp. less) than $P_{\text{nd}}^{(i)}$. Denote by $k_{\text{ad}}^{(i)}$ the number of vertices popped from the buckets $\mathcal{B}$ in the i-th iteration. Then in the $(i+1)$-th iteration, $k_{\text{ad}}^{(i+1)}$ is adaptively updated as

$$k_{\text{ad}}^{(i+1)} = k_{\text{ad}}^{(i)} + c_{\text{large}}^{(i)} - c_{\text{small}}^{(i)}. \tag{4.4}$$

Remark. Priority queue can guarantee the event with the highest priority is handled first, so it can minimize the total number of generated windows. However, inserting or deleting an element from a priority queue takes $O(\log n)$ time, which is expensive. In contrast, maintaining a bucket data structure takes only $O(1)$ time. Experimental results show that PVTP produces only slightly more windows than the original VTP algorithm, which justifies the effectiveness of the bucket data structure and the adaptive popping strategy.

4.2.3 K-Window-List Selection

Window list selection is a trivial operation in the original VTP algorithm since it considers window lists sequentially. However, PVTP is different since it propagates K window lists simultaneously. We observe that choosing a proper number of window lists for propagation is critical to the efficiency of parallelization. Our goal is to select as many window lists as possible

to get better parallelization efficiency, and meanwhile not downgrade the wavefront quality too much. We adopt the following strategy to determine the value of K.

Given the $k_{ad}^{(i)}$ vertices computed above, we count the number of window lists around them, which is denoted by $K_{\text{ad}}^{(i)}$. Then, we determine the number of window lists to be propagated $K^{(i)}$ by

$$K^{(i)} = \min(K_{\text{ad}}^{(i)}, 100T), \tag{4.5}$$

where T is the number of threads. We set an empirical threshold $100T$ to balance the wavefront quality and throughput of parallel propagation. Finally, we compute $k_v^{(i)}$ the number of actual popped vertices from buckets $\mathcal{B}$ which corresponds to the $K^{(i)}$ window lists.

The selected $K^{(i)}$ window lists are stored in array $\mathcal{A}_{\text{from}}$. The red segments in Figure 4.5(d1) denote the selected window lists. It is worth noting that if we encounter a saddle vertex that is popped from the buckets, we create windows on the one-ring opposite edges and add them into the corresponding window lists, which is due to the fact that a saddle vertex may serve as a pseudo-source vertex.

4.2.4 Parallel Window List Propagation

After selecting K window lists, PVTP propagates window lists in parallel with the aid of dual arrays $\mathcal{A}_{\text{from}}$ and $\mathcal{A}_{\text{to}}$ as shown in Figure 4.5(d1-d4) and Algorithm 4 (Line 12-16).

For each window list wl in $\mathcal{A}_{\text{from}}$, we inherit the window pruning strategies and propagation rules of the original VTP algorithm that are summarized in Section 3. During the propagation, if there is a vertex v gets a smaller distance, we push it to the buckets. Let wl be a window list. If

the newly-generated window lists wl_{left} and wl_{right}, the next generation of wl, are located inside the newly wavefront as the red segments shown in Figure 4.5(d2), we store wl_{left} and wl_{right} into $\mathcal{A}_{\text{to}}$. Considering that there may be possible data conflicts during the parallel window list propagation (different window lists may generate new window lists on the same half-edge; the distance of a vertex may be updated from different window lists), we utilize a buffer buf to store the conflicts temporarily and process the events in delay. After pushing the merged window lists into $\mathcal{A}_{\text{to}}$, we swap $\mathcal{A}_{\text{from}}$ and $\mathcal{A}_{\text{to}}$. In this way, we concurrently propagate window lists alternately with the help of $\mathcal{A}_{\text{from}}$ and $\mathcal{A}_{\text{to}}$ untill both of them become empty.

4.2.5 Correctness

A window-propagation-based algorithm is correct if it does not discard any useful windows. The correctness is independent of the window processing order. The central idea of all algorithms in the family is to effectively organize windows so that they can be propagated in a roughly near-to-far order and prune as many redundant windows as possible. The algorithms differ in the window organization and propagation strategies, and pruning rules. Since PVTP adopts the same pruning rules as VTP, it keeps all the useful windows and computes exact geodesic distances when the algorithm terminates.

4.2.6 Complexity Analysis

Given a manifold triangle mesh with n vertices, there are $O(n)$ half-edges and faces. Since a half-edge contains at most $O(n)$ windows, there are $O(n^2)$ windows generated [7]. As a result, parallel window list propagation takes empirical $O(n^2/T)$ time complexity, where T is the number of threads. Inserting (resp. deleting) an element to (resp. from) buckets takes $O(1)$

time. Therefore, both K-window list selection and pushing updated vertices into buckets have $O(n)$ time complexity. Window list merging in PVTP is as order-free as the original VTP (taking $O(1)$ time for each merging operation), so it also runs in $O(n)$ time. In summary, PVTP has an empirical time complexity $O(n^2/T + n) = O(n^2/T)$.

For regular models with uniform triangle faces, such as a sphere, the number of windows on different half-edge is approximately equal, the time complexity could be attained $O(n)$ when the number of threads is bigger than n. For anisotropic models with non-uniform triangle faces or extreme models, such as a very thin rod, the time complexity would be $O(n^2)$. Because this kind of models are not suitable for parallelization. In summary, the upper bound of time complexity is $O(n^2)$, and the lower bound is $O(n)$.

We adopt "buckets" structure in our parallel implementation, which could maintain a smooth wavefront as sequential VTP algorithm. Thus, PVTP has space complexity $O(n^2)$, which is the same as VTP.

4.2.7 Implementation Details

Adaptive choice of k_{ad} The number of vertices popped from the buckets $k_{\mathrm{ad}}^{(i)}$ is a variable. Obviously, if K_{ad}, the number of window lists selected for parallel propagation, is less than T, i.e. the number of threads, the parallel processing cannot bring performance gains. Herein, we require the number of selected vertices popped from the buckets, k_{ad}, must be larger than T. In this case, the number of selected window lists for parallel processing must be also larger than T.

Updated vertices inside wavefront For the original VTP algorithm, if an updated vertex v is a saddle vertex located in the area enclosed by the wavefront, new windows are created on each one-ring edge opposite to v and

these corresponding window lists have to be propagated until they arrive at the wavefront. In fact, for an anisotropic model with a large amount of obtuse triangles, it is quite often that the distance of a vertex v is updated multiple times. Note that the buckets cannot guarantee that the vertices are processed in a strictly ascending order of the distances. Therefore, in our implementation, the buckets contain two types of vertices: vertices on the wavefront and saddle vertices inside the swept region.

4.3 Approximate VTP

4.3.1 Motivation

Accuracy and speed are a pair of indicators for evaluating a geodesic algorithm. In fact, it is necessary to make a balance between accuracy and speed in practice. In this section, we aim at developing a fast and accuracy-controllable approximate discrete geodesic algorithm that does not require pre-computation.

Imagine that C is an isoline of the geodesic distance field. Suppose that the distances inside the area enclosed by C has been fixed. Now we take sufficiently many samples on C and continue propagating the distances. It is easy to show that the final distance field is almost identical to the real distance field. The observation motivates us to replace windows by vertices on the wavefront when the propagation goes across an appropriate distance gap.

Let L be the farthest geodesic distance from the source to all destinations, and D be the prescribed distance gap. Then the number of steps is $\lfloor \frac{L}{D} \rfloor$. Once the minimum distance reported by wavefront climbs to a new step, we clear all the windows and use the vertices on the wavefront as the new sources to continue the distance propagation. In this way, we can trade

accuracy for speed.

4.3.2 Key Ideas

Algorithm 5: Approximate VTP

Input: A triangle mesh $M = (V, E, F)$, source vertex $s \in V$, the speed and accuracy control parameter λ.

Output: The geodesic distance d_i for each vertex v_i.

```
1  Initialize as Algorithm 3: Line 1-6;
2  Compute the average edge length h.
3  R_num ← 0.
4  while Q is not empty do
5      v_i = Q.pop().
6      if d_i > (R_num + 1)λh then
           /* N: # of vertices on the current wavefront      */
7          for j ← 0 to N do
8              if d_j > (R_num + 1)λh + λ_g h then
9                  Preserve the windows on the window lists adjacent to v_j.
10             else
11                 Delete the windows on the window lists adjacent to v_j.
12         Mark all vertices on the wavefront as pseudo-sources and push
            them into Q.
13         R_num + +.
14     Propagate the wavefront as in Algorithm 3: Line 8-20.
```

The approximate geodesic algorithm is implemented by adding one step in the VTP algorithm as shown in Algorithm 5 and Figure 4.6. If the geodesic distance of the nearest vertex A on the current wavefront is larger than $(R_{num} + 1)\lambda h$, we delete all windows on the wavefront and take the vertices on wavefront as pseudo-sources, where λ denotes the prescribed distance gap, R_{num} denotes the number of steps that were passed, and h denotes the average edge length for input model. In order to avoid large errors for anisotropic or non-uniform triangle meshes, we set λ_g as a threshold: If the geodesic distance of a vertex on the current wavefront is larger than $(R_{num} + 1)\lambda h + \lambda_g h$, the windows on the edges adjacent to it would be

preserved. It is worth noting that AVTP can also be parallelized by the PVTP framework.

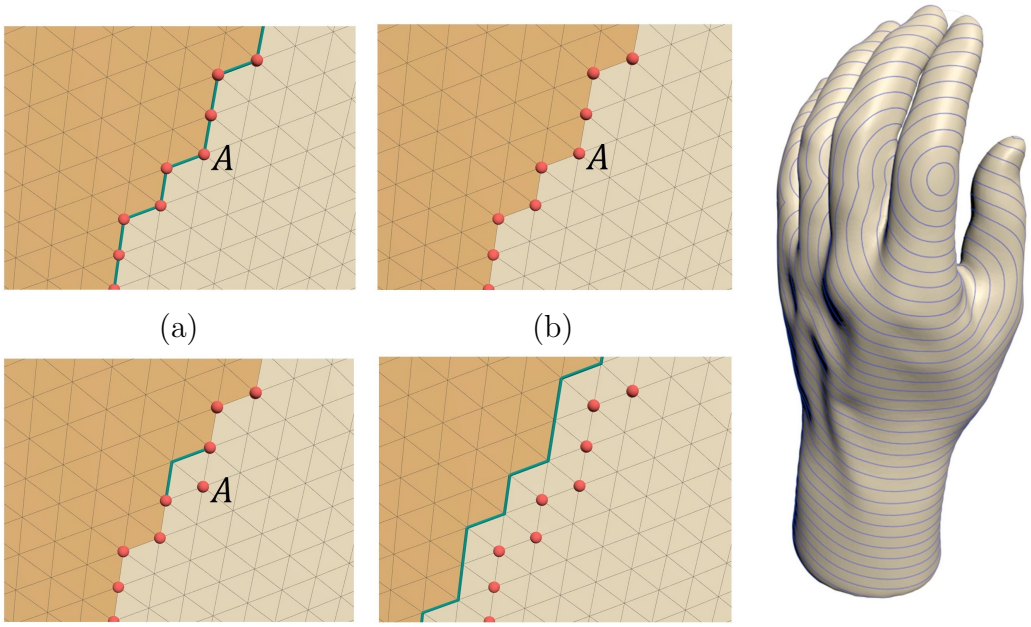


Figure 4.6: Illustration of a typical iteration of AVTP on the Hand model with $\lambda = 32$. (a) Let A be the nearest vertex to the source s. The edges containing active windows are drawn in green. (b) When $d(s, A) > \lambda h$, all windows on the wavefront are deleted. (c) Take the vertices on the current wavefront as new sources, from which windows are propagated. (d) A new wavefront is formed. (e) The isolines define the distance steps where we reset windows.

