BOURNEMOUTH UNIVERSITY

DOCTORAL THESIS

Fast and Exact Geodesic Computation using Edge-based Windows Grouping

Author:
Yipeng QIN

Supervisor:
Dr. Hongchuan YU
Prof. Jianjun ZHANG

A thesis submitted in partial fulfilment of the requirements of Bournemouth University for the degree of Doctor of Philosophy

National Centre for Computer Animation

July 20, 2017

Copyright Statement

This copy of the thesis has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and due acknowledgement must always be made of the use of any material contained in, or derived from, this thesis.

Abstract

Computing discrete geodesic distance over triangle meshes is one of the fundamental problems in computational geometry and computer graphics. As the "Big Data Era" arrives, a fast and accurate solution to the geodesic computation problem on large scale models with constantly increasing resolutions is desired. However, it is still challenging to deal with the speed, memory cost and accuracy of the geodesic computation at the same time.

This thesis addresses the aforementioned challenge by proposing the Edge-based Windows Grouping (EWG) technique. With the local geodesic information encoded in a "window", EWG groups the windows based on the mesh edges and processes them together. Thus, the interrelationships among the grouped windows can be utilized to improve the performance of geodesic computation on triangle meshes.

Based on EWG, a novel exact geodesic algorithm is proposed in this thesis, which is fast, accurate and memory-efficient. This algorithm computes the geodesic distances at mesh vertices by propagating the geodesic information from the source over the entire mesh. Its high performance comes from its low computational redundancy and management overhead, which are both introduced by EWG. First, the redundant windows on an edge can be removed by comparing its distance with those of the other windows on the same edge. Second, the windows grouped on an edge usually have similar geodesic distances and can be propagated in batches efficiently. To the best of my knowledge, the proposed exact geodesic algorithm is the fastest and most memory-efficient one among all existing methods.

In addition, the proposed exact geodesic algorithm is revised and employed to construct the geodesic-metric-based Voronoi diagram on triangle meshes. In this application, the geodesic computation is the bottleneck in both the time and memory costs. The proposed method achieves low memory cost from the key observation that the Voronoi diagram boundaries usually only cross a minority of the meshes' triangles and most of the windows stored on edges are redundant. As a result, the proposed method resolves the memory bottleneck of the Voronoi diagram construction without sacrificing its speed.

Contents

Copyright Statement **i**

Abstract **ii**

List of Figures **ix**

List of Tables **x**

List of Abbreviations **xi**

Acknowledgements **xii**

Declaration **xiii**

1 Introduction **1**

1.1 Background 1

1.1.1 Geodesics in Differential Geometry 2

1.1.2 Geodesics in Computational Geometry 3

1.2 Motivation 5

1.3 Contributions 6

1.4 Thesis Outline 7

1.5 List of Publications 8

2 Related Work **9**

2.1 Discrete Geodesic Algorithms 9

2.1.1 Computational Geometry Approach 10

2.1.1.1 Euclidean Cost Metric 10

2.1.1.2 Weighted Cost Metric 19

2.1.2 Partial Differential Equation (PDE) Approach 24

2.1.2.1 Discrete Eikonal Equation 24

2.1.2.2 Discrete Poisson Equation 26

2.2 Voronoi Diagram on Surfaces 28

2.2.1 Voronoi Diagrams on 2D Planes 28

2.2.2 Voronoi Diagrams on 3D Polyhedral Surfaces 31

3 Edge-based Windows Grouping 33

3.1 Discrete Geodesic Problem (DGP) Definition 33

3.1.1 Single-Source DGP (SS-DGP) 34

3.1.2 Voronoi Diagram oriented DGP (VD-DGP) 34

3.2 Preliminaries 35

3.2.1 Locally Shortest Paths on Triangle Meshes 35

3.2.2 Globally Shortest Paths on Triangle Meshes 38

3.3 Edge-based Windows Grouping (EWG) 38

3.3.1 Window Definition and Propagation 39

3.3.2 Applying EWG on Window Propagation 41

3.3.2.1 EWG Definition 42

3.3.2.2 EWG Window Propagation 43

3.3.2.3 EWG Performance Evaluation 45

3.3.3 EWG-based Solutions to Geodesic Problems 46

3.3.3.1 Solution to the SS-DGP problem 46

3.3.3.2 Solution to the VD-DGP problem 47

4 Fast and Exact SS-DGP Algorithm 50

4.1 Algorithm Overview 51

4.2 EWG in Window List Propagation Within a Triangle 52

4.2.1 Pairwise Window Pruning Within a Triangle 53
4.2.2 Principles for Window Pruning 59
4.2.3 EWG-based Window List Propagation 62
4.2.3.1 Window List Splitting 63
4.2.3.2 Window List Propagation 67
4.2.3.3 Window List Merging 68
4.2.4 Algorithmic Choices Justification 70
4.3 EWG in Wavefront Propagation Over a Mesh 74
4.3.1 Face-Sorted Wavefront Propagation 74
4.3.2 Vertex-Sorted Wavefront Propagation 79
4.3.3 Algorithmic Choices Justification 81
4.4 Complexity Analysis 83
4.5 Experimental Results 83
4.5.1 Overall Performance 84
4.5.2 Performance Profiling 84
4.5.3 Scalability 87
4.5.4 Robustness 87
4.6 Summary 92

5 Fast and Memory-Efficient Voronoi Diagram Construction 93
5.1 Redundant Window Removal (RWR) 94
5.1.1 Preliminaries 94
5.1.2 Redundant Windows Removal (RWR) 97
5.1.3 Performance Verification 100
5.2 Applying RWR in Geodesic Computation 102
5.2.1 Wavefront Collision 105
5.2.2 Priorities Definition 106
5.3 Complexity Analysis 107

5.4 Experimental Results 107

5.4.1 Comparison with Liu et al. (2011) 107

5.4.2 Comparison with Xu et al. (2014) 118

5.4.3 Comparison with VTP 119

5.4.4 Application to Remeshing 119

5.5 Summary 121

6 Conclusion and Future Work 123

6.1 Conclusion 123

6.2 Future Work 124

A Model Collection 135

B VTP Ablation Study 138

C VTP Variants Comparison 141

D Distribution of Window Propagations 146

E Performance Comparison between VTP and Others 148

F VTP Performance Profiling 153

G Longest Length in a Triangle 155

H RWR Performance Verification 157

I Performance Comparison among VD-DGP Algorithms 159

J Voronoi Diagram Construction Performance Profiling 164

List of Figures

1.1 Accuracy of the fast marching method and the heat method against the mesh quality. 6

2.1 The structure of related works on discrete geodesic algorithms. 9

2.2 Illustration of the incremental insertion method. 28

2.3 Illustration of the divide and conquer method. 30

2.4 Illustration of the plane-sweep method. 31

3.1 Convert the source point s to a vertex of the mesh by subdivisions. 34

3.2 Illustration of the two cases of a shortest path on a triangle mesh. 35

3.3 Illustration of the triangle strip and the planar unfolding. . . . 36

3.4 Illustration of a locally shortest path passing through a vertex. 37

3.5 Illustration of the geodesic visible cone in a triangle strip. . . . 39

3.6 Illustration of the window data structure. 40

3.7 Three cases of the window propagation in a face. 41

3.8 Illustration of EWG definitions. 42

3.9 The two cases of the EWG window propagation module. . . . 44

3.10 The reason for employing the EWG Definition 3.1 in the solution of the SS-DGP problem. 47

3.11 The reason for employing the EWG Definition 3.2 in the solution of the VD-DGP problem. 48

4.1 Three configurations of the separating point of a window. . . . 53

4.2 The three cases that w_0 and w_1 are propagated from the same edge to another edge. 54

4.3 The three cases that w_0 and w_1 are propagated from two edges to the third edge. 55

4.4 The three cases that w_0 and w_1 are propagated from the same edge to two other edges. 56

4.5 The three cases of checking windows with vertices. 57

4.6 The three cases that w_0 and w_1 are propagated from two edges to two edges. 58

4.7 Window configurations for Proposition 4.1. 59

4.8 Window configurations for Proposition 4.2. 60

4.9 Window configurations for Proposition 4.3. 61

4.10 Window configurations for Proposition 4.4. 62

4.11 One Angle Two Sides (Rule 1). 64

4.12 Ablation study on Rule 1. 66

4.13 Window List Propagation (Rule 2). 67

4.14 Window List Merging (Rule 3). 69

4.15 Ablation study on Rule 2 and 3. 71

4.16 Face-sorted wavefront propagation. 75

4.17 Order-free secondary merger. 78

4.18 Vertex-sorted wavefront propagation. 79

4.19 Comparison of running times of three common components on two models. 86

4.20 Comparison of scalability against recent geodesic algorithms. . 88

4.21 Comparison of robustness against anisotropic triangulation. . . 89

4.22 Robustness of individual components against anisotropic triangulation (window propagation component). 90

4.23 Robustness of individual components against anisotropic triangulation (window pruning component). 91

4.24 Robustness of individual components against anisotropic triangulation (window management component). 91

5.1 Voronoi diagram on the Buste model (3K faces) 95

5.2 Illustration of intersections between mesh edges and Voronoi cell boundaries. 96

5.3 Illustration of redundant primitives. 96

5.4 Illustration of the monotonicity for window propagations. . . . 97

5.5 Illustration of an inactive region. 98

5.6 Illustration of Proposition 5.1. 99

5.7 Performance verification on RWR. 101

5.8 Illustration of the triangle-oriented region expansion scheme. . 102

5.9 Vertex-sorted Triangle Propagation. 103

5.10 The window trimming rule in the MMP algorithm. 103

5.11 The collision of the propagation wavefront. 105

5.12 Illustration of the vertex's priority definition. 106

5.13 Examples of Voronoi diagrams on meshes. 108

5.14 Performance comparison between FWP-MMP based Voronoi diagram construction algorithm and ours on the number of sources. 112

5.15 Comparison of running times of four common components in Voronoi diagram construction on two models. 113

5.16 Comparison of scalability against FWP-MMP based Voronoi diagram construction algorithm. 115

5.17 Comparison of robustness against anisotropic triangulation (Time). 116

5.18 Comparison of robustness against anisotropic triangulation (Memory). 117

5.19 Illustration of remeshing. 120

G.1 Illustration of Lemma G.1. 155

List of Tables

4.1 Performance comparison between VTP-Exhaustive and VTP. . 72
4.2 Performance comparison between VTP-Trimming and VTP. . 73
4.3 Performance comparison between VTP-MMP, VTP-CH and VTP. 73
4.4 Performance comparison between FTP and VTP. 81
4.5 Performance comparison between OPVTP and VTP. 82
4.6 The mean and standard deviation of performance ratios between other algorithms and the proposed VTP algorithm. . . . 84
4.7 Performance comparison with state-of-the-art geodesic algorithms. 85

5.1 Performance comparison with Liu et al. (2011) 109
5.2 The mean and standard deviation of the performance ratios between other algorithms and the proposed *window*-VTP algorithm. 110
5.3 Performance comparison between MMP, FWP-MMP and the proposed *window*-VTP on five representative models. 110
5.4 Performance comparison with Xu et al. (2014) 118
5.5 Performance comparison with VTP 120
5.6 Performance comparison with the FWP-MMP version of Liu et al. (2011) on remeshing. 121

List of Abbreviations

2D	Two Dimensional
3D	Three Dimensional
AP-DGP	All-Pairs Discrete Geodesic Problem
CGAL	Computational Geometry Algorithms Library
CH	Chen-Han
DGP	Discrete Geodesic Problem
EWG	Edge-based Windows Grouping
FIFO	First-In-First-Out
FMM	Fast Marching Method
FTP	Face-oriented Triangle Propagation
FWP	Fast Wavefront Propagation
GIS	Geographic Information Systems
GPU	Graphics Processing Unit
ICH	Improved Chen-Han
MMP	Mitchell-Mount-Papadimitriou
NP	Non-deterministic Polynomial-time
ODE	Ordinary Differential Equation
OPFTP	Order-Preserving Face-oriented Triangle Propagation
OPVTP	Order-Preserving Vertex-oriented Triangle Propagation
PC	Personal Computer
PCH	Parallel Chen-Han
PDE	Partial Differential Equation
RWR	Redundant Window Removal
SS-DGP	Single-Source Discrete Geodesic Problem
VC	Voronoi Cell
VD-DGP	Voronoi Diagram oriented Discrete Geodesic Problem
VTP	Vertex-oriented Triangle Propagation

Acknowledgements

First of all, I would like to express my sincere gratitude to my supervisors Dr. Hongchuan Yu and Prof. Jianjun Zhang for their priceless advices and continuous support on my research. Their constant pursuit of cutting-edge research inspires me to dive deeper and deeper into my research topic.

Besides my supervisors, I would like to thank Prof. Yizhou Yu and Xiaoguang Han from the University of Hong Kong for their stimulating comments on my work and our sleepless nights before the deadline.

My sincere thanks also goes to Daniel Cox, Sunny Choi, Jan Lewis and other colleagues at Bournemouth University for their efforts on all the administrative works of my campus life. I would also like to thank all my labmates for the fun we have had in the past years.

Last but not the least, I would like to thank my parents and my wife for backing me up to overcome all the encountered difficulties during my research. Your encouragements always get me out of my frustrating moments.

Gratitude also goes to the financial support from Bournemouth University, which makes my PhD research possible.

Declaration

This thesis has been created by myself and has not been submitted in any previous application for any degree. The work in this thesis has been undertaken by myself except where otherwise stated.

Chapter 1

Introduction

1.1 Background

The shortest path problem is a classic problem in graph theory. It aims at finding a shortest path between two vertices of a graph. The most important shortest path algorithm so far is the Dijkstra's algorithm which solves the problem by dynamic programming (Dijkstra, 1959). Since many real-world problems can be represented with graphs, the shortest path algorithms are widely used in various areas, for example, route planning and navigation, geographic information systems (GISs), computer games, and sever selection. A detailed survey of the graph-based shortest path algorithms can be found in Sommer's work (Sommer, 2014). However, the demands for shortest paths are not limited to graphs.

In the 1970s and 1980s, applications in robotics and autonomous navigation motivated the study of optimal collision-free path planning algorithms in both two-dimensional (2D) and three-dimensional (3D) spaces (Lozano-Pérez and Wesley, 1979; O'Rourke et al., 1985; Mount, 1985a; Sharir and Schorr, 1986). For the 2D shortest path problem with polygonal obstacles, a common solution is to build a *visibility graph* (Lozano-Pérez and Wesley, 1979; Sharir and Schorr, 1986) and construct the shortest paths by searching it with the Dijkstra's algorithm (Dijkstra, 1959). The time complexity of this method is $O(n^2 \log n)$, where n is the number of obstacle corners. However, the 3D shortest path problem with polyhedral obstacles is much more difficult and is proven to be *Non-deterministic Polynomial-time (NP) hard* by Canny and Reif (1987).