4.3.3 Complexity Analysis

Figure 4.7 gives comparison statistics on the number of windows on the Hand model. By setting $\lambda = 32$, the relative mean error of AVTP is 0.00497% while the number of windows (red bar) is significantly reduced, thus greatly improving the performance. It is worth noting that AVTP reduces to to Dijkstra's algorithm when λh is less than the minimum edge length, while becoming the exact VTP algorithm when λh is greater than the maximum geodesic distance L. The sequential AVTP algorithm has an $O(n\lambda)$ time

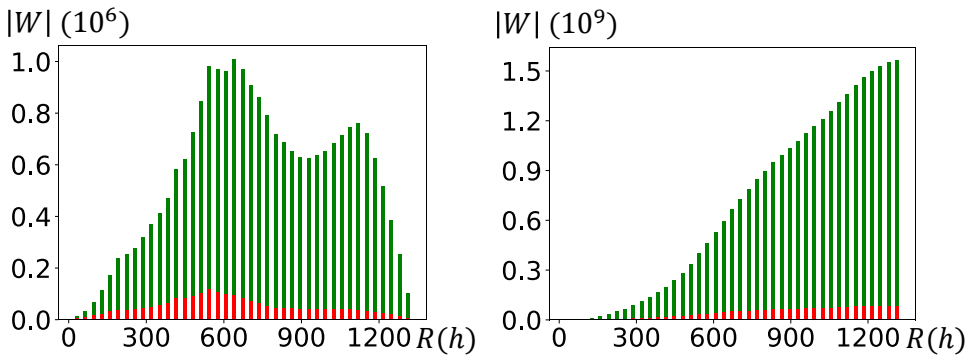


Figure 4.7: Window distribution of AVTP with $\lambda = 32$ (red) and the exact VTP (green) on an isotropic hand model with 2.3 million vertices. The horizontal axis is the wavefront radius, expressed as a multiple of the average edge length h. The vertical axes in (a) and (b) are the number of windows on the wavefront and the accumulated number of windows, respectively. AVTP generates significantly fewer windows than VTP.

complexity because there are at most $O(\lambda)$ windows on each edge during window propagation [9]. The parallel AVTP algorithm has an empirical $O(n\lambda/T)$ time complexity.

4.4 Experimental Results and Discussions

4.4.1 Parallel Algorithm

4.4.1.1 Efficiency

To test the efficiency of PVTP, we compare it with the sequential VTP on both isotropic and anisotropic meshes. Table 4.1 and Figure 4.8 report the detailed statistics. We observe that for regularly tessellated models, PVTP is 2∼3 times faster than the sequential VTP on $T = 4$ threads and 3∼5 times faster when $T = 8$. Moveover, the memory consumption and window complexity of PVTP is similar to VTP. As Table 4.1 shows, the total propagated windows of PVTP is only 2.54% more than VTP under 4 threads and 3.16% more under 8 threads. The memory cost of PVTP is

Model	Performance	Algorithms						
		VTP	PVTP		$\lambda = 8$		$\lambda = 32$	
			$T = 4$	$T = 8$	AVTP	PAVTP	AVTP	PAVTP
Armadillo	Speedup	1	2.443	3.375	2.529	4.276	1.768	3.459
$\|V\|$: 692K	Peak mem.(MB)	6.549	6.736	6.802	1.594	1.654	4.097	4.200
τ: 1.347	#windows (K)	66,223	68,770	69,279	11,129	11,639	28,486	29,433
	Error (%)	0	0	0	0.080564	0.080564	0.014819	0.014819
Bunny	Speedup	1	2.189	2.526	2.400	3.243	1.590	2.748
$\|V\|$: 72K	Peak mem.(MB)	1.519	1.594	1.643	0.466	0.492	1.195	1.238
τ: 1.258	#windows (K)	4,879	5,086	5,186	1,004	1,058	2,445	2,540
	Error (%)	0	0	0	0.086441	0.086441	0.013227	0.013227
Gargoyle	Speedup	1	2.672	3.489	3.376	5.900	2.294	4.701
$\|V\|$: 3,200K	Peak mem.(MB)	46.648	46.979	47.080	6.131	6.268	18.242	18.463
τ: 1.407	#windows (K)	569,785	577,395	578,867	60,942	63,312	154,119	158,462
	Error (%)	0	0	0	0.116999	0.116999	0.012781	0.012781
Golfball	Speedup	1	2.912	4.017	16.130	29.299	10.343	21.928
$\|V\|$: 7,864K	Peak mem.(MB)	203.751	206.407	206.645	5.225	5.388	16.778	17.250
τ: 1.731	#windows (K)	8,749,979	8,907,732	8,922,047	169,496	178,567	451,940	470,661
	Error (%)	0	0	0	0.045374	0.045374	0.003834	0.003834
Hand	Speedup	1	3.073	5.176	17.559	27.973	11.363	22.960
$\|V\|$: 2,298K	Peak mem.(MB)	75.562	75.644	75.776	2.847	2.892	9.455	9.500
τ: 1.012	#windows (K)	1,481,158	1,484,110	1,485,675	31,252	31,883	85,861	86,690
	Error (%)	0	0	0	0.059886	0.059886	0.00497	0.00497
Lucy	Speedup	1	2.396	3.318	2.697	4.281	1.855	3.444
$\|V\|$: 1,052K	Peak mem.(MB)	11.733	12.133	12.206	2.702	2.810	7.315	7.552
τ: 1.604	#windows (K)	117,416	123,903	124,886	18,977	20,123	45,661	47,861
	Error (%)	0	0	0	0.087376	0.087376	0.01201	0.01201
Rockerarm	Speedup	1	2.943	4.267	9.599	17.212	6.207	13.252
$\|V\|$: 2,571K	Peak mem.(MB)	70.588	70.917	70.949	2.822	2.909	8.657	8.849
τ: 1.515	#windows (K)	1,447,435	1,459,454	1,461,652	57,004	59,759	147,407	153,163
	Error (%)	0	0	0	0.139333	0.139333	0.011661	0.011661

Table 4.1: Performance comparison of VTP and the proposed parallel and approximate variants (PVTP, AVTP, PAVTP ($T = 4$)). τ is the anisotropy degree of the input meshes and error is the relative mean error.

Model	τ	Performance	Algorithms			
			PCH	AWP-CH	PVTP	
					$T = 4$	$T = 8$
Armadillo $\|V\|$: 692K	1.347	Peak mem.(MB)	134.783	167.343	6.736	6.802
		Total #windows	243,998,069	441,738,351	68,769,693	69,279,435
Bunny $\|V\|$: 72K	1.258	Peak mem.(MB)	50.688	25.810	1.594	1.643
		Total #windows	33,892,153	17,832,190	5,085,895	5,185,930
Fertility $\|V\|$: 30K	1.267	Peak mem.(MB)	14.106	11.344	1.369	1.405
		Total #windows	7,446,597	4,224,471	2,025,663	2,050,153
Gargoyle $\|V\|$: 247K	1.025	Peak mem.(MB)	68.822	66.331	2.882	2.922
		Total #windows	110,457,175	67,589,361	13,287,453	13,385,292
Golfball $\|V\|$: 123K	1.731	Peak mem.(MB)	35.945	49.625	2.601	2.676
		Total #windows	46,999,069	52,180,166	14,016,515	14,270,891
Hand $\|V\|$: 125K	1.017	Peak mem.(MB)	58.695	40.434	5.516	5.561
		Total #windows	66,360,823	26,118,914	16,215,356	16,300,920
Lucy $\|V\|$: 1052K	1.604	Peak mem.(MB)	186.742	228.298	12.133	12.206
		Total #windows	238,212,237	222,127,471	123,902,626	124,886,259

Table 4.2: Memory consumption of PVTP (with 4 and 8 threads), PCH and AWP-CH. Our method generates significantly fewer windows and consumes much less memory than the other parallel algorithms.

1.97% higher than VTP under 4 threads and 2.74% higher under 8 threads.

It is no wonder that PVTP does not have a big advantage over the sequential VTP if the model resolution is small since there is a large timing cost spent on the sequential computation part; see the Bunny model with 72K vertices in Table 4.1. Therefore, we subdivide the anisotropic models into at least 1M vertices and observe the improvement of the performance. We randomly choose 300 models from the Thingi10k dataset[2], including (a) 100 models whose anisotropic degree τ is less than 2, (b) 100 models whose τ is between 2 and 10, and (c) 100 models whose τ is larger than 10.

As an exact algorithm, PVTP produces the same numerical results on all models as the sequential VTP with window interval threshold 10^{-6} [3][3]. The statistics in Figure 4.8 show that our parallel framework has an advantage over the original VTP for anisotropic models with $\tau < 10$. In detail, our parallel-VTP is at least 1.5∼3 times faster than VTP under 4 threads on the 300 models. If we consider the models with $\tau < 2$, the speedup amounts to 2.5∼3 times on 94% of the models. If we allow the anisotropic factor τ to

[2] https://ten-thousand-models.appspot.com/

[3] Windows whose lengths are less than the threshold are discarded.