Thanks to its usefulness as an important distance metric of surfaces, an important special case of the general 3D shortest path problem attracts research interests from the computational geometry community. This special

case is formulated as the ***discrete geodesic problem*** **(DGP)** by Mitchell et al. (1987) and it aims to find a shortest path between two vertices on a polyhedral surface in the 3D Euclidean space. These shortest paths are known as the *geodesic paths* and their lengths are known as the *geodesic distances*. According to the number of source and destination vertices, the DGP problem has three variants:

(1) The *Discrete Geodesic Problem* (DGP), which finds the geodesic path between one source vertex and one destination vertex.

(2) The *Single-Source Discrete Geodesic Problem* (SS-DGP), which finds the geodesic paths from one source vertex to all the other vertices of the polyhedral surface.

(3) The *All-Pairs Discrete Geodesic Problem* (AP-DGP), which finds the geodesic paths between all pairs of vertices on the polyhedral surface.

This thesis focuses on solving the SS-DGP problem since it can be extended to solve the other two problems. For the DGP problem, it can be solved by performing the SS-DGP algorithm at its source vertex since its source and destination vertices are a subset of those of the SS-DGP problem. For the AP-DGP problem, the all-pairs geodesic paths it requires can be obtained by performing the SS-DGP algorithm at each vertex of the polyhedral surface. However, these extensions are not necessarily the optimal solution to the DGP and AP-DGP problems.

Traditionally, the geodesics are studied in two fields of mathematics: the differential geometry and the computational geometry. Thus, an overview of geodesics from the perspectives of these two fields is presented in the following two sections.

1.1.1 Geodesics in Differential Geometry

Kline (1990) reviewed the development of differential geometry as a chapter in his book. Originally, the theory of surfaces is founded by Euler and extended by Monge. Then, immense amount of work on this subject is made by Gauss, who started working on geodesy and map-making in 1816. In 1827, Gauss published his definitive paper titled "General Investigations of Curved Surfaces", in which Gauss solved the problem of finding the *geodesics* on surfaces by the calculus of variations. Note that the term *geodesic* is introduced

by Liouville in 1850 from the word *geodesy*. Gauss also proposed the idea that a surface is a space in itself. This idea was generalized by Riemann and opened up the new field of Riemannian geometry in non-Euclidean geometry. Thus, the geodesics play a fundamental role in the development of differential geometry.

Intuitively, the *geodesic* is a generalized notion of the *straight line* from planes to curved spaces, which is locally shortest. According to Polthier and Schmies (1998), a curve is a geodesic if its geodesic curvature is zero, and thus the geodesics on smooth surfaces can be computed by solving a ordinary differential equation (ODE) related to the geodesic curvature.

In practice, computers have a discrete nature and the data they process are usually discretized in advance. Thus, the smooth surfaces are usually discretized as triangle meshes to be processed by computers. However, this discretization results in zero curvature points on the triangle faces of the meshes. Thus, the concepts and methods from the differential geometry cannot be applied in this scenario and should be discretized as well. For example, Polthier and Schmies (1998) proposed a discrete version of the geodesic curvature and used the Runge-Kutta method to solve the ODE associated with it. However, the geodesic paths computed by their method are *locally* shortest, which are less useful in many real-world applications (e.g. path planning) requiring the *globally* shortest paths. To compute globally shortest geodesics, some other methods based on the relationship between geodesic distances and physical phenomena (e.g. wave, heat) on surfaces are proposed (Kimmel and Sethian, 1998; Crane et al., 2013). However, these methods cannot avoid introducing errors and thus only yield an approximate solution.

1.1.2 Geodesics in Computational Geometry

In computational geometry, surfaces are assumed to be polyhedral, which are usually represented by triangle meshes. Although the general 3D shortest path problem is proven to be *NP-hard* (Canny and Reif, 1987), the DGP problem is not and the first concern of it is its computational complexity.

The pioneer work on this issue is the algorithm proposed by Sharir and Schorr (1986), which solves the DGP problem on a convex polyhedral surface in $O(n^3 \log n)$ time, where n is the number of vertices of the surface. Although their algorithm only operates on convex polyhedral surfaces, the

planar unfolding technique employed by it laid the foundation of many following works since it converts the problem from 3D polyhedral surfaces to 2D planes. Sharir and Schorr's algorithm is then improved by Mount (1985a) to achieve an $O(n^2 \log n)$ time complexity and extended by O'Rourke et al. (1985) to operate on possibly non-convex polyhedral surfaces in $O(n^5)$ time. Although the algorithm proposed by O'Rourke et al. (1985) has a high time complexity of $O(n^5)$, it proves that the DGP problem can be solved in polynomial time. After that, the major concern of solving the DGP problem moves to its asymptotic complexity.

Mitchell et al. (1987) improved the time and space complexities of solving the SS-DGP problem on polyhedral surfaces to $O(n^2 \log n)$ and $O(n^2)$ respectively. Chen and Han (1990) further improved them to $O(n^2)$ and $O(n)$, which are the best asymptotic complexities so far.[1] However, it is interesting that Mitchell et al.'s algorithm runs much faster than Chen and Han's algorithm in practice although having a larger asymptotic complexity (Surazhsky et al., 2005). Since then, the major concern moves to improve the practical performance of geodesic algorithms.

Xin and Wang (2009) improved the algorithm proposed by Chen and Han (1990) by reducing its redundancy effectively. However, performing such redundancy reduction requires a priority queue. Although employing a priority queue increases the time complexity of their algorithm to $O(n^2 \log n)$, it runs thousands times faster than the original Chen and Han's algorithm. To further accelerate it, Ying et al. (2014) proposed an parallel implementation of Xin and Wang's algorithm which runs an order of magnitude faster. Observing that the priority queue consumes the majority of the time in the algorithms proposed by Mitchell et al. (1987) and Xin and Wang (2009), Xu et al. (2015) proposed to replace the priority queue with a bucket structure and their method can improve the speed by a factor of 3-10.

Apart from solving the DGP problem exactly, some approximation algorithms are also proposed. For example, the algorithms which approximate the geodesic paths by constructing and searching a graph (Agarwal et al., 2002; Ying et al., 2013) and those by iteratively refining a rough initial path (Kanai and Suzuki, 2001).If different weights are assigned to the faces of the polyhe-

[1] Kapoor (1999) proposed an algorithm and claimed it runs in $O(n \log^2 n)$ time. However, it is not widely accepted by the academia since its details are too complex (O'Rourke, 1999).

dral surfaces, the geodesics on them become quite different and are discussed in some other works (Mitchell and Papadimitriou, 1991; Sun and Reif, 2006; Aleksandrov et al., 2010).

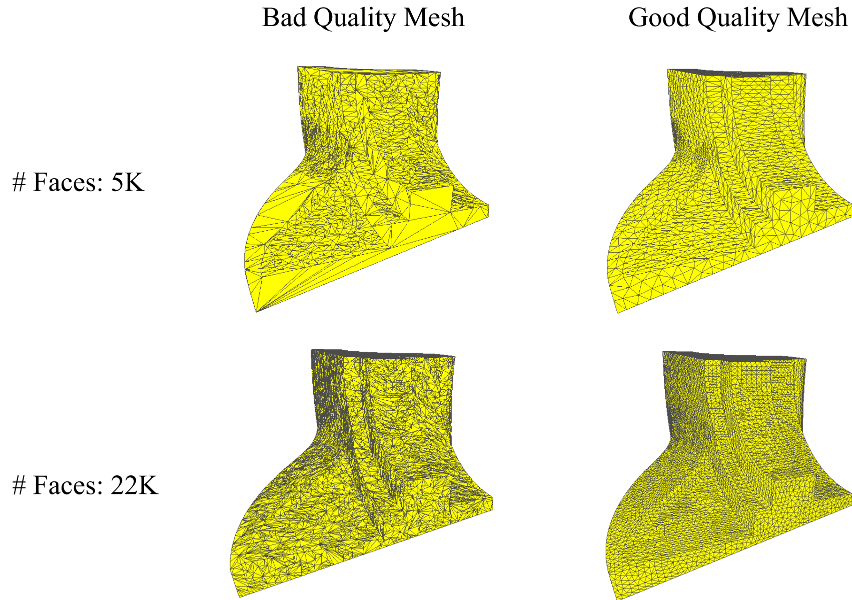
1.2 Motivation

The research on geodesics is motivated by both the theoretical studies and the practical applications:

- Theoretically, finding the geodesics is an important topic in both differential geometry and computational geometry (Section 1.1). Thus, the analysis and improvement on geodesic computations will push forward the research frontiers of these two fields.
- In practice, the geodesic distances and paths are widely used in computer graphics. For example, Voronoi diagram construction (Liu et al., 2011; Xu et al., 2014) , remeshing (Peyré and Cohen, 2006; Liu et al., 2011), skeleton extraction (Liu et al., 2011; Liu, 2015), water wave animation (Liu et al., 2006), mesh segmentation (Lai et al., 2006), cloth simulation (Zink and Hardy, 2007; Kim et al., 2012) and point pattern analysis (Liu et al., 2011). Thus, the improvement on geodesic computations will make these applications perform better.

Within the field of geodesic computations, this thesis focuses on the *exact* geodesic computation since it yields a robust and accurate solution to the DGP problem, which cannot be achieved by the state-of-the-art approximation algorithms (Figure 1.1).

However, the exact geodesic algorithms are usually slow and consume lots of memory. In some applications, the geodesic computation is even the performance bottleneck. For example, the geodesic algorithm proposed by Surazhsky et al. (2005) is employed by Liu et al. (2011) to construct Voronoi diagrams on triangle meshes and it consumes the majority of the time and memory. Thus, it is desired to improve the exact geodesic computations on triangle meshes, which is the central topic in this thesis. In addition, an improved exact geodesic algorithm can be used as a better alternative to those used in geometry processing softwares. For example, the algorithm proposed by Xin and Wang (2009) is employed by the Computational Geometry Algorithms Library (CGAL) (Kiazyk et al., 2016).



Model Size (# Faces)	Mesh Quality	Fast Marching Method		Heat Method	
		Average Error	Max Error	Average Error	Max Error
5k	Good	2.16%	12.67%	3.52%	25.12%
22k		1.08%	13.59%	0.74%	40.25%
5k	Bad	3.97%	40.67%	Failed	
22k		3.33%	93.33%	Failed	

FIGURE 1.1: Accuracy of the fast marching method (Kimmel and Sethian, 1998) and the heat method (Crane et al., 2013) against the mesh quality. The results show that these two algorithms produce large errors and may even fail on bad quality meshes.

1.3 Contributions

This thesis presents a new technique named *Edge-based Windows Grouping* (EWG) to improve the exact geodesic computation on triangle meshes, where the *window* is a basic data structure encoding the geodesic information. The contribution of the EWG technique is:

- **A novel structure to process nearby windows in batches.** The EWG technique groups all the windows on an edge into one or two window lists associated with the edge, and processes the window lists instead of the individual windows.

Based on the EWG technique, a novel exact geodesic algorithm is proposed to solve the SS-DGP problem on triangle meshes. Its contributions are:

- **A new geodesic computation framework based on EWG.** The proposed geodesic algorithm has an $O(n^2)$ time complexity. It runs 4-15 times faster than the MMP (Surazhsky et al., 2005) and ICH (Xin

and Wang, 2009) algorithms, 2-4 times faster than the FWP-MMP and FWP-CH algorithms (Xu et al., 2015), and also incurs the least memory usage.

- **A complete list of scenarios for pairwise window cross checking and pruning inside a triangle.**
- **A set of rules and algorithms for window list propagation within a triangle.**

Then, the proposed exact geodesic algorithm is revised to improve the geodesic-based Voronoi diagram construction on triangle meshes. Its contributions are:

- **A novel Redundant Window Removal (RWR) method to remove redundant windows during the Voronoi diagram construction.**
- **The high efficiency of Voronoi diagram construction.** The proposed method runs 3-8 times faster than Liu et al.'s (2011) method, 1.2 times faster than its FWP-MMP variant and more importantly uses 10-70 times less memory than both of them, which is ideal for large scale models.

1.4 Thesis Outline

This chapter presents the background and motivation of the exact geodesic computation, and lists the contributions of this work. The following chapters of this thesis is organized as follows:

Chapter 2 reviews the related works of discrete geodesic algorithms and Voronoi diagrams.

Chapter 3 defines the geodesic problems to be solved, introduces the preliminaries to understand the following chapters and proposes the key technique, *Edge-based Windows Grouping* (EWG), in this thesis.

Chapter 4 proposes a novel exact geodesic algorithm to solve the SS-DGP problem on triangle meshes based on the EWG technique. This chapter shows how EWG is applied to improve the exact geodesic computation by reducing its redundancy and management cost.

Chapter 5 shows how the proposed exact geodesic algorithm is revised to improve the geodesic-based Voronoi diagram construction on triangle meshes.

Chapter 6 concludes this thesis and suggests some possible future directions of the current work.

1.5 List of Publications

The research of this thesis has led to the following publications in peer reviewed journals and conferences:

- Yipeng Qin*, Xiaoguang Han*, Hongchuan Yu, Yizhou Yu, and Jianjun Zhang (*Joint first authors). 2016. Fast and exact discrete geodesic computation based on triangle-oriented wavefront propagation. ACM Trans. Graph. 35, 4, Article 125 (July 2016), 13 pages (Link).
- Yipeng Qin, Hongchuan Yu, Jianjun Zhang. 2017. Fast and Memory-Efficient Voronoi Diagram Construction on Triangle Meshes. Computer Graphics Forum, 36, 5 (July 2017), 12 pages (to appear).

Chapter 2

Related Work

Computing discrete geodesic distance over triangle meshes is one of the fundamental problems in computational geometry and computer graphics (Section 1.2). This chapter reviews the related works on the discrete geodesic algorithms (Section 2.1) and the Voronoi diagram on surfaces (Section 2.2), which is an important application of the discrete geodesic algorithms.

2.1 Discrete Geodesic Algorithms

This section reviews important discrete geodesic algorithms hierarchically by classifying them according to their approaches, cost metrics and object types (Figure 2.1). Without loss of generality, let $M = (V, E, F)$ be a triangle mesh representing a polyhedral surface, where V, E, F are respectively the sets of vertices, edges and faces. Let s be the geodesic source, t be the geodesic destination and n be the number of vertices of mesh M.

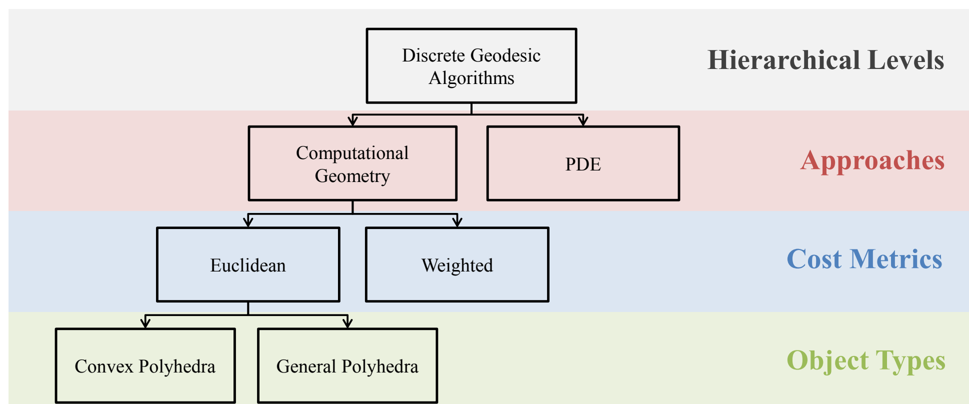


FIGURE 2.1: The structure of related works on discrete geodesic algorithms.

2.1.1 Computational Geometry Approach

In computational geometry, the discrete geodesic paths on mesh M consist of straight line segments passing through faces of M and their lengths (geodesic distances) rely on the employed cost metric. Hence, Section 2.1.1.1 reviews the discrete geodesic algorithms under an important special cost metric, the *Euclidean cost metric*, and Section 2.1.1.2 reviews those under a more general cost metric, the *Weighted cost metric*.

2.1.1.1 Euclidean Cost Metric

Employing Euclidean distances as the cost metric, the shortest path between any two points on a face $f \in M$ is a straight line segment since f is a planar triangle. More generally, the shortest path between any two points on different faces of mesh M is a polyline $l = (A_1, A_2, A_3, ..., A_n)$, where $A_i (1 \leq i \leq n)$ are the vertices of the polyline. Note that despite the first vertex A_1 and the last vertex A_n, the inner vertices $A_i (2 \leq i \leq n-1)$ reside on:

(1) *Edges* for convex polyhedra (Sharir and Schorr, 1986);

(2) *Vertices* or *edges* for general polyhedra (Mitchell et al., 1987).

Due to this difference, the discrete geodesic algorithms which operate on *convex polyhedra* and *general polyhedra* (possibly non-convex) are reviewed respectively as follows.

Convex Polyhedral Surface According to the methods used, the discrete geodesic algorithms for convex polyhedral surfaces are classified into four categories as follows.

(A) *Ridge theory based algorithms* Sharir and Schorr (1986) proposed an algorithm to compute discrete geodesics on convex polyhedral surfaces in $O(n^3 \log n)$ time. Their key idea is to subdivide mesh M into $O(n)$ disjoint "peels" in the pre-processing stage, and unfold each "peel" into a common plane on which the geodesic computations are performed. This subdivision process relies on the *ridge theory*: a point p on mesh M is defined as a *ridge* point if there exists more than one shortest paths from the source point s to p. Furthermore, they proved that:

- The shortest paths on M cannot pass through vertices or ridge points;

- Ridge points form finite many straight line segments;
- The set R of vertices and ridge points of M is closed, connected and contains no closed path.

Thus, R is a tree whose leaves are vertices of M and it is the boundary of the subdivided peels of M. Algorithmically, the construction of R follows the idea of the Dijkstra's algorithm (Dijkstra, 1959) which organises events in a priority queue and processes them from near to far. Let $\sigma = (\theta_1, \theta_2)$ be a peel in a polar coordinate system with the source point s as the origin. In the initialisation process, the peels adjacent to s are created and inserted into the priority queue. In the main loop, the peel σ on edge e closest to source point is extracted from the priority queue and trimmed with other peels on e. Then, the child peel of σ is inserted into the priority queue. The loop stops when the priority queue is empty. At this point, mesh M is subdivided into $O(n)$ peels and the shortest path query between any destination point t and the source point s can be answered in $O(n)$ time. This construction procedure costs $O(n^3 \log n)$ time and uses $O(n^2)$ space to store the peels tree R. The limitations of Sharir and Schorr's algorithm are:

- Only on convex polyhedra that the ridge points form straight line segments. Thus, it is difficult to extend this algorithm to general polyhedra.
- A $O(n^3 \log n)$ pre-processing procedure is required, which is difficult to implement and time-consuming.

To improve the performance of Sharir and Schorr's algorithm, Mount (1985a) further investigated the ridge theory and observed that the constructed peels are the Voronoi diagram of a set P containing the unfolded images of the source point s. Since the image points in P and the peels are in one-to-one correspondence, the size of P in a face is bounded by $O(n)$. Based on this observation, Mount's algorithm avoids explicitly generating ridges and represents the peels using P's Voronoi diagram. In this way, Mount's algorithm improved the time complexity of the pre-processing stage in Sharir and Schorr's algorithm to $O(n^2 \log n)$ using $O(n^2)$ space. After pre-processing, the shortest path from the source point s to any destination point $t \in M$ can be constructed in $O(k + \log n)$ time by locating t in P's Voronoi diagram and back-tracing, where k is the number of faces passed by the shortest path. For the $O(n^2)$ space cost to store P's Voronoi diagram, Mount showed that

it can be reduced to $O(n \log n)$ by efficiently storing $O(n)$ different but similar lists with $O(n)$ elements each. However, this technique is applied after constructing P's Voronoi diagram and thus the peak memory cost of Mount's algorithm is still $O(n^2)$. To reduce it, Mount (1987) proposed to record the ordered incidence of geodesics on an edge by a tree structure, and thus the overall memory cost of Mount's algorithm is reduced to $O(n \log n)$ by sharing common sub-trees.

(B) ***Path mapping algorithms*** Hershberger and Suri (1998) proposed an algorithm to construct an approximate shortest path between two points on mesh M in $O(n)$ time with bounded error factor of 2. Their main idea is to build a simplified representation of mesh M, on which the approximate shortest path is computed and mapped back to M without increasing the length. In more details, mesh M is approximated using *wedges*, which is extended from the *bounding box* and is independent to M's complexity. Note that the wedges always contain M from convexity. Let f_s and f_t be the two faces contain the source point s and the destination point t on mesh M. Let H_s and H_t be the two planes defined by f_s and f_t. According to the dihedral angle $\alpha(H_s, H_t)$, their algorithm contains two cases:

(1) $\alpha(H_s, H_t) \geq \pi/3$. In this case, the wedge is constructed as a 2-plane wedge $W(H_s, H_t)$. Then, the approximate shortest path is constructed on $W(H_s, H_t)$ and mapped to mesh M.

(2) $\alpha(H_s, H_t) < \pi/3$. In this case, multiple 3-plane wedges are constructed by combining the *horizon plane* $H_i (1 \leq i \leq k, k \leq n)$ with H_s and H_t as $W(H_s, H_i, H_t)$. Here, the *horizon plane* is defined as follows. First, divide all the faces of mesh M into two groups according to the sign of $\vec{i_z} \cdot \vec{n_f}$, where $\vec{i_z}$ is the unit vector along positive z-axis and $\vec{n_f}$ is the outer normal of a face f. Then, the edges which form the boundary between the two groups are defined as the *horizon edges* and the vertical planes passing through them are defined as the *horizon planes*. Obviously, the number of horizon planes $k \leq n$. For all the constructed wedges, compute the shortest path distances on them. Then, map the path providing the minimum distance to mesh M.

Their algorithm can be extended to solve the SS-DGP geodesic problem under the same error bound in $O(n \log n)$ time by building an approximate shortest path tree.

Although Hershberger and Suri's algorithm is simple and achieves linear time complexity, its error bound factor of 2 is high. To this end, Har-Peled et al. (1996) extended Hershberger and Suri's algorithm to achieve an improved error bound factor as $(1+\epsilon)$, where $0<\epsilon<1$. Their main idea is to construct a closer approximation G for mesh M with a grid lattice rather than the wedges used by Hershberger and Suri (1998). To construct the approximate mesh G, mesh M is first expanded to M' by a factor $r \approx \epsilon^{1.5}d$, where d is the approximate shortest distance between the source point s and the destination point d computed by Hershberger and Suri's algorithm. Then, G is constructed between M and M' that $M \subseteq G \subseteq M'$. Since M is in the interior of G, the approximate shortest paths on G cannot be smaller than the true shortest ones on M. Similar to Hershberger and Suri's algorithm, the approximate shortest path on G is mapped to mesh M without increasing the length. However, at the cost of flexible error bounds, the time complexity of Har-Peled et al.'s algorithm depends on both n and ϵ as:

$$O(n \cdot \min\{1/\epsilon^{1.5}, \log n\} + 1/\epsilon^{4.5} log(1/\epsilon))$$

Their algorithm can also be extended to solve the SS-DGP geodesic problem in $O(n/\epsilon^{4.5}(\log n + \log 1/\epsilon))$ time. Agarwal et al. (1997) improved Har-Peled et al.'s algorithm by constructing another approximate mesh Q between M and M' using the approximation scheme proposed by Dudley (1974). Their algorithm achieves a time complexity of $O(n\log(1/\epsilon) + 1/\epsilon^3)$ for the approximate shortest path construction between two points on M and $O(n/\epsilon^3 + (n/\epsilon^{1.5})\log n)$ for solving the SS-DGP geodesic problem. Agarwal et al.'s algorithm is further improved by Har-Peled (1999a) with an $O(n)$ pre-processing procedure. After pre-processing, the time complexity achieves $O(\log n/\epsilon^{1.5} + 1/\epsilon^3)$ for the approximate shortest path construction between two points on M and $O(n(1 + \log n/\epsilon^{1.5} + 1/\epsilon^3))$ for solving the SS-DGP geodesic problem.

(C) *Graph-based algorithms* Agarwal et al. (2002) proposed an graph-based algorithm to compute the approximate shortest path between two points on mesh M with an error bound factor of $(1+\epsilon)$. Their algorithm contains three major steps:

(1) Construct a graph G in the vicinity of mesh M in $O(n/\sqrt{\epsilon})$ time with $O(1/\epsilon^4)$ space. This makes the size of G independent from M;

(2) Construct a shortest path on G and project it to M in $O(n/\epsilon)$ time;

(3) Refine the resulting path heuristically to improve its quality.

Their experiments show that constructing the graph by a very coarse grid and applying their shortcutting heuristic yield the best result.

(D) *Conforming subdivision algorithms* Schreiber and Sharir (2007) proposed an exact algorithm to solve the SS-DGP geodesic problem in optimal time $O(n \log n)$ using $O(n \log n)$ space. Their algorithm follows the continuous Dijkstra paradigm (Mitchell et al., 1987) which propagates the "wavefront" from the source point s over the mesh M from near to far. The key point of their algorithm is to employ the *conforming subdivision* technique (Hershberger and Suri, 1999) and generalize it to 3D polyhedral surfaces. This generalized technique constructs an oct-tree-like structure on vertices of mesh M and uses it to control the wavefront propagation. After the propagation, an implicit representation of the shortest paths on mesh M is computed, and the shortest path queries between any destination point t and the source point s can be reported in $O(\log n + k)$ time, where k is the number of faces passed by the shortest path. Schreiber (2009) extended this algorithm to three kinds of *realistic polyhedron* which are possibly non-convex:

(1) *Terrain polyhedron* T. The maximum face slope of T is bounded by a fixed constant.

(2) *Uncrowded polyhedron* U. Each axis-parallel square of side length l whose distance to any vertex of U is at least l is intersected by at most a constant number of faces of U.

(3) *Self-conforming polyhedra* S. For each edge e of S, the number of faces within the shortest path distance $O(\|e\|)$ from e is bounded by a constant.

General Polyhedral Surface According to the methods used, the discrete geodesic algorithms for general (possibly non-convex) polyhedral surfaces are classified into four categories as follows.

(A) *Window propagation algorithms* Mitchell et al. (1987) proposed an important discrete geodesic algorithm, namely the MMP (Mitchell-Mount-Papadimitriou) algorithm, to solve the SS-DGP problem in $O(n^2 \log n)$ time and $O(n^2)$ space. Their algorithm employs the *continuous Dijkstra* technique extended from the Dijkstra's algorithm (Dijkstra, 1959), and propagates the

"wavefront" from the source point s over mesh M from near to far. The wavefront consists of *intervals*, which is similar to the *peels* (Sharir and Schorr, 1986) and encodes the geodesic information of an unfolded face sequence from the source point s to an edge. That is, for any two points p and q in an interval, the shortest paths from the source point s to p and q pass through the same face sequence. Note that the face sequences may contain saddle vertices as *pseudo-sources* since the object mesh may be non-convex. Then, the SS-DGP problem is solved by keeping track the intervals on edges during the wavefront propagation. To minimize the redundancy among intervals on an edge, Mitchell et al. suggested to trim the intervals into disjoint ones and organize them in an ordered list according to their positions. The algorithm terminates when each edge of the mesh is subdivided into a list of end-to-end linked intervals and these intervals contain the geodesic information of any point on the mesh. With such subdivisions, the geodesic path queries can be reported in $O(k + \log n)$ time, where k is the number of faces passed by the shortest path.

Surazhsky et al. (2005) implemented the MMP algorithm (Mitchell et al., 1987) and tested it on various laser-scanned models. In their implementation, the *intervals* used by Mitchell et al. are called *windows*, and the trimming procedure between windows is fulfilled by solving a quadratic equation. Their experimental results show that the MMP algorithm runs in sub-quadratic time, which is much faster than the worst-case time complexity of $O(n^2 \log n)$. In addition, they proposed an approximate version of the MMP algorithm by merging adjacent windows under bounded error during the wavefront propagation. This approximation algorithm is compared to the popular fast marching method (Kimmel and Sethian, 1998) and the results show that it outperforms the fast marching method in both the running time and the accuracy.

Since then, several incremental works are done to improve the performance of the MMP algorithm. Liu et al. (2007) observed that the degenerate cases induced by floating point calculation occur frequently when applying the MMP algorithm to real world models. Thus, they systematically analysed all the degenerated cases and handled them accordingly, which makes the MMP algorithm more robust. Addressing the redundancy caused by the employed data structure, Liu (2013) proposed to use an edge-based data structure rather than the half-edge data structure (Surazhsky et al., 2005) in implementing the MMP algorithm. Experimental results show that the running time is

reduced by 44% and the memory cost is reduced by 29% on average. Observing that the priority queue costs the majority of the running time in the MMP algorithm, Xu et al. (2015) proposed to replace it with a bucket structure. As a result, their method runs 3-10 times faster than the original MMP algorithm.

Chen and Han (1990) proposed an important algorithm, namely the CH (Chen-Han) algorithm, to solve the SS-DGP problem using an approach different from the continuous Dijkstra technique. Their main idea is to build a sequence tree using the *First-In-First-Out* (FIFO) queue, and each node in the sequence tree contains the geodesic information of an unfolded face sequence where geodesic paths reside. Note that the sequence tree can be viewed as a dual of the peels structure proposed by Sharir and Schorr (1986). However, the size of the tree can be exponential without an effective redundancy reduction rule. To this end, Chen and Han proposed the "one angle, one split" rule to remove the redundant nodes at vertices of the object mesh. With this rule, they proved that the sequence tree contains only $O(n)$ leaf nodes for convex polyhedra and at most additional $O(n)$ leaf nodes (caused by saddle vertices) for non-convex polyhedra. Thus, the sequence tree can be built in $O(n^2)$ time using $O(n)$ space, which are the best asymptotic complexities for the exact discrete geodesic algorithms so far. After building the sequence tree, the geodesic path queries can be reported in $O(k + \log n)$ time, where k is the number of faces passed by the shortest path. The CH algorithm is first implemented by Kaneva and O'Rourke (2000). Their experimental results verify the $O(n^2)$ time complexity and $O(n)$ space complexity in practice. However, Surazhsky et al. (2005) reported that their implementation of the MMP algorithm is many times faster than Kaneva and O'Rourke's implementation of the CH algorithm, although having a larger time complexity $O(n^2 \log n)$.