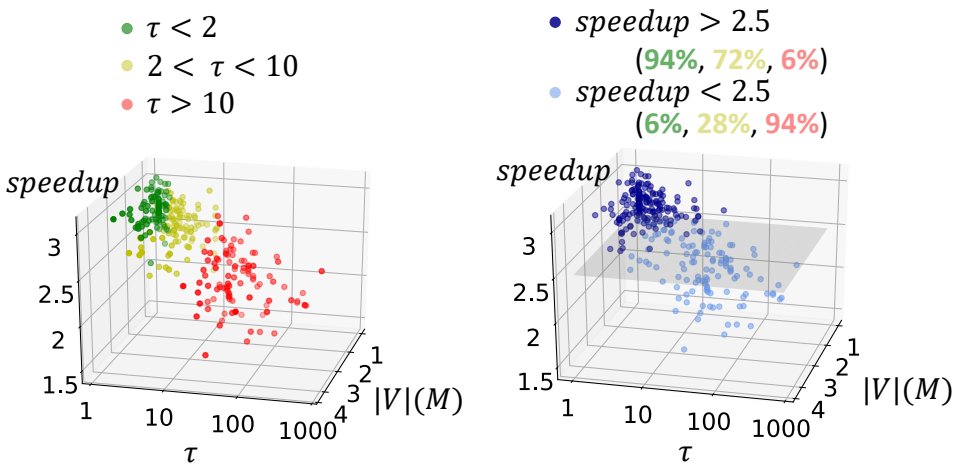


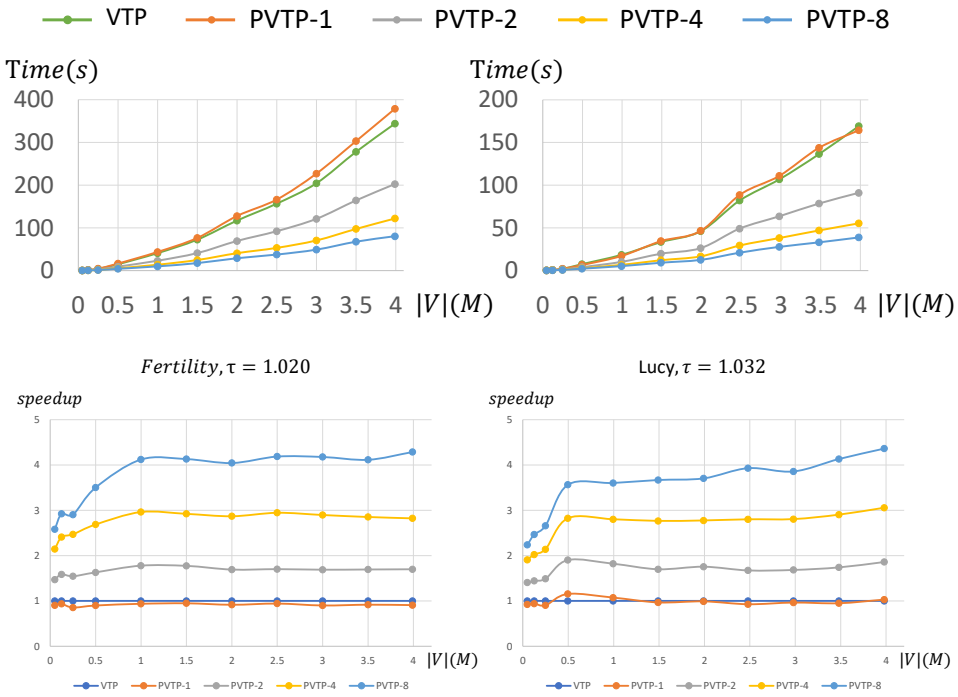
Figure 4.8: Performance statistics of PVTP (4 threads) on 300 models with varying anisotropy degree τ, which are randomly selected from the Thingi10k dataset. We plot the speedup factor in 3D diagrams. (a) Meshes with low, middle and high degree of anisotropy are drawn in green, yellow and red, respectively. (b) The horizontal plane denotes the speedup factor 2.5.

range between 2 and 10, a 2.5∼3 times speedup can also be obtained on 72% of the models. It is worth noting that the speedup effect decreases with the increasing of anisotropic degree, which is due to the fact that a high-quality wavefront (close to a isocontour) is difficult to achieve unless we rigorously follow the priorities of vertices.

4.4.1.2 Scalability

We use the Fertility and the Lucy models to observe whether the performance gain exists independent of specific shapes. As Figure 4.9 shows, the speedup factor depends on the number of vertices and is insensitive to different geometric variations.

We also compare the total propagated windows and memory consumption on commonly used models in Table 4.2. The total propagated windows of PCH (resp. AWP-CH) is 4.46 (resp. 3.42) times more than PVTP with $T = 8$ on average. The memory cost of PCH (resp. AWP-CH) is 17.65



(a) Fertility (b) Lucy

Figure 4.9: The time complexity and speedup of PVTP with $T = 1$, 2, 4 and 8 on two representative models. The horizontal axis shows the number of vertices and the vertical axis is the running time/speedup factor.

(resp. 16.52) times higher than PVTP with $T = 8$ on average.

4.4.1.3 Profiling

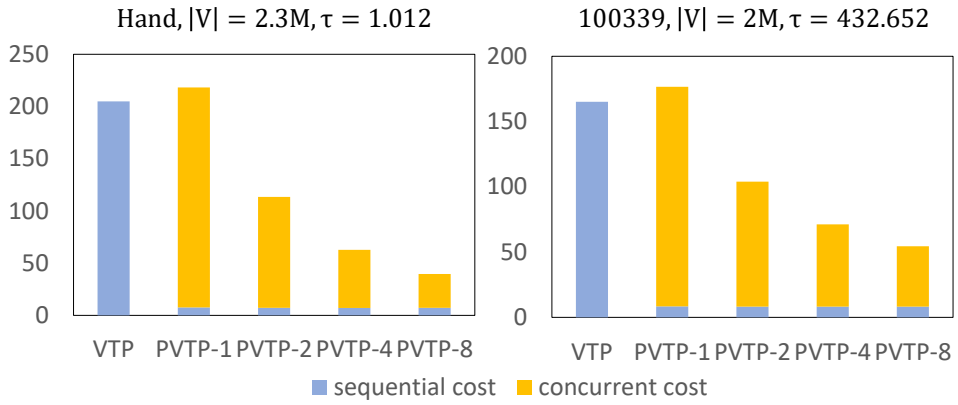


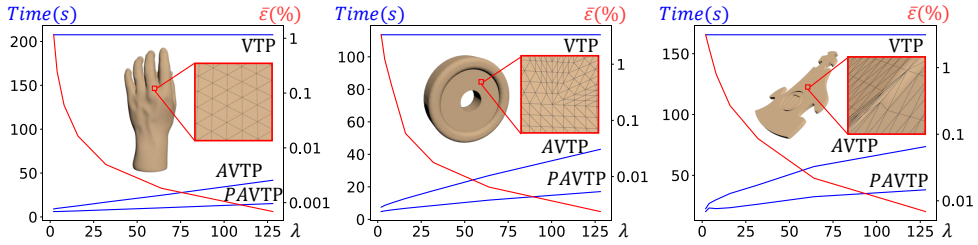
Figure 4.10: Profiling of PVTP on an isotropic model (Hand) and an anisotropic model (Model 100339 from the Thingi10k dataset). The yellow and blue parts are the running time of the parallel step and the sequential steps respectively. PVTP-i denotes the parallel-VTP algorithm with $T = i$ threads.

In order to observe the parallelization effect, we divide the total running time into two parts, one spent in sequential operations and the other spent in parallelization operations. Based on the statistics given by Figure 4.10, we can see that the timing cost of sequential steps (blue part of histogram) under one thread is lower than 5%. The majority of the operations are executed in a parallel style, which accounts for the fact that the performance improvement of PVTP against VTP correlates with the number of threads in CPU.

4.4.2 Approximate Algorithm

4.4.2.1 Performance and Accuracy

We test our AVTP on regularly tessellated models as shown in Table 4.1 and anisotropic models from Thingi10k. Recall that we use a single parameter λ to make the balance between performance and accuracy. Figure 4.11



$|V| = 2.3\text{M}, \tau = 1.012$ $|V| = 1.2\text{M}, \tau = 2.621$ $|V| = 2\text{M}, \tau = 432.652$

Figure 4.11: Performance and accuracy trade-off. We select three representative models with low, middle and high degree of anisotropy for test.

plots how λ affects the trade off between performance and accuracy. Here the three test models are with different anisotropic degrees. Based on the experimental results, we find (1) the time complexity is roughly $O(n\lambda)$, and (2) the accuracy could be improved to 2^a times by doubling λ, where $a \in [1, 2]$. The biggest advantage of AVTP/PAVTP is that they allow users to balance performance and accuracy using λ.

4.4.2.2 Comparison with Approximate-ICH

We compare AVTP with AICH [14], which is an approximate variant of the ICH algorithm [2]. Both AVTP and AICH do not require pre-computation. Using a user-specified parameter $\lambda_a \in [0, 1]$, AICH adopted an over-filtering strategy to control the accuracy. Figure 4.12 shows the distribution of relative mean error for models with varying degrees of anisotropy. We compute $x_{\bar{\varepsilon}}$ by averaging the mean errors of models with $\tau < 10$ and define $\delta \triangleq x_{\bar{\varepsilon}} \times 0.8$. We observe that 67.7% (resp. 69% and 71%) of the models have an error in the range $[x_{\bar{\varepsilon}} - \delta, x_{\bar{\varepsilon}} + \delta]$ when we set $\lambda = 8$ (resp. 32 and 128) in AVTP. For the comparison purpose, we take $\lambda_A = 0.1$ for the AICH algorithm. With the parameter setting, AICH is not so accurate as our AVTP but more timing consuming. In detail, the average error of AICH is around $x_{\bar{\varepsilon}}$ for 41.3% models and even worse for the remaining models.

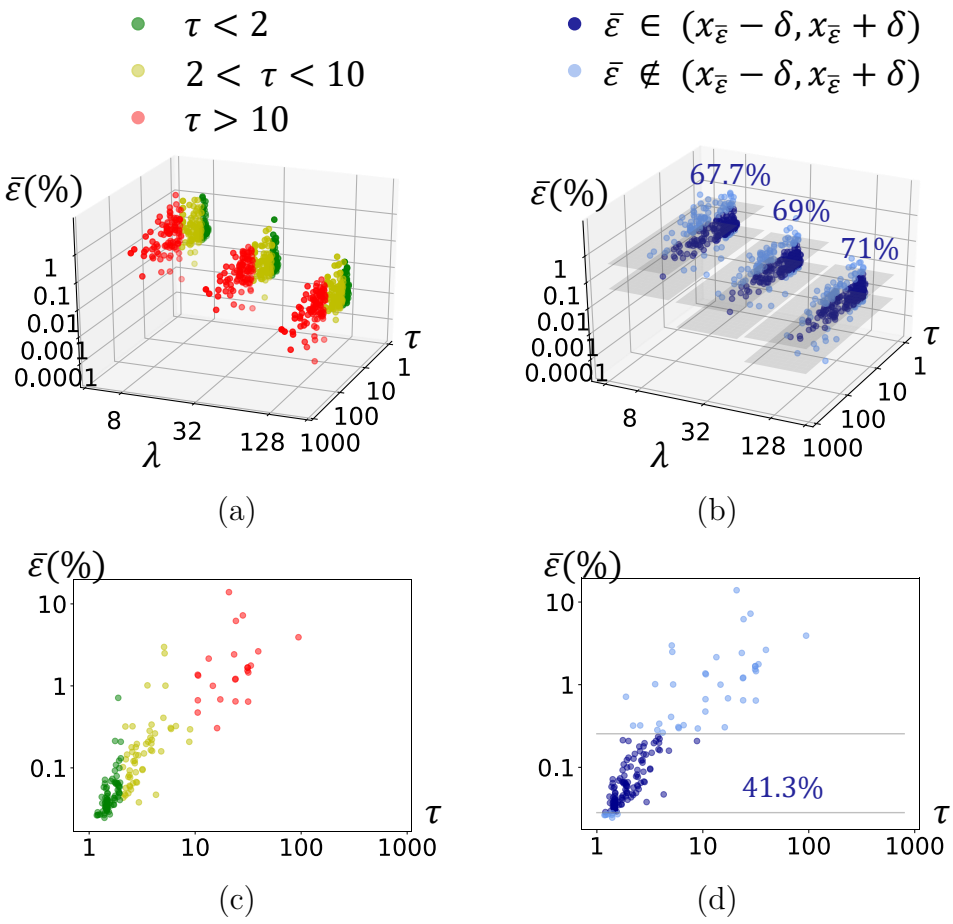


Figure 4.12: Relative mean error distribution of AVTP (row 1) and AICH (row 2). (a) Results of AVTP with $\lambda = 8, 32$, and 128, respectively. (b) Models around average value $x_{\bar{\varepsilon}}$ are marked in dark blue for AVTP. (c) Results of AICH with $\lambda_A = 0.1$. (d) Models around average value $x_{\bar{\varepsilon}}$ are marked in dark blue for AICH.

4.4.2.3 Comparison with Heat Method

In addition, we compare our AVTP with Heat Method(HM) [18] [19], the classical approximate geodesic algorithm. Heat method utilizes a partial differential equation approach to compute approximate geodesic distance from one source to all destinations. This approach doesn't exist bounded error and has a preprocessing step. Let HM attain the minimal average relative error, Table 4.3 presents the performance comparison between AVTP and HM under same precise.