Investigating in this abnormal phenomenon, Xin and Wang (2009) proposed the ICH (Improved Chen-Han) algorithm as an improved version of the CH algorithm. In their algorithm, the sequence tree nodes are called *windows* as Surazhsky et al. (2005) did. Observing that more than 99% of the windows generated in the CH algorithm are useless, they proposed an effective window filtering rule to filter out such useless windows. Furthermore, they employed the continuous Dijkstra technique used in the MMP algorithm (Mitchell et al., 1987) to organize the windows by a priority queue. Note that introducing the priority queue increases the time complexity of their algorithm to $O(n^2 \log n)$ and makes it difficult to prove that its space complexity is still $O(n)$. However, their experimental results show that their method greatly outperforms

the CH algorithm and runs comparable to the MMP algorithm while using considerably less space. Ying et al. (2014) proposed a parallel version of the ICH algorithm, which runs an order of magnitude faster than the serial ICH algorithm. Since the ICH algorithm also employs the continuous Dijkstra technique, its priority queue costs the majority of the running time as the MMP algorithm does. To this end, Xu et al. (2015) proposed to replace it with a bucket structure and and achieves a speed-up factor of 2-5.

(B) ***Graph-based algorithms*** O'Rourke et al. (1985) extended Sharir and Schorr's (1986) algorithm to solve the DGP problem on general polyhedra, only requiring the surfaces to be orientable. Assuming both the source and destination points to be vertices of mesh M, their algorithm is performed in two steps:

(1) Build a vertex-to-vertex graph by computing the shortest *straight-line distances* between all pairs of vertices on mesh M. The *straight-line distance* is the length of a path between two vertices that the path is a straight line on the unfolded face sequence (Sharir and Schorr, 1986) and does not contain any intermediate vertices.

(2) Construct the shortest path by searching the vertex-to-vertex graph.

This algorithm has a high time complexity of $O(n^5)$. Thus, it has little practical value and has not been implemented so far. However, it shows that the DGP problem on general polyhedra can be solved in polynomial time.

Similar to O'Rourke et al.'s idea, Balasubramanian et al. (2009) proposed a discrete geodesic algorithm to solve the AP-DGP problem. Their algorithm also constructs the vertex-to-vertex graph using the minimal geodesic distances, i.e. the straight-line distances used by O'Rourke et al. (1985), and computes the all-pairs shortest distances by searching the graph. Although the theoretical time complexity of their algorithm is exponential, they observed that its practical running time is no more than $O(n^3)$.

Ying et al. (2013) proposed an approximation algorithm to answer frequent shortest path queries on a mesh by searching a related graph. Compared to the graphs constructed by O'Rourke et al. (1985) and Balasubramanian et al. (2009), their graph is sparser since they only compute the shortest distances in the user-defined geodesic disks around each vertex rather than the entire mesh. The shortest distances in these geodesic disks are computed by either the MMP algorithm (Surazhsky et al., 2005) or the ICH algorithm

(Xin and Wang, 2009). Let K be the user-defined size of each geodesic disk ($K < n$), their graph can be constructed in $O(nK^2 \log K)$ time. Then, the shortest path queries can be answered by searching the constructed graph. Not counting the graph construction time, their experimental results show that their method outperforms the state-of-the-art approximate geodesic algorithms (Kimmel and Sethian, 1998; Surazhsky et al., 2005; Crane et al., 2013) in both the running time and the accuracy.

To answer the open problem raised by Agarwal et al. (1997) that whether the DGP problem on general polyhedra can be solved approximately in sub-quadratic time, Varadarajan and Agarwal (2000) proposed two algorithms:

(1) The first one runs in $O(n^{5/3} \log^{5/3} n)$ time with an error bound factor of $7(1 + \epsilon)$;

(2) The second one runs slightly faster in $O(n^{8/5} \log^{8/5} n)$ time but has an larger error bound factor of $15(1 + \epsilon)$.

In these two algorithms, mesh M is first partitioned into $O(n/r)$ patches and each patch contains at most r faces, where r is a chosen parameter. Then, for each patch P_i, a set of points are added to its boundary and the approximate shortest paths between all pairs of such points form a graph G_i. Finally, these graphs G_i are merged to form a global graph G. By ensuring that the source and destination points are vertices of G, the shortest path between them is constructed by performing the Dijkstra's algorithm (Dijkstra, 1959) on G.

(C) ***Iterative refining algorithms*** Kanai and Suzuki (2001) proposed an approximation algorithm to solve the DGP problem on general polyhedra by iteratively refining an rough initial path. In each iteration, their algorithm performs the following three steps:

(1) Perform the Dijkstra's algorithm (Dijkstra, 1959) on the graph G_i to construct a path p_i between the source and destination points.

(2) Find the region R_i where the shortest path may reside based on p_i.

(3) Refine the region R_i by adding extra points on its edges and form the new graph G_{i+1}.

Let G_0 be the graph of the mesh M, the algorithm operates by replacing G_i by G_{i+1} iteratively. As the region R_i becomes narrower, the constructed path p_i

becomes closer to the shortest path. In their algorithm, the trade-off between accuracy and performance depends on the number of extra points added to each edge, which is user-defined. The experimental results show that their algorithm runs 100 to 1000 times faster than the CH algorithm (Chen and Han, 1990) with 0.4% error.

(D) ***Others*** Kapoor (1999) proposed a controversial algorithm to solve the DGP problem on general polyhedra and claimed that it runs in $O(n \log^2 n)$ time using $O(n)$ space. This algorithm also employs the continuous Dijkstra technique and maintains the wavefront using a sequence of circular arcs. However, according to O'Rourke (1999) and Surazhsky et al. (2005), the details of this algorithm is too complex because it calls many other complicated computational geometry algorithms as subroutines. Thus, it is not widely accepted by academia and may never been implemented.

Har-Peled (1999b) generalized his previous algorithm (Har-Peled, 1999a) to solve the SS-DGP problem on general polyhedra approximately. Let ϵ be the error bound factor ($0 < \epsilon \leq 1$), his main idea is to construct a subdivision of mesh M of size $O((n/\epsilon) \log(1/\epsilon))$ such that the geodesic distance queries can be reported in $O(\log(n/\epsilon))$ time. The time complexity of this construction is:

(1) $O(n^2 \log n + (n/\epsilon) \log(1/\epsilon) \log(n/\epsilon))$ for general polyhedra;

(2) $O((n/\epsilon^3) \log 1/\epsilon + (n/\epsilon^{1.5}) \log(1/\epsilon) \log n)$ for convex polyhedra.

However, the performance of this algorithm on general polyhedra can never be better than the MMP algorithm (Mitchell et al., 1987) since it uses the exact shortest distance map generated by MMP as an input.

2.1.1.2 Weighted Cost Metric

To model the difficulty of the paths passing through some pieces of a surface, the faces of a mesh are required to be weighted in some applications. For example, in the path planning task of computer games, walking on grass lands or mountains usually costs more time than walking on flat roads. To meet this demand, a weight w_i is associated with each face f_i of the mesh M. Then, the length of the sub-path crossing face f_i is multiplied by w_i, and the length of a path is computed by summing up the lengths of all its sub-paths. Due to the diversity of the weighting schemes, it is difficult to compute the geodesic paths on weighted polyhedral surfaces. According to the methods used, the

discrete geodesic algorithms for weighted polyhedral surfaces are classified into two categories as follows.

(A) ***Refraction-based algorithms*** Mitchell and Papadimitriou (1991) proposed an approximation algorithm to solve the SS-DGP problem on weighted polyhedral surfaces with an error bound factor of $(1+\epsilon)$ that $\epsilon > 0$. Their main idea is to employ the continuous Dijkstra technique (Mitchell et al., 1987) and apply the *Snell's Law of Refraction* from the optics theory to control the direction of a path crossing an edge of a mesh. The rationale behind this application of the *Snell's Law* is that the path light traverses is always the one with the minimum propagation time (the *Fermat's Principle*). Following the *Snell's Law*, a shortest path from the source point is viewed as a ray of light and achieves local optimality by being "bent" at mesh edges. Their algorithm runs in $O(ES)$ time using $O(E)$ space, where E is the number of "events" processed by the continuous Dijkstra algorithm and is bounded by $O(n^4)$; S is the time cost of performing a numerical search procedure to find a $(1+\epsilon)$-approximate path within a given face sequence and is bounded by $O(n^4 \log \frac{nNW}{\epsilon w})$, where N is the maximum integer coordinate of any vertex of the mesh, W and w are the maximum and minimum finite integer weights assigned to faces of the mesh respectively. Thus, the worst-case time cost of their algorithm is $O(n^8 \log \frac{nNW}{\epsilon w})$ and the worst-case memory cost is $O(n^4)$. However, since it is a worst-case analysis, their algorithm may perform much better in practice.

(B) ***Graph-based algorithms*** Mata and Mitchell (1997) proposed an approximation algorithm to solve the DGP problem on a weighted polyhedral surface by constructing and searching a graph. In their algorithm, the graph is constructed in two steps:

(1) k evenly-spaced cones are defined to approximate the round angle of each vertex v of mesh M.

(2) Within each cone of v, at most one *link* is built to connect it with another vertex or a critical point on some edge of M.

By searching the constructed graph using the Dijkstra's algorithm (Dijkstra, 1959), approximate shortest paths with an error bound factor of $(1+O(\frac{W}{wk\theta}))$ can be constructed, where W are w are the maximum and minimum finite integer weights assigned to faces of the mesh respectively, θ is the minimum

interior face angles of the mesh. The size of the graph is $O(kn)$ and its construction costs $O(kn^3)$ time. To test its performance, their algorithm is implemented and compared with some other simple heuristic algorithms. The experimental results show that their algorithm outperforms others in both the query time and the accuracy.

Lanthier et al. (1997, 2001) proposed another graph-based approximation algorithm with three variants to solve the DGP problem on weighted polyhedral surfaces. Compared to Mata and Mitchell's algorithm which links vertices or critical points by approximating the round angles, their algorithm constructs the graph by placing and linking extra points on edges of the mesh M. According to the point-placing scheme employed, their algorithm has three variants:

(1) *Fixed scheme*. This scheme places k points evenly on each edge of mesh M, where k is a positive integer. For each face $f_i \in M$, a graph G_i is constructed by linking every pair of points (or vertices) on different edges of f_i. The final graph G is constructed by merging all the face graphs G_i. By applying the Dijkstra's algorithm (Dijkstra, 1959) on G, an approximate shortest path between two points on M with an additive error bound factor of $W|L|$ can be constructed in $O(n^5)$ time, where W is the maximum weight assigned to faces of M and L is the longest edge of M.

(2) *Interval scheme*. This scheme can be viewed as an improved version of the *Fixed Scheme*. Instead of placing a fixed number of points on each edge, this scheme places points on edges with a fixed interval length, e.g. $|L|/(k+1)$, where L is the longest edge of M and k is a positive integer. As a result, the number of placed points is considerably reduced in practice. However, the worst case analysis of this scheme remains the same as the *Fixed Scheme*.

(3) *Spanner scheme*. To improve the time complexity of the *Fixed Scheme* and the *Interval Scheme*, Lanthier et al. proposed to reduce the number of links between placed points in each face, which is implemented by introducing the notion of the *spanner* (Clarkson, 1987). As a result, the algorithm's time complexity is improved to $O(n^3 \log n)$ at the cost of a larger approximation error. In more details, the length of an obtained approximate path is increased to $\beta p + W|L|$, where $\beta > 1$, p is the

length of the exact shortest path, W is the maximum weight assigned to faces of M and L is the longest edge of M.

Their experimental results on typical terrain data show that adding a smaller constant number of points (i.e. six) per edge is sufficient to obtain a near-optimal result and thus the practical running time of their algorithm is reduced to $O(n \log n)$.

Lanthier et al. (2003) proposed a parallel implementation of their algorithm (Lanthier et al., 1997, 2001) to speed up the computation. In their implementation, a general spatial indexing storage structure, namely *multidimensional fixed partition*, is employed to achieve load balancing and reduce the processors' idle time. To make a fair performance evaluation, they tested the implementation on different platforms including, a network of workstations, a Beowulf cluster and a symmetric multiprocessing architecture.

Given an error bound factor $(1+\epsilon)$ that $\epsilon > 0$, Aleksandrov et al. (1998) proposed a graph-based approximation algorithm to solve the DGP problem on weighted polyhedral surfaces. Similar to the idea of Lanthier et al. (1997, 2001), their algorithm places extra points on edges of mesh M to generate an augmented graph G. In more details, they place $k = O(\log_\delta |L|/r)$ points on each edge of M in a geometric progression manner, where L is the longest edge of M, r equals the minimum distance from any vertex of M to the boundary of the union of its incident faces times ϵ, θ is the minimum of all the face angles of M and $\delta \geq 1 + \epsilon \sin\theta$. Then, the approximate shortest paths are obtained by performing the Dijkstra's algorithm (Dijkstra, 1959) on G. The time cost of their algorithm is $O(mn \log mn + nm^2)$, where m is the total number of extra points added. To improve this algorithm, Aleksandrov et al. (2000) proposed to:

(1) Reduce the number of points placed by modifying the point placing scheme. Let p_i, p_{i+1} be two adjacent points placed on an edge $e \in M$ in a geometric progression manner, x be a point on the boundary of the union of e's two adjacent faces. Then, it is required that the angle $\angle axb < \frac{\pi\epsilon}{2}$.

(2) Prune the search of the Dijkstra's algorithm using the *Snell's Law*. Let a_i be a node in the graph and a_{i-1} be its preceding node in a_i's shortest path. Then, the next node to be updated from a_i must be in the *geodesic cone* of $a_{i-1}a_i$ defined by the *Snell's Law*.

As a result, the time cost is reduced to $O(\frac{n}{\epsilon}\log\frac{1}{\epsilon}(\frac{1}{\sqrt{\epsilon}}+\log n))$ while the error bound factor is still $(1+\epsilon)$.

Sun and Reif (2006) improved the algorithm proposed by Aleksandrov et al. (2000) by replacing the pruned Dijkstra's algorithm with their *BUSH-WHACK* algorithm to search the constructed graph G. Similar to Aleksandrov et al.'s idea, they dynamically maintain a small number of incident edges to each vertex of G which may contribute to a shortest path. However, compared to the overlapping geodesic cones used by Aleksandrov et al. (2000), the *intervals* used in their algorithm are mutually exclusive. Thus, the search space of finding a shortest path is further reduced and the time cost of their algorithm is $O(\frac{n}{\epsilon}\log\frac{1}{\epsilon}(\log\frac{1}{\epsilon}+\log n))$. In addition, they improved the point placing scheme used by Aleksandrov et al. (2000) to make the time complexity of their algorithm independent of $\frac{W}{w}$, where W and w are the maximum and minimum weights assigned to faces of the mesh respectively.

Aleksandrov et al. (2005) improved their previous algorithm (Aleksandrov et al., 2000) by introducing a new point-placing scheme. In their algorithm, the extra points are placed in a geometric progression way on the *bisectors* rather than the edges of each face of the object mesh M. They proved that:

- For a bisector l of angle α at a vertex v, the number of extra points placed on it is bounded by $\frac{1.61}{\sin\alpha}\log_2\frac{2|l|}{r(v)}\frac{1}{\sqrt{\epsilon}}\log_2\frac{2}{\epsilon}$, where $r(v)$ represents the weighted radius $r(v)$ for each face incident to v.
- The total number of extra points placed is bounded by $C(P)\frac{n}{\sqrt{\epsilon}}\log_2\frac{2}{\epsilon}$, where $C(P) < 4.83\Gamma\log_2 2L$, L is the maximum of the ratios $\frac{|l(v)|}{r(v)}$ among all vertices $v\in M$, Γ is the average of the reciprocals of the sinuses of angles on mesh M.

As a result, their algorithm runs in $O(C(P)\frac{n}{\sqrt{\epsilon}}\log\frac{n}{\epsilon}\log\frac{1}{\epsilon})$ time with the same error bound $(1+\epsilon)$. Based on this algorithm, Aleksandrov et al. (2010) proposed two approximation algorithms to solve the SS-DGP and AP-DGP query problems on weighted polyhedra surfaces respectively:

- To solve the SS-DGP query problem, they employed a single-source query data structure *SSQ* which can answer the shortest distance queries in logarithmic time with an error factor of $(1+\epsilon)$. The *SSQ* data structure can be constructed in $O(\frac{n}{\sqrt{\epsilon}}\log\frac{n}{\epsilon}\log\frac{1}{\epsilon})$ time using $O(\frac{n}{\sqrt{\epsilon}}\log\frac{1}{\epsilon})$ space (Aleksandrov et al., 2005).

- To solve the AP-DGP query problem, they first partition the object mesh M into regions with "small-cost" boundaries by generalizing a partitioning algorithm for planar graphs (Lipton and Tarjan, 1979). Then, an all-pairs query data structure *APQ* is constructed as a collection of the *SSQ* data structures based on the partitioned regions. With the constructed *APQ*, the shortest distance query can be answered in $O(q)$ time, where $q \in (\frac{1}{\sqrt{\epsilon}} \log^2 \frac{1}{\epsilon}, \frac{(g+1)^{2/3} n^{1/3}}{\sqrt{\epsilon}})$ is a user-defined parameter, g is the genus of the mesh. The *APQ* data structure can be constructed in $O(\frac{(g+1)n^2}{\epsilon^{3/2} q} \log \frac{n}{\epsilon} \log^4 \frac{1}{\epsilon})$ time using $O(\frac{(g+1)n^2}{\epsilon^{3/2} q} \log^4 \frac{1}{\epsilon})$ space.

2.1.2 Partial Differential Equation (PDE) Approach

The PDE-based discrete geodesic algorithms compute geodesics on polyhedral surfaces by solving discretized Partial Differential Equations (PDEs). These algorithms are usually fast and easy to implement. However, they cannot provide an exact solution. Furthermore, their approximation errors are usually unbounded and depend heavily on the mesh quality. According to the PDE solved, this section reviews two kinds of important PDE-based discrete geodesic algorithms: the ones solving the *discrete Eikonal equation* (Section 2.1.2.1) and those solving the *discrete Poisson equation* (Section 2.1.2.2).

2.1.2.1 Discrete Eikonal Equation

The Eikonal equation $|\nabla T| F = 1$ is a partial differential equation characterising the propagation of waves, where T is the time when the wavefront arrives at a point, F is the speed of the propagating wavefront. It shows that the gradient of the arrival time is inversely proportional to the speed of the wavefront (Sethian, 1996). By setting the speed of the wavefront $F \equiv 1$ on mesh M, the arrive time of each point $p \in M$ is equivalent to p's geodesic distance. According to the types of the mesh, the algorithms to solve the discrete Eikonal equation are classified into two categories as follows.

(A) ***Regular grids*** Sethian (1996) proposed an important algorithm to solve the discrete Eikonal equation on regular grids, which is known as the *Fast Marching Method* (FMM) algorithm. Their main idea is to compute the arrival time T for each vertex of the grid by propagating the wavefront outward in an "upwind" fashion. That is, the wavefront vertices are maintained

in a priority queue and the one with the minimum arrival time T is first processed. The computation of T follows the discrete Eikonal Equation,

$$[\max(\max(D_{ij}^{-x}T, 0) - \min(D_{ij}^{+x}T, 0))^2 + \max(\max(D_{ij}^{-y}T, 0) - \min(D_{ij}^{+y}T, 0))^2] = 1/F_{ij}^2$$

In the above equation, the gradient of T is discretized by the finite difference method, for example, $D_{ij}^{+x}T = (T_{i+1,j} - T_{i,j})/(\Delta x)$. The time cost of their algorithm is $O(n \log n)$.

Yatziv et al. (2006) improved the time cost of the FMM algorithm to $O(n)$ by replacing the priority queue used with a bucket structure called the *untidy queue*. The error bound of their algorithm is of the same magnitude with the original FMM algorithm.

Bertelli et al. (2006) extended the FMM algorithm to compute all-pairs distance on rectangular grids. Their key observation is that more than 90% of the distance calculations are repeated when naively running the FMM algorithm n times for each vertex. To remove such redundancy, they proposed a method to reuse the previous calculated distances. The experimental results show that their algorithm not only reduced the redundancy but also improved the accuracy of distance calculations. However, the theoretical time complexity of their algorithm remains $O(n^2 \log n)$, which is the same as naively running the FMM algorithm n times.

(B) *Triangle meshes* Kimmel and Sethian (1998) extended the FMM algorithm proposed by Sethian (1996) to solve the SS-DGP problem on triangle meshes. Similar to the ideas of Dijkstra (1959) and Sethian (1996), their algorithm computes the geodesic distances of vertices by processing them from near to far using a priority queue. Let p be a vertex to be processed, the geodesic distance of p is updated by the triangles containing processed vertices in p's 1-ring neighbourhood. This update contains two cases:

(1) The triangle contains two processed vertices. In this case, the geodesic distance of p is computed by solving the discrete Eikonal equation in the triangle, which is essentially a quadratic equation.

(2) The triangle is obtuse and contains only one processed vertex. Let a be the processed vertex. In this case, the adjacent triangles opposing the obtuse angle are unfolded to a plane until another processed vertex b is found. Then, the vertices p, a, b form a virtual triangle and the geodesic

distance of p is computed by solving the discrete Eikonal equation in this virtual triangle.

The error of the first case is $O(e_{\max})$ and that of the second case is $O(\frac{e_{\max}}{\pi-\theta_{\max}})$, where $e_{\max}$ is the length of the longest edge and $\theta_{\max}$ is the widest angle of the mesh. Thus, the accuracy of their algorithm depends on the mesh quality. For near-degenerate meshes, the error becomes unbounded and may yield poor results. After the distance field computation, the geodesic paths can be constructed by performing the gradient descent method.

Martínez et al. (2005) proposed an algorithm to refine the geodesic path generated by the FMM algorithm. Their main idea is to refined the initial path constructed by FMM iteratively by a discrete geodesic flow based on the straightest geodesics theory (Polthier and Schmies, 1998). They proved that the refined path has a shorter length and converges to a local minimum.

Xin and Wang (2007) proposed an algorithm to construct an exact locally shortest geodesic path on triangle meshes based on the FMM algorithm. Similar to the idea of Martínez et al. (2005), they proposed to construct the locally shortest path by iteratively refining a rough initial path. First, to obtain a more accurate initial path, they improved the FMM algorithm by classifying all edges into seven types to control the wavefront propagation more precisely. Second, the obtained initial path is refined by iteratively optimizing the face sequence containing the path. The optimization continues until the face sequence contains the exact locally shortest path. Note that the shortest path in a face sequence is computed by a visibility-based algorithm inspired by the MMP algorithm (Mitchell et al., 1987) rather than the discrete geodesic flow used by Martínez et al. (2005).

2.1.2.2 Discrete Poisson Equation

The geodesic distance field d on a triangle mesh M can be obtained by solving the Poisson equation $\Delta d = \nabla \cdot X$, where X is a vector field constructed from M representing the approximation of d's gradient ∇d. This is equivalent to solving the minimization problem $\int_M |\nabla d - X|$.

Xin et al. (2012) proposed an algorithm to compute the geodesic distance fields on broken meshes (e.g. containing holes, gaps and short-cuts) in an

iterative manner. The initial distance field d_0 is defined as the Euclidean distances from the source vertex to all the other vertices of the mesh. In each iteration, their algorithm contains three steps:

(1) Smooth d_i and normalize its gradient field such that $\|\nabla d_i\| = 1$.

(2) Compute the distance field d_{i+1} by solving Poisson equation with ∇d_i.

(3) Remove the minimal points other than the source in d_{i+1} by smoothing.

Their experimental results show that the computed distances are insensitive to the defects of the mesh but smoother than the geodesic distances. In addition, their algorithm is not guaranteed to converge.

Crane et al. (2013) proposed an algorithm to compute geodesic distances on triangle meshes using the Varadhan's formula, which describes the relationship between heat and geodesic distance on a Riemannian manifold as,

$$d(x, y) = \lim_{t \to 0} \sqrt{-4t \log k_{t,x}(y)}$$

where $d(x, y)$ is the geodesic distance between two points x and y, t is the time, $k_{t,x}(y)$ is the heat kernel. Let d be the geodesic distance field to be computed, their algorithm divides the geodesic computation into three steps:

(1) Compute the heat flow for a fixed time t over the mesh to approximate the gradient field ∇d of d.

(2) Normalizes ∇d such that $\|\nabla d\| = 1$.

(3) Solve the Poisson equation with ∇d to recover d.

Their algorithm can be implemented in an efficient way to run in near-linear time by pre-factoring the Laplacian matrix used. Similar to the algorithm proposed by Xin et al. (2012), the distances calculated by their method is a smoothed approximation of the geodesic distances. The experimental results show that their method runs faster than the FMM algorithm (Kimmel and Sethian, 1998) while producing similar errors.

2.2 Voronoi Diagram on Surfaces

According to Aurenhammer (1991), the Voronoi diagram of a set of sources partitions the surface into a collection of regions. Each such region is associated to a unique source s_i and all the points in the region are closer to s_i than any other sources. Thus, the Voronoi diagram is closely related to the shortest path problem. This section reviews the methods to construct Voronoi diagrams on two important types of surfaces: the 2D planes (Section 2.2.1) and the 3D polyhedral surfaces (Section 2.2.2).

2.2.1 Voronoi Diagrams on 2D Planes

The Voronoi diagrams on 2D planes are well-studied in the mid-late 20th century and three typical algorithms to construct them are reviewed as follows.

The Incremental Insertion method Green and Sibson (1978) proposed to construct the Voronoi diagram on the 2D plane by incrementally inserting sources. Figure 2.2 illustrates the process of inserting a new source v into an existing Voronoi diagram. This process contains two steps:

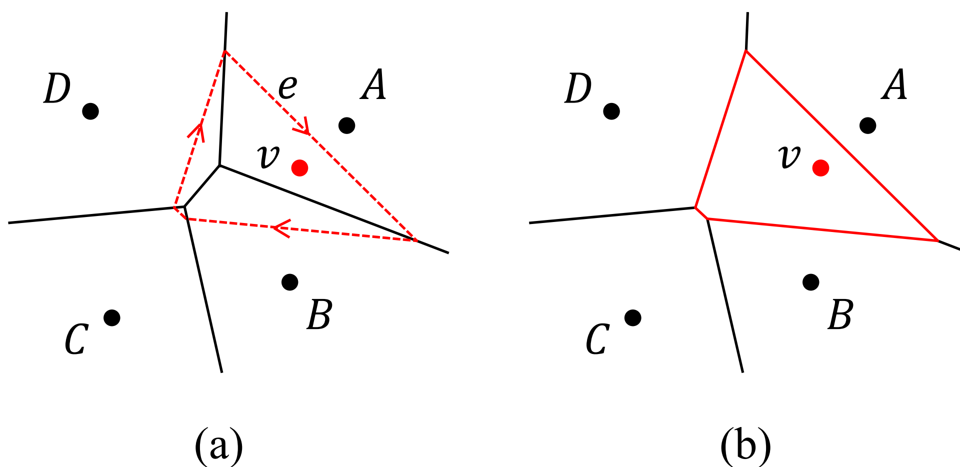


FIGURE 2.2: Illustration of the incremental insertion method. (a) Source v is inserted. (b) The resulting Voronoi diagram.

(1) First, find the nearest source of v in the existing Voronoi diagram. For example in Figure 2.2 (a), A is the nearest source to v and v falls in A's Voronoi region. The perpendicular bisector between v and A contributes to a boundary edge of v's Voronoi region, i.e. edge e.

(2) Second, the boundary of v's Voronoi region is constructed recursively edge by edge. For example in Figure 2.2 (a), the construction starts from e and is performed along the red arrows. During the construction, the Voronoi diagram is updated by traversing and removing the redundant parts of the old boundaries, which reside in v's region. Figure 2.2 (b) shows the resulting Voronoi diagram after the insertion.

Let k be the number of sources inserted. Then, the worst time complexity of the first stage is $O(k)$. However, Green and Sibson (1978) found that utilizing the boundaries of the existing Voronoi diagram as a heuristic can reduce the expect time cost to $O(\sqrt{k})$. For the second stage, its worst time complexity is also $O(k)$. In practice, Green and Sibson (1978) observed that the average number of a source's boundary edges approaches *six* in large configurations. Thus, the expected time cost of the second stage is $O(1)$. In summary, the worst time cost of this algorithm is $O(n^2)$ for n sources, while its expected time cost is $O(n^{1.5})$.

Since the performance bottleneck of the *incremental insertion method* is the nearest neighbour search when inserting a new source, Ohya et al. (1984) accelerated it by ordering the insertions appropriately. In their method, the plane is partitioned into unit square grids, and the sources are placed into these grids according to their positions. Then, the insertion of sources is ordered according to the grids' adjacency. As a result, the practical running time of the nearest neighbour search is reduced to $O(1)$ and the overall algorithm runs in $O(n)$ time for n sources, which reaches the lower bound of constructing 2D planar Voronoi diagrams.