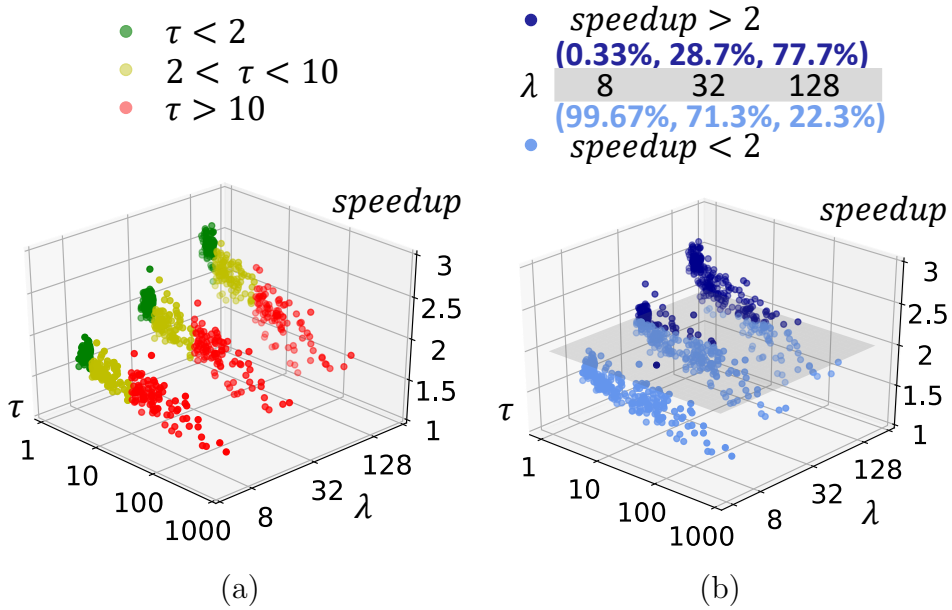
The rules of precise - parameter m are parabolas in Heat method, whereas accuracy is directly proportional to parameter λ in our AVTP. As a result, the precise of AVTP is more controllable. What's more, our AVTP is more effective than Heat method under same precise. See Table 4.3. For models with different anisotropy degree, our AVTP is much faster than Heat method($T < T_{pre} + T_{run}$).

Model	$\|V\|$(K)	τ	Heat Method				AVTP(equal error)	
			m	Mean error(%)	T_{pre}(s)	T_{run}(s)	λ	T(s)
Hand	2298	1.012	3.409	0.221750	295.497	2.559	3.994	15.048
Bunny	72	1.258	1.257	0.489849	2.690	0.060	2.842	0.315
Lucy	1052	1.604	2.904	0.570717	100.453	2.205	2.743	7.148
Golfball	7864	1.731	110.876	0.049740	904.670	25.832	7.665	83.716
1063861	2750	3.785	581.859	1.554662	424.419	3.023	0.523	17.347
108771	1231	8.959	395.848	4.840904	68.871	0.993	0.976	11.3
1036316	1309	29.812	91.671	0.789807	86.275	1.150	8.037	29.548
100343	1256	172.550	23.236	0.885456	175.182	2.243	4.358	55.191

Table 4.3: Performance comparison between AVTP and Heat method under same mean error. T_{pre} denotes the time of preprocess step and T_{run} denotes the execution time of Heat method. T denotes the time of AVTP.

4.4.3 Parallel and Approximate Algorithm

It is worth noting that the proposed parallelization and approximation techniques can be either applied separately or combined together. The previous



(a) (b)

Figure 4.13: Speedup factor of PAVTP with $T = 4$ over AVTP. (a) Speedup factor under different λ and anisotropy degree τ. (b) Models whose speedup factor is bigger than 2 are marked in dark blue.

subsections report the performance of each technique separately. In this subsection, we discuss the combination of the two techniques.

We test our parallel approximate VTP (PAVTP) for regularly tessellated models. See Table 4.1. Herein, we list the performance of PAVTP with $\lambda = 8, 32$ under 4 threads. The performance advantage of PAVTP is even more significant on models with higher resolution or bulging shapes (with a long wavefront during distance propagation). For example, PAVTP is 29 times faster than VTP on the Golfball model with $\lambda = 8$. Figure 4.11 also illustrates the performance of PAVTP under different λ, which shows that PAVTP runs faster under the same accuracy requirement.

We compare the performance of PAVTP and AVTP on anisotropic models in Figure 4.13. We observe that the speedup factor depends on the parameter λ: the larger the parameter λ, the higher the speedup. For example, 77.7% models enjoy a speedup factor greater than 2 with $\lambda = 128$. The per-

formance improvement of PAVTP also depends on the triangulation quality of the input meshes, which is similar to the observations of PVTP. See Section 4.4.1.1.

4.5 Summary

We extend the vertex-oriented triangle propagation algorithm [3] - the state-of-the-art method for exact discrete geodesics - via parallelization and approximation. To avoid conflicts in data writing, the parallel VTP algorithm proceeds with 3 steps in each iteration: K-nearest window list selection, parallel window list propagation, and vertex distance updating and window list merging. PVTP has an empirical $O(\frac{n^2}{T})$ time complexity, where T is the number of threads. Experimental results show that PVTP runs 2.5$\sim$3 times faster than the sequential VTP algorithm using 4 threads and 4$\sim$5 times faster using 8 threads on triangle meshes with 1 million vertices and fairly regular tessellations. For challenging anisotropic meshes, PVTP is still 1.5$\sim$3 times faster than VTP. We also propose an approximate variant of VTP that balances accuracy and speed by a global parameter λ. Our idea is to reset window on the wavefront when its radius is a multiple of λh, where h is the average length of mesh edges. AVTP becomes Dijkstra's algorithm when λh is less than the minimal edge length and becomes the exact VTP algorithm when λh is greater than the longest geodesic distance on the model. We prove that AVTP has $O(n\lambda)$ time complexity. It is worth noting that AVTP can also be parallelized under the same framework of PVTP. Our source code is available at https://github.com/djie-0329/PVTP.

Chapter 5

Application: Repetitive Pattern Detection on 3D Models

This chapter introduces an application scene of computing geodesics in real world. We present a new method for detecting repetitive patterns on 3D models. Given a manifold triangle mesh M and the user-specified to-be-detected pattern P, our method detects all nearby potential patterns in two main steps. First, it forms and classifies matches between pattern P and nearby potential patterns. Second, it refines the rough matches via a continuous optimization. Every possible pattern could be detected by iterating these two steps during region growing scheme. All steps require the query of geodesic distance between vertices on triangular mesh M. In contrast to the existing methods, our method does not assume the patterns are (near) regular and of similar sizes. As a result, it works for both isotropic and anisotropic patterns, non-regularly distributed patterns as well as mixed patterns.

5.1 Motivation

Near repetitive patterns are ubiquitous in man-made and natural objects. Examples include wall decoration, animals' scales and so on, as illustrated in Figure 5.1. Many of these patterns have various sizes and degrees of anisotropy, and they may be arranged non-uniformly. Discovering such patterns and their arrangement is helpful in understanding and analyzing geometric property which is desired to many applications, such as shape editing, repairing , compression, etc.

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

Figure 5.1: Near repetitive patterns are commonly found in man-made and natural objects. Some patterns are isotropic and equally spaced to each other as (a) and (b) shows, whereas others may be anisotropy, different sizes, orientations and arranged non-uniformly as (c) and (d) displays.

Although pattern detection in 2D domain has been studied extensively [66, 73,75,93], it is difficult to extend the existing 2D techniques to 3D models due to two main reasons. First, 2D images are regular structures with a natural coordinate system. Therefore, the pattern detection problem could be easily formulated by adopting matrices and solved by the powerful optimization tools. Except for some simple models(such as spheres, cylinders), regular representation is a luxury for real-world 3D models. Therefore, there is no global coordinate system on these models. Second, the 2D problem focuses on the affine-invariant patterns, whereas the 3D counterpart must deal with the scale and isometric-invariant patterns defined on highly curved geometry with non-trivial topology. Thus, detecting near repetitive patterns on 3D models is a challenging problem.

Huang et al. [89] proposed a novel method for discovering near-regular structure on 3D models. They formulated the problem as a constrained optimization with both geometric and topological aspects, where the former is to determine the locations of the patterns required and the latter is to find the connectivity relationship among the patterns. They showed that this kind of mixed discrete-continuous problem can be efficiently solved using linear programming techniques. Their method is theoretically sound and robust with respect to geometric and topological noises which are commonly found in real-world models. However, their method assumed the repeated patterns are isotropic, of similar sizes, and regularly arranged on the surface. As a result, it does not work for patterns with high degree of anisotropy or non-uniformly distributed patterns with varying sizes. Herein, we present a new method for detecting repetitive patterns on 3D models. Observing that repetitive patterns have consistent matching between the corresponding feature points, we tackle the above-mentioned challenges by a robust and efficient local feature matching algorithm.

Given a manifold triangle mesh M and the user-specified pattern P, our method then solves the pattern detection problem in two steps as follows. First, with the extracted signature from P, it maps each sample point inside of pattern P to candidate points around P. And then, it clusters these formed matches to find out respective correspondence of detected patterns. Second, it refines the optimal correspondence around the pattern via a continuous energy optimization, which not only matches the corresponding features between two patches but also considers the feature consistency within each patch. Finally, the arrangement graph for all detected patterns are able to built by computing the Voronoi diagram of the detected patterns. Our method does not assume the patterns are (near) regular and of similar sizes, thereby it can work for both isotropic, anisotropic and adaptive patterns.

5.2 Method

5.2.1 Overview

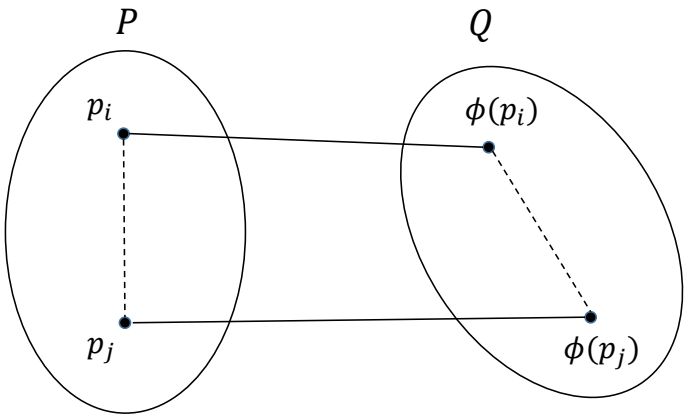


Figure 5.2: The points p_i, p_j in pattern P and their images $\phi(p_i), \phi(p_j)$ in to-be-discovered pattern Q. $g(p_i, p_j) \approx g(\phi(p_i), \phi(p_j))$, where $g(p_i, p_j)$ denotes the geodesic distance between points p_i and p_j.

Our method is motivated by the following observation. Denote by $Q \subset M$ a patch on mesh M and $P \subset M$ a known pattern. When Q is an instance of pattern P, there exists a (near) bijective mapping $\phi : P \to Q$ such that the salient features in P are mapped to the corresponding features in Q. Such a mapping ϕ should also preserve the local isometry. See Figure 5.2. Given two points $p_i, p_j \in P$ and their images $\phi(p_i), \phi(p_j) \in Q$, the geodesic distance $g(p_i, p_j)$ between p_i and p_j is approximately equal to $g(\phi(p_i), \phi(p_j))$.

The user-specified pattern covers a set of pre-defined samples, and therefore our method's target is finding potential patterns by constructing discrete mapping for samples. Obviously, a brutal force search is not practical due to the huge computational cost. Our solution is inspired by a spectral technique [94] [95] which can efficiently solve the shape correspondence problem [36]. Our framework utilizes two steps to detect patterns around user-specified patch.