The Divide and Conquer method The classic *divide and conquer* paradigm is widely used in designing fast algorithms. Shamos and Hoey (1975) first proposed to apply it in constructing the Voronoi diagrams on the 2D plane. In their method, the given n sources are first divided into two groups L and R by a line, each containing $n/2$ source. Suppose the Voronoi diagrams of L and R are constructed, they showed that these two Voronoi diagrams can be merged in linear time by constructing the polyline dividing them (e.g. polyline P in Figure 2.3). Note that the constructed polyline is part of the boundaries of the resulting Voronoi diagram. As a result, their method can construct the Voronoi diagram in $O(n \log n)$ time by recursively dividing the n sources. Here, $O(n \log n)$ is both the worst-case and expected time complexities.

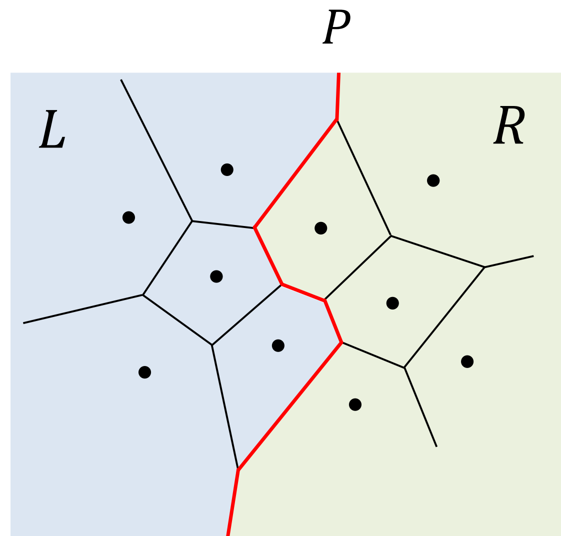


FIGURE 2.3: Illustration of the divide and conquer method.

The Plane-Sweep method Fortune (1986) proposed an algorithm to construct the Voronoi diagrams by simulating a line sweeping through the 2D plane. This line is denoted as the *sweep line* and considered as a source in the construction as well. In their algorithm, the Voronoi diagram is only constructed in the region swept by the *sweep line*. As Figure 2.4 shows, the concepts used are introduced as follows:

- **Sweep Line.** A horizontal line sweeping from top to bottom.
- **Swept Region.** A region containing the swept sources and the constructed part of the Voronoi diagram.
- **Beach Line.** Since the bisector between a line and a point is a parabola, the *beach line* is a chain of such parabola arcs between the *sweep line* and the swept sources.
- **Break Point.** The point shared by two parabola arcs.

During the sweeping, the *sweep line* moves from the top to the bottom of the plane. The *beach line* follows the *sweep line* and its structure changes when a new parabola arc is added (i.e. a new source is swept), or an old one vanishes. Then, the Voronoi diagram is constructed by tracking the *break points* since they reside on the Voronoi boundaries. In their algorithm, the moves of the *sweep line* is implemented by sorting sources' y-coordinates in a priority queue, and the *beach line* is organized using a binary tree for efficient searching and updating. Fortune (1986) proved that the the *beach*

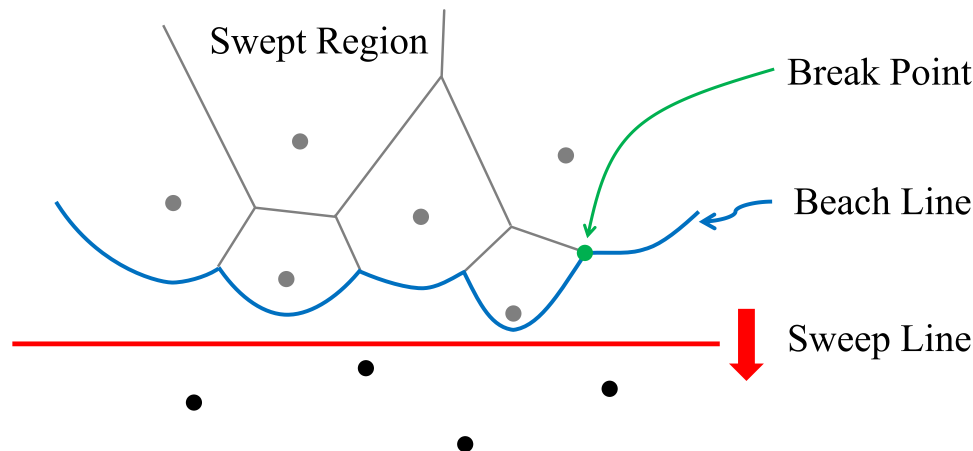


FIGURE 2.4: Illustration of the plane-sweep method.

line's structure changes at most $O(n)$ times. Thus, the time complexity of their algorithm is $O(n \log n)$.

2.2.2 Voronoi Diagrams on 3D Polyhedral Surfaces

In computer graphics and computational geometry, 3D surfaces are usually assumed to be polyhedral and represented by triangle meshes. In this context, geodesics are used as the distance metric because they reflect the intrinsic properties of the surfaces and are invariant to isometric deformations. Thus, the Voronoi diagrams on 3D polyhedral surfaces are also known as the *geodesic Voronoi diagrams*, which can be constructed by the computational geometry approach or the PDE approach.

Computational Geometry Approach Mount (1985b) first proposed an algorithm to construct the geodesic Voronoi diagrams based on the MMP algorithm (Mitchell et al., 1987). The distinct advantage of the MMP algorithm is to bring necessary geodesic information to edges of the mesh, and thus the geodesic Voronoi diagram can be constructed face by face. In such faces, the problem degenerates to the Voronoi diagram construction on 2D planes. Thus, Mount (1985b) proposed to solve it by employing the divided-and-conquer method (Shamos and Hoey, 1975). Note that the Voronoi boundaries on 3D polyhedral surfaces contain not only straight line segments but also hyperbolic segments (caused by saddle vertices), which are more complex than those on 2D planes. Let n be the number of mesh vertices, this algorithm runs in $O(n^2 \log n)$ time using $O(n^2)$ space. However, it lacks explicit details and has not been implemented so far.

Following Mount's work, Liu et al. (2011) proposed a practical algorithm to construct the geodesic Voronoi diagram on triangle meshes based on the necessary geodesic information provided by the MMP algorithm (Mitchell et al., 1987). Their algorithm is essentially a triangle-marching scheme which tracks the Voronoi boundaries. However, to make the marching scheme simple, the triangle faces must be subdivided to guarantee that each edge contains at most one intersection point with the Voronoi boundaries. Let k be the number of triangles passed by the Voronoi boundaries, their algorithm runs in $O(k \log k)$ time. In practice, since the Voronoi diagram is usually more sparse than the meshes, $k \ll n$, where n is the number of mesh vertices. Thus, the performance bottleneck of their method is the geodesic computation part which costs $O(n^2 \log n)$ time and $O(n^2)$ space.

Xu et al. (2014) extended Liu et al.'s work to construct polyline-sourced geodesic Voronoi diagrams on triangle meshes. In their algorithm, a method is proposed to reduce the memory cost of the geodesic Voronoi diagram construction. In more details, this method performs the redundancy checks repeatedly on the geodesic information stored on mesh edges. Since the cost of the redundancy check is large, performing it frequently is time-consuming. Thus, their method suffers from the trade-off between the running time and the memory-cost.

PDE Approach Kimmel and Sethian (1999) proposed to construct geodesic Voronoi diagrams approximately using the Fast Marching Method (FMM) algorithm (Kimmel and Sethian, 1998). First, the geodesic distances at vertices are calculated by performing the FMM algorithm on all the sources simultaneously. Then, the Voronoi boundaries are constructed by marching along the triangles whose vertices' geodesic distances are defined by different sources. For each such triangle, the Voronoi boundary curve is linearly interpolated by the different distance maps provided by its vertices. Let n be the number of mesh vertices, the geodesic computation part costs $O(n \log n)$ time (Kimmel and Sethian, 1998) and the Voronoi diagram construction part costs $O(n)$ time. Although this algorithm is fast, potentially large errors may occur on near-degenerate meshes since it is based on PDE.

Chapter 3

Edge-based Windows Grouping

Mitchell et al. (1987) showed that the exact geodesics on triangle meshes can be computed by performing *window propagations*. Here, a *window* encodes the information of a geodesic cone originating from the source point. Then, the exact geodesics over a mesh surface are computed by progressively propagating and updating *windows*. The organization of this chapter is shown as follows:

- Section 3.1 defines the two geodesic problems to be solved in this thesis.
- Section 3.2 presents the geometric preliminaries of the exact geodesics on triangle meshes.
- Section 3.3 proposes the Edge-based Windows Grouping (EWG) technique, which is the foundation of this research.

3.1 Discrete Geodesic Problem (DGP) Definition

First, a polyhedral surface M in R^3 is given, which is defined by its associated vertices V, edges E and faces F, i.e. $M = (V, E, F)$. Then, a special point s is given as the *source*. Without loss of generality,

(1) M is assumed to be a *triangle mesh* since the polygonal faces of M can be triangulated in linear time (Chazelle, 1991).

(2) s is assumed to be a vertex of M since the source points on edges or faces of M can be converted to vertices in constant time (Figure 3.1).

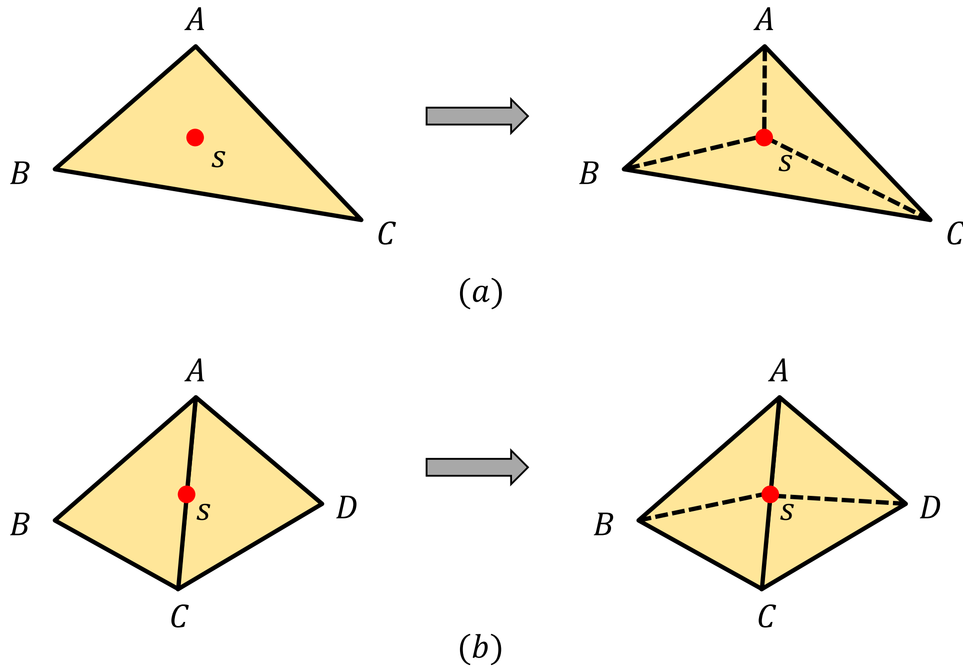


FIGURE 3.1: Convert the source point s to a vertex of the mesh by subdivisions. (a) The source point s is in the interior of ΔABC. Link s to vertices A, B, C. Then, ΔABC is subdivided into ΔsAB, ΔsBC, ΔsAC respectively and s is converted to a vertex of the mesh. (b) The source point s is on the edge AC shared by ΔABC and ΔACD. Link s to vertices B, D. Then, ΔABC is subdivided into ΔsAB and ΔsBC, ΔACD is subdivided into ΔsAD and ΔsDC. s is converted to a vertex of the mesh.

3.1.1 Single-Source DGP (SS-DGP)

Following Mitchell et al. (1987), the classic single-source discrete geodesic problem on triangle meshes is defined as:

Single-Source Discrete Geodesic Problem (SS-DGP).

Instance: A source vertex s on a triangle mesh M.

Question: Find the geodesic distances from s to all the other vertices of M in the Euclidean metric such that the corresponding geodesic paths stay on the mesh M.

3.1.2 Voronoi Diagram oriented DGP (VD-DGP)

An important application of the discrete geodesic algorithm is to construct Voronoi diagrams on triangle meshes (Liu et al., 2011). This application requires retaining multi-source geodesics on edges of a mesh M. Thus, the Voronoi diagram oriented discrete geodesic problem is defined as:

Voronoi Diagram oriented Discrete Geodesic Problem (VD-DGP).

Instance: A set of source vertices $s_0, s_1, ..., s_n$ on a triangle mesh M.

Question: Find and retain the geodesic information from $s_0, s_1, ..., s_n$ to E_{vd} on M in the Euclidean metric such that the corresponding geodesic paths stay on mesh M, where $E_{vd} \subseteq E$ is the set of edges which contribute to the Voronoi diagram construction.

3.2 Preliminaries

3.2.1 Locally Shortest Paths on Triangle Meshes

Starting from the simple case: given a face f of a mesh M, the shortest path between two points s and d in f is the straight line segment linking them (Figure 3.2 (a)).

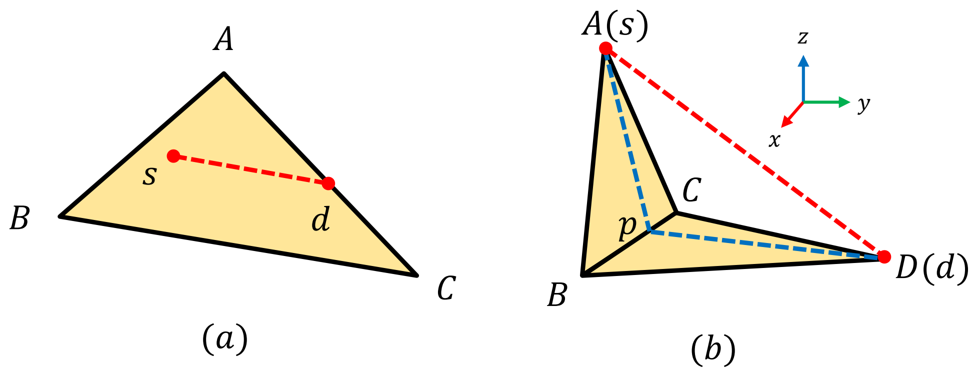
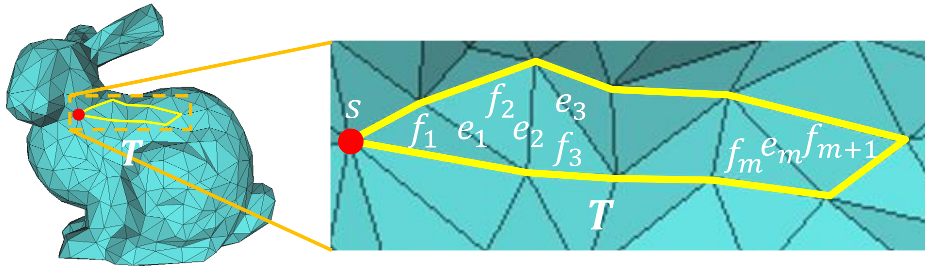


FIGURE 3.2: Illustration of the two cases of a shortest path on a triangle mesh. (a) The shortest path between two points s and d in a face ΔABC is a straight line segment $\overline{sd}$. Note that s and d can be located in the interior or on the boundary of ΔABC. (b) Two points s and d are in ΔABC and ΔBCD respectively. The shortest path between them is a polyline $\overline{spd}$ but not the straight line segment $\overline{sd}$ connecting them. Note that s and d may coincide with the vertices of the faces, e.g. the vertices A and D in the figure.