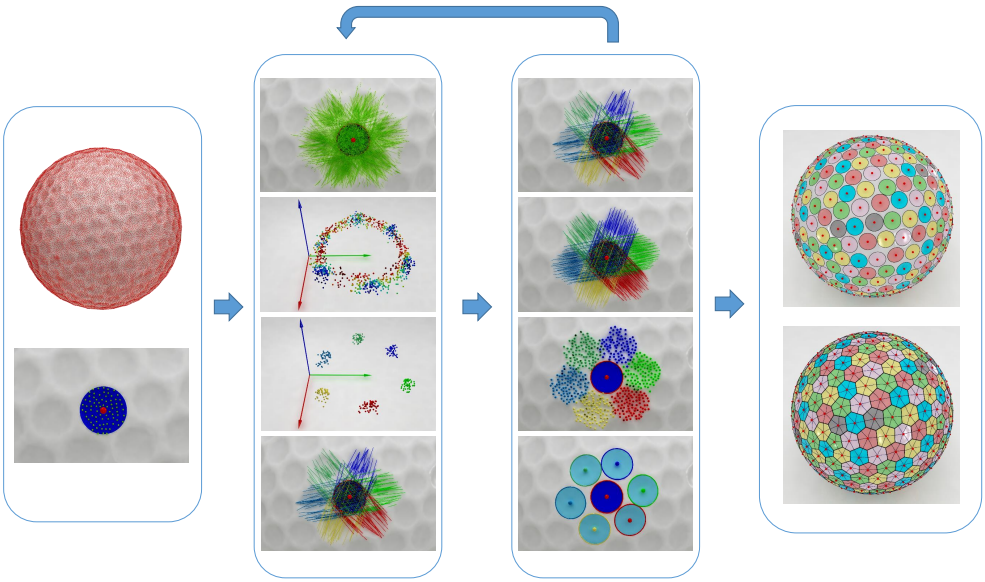


Figure 5.3: Pipeline of our method on a golfball model. (a) Input model and user-specified patch; (b) Forming and Classifying candidate matches; (c) Energy optimization; (d) Arrangement for all patterns.

First, it finds the sample points whose signature are similar to the sample points on fixed pattern so that constructing candidate matches. Then, it transfers the matches into the match-sample-points and finds the clusters among these match-sample-points to classify candidate matches. All possible patterns would be detected with the aid of match-sample-points clustering method and region growing scheme. It should be noted that the distribution of these possible patches may deviate from real patterns because of the irregular distribution of patterns, the noisy of triangle mesh and so on. In order to address this problem, we bring in energy optimization for second step. After detecting all nearby potential patterns, it builds the arrangement. Center points of all patterns are considered as the seeds to compute the geodesic Voronoi diagram [30] and its dual graph Delaunay triangulation network. Figure 5.3 visualizes the pipeline of our method on a golfball model.

5.2.2 Preprocessing: Sampling and Feature Extraction

There are two preprocessing steps before executing our method on input model.

5.2.2.1 Sampling

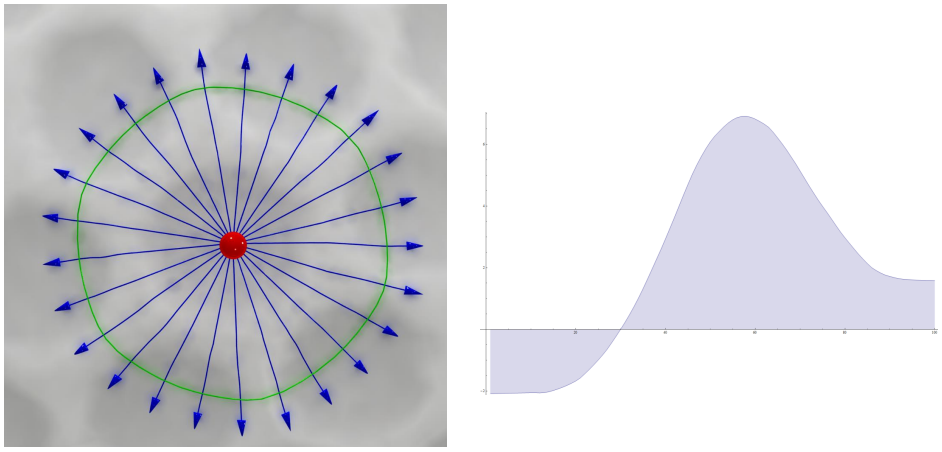
It's highly time-consuming to cluster matches linking each vertices for adjacent patches. Therefore, we pre-compute sample points for input model. Herein, we adopt Farthest Point Sampling(FPS) [38] algorithm with the help of our Parallel-VTP geodesic algorithm. FPS takes the farthest vertex to the already selected samples as a new sample point. Let V be the set of vertices and S be the set of sample points on the mesh respectively. Assume there are k chosen sample points $p_0, p_1, ..., p_k \in S$, we then fix p_{k+1} as

$$p_{k+1} = \arg\max_{p \in V} \min_{0 \leq i \leq k} g(p_i, p),$$

where $g(p_i, p)$ denotes the geodesic distance between p_i and p.

5.2.2.2 Feature Extraction

During matching samples with common saliency in our first step, we assume that each sample point has a pre-computed signature. Therefore, there is a preprocessing step of extracting feature for each sample on input triangle mesh. Our method refers to [96]. For each point, we first compute a geodesic disk and sample K geodesic paths. See Figure 5.4(a). Then, for each sampled geodesic path, we averagely sample N points and utilize the principle curvatures of these points to form a feature histogram. See Figure 5.4(b). We adopt the average and variance histogram of these K geodesic radiuses on behalf of each sample's signature.



(a) (b)

Figure 5.4: Feature extraction for each sample point.(a) Sampled radius line on geodesic disk;(b) Feature descriptor of one sampled radius.

Given a sample point p, we use $f(p,[0,r])$ represents the signature based on above geodesic disk whose geodesic radius is r. Note that $f(p,[0,r_1])$ and $f(p,[0,r_2])$ have same signature histogram in interval $[0,r_1]$ when $r_1 < r_2$. In general, we choose larger radius to extract sample's signature so as to avoid repeatedly pre-computing signatures in the situation of different scale-level patterns on same model. Without loss of generality, the signature of point p is represented by $f(p)$.

5.2.3 Match Clustering

Between the input pattern P and its nearby potential pattern Q, there exist a continuous bijective mapping $\phi : P \to Q$. Herein, we discretize the mapping by matching sample points whose signatures are similar. Given two samples $p, q \in S$, we call they are matched if their signatures are approximately equal $f(p) \approx f(q)$. Denote by $\widehat{pq}$ the match. After finding local matches around the input pattern P, we get a set of candidate matches. Then, we adopt the spectral clustering method [95] to classify these candidate matches into the correspondences for different potential patterns.

Algorithm 6: Find Clusters of candidate matches in embedding space $\mathcal{R}^3$.

Input: Candidate matches set $\mathcal{M}$, the number of sample points on input pattern k.

Output: Clusters $\mathcal{C} = \{C_i, i = 1, 2, 3..\}$ in $\mathcal{M}$.

```
Build similarity matrix W.
Compute the first three eigenvectors u_1, u_2, u_3.
Find spectral embedding coordinates for each match m_i in M.
/* N_M : # of matches in M.                                   */
for i ← 0 to N_M do
    Compute the weight w_i for match-sample-point m_i.
Sort ∀m_i ∈ M by their weight w_i.
for i ← 0 to N_M do
    Use mean shift algorithm to move m_i.
    if the nearest k match-sample-points of m_i don't belong to any
     cluster then
        Create new cluster C_i for e_i and its k neighbors.
```

Let $\mathcal{M} = \{m_i = \widehat{p_i q_i} | p_i \in S \cap P, q_i \notin S \cap P\}$ be the set of all candidate matches, where P denotes fixed pattern, S denotes the set of sample points. The similarity matrix W is constructed as

$$W_{ij} = \begin{cases} \frac{c_{ij} - c_0}{1 - c_0}, & \text{if } c_{ij} > c_0, i \neq j \\ 0, & \text{otherwise.} \end{cases} \tag{5.1}$$

Herein, we define the similarity value between two matches as

$$c_{ij} = \min\left(\frac{g(p_i, p_j)}{g(q_i, q_j)}, \frac{g(q_i, q_j)}{g(p_i, p_j)}\right), \tag{5.2}$$

and c_0 denotes the cutoff value. Each match $m_i \in \mathcal{M}$ could be transferred into a match-sample-point in the embedding space $\mathcal{R}^3$ by computing the first three eigenvectors of matrix W. See Figure 5.3(b).

Since the number of potential patterns is unknown, we adopt mean shift clustering [97,98] algorithm to cluster points in embedding space. See Algo-

Algorithm 7: Search all neighboring patterns by region growing method.

Input: User-specified pattern P, model M
Output: The set T of all detected patterns

```
Insert pattern P into set T, T = {P}.
Insert pattern P into set NewPatchSet, NewPatchSet = {P}.
while NewPatchSet is not empty do
    Pop a patch P_i from NewPatchSet.
    Compute candidate matches set 𝔐 for patch P.
    Compute clusters set C by Algorithm 6.
    for ∀C_i ∈ C do
        Optimize the group matches.
        if Q is a new detected pattern then
            Insert Q into NewPatchSet.
            Insert Q into set T.
```

rithm 6. In practice, it is more efficient to bring in weight function in mean shift clustering, since the matches in the local correspondence have higher similarity. We define the weight function of match-sample-point m as

$$Weight(m) = \sum_{i \in Nk(m)} \sum_{j \in Nk(m)} c_{ij}, \tag{5.3}$$

where $Nk(m)$ denotes the set of k-nearest match-sample-points of m and k is the number of sample points on the initial user-specified pattern. After spectral clustering algorithm, the candidate matches are categorized to different possible patterns. Finally, we adopt region growing method to search all candidate patches. See Algorithm 7.

5.2.4 Energy Optimization

Algorithm 6 remarkably indicates which matches may be local shape correspondence between two patterns. However, the correspondences are coarse, which requires more optimizations.

We define the energy function as

$$
\begin{aligned}
F(Q) = \alpha & \sum_{\substack{p_i \in P \\ \phi(p_i) \in Q}} \|f(p_i) - f(\phi(p_i))\| \\
& + \beta \sum_{\substack{p_i \in P \\ \phi(p_i) \in Q}} \sum_{\substack{p_j \in P \\ \phi(p_j) \in Q}} (1 - c(\widehat{p_i \phi(p_i)}, \widehat{p_j \phi(p_j)})) \\
& + \gamma \|f(p_l) - f(\phi(p_l))\|, \qquad (5.4)
\end{aligned}
$$

where P represents the fixed pattern and Q represents the nearby possible pattern, $f(p_i)$ denotes the signature of point p_i, $c(m_i, m_j)$ denotes the similarity between m_i and m_j, p_l denotes the landmark point in pattern P. α, β and γ are parameters to balance the weight.

In the ideal correspondence, for each sample point p_i, $f(p_i)$ is equal to $f(\phi(p_i))$; and for any two matches m_i, m_j, the similarity c is 1. Thereby, we minimize the energy value for group matches during region growing. There exist continuous iterations in the process of region growing. To avoid the increasing of deviation, we amplify the *landmark* point in each patch. If the patch is isotropic, we choose the center as its landmark point; if the patch is anisotropic, we choose the midpoint of median axis. Finally, we attain a group match $\mathcal{M}$ which is closer to the ideal discrete local shape correspondence by minimizing 5.4. See Figure 5.3(c).

5.3 Experimental Results & Comparison

5.3.1 Performance Analysis

The consumption of our method focuses on its two main steps. See Table 5.1. Note that the overall consumed time is linearly related to the number of patterns which could be detected as $T(overall) = (t(clustering) +$

Figure 5.5: Repetitive pattern detection results on asian dragon model.

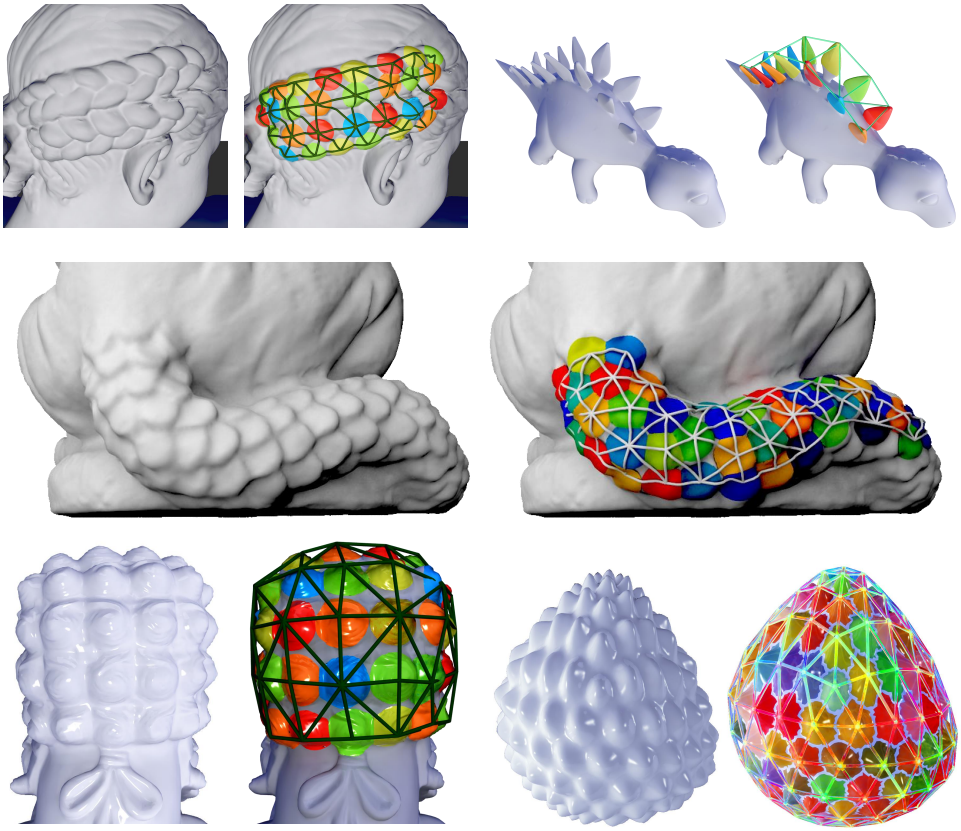


Figure 5.6: Repetitive pattern detection results on different models.

$t(optimization)) * N_p$, where N_p is the number of patterns.

Model	# vertices	$T_c(s)$	$T_o(s)$	$T(s)$
Ice dragon neck	50K	1	15	16
Ice dragon body	50K	1	47	48
Dragon	100K	5	20	25
Gargoyle tail	350K	3	30	33
Octopus	106K	4	57	61

Table 5.1: Runtime for detecting repetitive patterns on different models. T_c denotes the runtime of spectral clustering, T_o denotes the runtime of optimization.

5.3.2 Anisotropic Patterns

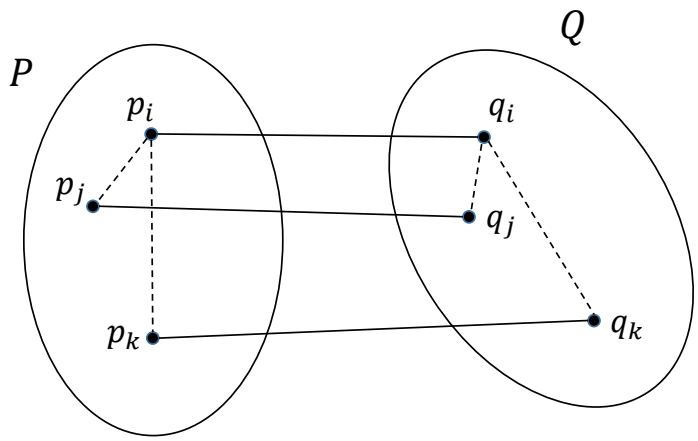


Figure 5.7: Schematic diagram for anisotropic pattern. Point p_i, p_j, p_k in pattern P and their mapping q_i, q_j, q_k in pattern Q. Local isometry invariant as $g(p_i, p_j) \approx g(q_i, q_j), g(p_i, p_k) \approx g(q_i, q_k)$.

Figure 5.5 and 5.6 exhibits the experiments results on different models. Apart from isotropic patterns, our method also be viable for anisotropic patterns. See Figure 5.7. The matches between samples on two anisotropic patches also preserve local isometry. There is no difference between isotropic and anisotropic patterns by our scheme. Figure 5.8 displays the detection result on a model with anisotropic patterns.

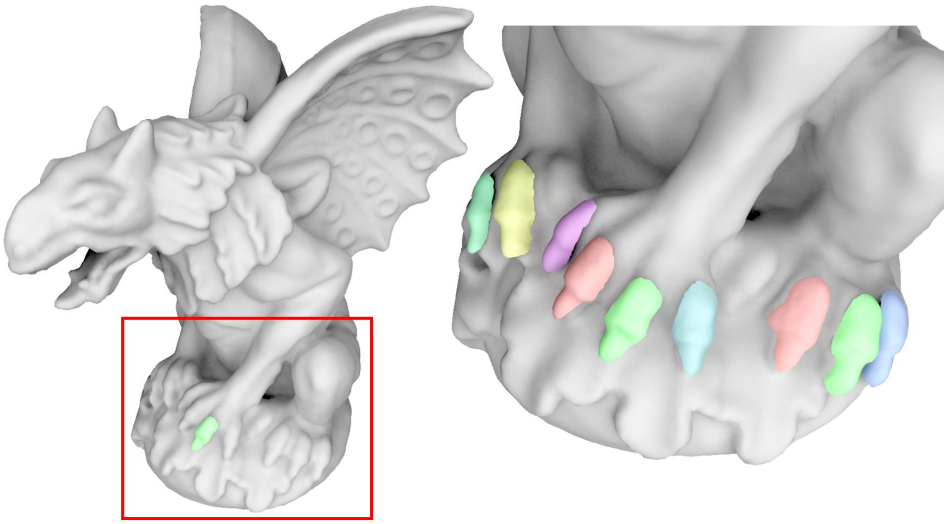


Figure 5.8: The result of our method for anisotropic patterns.

5.3.3 Adaptive Patterns

For the adaptive case, our method requires user to draw the stroke so as to confirm the ratio value of density λ. Then, feature extraction method of each point and the similarity computation between two matches need to be adjusted accordingly. Suppose $f(p, [0, r_p])$ denotes the signature of point p in pattern P, $f(q, [0, r_q])$ denotes the signature of point q in pattern Q, then we consider p and q are matched only if

$$\|f(p, [0, r_p]) - f(q, [0, r_q] * \lambda)\| < \epsilon, \tag{5.5}$$

where $\lambda = \frac{\rho_p}{\rho_q}$, ρ_p denotes the density of point p and ρ_q denotes the density of q respectively. See Figure 5.9.

Equally, we bring in the factor λ to Equation 5.2 as follows,

$$c_{ij} = min\{\frac{2 * g(p_i, p_j)}{g(q_i, q_j) * (\lambda_i + \lambda_j)}, \frac{g(q_i, q_j) * (\lambda_i + \lambda_j)}{2 * g(p_i, p_j)}\}. \tag{5.6}$$

Figure 5.10 illustrates the detection result of adaptive patterns.

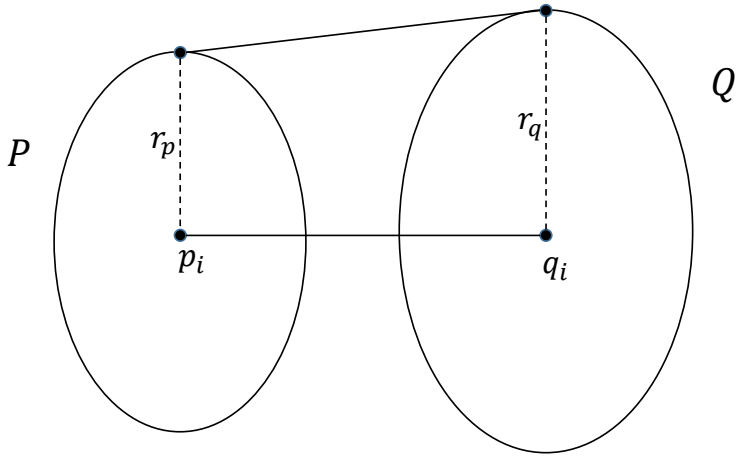


Figure 5.9: Schematic diagram of adaptive pattern. Point p_1 and p_2 in pattern P and Q respectively.

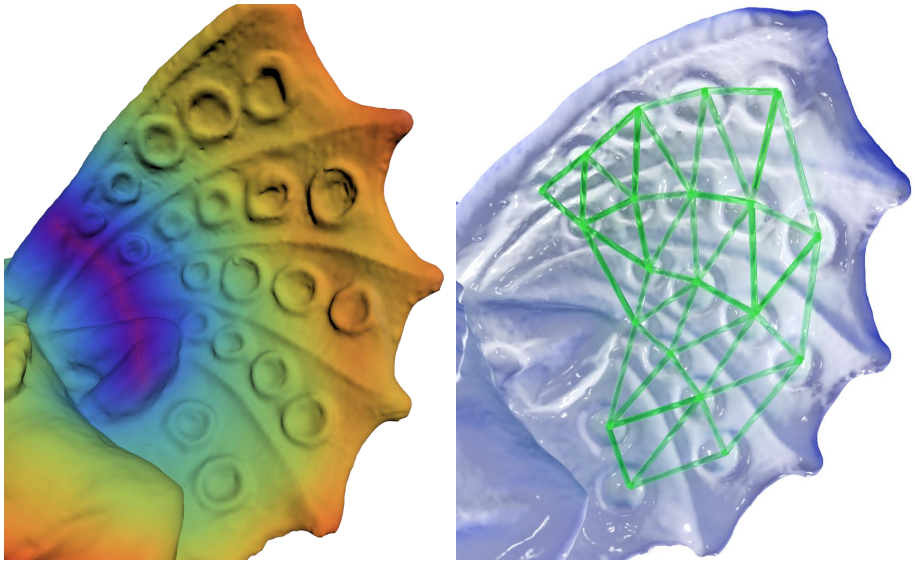


Figure 5.10: The result of our method for adaptive patterns.

5.3.4 Discussion

The proposed method with linear programming techniques in [89] imposes restrictions on the input models that their patterns are near-regular and of similar size. Our method works for both isotropic and anisotropic patterns, non-regularly distributed patterns as well as adaptive patterns with the aid of a user-interface. Moreover, as a localized approach, it does not require any global numerical solver, therefore it is efficient and scales well on large models, and can also be implemented in parallel.