However, the shortest path between two points s and d is no longer a simple straight line segment if s and d are on two different faces of mesh M respectively. In this case, the straight line segment linking s and d may not stay on M and the shortest path between them is a polyline crossing a sequence of faces with its vertices on edges of M (Figure 3.2 (b)). Such a face sequence is denoted as a *triangle strip*.

Triangle strip Let s be the source vertex, a triangle strip T starting from s is defined as a sequence of faces $f_1, f_2, ..., f_{m+1}$ that two consecutive faces f_i and f_{i+1} $(1 \leq i \leq m)$ are adjacent on the mesh by sharing a common edge e_i (Figure 3.3 (a)). Unless specified, the triangle strips mentioned later are all simple, i.e. a face appears at most once in a triangle strip.



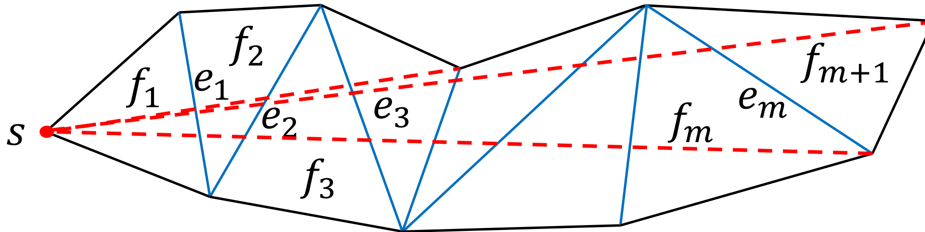


FIGURE 3.3: Illustration of the triangle strip and the planar unfolding. (a) A triangle strip T starting from a vertex s. (b) The obtained 2D triangle strip T' by performing *planar unfolding* on the 3D triangle strip T. The red dashed lines show some shortest paths on T', which are straight line segments. Note that the straight line segments are restricted to fall inside T'. Otherwise, the computed 2D shortest paths cannot be mapped back to the 3D triangle strip T.

Although the triangle strips simplify the search space of shortest paths from the entire mesh, it remains difficult to find shortest paths on them due to their 3D nature. Thus, to further simplify the problem and utilize the extensive knowledge from 2D computational geometry, a technique named *planar unfolding* is employed to unfold 3D triangle strips onto 2D planes.

Planar Unfolding Given a triangle strip $T = (f_1, f_2, ..., f_{m+1})$ and its corresponding edge sequence $e_1, e_2, ..., e_m$ (Figure 3.3 (a)), T is unfolded in this way: rotate f_1 around e_1 until its plane coincides with that of f_2, rotate f_1 and f_2 around e_2 until their plane coincides with that of f_3, continue in this way until all faces $f_1, f_2, ..., f_m$ lie in the plane of f_{m+1}.

The planar unfolding process only involves rotations, and thus no distortion is introduced. As Figure 3.3 (b) shows, a triangle strip is unfolded from 3D to a 2D plane where the shortest path between two points is the straight line segment linking them. Note that the shortest path computed on the unfolded triangle strip can be mapped back to the 3D mesh without distortion.

Based on the above discussions, Mitchell et al. (1987) characterised the locally shortest paths on triangle meshes as follows:

Lemma 3.1. *The general form of a locally shortest path is a path that goes through an alternating sequence of vertices and edge sequences such that the unfolded image of the path along any edge sequence is a straight line segment and both the angles of the path passing through a vertex are greater than or equal to* π.

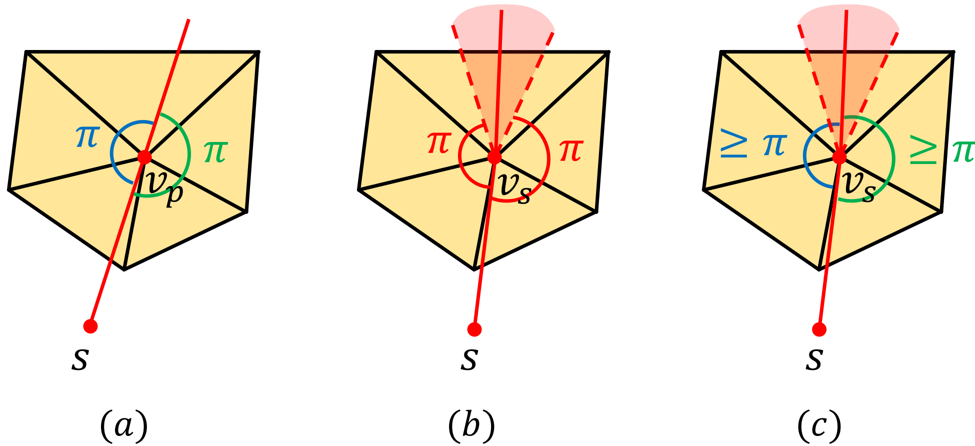


FIGURE 3.4: Illustration of a locally shortest path passing through a vertex. (a) The vertex v_p is a planar vertex. Thus, both the angles of the locally shortest path passing through it is π. (b) The vertex v_s is a saddle vertex. Thus, the locally shortest path passing through it must lie in the "invisible" region it causes (the red shaded region). The "invisible" region is bounded by the two locally shortest paths passing through v_s whose angles are π. (c) The vertex v_s is a saddle vertex. Thus, at least one of the angles of the locally shortest path passing through it is larger than π.

Lemma 3.1 mentioned two cases of the locally shortest path according to the types of the vertex passed:

- A *planar* vertex v_p whose round angle equals 2π. In this case, both the angles of the path passing through v equals π (Figure 3.4 (a)).
- A *saddle* vertex v_s whose round angle is larger than 2π. In this case, at least one of the two angles of the path passing through v_s is larger than π (Figure 3.4 (b)(c)).

In geodesic studies, the second case attracts more attention since it causes "invisible" regions on the mesh from the source vertex. The geodesic paths of all the points in an "invisible" region pass through the corresponding saddle vertex. Thus, the saddle vertices are usually referred to as *pseudo sources*.

3.2.2 Globally Shortest Paths on Triangle Meshes

The preceding section characterises the locally shortest paths on a triangle mesh M by introducing the *triangle strip* and *planar unfolding*. However, there may be multiple locally shortest paths from a source vertex s to a destination vertex d on M. Among them, only the ones with the shortest length are the globally shortest paths. Thus, the globally shortest paths from s to d can be found by comparing the lengths of all the locally shortest paths linking them. In addition, the geodesic distance of d is the length of the globally shortest paths and is ***unique***. Mitchell et al. (1987) characterised the globally shortest paths formally as follows:

Lemma 3.2. *A globally shortest path is a geodesic, and it has the additional property that no edge can appear in more than one edge sequence and each edge sequence must be simple. If $p(x)$ and $p(y)$ are geodesic paths from s to points x and y, then they can intersect only at vertices of M, and if they do intersect at v, then that subpath of $p(x)$ from s to v has the same length as that subpath of $p(y)$ from s to v.*

In Lemma 3.2, the vertex v is essentially a saddle vertex and it states that the geodesic paths do not intersect on edges or faces of a triangle mesh M.

3.3 Edge-based Windows Grouping (EWG)

The Edge-based Windows Grouping (EWG) technique is proposed to improve the exact geodesic computation on triangle meshes. Its main idea is to group the windows on an edge into one or two window lists and process them in batches. Before discussing the details of EWG, the definition and propagation of windows are first proposed as follows.

3.3.1 Window Definition and Propagation

The feature of an individual geodesic path is discussed in Section 3.2. However, since a source vertex s can emanate an infinite number of geodesic paths on a triangle mesh, an efficient data structure is required to encode the geodesic information emanated from s for computational purpose. Based on the observation that many adjacent geodesic paths are from the same triangle strip, Mitchell et al. (1987) proposed such a data structure called *window*, which is also employed by the work of this thesis.

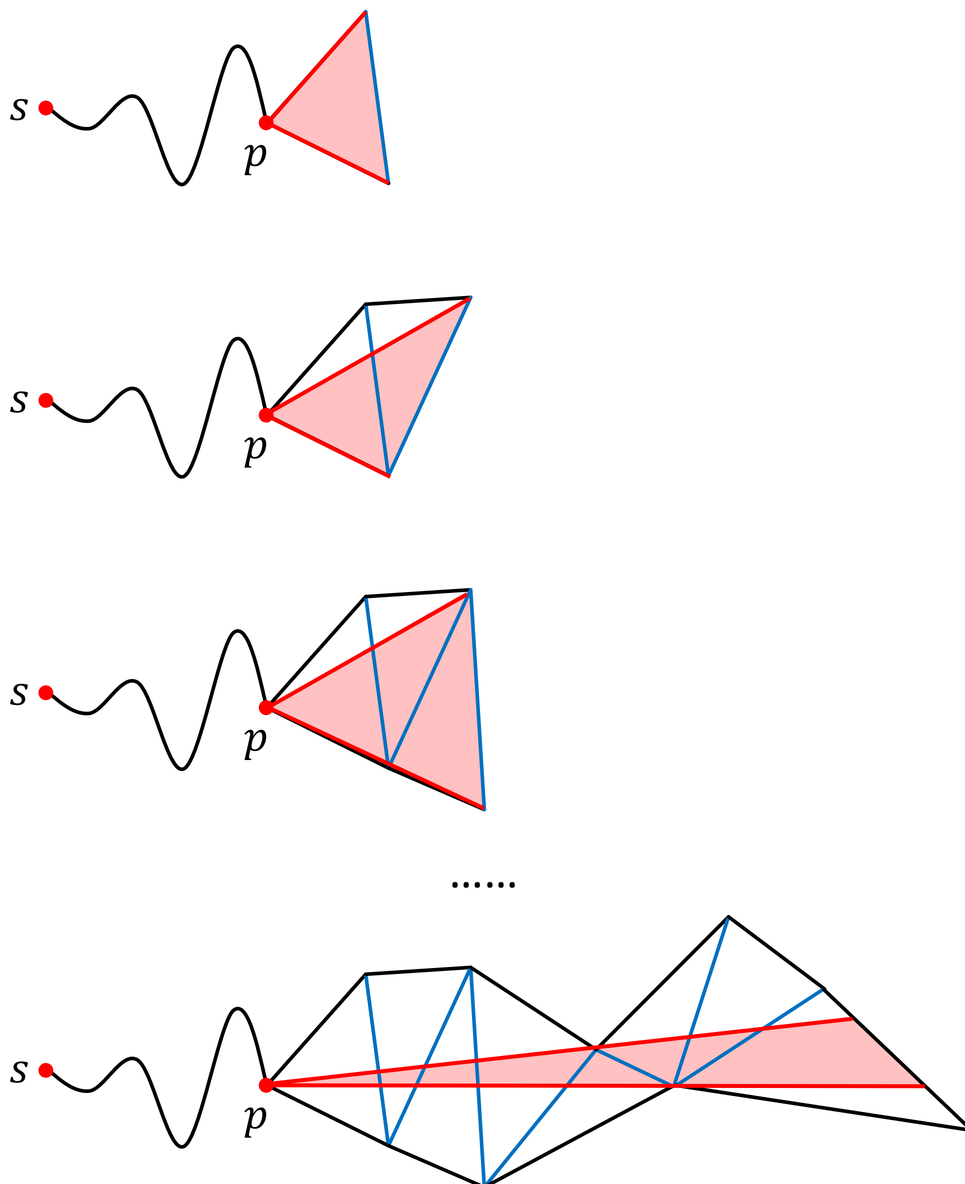


FIGURE 3.5: Illustration of the geodesic visible cone in a triangle strip. s is the source vertex. p is the projection of the pseudo source, which must be a saddle vertex.

According to the discussion in Section 3.2.1, the shortest paths in an unfolded triangle strip are straight line segments and these shortest paths form a visible cone. Figure 3.5 shows how the visible cone changes when the triangle strip is unfolded from p. A *window* is a data structure encoding the geodesic information of an unfolded triangle strip including the visible cone, which is defined as follows:

Window definition A window is defined as $w = (\Delta ABC, a_0, a_1, p, d_0, d_1, \sigma)$, where ΔABC stands for the triangle it enters from AB and shows the unfolding direction of the triangle strip (this is equivalent to the half edge used by Surazhsky et al. (2005)). Two scalar parameters, a_0 and a_1, mark the two endpoints of w, which lies on the edge AB, and a_0 denotes the endpoint closer to A. Every window w is created by the source vertex s or a pseudo source, which must be a saddle vertex. Here p represents the projection of this pseudo source on the plane determined by ΔABC, and d_0, d_1 are the distances from a_0, a_1 to p respectively. a_0, a_1, p, d_0, d_1 together encode the information of the visible cone from p. σ denotes the geodesic distance from the pseudo source to the source vertex s.

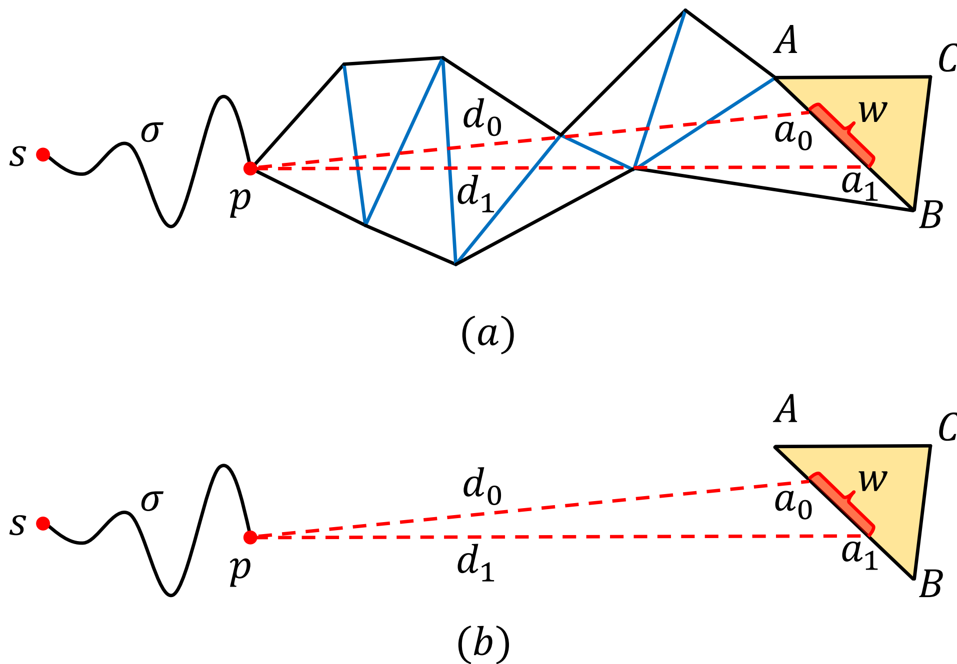


FIGURE 3.6: Illustration of the *window* data structure.