5.4 Summary

We propose a novel framework for detecting repetitive patterns in this chapter. There are two main steps in our key idea: detecting potential nearby patterns by clustering match-sample-points and refining the correspondence via optimizations. Finally, all potential patterns around user-specified patch are discovered and their arrangement are constructed by computing geodesic Voronoi diagram and dual graph Delaunay triangulation. Different from existing pattern detection method [89] on 3D models, our method does not assume the potential patterns are regular. Our method works for both isotropic, anisotropic, non-regularly distributed patterns as well as adaptive patterns with a user-interface.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis, we focus on the research of developing faster discrete geodesic algorithms with the aid of parallel framework.

At first, we propose a parallel framework based on MMP algorithm. Our parallel-MMP algorithm includes five main steps: window selection, window propagation, window organization, window clipping and window pushing. The two most time consuming steps are parallelized. Our experimental results show that the speed of our parallel MMP algorithm is 1.6-1.7 times faster than FWP-MMP algorithm and 5-7 times faster than MMP algorithm based on 4 CPU threads. Our parallel-MMP algorithm is the fastest geodesic algorithm to preserve the geodesic paths from source to all destinations so far.

Then, we move to develop parallel framework based on VTP algorithm, the fastest sequential exact geodesic algorithm so far. Our parallel-VTP framework includes three main steps: K-window-list selection, parallel window list propagation, vertex distance updating and window list merging. Al-

though there is only one concurrent step, window list propagation, the time complexity of other three steps is only $O(n)$. Thus, our algorithm has an empirical $O(\frac{n^2}{T})$ time complexity, where n is the number of vertices and T is the number of threads. We evaluate PVTP on a wide range of 3D models and observe that it improves the performance of the sequential VTP by a factor of 2.5$\sim$3 on a 4-core CPU and 4$\sim$5 on an 8-core CPU for most of regular triangle meshes with over 1 million vertices. For challenging anisotropic meshes, PVTP is also 1.5$\sim$3 times faster than VTP. We also propose an approximate variant of VTP that controls accuracy and speed by a global parameter λ. The AVTP becomes Dijkstra's algorithm when λh is smaller than minimal edge and becomes the exact VTP when λh is larger than the farthest geodesic distance, where h is the average length of mesh edges. We prove that AVTP has $O(n\lambda)$ time complexity. It is worth noting that AVTP can also be parallelized under the same framework of PVTP.

At last, we present an application of discrete geodesic algorithm, that is near repetitive pattern detection on 3D models. Taking a manifold triangle mesh M and the user-specified to-be-detected pattern P as input, our algorithm then solves the pattern detection problem in two steps: detecting potential nearby patterns by clustering match-sample-points and refining the correspondence via optimizations. Finally, all potential patterns around user-specified patch are discovered and their arrangement are constructed by computing geodesic Voronoi diagram and dual graph Delaunay triangulation. Different from existing pattern detection method on 3D models, our method does not assume the potential patterns are regular. As a result, our scheme works for both isotropic and anisotropic, even non-regularly distributed patterns.

6.2 Future Work

We implemented PVTP on CPUs, which have relatively few cores comparing to GPUs. Since the bucket data structure and K-window-list selection are effective to control the total number of propagated windows, we will explore their potential on GPUs in the near future.

Though the study of discrete algorithms in this thesis focuses on the single-source-all-destinations geodesics, applying PVTP *locally* can improve the performance of constructing saddle vertex graphs [10] and discrete geodesic graphs [13, 20].

Motion planning problem in robotics domain is to find a feasible trace from the source to destinations in real world which might include polyhedron obstacles. This problem could be transferred into discrete geodesic problem on manifold as well. Therefore, developing more efficient algorithms for motion planning problem with our algorithms is also a valuable research direction.

Besides, our parallel-VTP algorithm is able to improve the performance of Geodesic Voronoi Diagram problem.

References

[1] Le Tien Hung. *Parallel Computation of Discrete Geodesics and Its Applications*. PhD thesis, Nanyang Technological University, School of Computer Science and Engineering, 2017.

[2] Shi-Qing Xin and Guo-Jin Wang. Improving chen and han's algorithm on the discrete geodesic problem. *ACM Trans. Graph.*, 28(4):104:1–104:8, September 2009.

[3] Yipeng Qin, Xiaoguang Han, Hongchuan Yu, Yizhou Yu, and Jianjun Zhang. Fast and exact discrete geodesic computation based on triangle-oriented wavefront propagation. *ACM Trans. Graph.*, 35(4):125:1–125:13, July 2016.

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

[5] Dirk J. Struik. *Lectures on Classical Differential Geometry. Second Edition*. Reading, Mass., Addison-Wesley Pub. Co, 1988.

[6] Pavel Pokorny. Geodesics revisited. *Chaotic Modeling and Simulation*, pages 281–298, 2012.

[7] Joseph S. B. Mitchell, David M. Mount, and Christos H. Papadimitriou. The discrete geodesic problem. *SIAM J. Comput.*, 16(4):647–668, 1987.

[8] Jindong Chen and Yijie Han. Shortest paths on a polyhedron. In *Proceedings of the Sixth Annual Symposium on Computational Geometry*, SCG '90, pages 360–369, New York, NY, USA, 1990. ACM.

[9] Vitaly Surazhsky, Tatiana Surazhsky, Danil Kirsanov, Steven J. Gortler, and Hugues Hoppe. Fast exact and approximate geodesics on meshes. *ACM Trans. Graph.*, 24(3):553–560, July 2005.

[10] Xiang Ying, Xiaoning Wang, and Ying He. Saddle vertex graph (svg): A novel solution to the discrete geodesic problem. *ACM Trans. Graph.*, 32(6):170:1–170:12, November 2013.

[11] Xiang Ying, Shi-Qing Xin, and Ying He. Parallel chen-han (pch) algorithm for discrete geodesics. *ArXiv*, abs/1305.1293, 2013.

[12] Chunxu Xu, Tuanfeng Y Wang, Yong-Jin Liu, Ligang Liu, and Ying He. Fast wavefront propagation (fwp) for computing exact geodesic distances on meshes. *IEEE transactions on visualization and computer graphics*, 21(7):822–834, 2015.

[13] Xiaoning Wang, Zheng Fang, Jiajun Wu, Shi-Qing Xin, and Ying He. Discrete geodesic graph (dgg) for computing geodesic distances on polyhedral surfaces. *Computer Aided Geometric Design*, 52:262 – 284, 2017. Geometric Modeling and Processing 2017.

[14] Shi-Qing Xin and Guo-Jin Wang. Applying the improved chen and han's algorithm to different versions of shortest path problems on a polyhedral surface. *Computer-Aided Design*, 42(10):942 – 951, 2010.

[15] Shi-Qing Xin, Xiang Ying, and Ying He. Constant-time all-pairs geodesic distance query on triangle meshes. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '12, pages 31–38, New York, NY, USA, 2012. ACM.

[16] Mukund Balasubramanian, Jonathan R. Polimeni, and Eric L. Schwartz. Exact geodesics and shortest paths on polyhedral surfaces. *IEEE Trans. Pattern Anal. Mach. Intell.*, 31(6):1006–1016, June 2009.

[17] R. Kimmel and J. A. Sethian. Computing geodesic paths on manifolds. *Proceedings of the National Academy of Sciences*, 95(15):8431–8435, 1998.

[18] Keenan Crane, Clarisse Weischedel, and Max Wardetzky. Geodesics in heat: A new approach to computing distance based on heat flow. *ACM Trans. Graph.*, 32(5):152:1–152:11, October 2013.

[19] Keenan Crane, Clarisse Weischedel, and Max Wardetzky. The heat method for distance computation. *Commun. ACM*, 60(11):90–99, October 2017.

[20] Yohanes Yudhi Adikusuma, Zheng Fang, and Ying He. Fast construction of discrete geodesic graphs. *ACM Transactions on Graphics*, 39(2):14:1–14, 2020.

[21] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1(1):269–271, December 1959.

[22] Xiang Ying, Caibao Huang, Xuzhou Fu, Ying He, Ruiguo Yu, Jianrong Wang, and Mei Yu. Parallelizing discrete geodesic algorithms with perfect efficiency. *Computer-Aided Design*, 115:161 – 171, 2019.

[23] Ofir Weber, Yohai S. Devir, Alexander M. Bronstein, Michael M. Bronstein, and Ron Kimmel. Parallel algorithms for approximation of distance maps on parametric surfaces. *ACM Trans. Graph.*, 27(4):104:1–104:16, November 2008.

[24] Jiong Tao, Juyong Zhang, Bailin Deng, Zheng Fang, Yue Peng, and Ying He. Parallel and scalable heat method. *CoRR*, abs/1812.06060,

2018.

[25] Xiang Ying. *Efficient and Practical Algorithms for Discrete Geodesics*. PhD thesis, Nanyang Technological University, School of Computer Science and Engineering, 2012.

[26] Prosenjit Bose, Anil Maheshwari, Chang Shu, and Stefanie Wuhrer. A survey of geodesic paths on 3d surfaces. *Computational Geometry*, 44(9):486 – 498, 2011.

[27] Leila De Floriani, Enrico Puppo, and Paola Magillo. Applications of computational geometry to geographic information systems. *Handbook of computational geometry*, pages 333–388, 1999.

[28] Guo-Jin Wang, Kai Tang, and Chiew-Lan Tai. Parametric representation of a surface pencil with a common spatial geodesic. *Computer-Aided Design*, 36(5):447–459, 2004.

[29] Pekka J Toivanen. New geodosic distance transforms for gray-scale images. *Pattern Recognition Letters*, 17(5):437–450, 1996.

[30] Xiaoning Wang, Xiang Ying, Yong-Jin Liu, Shi-Qing Xin, Wenping Wang, Xianfeng Gu, Wolfgang Mueller-Wittig, and Ying He. Intrinsic computation of centroidal voronoi tessellation (cvt) on meshes. *Computer-Aided Design*, 58:51 – 61, 2015. Solid and Physical Modeling 2014.

[31] Chunxu Xu, Yong-Jin Liu, Qian Sun, Jinyan Li, and Ying He. Polyline-sourced geodesic voronoi diagrams on triangle meshes. *Comput. Graph. Forum*, 33(7):161–170, October 2014.

[32] Gabriel Peyré and Laurent D. Cohen. Geodesic remeshing using front propagation. *International Journal of Computer Vision*, 69(1):145, May 2006.

[33] Y. J. 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, Aug 2011.

[34] Yajie Yan, Kyle Sykes, Erin Chambers, David Letscher, and Tao Ju. Erosion thickness on medial axes of 3d shapes. *ACM Transactions on Graphics (TOG)*, 35(4):38, 2016.

[35] Xiang Ying, Zhenhua Li, and Ying He. A parallel algorithm for improving the maximal property of poisson disk sampling. *Computer-Aided Design*, 46:37 – 44, 2014. 2013 SIAM Conference on Geometric and Physical Modeling.

[36] Qi-Xing Huang, Bart Adams, Martin Wicke, and Leonidas J. Guibas. Non-rigid registration under isometric deformations. In *Proceedings of the Symposium on Geometry Processing*, number 9 in SGP '08, pages 1449–1457, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.

[37] Yong-Jin Liu, Dian Fan, Chun-Xu Xu, and Ying He. Constructing intrinsic delaunay triangulations from the dual of geodesic voronoi diagrams. *ACM Trans. Graph.*, 36(2):15:1–15:15, April 2017.

[38] Gabriel Peyré, Mickael Péchaud, Renaud Keriven, and Laurent D. Cohen. Geodesic methods in computer vision and graphics. *Foundations and Trends in Computer Graphics and Vision*, 5(3-4):197–397, 2010.

[39] F. Cazals and M. Pouget. Estimating differential quantities using polynomial fitting of osculating jets. *Computer Aided Geometric Design*, 22(2):121 – 146, 2005.

[40] Joseph S.B. Mitchell. Geometric shortest paths and network optimization. In *Handbook of Computational Geometry*, pages 633–701. Elsevier Science Publishers B.V. North-Holland, 1998.

[41] Micha Sharir(and Amir Schorr. On shortest paths in polyhedral spaces. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, STOC '84, pages 144–153, New York, NY, USA, 1984. ACM.

[42] David M. Mount. On finding shortest paths on convex polyhedra. 1985.

[43] Joseph O'Rourke, Subhash Suri, and Heather Booth. Shortest paths on polyhedral surfaces. In K. Mehlhorn, editor, *STACS 85*, pages 243–254, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg.

[44] Biliana Kaneva and Joseph O'Rourke. An implementation of chen & han's shortest paths algorithm. In *CCCG*, 2000.

[45] J A Sethian. A fast marching level set method for monotonically advancing fronts. *Proceedings of the National Academy of Sciences*, 93(4):1591–1595, 1996.

[46] Christos H. Papadimitriou. An algorithm for shortest-path motion in three dimensions. *Information Processing Letters*, 20(5):259 – 263, 1985.

[47] K. Clarkson. Approximation algorithms for shortest path motion planning. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, page 56–65, New York, NY, USA, 1987. Association for Computing Machinery.

[48] Joonsoo Choi, Jürgen Sellen, and Chee-Keng Yap. Approximate euclidean shortest path in 3-space. In *Proceedings of the Tenth Annual Symposium on Computational Geometry*, SCG '94, page 41–48, New York, NY, USA, 1994. Association for Computing Machinery.

[49] John Hershberger and Subhash Suri. Practical methods for approximating shortest paths on a convex polytope in r3. *Computational Geometry*, 10(1):31 – 46, 1998.

[50] M Novotni and Reinhard Klein. Computing geodesic distances on triangular meshes. 02 2002.

[51] Lyudmil Aleksandrov, Anil Maheshwari, and Jörg-Rüdiger Sack. An improved approximation algorithm for computing geometric shortest paths. In Andrzej Lingas and Bengt J. Nilsson, editors, *Fundamentals of Computation Theory*, pages 246–257, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

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

[53] Shi-Qing Xin and Guo-Jin Wang. Efficiently determining a locally exact shortest path on polyhedral surfaces. *Computer-Aided Design*, 39(12):1081 – 1090, 2007.

[54] Pankaj Agarwal, Boris Aronov, Joseph O'Rourke, and Catherine Schevon. Star unfolding of a polytope with applications. *SIAM J. Comput.*, 26:1689–1713, 12 1997.

[55] Atlas F. Cook and Carola Wenk. Shortest path problems on a polyhedral surface. In Frank Dehne, Marina Gavrilova, Jörg-Rüdiger Sack, and Csaba D. Tóth, editors, *Algorithms and Data Structures*, pages 156–167, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[56] Uri Zwick. All pairs shortest paths in weighted directed graphs exact and almost exact algorithms. *In Proceedings of the 39th Annual Symposium on Foundations of Computer Science, FOCS '98*, page 310, 1998.

[57] Uri Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of the ACM*, 49, 12 2000.

[58] Surender Baswana, Ramesh Hariharan, and Sandeep Sen. Maintaining all-pairs approximate shortest paths under deletion of edges. In *Pro-*

ceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '03, page 394–403, USA, 2003. Society for Industrial and Applied Mathematics.

[59] Dorit Dor, Shay Halperin, and Uri Zwick. All-pairs almost shortest paths. *SIAM Journal on Computing*, 29(5):1740–1759, 2000.

[60] Michael Breuß, Emiliano Cristiani, Pascal Gwosdek, and Oliver Vogel. An adaptive domain-decomposition technique for parallelization of the fast marching method. *Applied Mathematics and Computation*, 218(1):32 – 44, 2011.

[61] Adam Chacon and Alexander Vladimirsky. A parallel two-scale method for eikonal equations. *SIAM J. Sci. Comput*, 37(1):A156 – A180, 2015.

[62] Jianming Yang and Frederick Stern. A highly scalable massively parallel fast marching method for the eikonal equation. *Journal of Computational Physics*, 332:333 – 362, 2017.

[63] Thomas Leung and Jitendra Malik. Detecting, localizing and grouping repeated scene elements from an image. In Bernard Buxton and Roberto Cipolla, editors, *Computer Vision — ECCV '96*, pages 546–555, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[64] Frederik Schaffalitzky and Andrew Zisserman. *Geometric Grouping of Repeated Elements within Images*, pages 165–181. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[65] Chris Harris and Mike Stephens. A combined corner and edge detector. In *In Proc. of Fourth Alvey Vision Conference*, pages 147–151, 1988.

[66] James Hays, Marius Leordeanu, Alexei A. Efros, and Yanxi Liu. *Discovering Texture Regularity as a Higher-Order Correspondence Problem*, pages 522–535. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[67] Yanxi Liu, Robert T. Collins, and Yanghai Tsin. A computational model for periodic pattern perception based on frieze and wallpaper groups. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(3):354–371, March 2004.

[68] M. Park, K. Brocklehurst, R. T. Collins, and Y. Liu. Deformed lattice detection in real-world images using mean-shift belief propagation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(10):1804–1816, Oct 2009.

[69] L. Lettry, M. Perdoch, K. Vanhoey, and L. Van Gool. Repeated pattern detection using cnn activations. In *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 47–55, 2017.

[70] Jingchen Liu and Yanxi Liu. Grasp recurring patterns from a single view. In *Proceedings of the 2013 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '13, page 2003–2010, USA, 2013. IEEE Computer Society.

[71] Carlos Rodriguez-Pardo, Sergio Suja, David Pascual, Jorge Lopez-Moreno, and Elena Garces. Automatic extraction and synthesis of regular repeatable patterns. *Computers & Graphics*, 83:33 – 41, 2019.

[72] Tinne Tuytelaars, Andreas Turina, and Luc Van Gool. Noncombinatorial detection of regular repetitions under perspective skew. *IEEE Trans. Pattern Anal. Mach. Intell.*, 25(4):418–432, April 2003.

[73] Y. Cai and G. Baciu. Detecting, grouping, and structure inference for invariant repetitive patterns in images. *IEEE Transactions on Image Processing*, 22(6):2343–2355, June 2013.

[74] Song Chun Zhu and Alan Yuille. Region competition: Unifying snakes, region growing, and bayes/mdl for multiband image segmentation.

IEEE Trans. Pattern Anal. Mach. Intell., 18(9):884–900, September 1996.

[75] J. Pritts, O. Chum, and J. Matas. Rectification, and segmentation of coplanar repeated patterns. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 2973–2980, June 2014.

[76] J Matas, O Chum, M Urban, and T Pajdla. Robust wide-baseline stereo from maximally stable extremal regions. *Image and Vision Computing*, 22(10):761 – 767, 2004. British Machine Vision Computing 2002.

[77] Stepán Obdrzálek and Jiri Matas. Object recognition using local affine frames on distinguished regions. 01 2002.

[78] David Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60:91–, 11 2004.

[79] M. Li, F. C. Langbein, and R. R. Martin. Constructing regularity feature trees for solid models. In *Proceedings of the 4th International Conference on Geometric Modeling and Processing*, GMP'06, page 267–286, Berlin, Heidelberg, 2006. Springer-Verlag.

[80] Niloy J. Mitra, Leonidas J. Guibas, and Mark Pauly. Partial and approximate symmetry detection for 3d geometry. In *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, page 560–568, New York, NY, USA, 2006. Association for Computing Machinery.

[81] Joshua Podolak, Philip Shilane, Aleksey Golovinskiy, Szymon Rusinkiewicz, and Thomas Funkhouser. A planar-reflective symmetry transform for 3d shapes. *ACM Trans. Graph.*, 25(3):549–559, July 2006.

[82] Maks Ovsjanikov, Jian Sun, and Leonidas Guibas. Global intrinsic symmetries of shapes. In *Proceedings of the Symposium on Geometry*

Processing, SGP '08, page 1341–1348, Goslar, DEU, 2008. Eurographics Association.

[83] Kai Xu, Hao Zhang, Andrea Tagliasacchi, Ligang Liu, Guo Li, Min Meng, and Yueshan Xiong. Partial intrinsic reflectional symmetry of 3d shapes. *ACM Trans. Graph.*, 28(5):1–10, December 2009.

[84] Shenglan Liu, Ralph Martin, Frank Langbein, and Paul Rosin. Segmenting periodic reliefs on triangle meshes. volume 4647, 04 2007.

[85] P. J. Besl and N. D. McKay. A method for registration of 3-d shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(2):239–256, 1992.

[86] David W. Eggert, Andrew W. Fitzgibbon, and Robert B. Fisher. Simultaneous registration of multiple range views for use in reverse engineering of cad models. *Computer Vision and Image Understanding*, 69(3):253 – 272, 1998.

[87] Mark Pauly, Niloy J. Mitra, Johannes Wallner, Helmut Pottmann, and Leonidas J. Guibas. Discovering structural regularity in 3d geometry. *ACM Trans. Graph.*, 27(3):1–11, August 2008.

[88] Niloy J. Mitra, Alex Bronstein, and Michael Bronstein. Intrinsic regularity detection in 3d geometry. In Kostas Daniilidis, Petros Maragos, and Nikos Paragios, editors, *Computer Vision – ECCV 2010*, pages 398–410, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[89] Qixing Huang, Leonidas J. Guibas, and Niloy J. Mitra. Near-regular structure discovery using linear programming. *ACM Trans. Graph.*, 33(3):23:1–23:17, June 2014.

[90] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science &*

Engineering, IEEE, 5(1):46–55, 1998.

[91] Qin Yipeng. *Fast and Exact Geodesic Computation using Edge-based Windows Grouping.* PhD thesis, Bournemouth University, National Centre for Computer Animation, 2017.

[92] Zichun Zhong, Xiaohu Guo, Wenping Wang, Bruno Lévy, Feng Sun, Yang Liu, and Weihua Mao. Particle-based anisotropic surface meshing. *ACM Trans. Graph.*, 32(4), July 2013.

[93] Y. Cai and G. Baciu. Translation symmetry detection: A repetitive pattern analysis approach. In *2013 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 223–228, June 2013.

[94] Marius Leordeanu and Martial Hebert. A spectral technique for correspondence problems using pairwise constraints. In *Proceedings of the Tenth IEEE International Conference on Computer Vision - Volume 2*, ICCV '05, page 1482–1489, USA, 2005. IEEE Computer Society.

[95] Ulrike von Luxburg. A tutorial on spectral clustering. *Statistics and Computing*, 17(4):395–416, Dec 2007.

[96] Timothy Gatzke, Michael Garland, Steve Zelinka, and Cindy Grimm. Curvature maps for local shape comparison. *Proceedings. International Conference on Shape Modeling and Applications*, 00:246–255, 2005.

[97] K. Fukunaga and L. Hostetler. The estimation of the gradient of a density function, with applications in pattern recognition. *IEEE Trans. Inf. Theor.*, 21(1):32–40, September 2006.

[98] Yizong Cheng. Mean shift, mode seeking, and clustering. *IEEE Trans. Pattern Anal. Mach. Intell.*, 17(8):790–799, August 1995.