Figure 3.6 (a) shows an illustration of window w in an unfolded triangle strip. It can be seen that w is narrowed by the boundary vertices of the triangle strip. Since the parameters of a window contain all the geodesic information of an unfolded triangle strip, the geodesic computation can be implemented

by propagating windows without considering the corresponding triangle strip (Figure 3.6 (b)). That is, the operation of unfolding triangle strips is converted to updating the parameters of windows. Let w be a window to be propagated across ΔABC from the edge AB. As Figure 3.7 shows, the three points a_0, a_1 and p of w form a 2D visible cone from p whose boundaries are the rays $\overrightarrow{pa_0}$ and $\overrightarrow{pa_1}$. According to the positions of the intersection points x, y between $\overrightarrow{pa_0}$, $\overrightarrow{pa_1}$ and the edges of ΔABC, the propagation of w contains three cases:

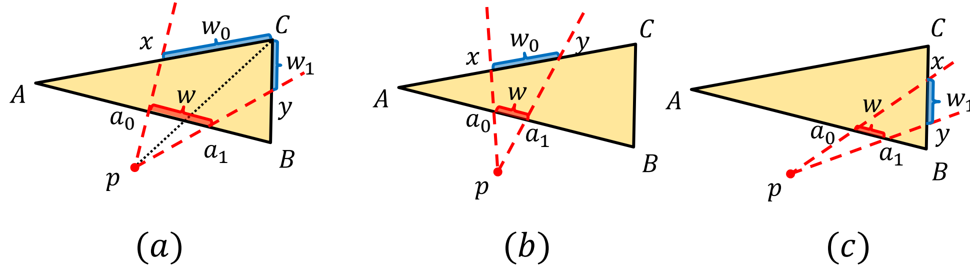


FIGURE 3.7: Three cases of the window propagation in a face. (a) A window w is propagated from the edge AB to the edges AC and BC as the windows w_0 and w_1 respectively. (b) w is propagated from AB to AC as w_0. (c) w is propagated from AB to BC as w_1.

(1) If x and y are on the edges AC and BC respectively, there are two valid propagation directions. First, the window w is propagated to AC as the window w_0 whose endpoints are x and C. Second, w is propagated to BC as the window w_1 whose endpoints are C and y.

(2) If both x, y are on the edge AC, there is only one valid propagation direction. The window w is propagated to AC as the window w_0 whose endpoints are x and y.

(3) If both x, y are on the edge BC, there is only one valid propagation direction. The window w is propagated to BC as the window w_1 whose endpoints are x and y.

With the above knowledge, EWG groups nearby windows on an edge into window lists and process them in batches, whose details are shown as follows.

3.3.2 Applying EWG on Window Propagation

This section defines EWG and shows how to propagate the windows grouped by EWG in batches.

3.3.2.1 EWG Definition

According to the number of window lists associated with an edge, the definition of EWG has two variants and are presented respectively as follows (Figure 3.8):

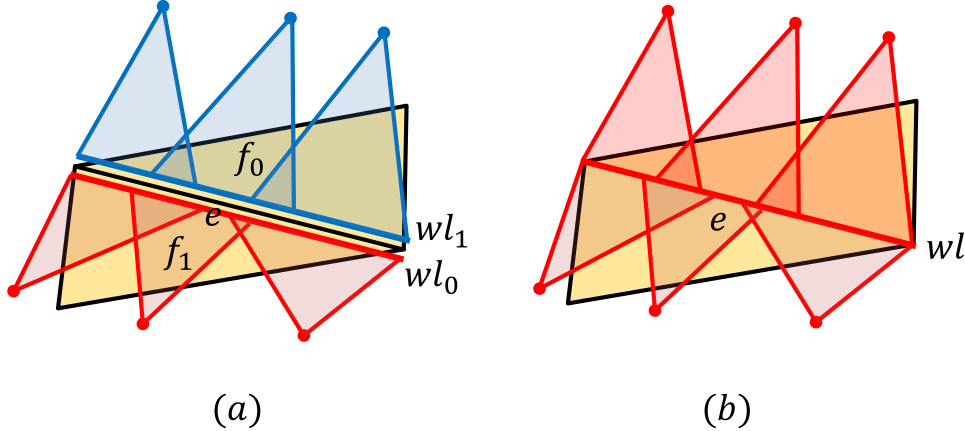


FIGURE 3.8: (a) Illustration of Definition 3.1. (b) Illustration of Definition 3.2.

Definition 3.1. *For an edge e of a mesh M, two window lists wl_0 and wl_1 (corresponding to the two faces f_0 and f_1 sharing e) are defined and associated with e. wl_0 contains the windows to be propagated through f_0 and wl_1 contains the windows to be propagated through f_1. The range and operations of this definition is introduced as follows,*

Range: *the window lists wl_0 and wl_1.*

Operations:

(1) ***Window list query.*** *For an edge e of a mesh M, this operation queries the window lists wl_0 and/or wl_1 associated with it.*

(2) ***Window query.*** *For a queried window list (e.g. wl_0 or wl_1), this operation queries the window(s) stored in it.*

Definition 3.2. *For an edge e of a mesh M, a window list wl is defined and associated with e. wl contains all the windows propagated to e. The range and operations of this definition is introduced as follows,*

Range: *the window list wl.*

Operations:

(1) ***Window list query.*** *For any edge e of mesh M, this operation queries the window list wl associated with it.*

(2) ***Window query.*** *For a queried window list wl, this operation queries the window(s) stored in it.*

To illustrate how to use the defined EWG technique, the window propagations are implemented in terms of EWG as follows.

3.3.2.2 EWG Window Propagation

The exact geodesics on a triangle mesh M can be computed by iteratively propagating the windows edge by edge from a source vertex s or a set of source vertices $s_0, s_1, ..., s_n$ over M (Mitchell et al., 1987). Compared to the previous methods which organises such propagations by individual windows (Surazhsky et al., 2005; Xin and Wang, 2009; Xu et al., 2015), the work in this thesis applies the EWG technique to group nearby windows in window lists and propagate them together in batches. This process is summarized as the ***EWG-based window propagation framework***, which is outlined as follows,

Initialization. Create the initial window lists on the edges in the 1-ring neighbourhood of a source vertex s or a set of source vertices $s_0, s_1, ..., s_n$. Then, push all these window lists into a priority queue Q.

EWG-based Window Propagation.
Step 1. Pop a window list wl from Q.
Step 2. Propagate each window in wl across its corresponding triangle face and update the corresponding window list(s).
Step 3. Remove the redundant windows in the updated window list(s). Push these window lists into Q if they can be further propagated.
Step 4. If Q is empty, finish; otherwise, goto Step 1.

The above framework can be divided into three modules, which are shared by all the window-based exact geodesic algorithms (Surazhsky et al., 2005; Xin and Wang, 2009; Xu et al., 2015):

- **Window propagation (Step 2).** This module performs window propagations across the faces of a mesh M.
- **Window redundancy reduction (Step 3).** This module identifies the redundant windows and remove them during the propagation. It also guarantees the termination of the algorithm.
- **Window management (Step 1,4).** This module manages the window propagations in order, which makes the *window redundancy reduction*

module more effective. The proposed framework employs the continuous Dijkstra technique (Mitchell et al., 1987) and propagates the windows grouped by EWG from near to far according to their distances from the source vertex or vertices. Thus, the redundant windows can be removed at the earliest stage.

Among the three modules of the proposed framework, the *window propagation* module is implemented by propagating all the windows in a window list defined in the two EWG definitions (Definition 3.1 and Definition 3.2). Thus, the propagation of a window list wl has two cases:

- wl is a window list defined in Definition 3.1. In this case, all the windows in wl have the same propagation direction. Thus, wl is propagated by simply traversing and propagating each window in it across the same face (Figure 3.9 (a)).
- wl is a window list defined in Definition 3.2. In this case, the windows in wl may have different propagation directions. Hence, wl is propagated by traversing and propagating each window in it across the face recorded by the window (Figure 3.9 (b)).

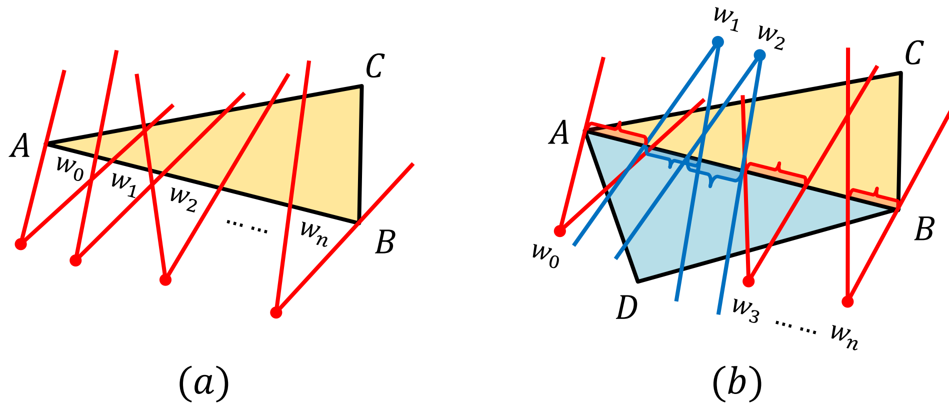


FIGURE 3.9: The two cases of the EWG window propagation module. (a) The window list $wl = (w_0, w_1, w_2, ..., w_n)$ is defined by Definition 3.1. Thus, all the windows in it are propagated across ΔABC. (b) wl is defined by Definition 3.2. Thus, the windows in it are propagated across ΔABC and ΔABD respectively according to their recorded directions.

The other two modules *window redundancy reduction* and *window management* are the key modules since they determine the time costs of the exact geodesic algorithms. To illustrate how EWG is applied in these two modules and achieves high performance, the following section evaluates the performance of EWG by comparing it with existing methods.

3.3.2.3 EWG Performance Evaluation

All the state-of-the-art exact geodesic algorithms (Surazhsky et al., 2005; Xin and Wang, 2009; Xu et al., 2015) employ an individual-window-based propagation framework, in which the window propagation, window redundancy reduction and window management are all based on individual windows rather than the grouped window lists. Compared to them, the proposed EWG-based window propagation framework has the following distinct advantages,

- **Low window redundancy** EWG groups the windows on edges together into window lists. Thus, the extra *inter-window geodesic information* among the windows in a window list can be utilized to removed the redundancy more thoroughly.
- **Low window management cost** Both the proposed framework and the existing exact geodesic algorithms employ the continuous Dijkstra technique (Mitchell et al., 1987) to manage the window propagation from near to far by a priority queue. However, compared to the existing methods whose priority queues employ individual windows as elements, the proposed framework employs window lists as the priority queue elements. Since the number of window lists is much smaller that that of windows, the window management cost of the proposed framework is dramatically reduced.

The rationale of these advantages is from the uniqueness of geodesic distances (Section 3.2.2). Such uniqueness shows that the redundant part of a window can be removed by constructing shorter paths to the points in it from other windows. However, since the total number of the windows on a mesh is large, it is infeasible and unnecessary to remove a window's redundancy by checking it with all the other windows on the mesh. Thus, a natural choice is to check a window with the nearby ones on the same edge. After this redundancy removal process, the valid windows on an edge will have similar distances whose range is related to the edge's length. This makes the propagation of *window lists* reasonable.

The above-mentioned advantages of EWG make the solutions of the SS-DGP and VD-DGP problems (Section 3.1) fast and memory-efficient, which will be discussed in the following section.

3.3.3 EWG-based Solutions to Geodesic Problems

3.3.3.1 Solution to the SS-DGP problem

The SS-DGP problem focuses on computing the geodesic distances from a source vertex to all the other vertices of a triangle mesh (Section 3.1.1). To maintain such geodesic distances, a vector $D = (d_1, d_2, ..., d_n)$ is employed, where n is the number of mesh vertices. Then, the SS-DGP problem can be solved by constantly updating D during window propagation.

Challenge: a both fast and memory-efficient SS-DGP algorithm The SS-DGP problem can be solved by all the four state-of-the-art exact geodesic algorithms (Surazhsky et al., 2005; Xin and Wang, 2009; Xu et al., 2015):

- The Mitchell-Mount-Papadimitriou (MMP) algorithm.
- The Improved Chen-Han (ICH) algorithm.
- The Fast-Wavefront-Propagation MMP (FWP-MMP) algorithm.
- The Fast-Wavefront-Propagation CH (FWP-CH) algorithm.

Among them, the MMP and FWP-MMP algorithms usually run faster than the ICH and FWP-CH algorithms but consume thousands times more memory (Xin and Wang, 2009; Xu et al., 2015). Thus, it is still challenging to design a both fast and memory-efficient SS-DGP algorithm.

Employed EWG Definition: Definition 3.1 The geodesic distance of a vertex v on the mesh is determined by the windows propagated to it from the opposite edges in its 1-ring neighbourhood. On each such opposite edge e, it is straightforward to group the windows which may contribute to v's geodesic distance as a window list. Since these windows have the same propagation direction, Definition 3.1 is employed in this scenario (Figure 3.10).

Solution The challenge of the SS-DGP solution is addressed as follows,

- ***Fast speed*** As discussed in Section 3.3.2.3, the proposed SS-DGP algorithm is fast because it has lower window redundancy and window management cost than the existing methods.
- ***Low memory cost*** Since the SS-DGP problem only involves computing the geodesic distances at vertices, the propagated windows are deleted but not retained on the edges of meshes as the MMP and FWP-MMP

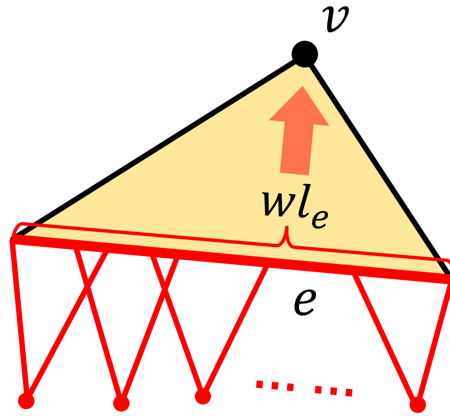


FIGURE 3.10: The reason for employing the EWG Definition 3.1 in the solution of the SS-DGP problem. The bold red line denote a 1-ring opposite edge e of vertex v. The window list wl_e on edge e is defined according to Definition 3.1, which contains the windows propagating to the direction of v.

algorithms do. In addition, applying EWG makes the window redundancy lower than the existing methods (Section 3.3.2.3). Thus, the proposed SS-DGP algorithm is memory-efficient.

Based on the above ideas, a fast and memory-efficient SS-DGP algorithm is proposed in Chapter 4.

3.3.3.2 Solution to the VD-DGP problem

The VD-DGP problem requires retaining the geodesic information on a set of edges E_{vd} of a triangle mesh which contribute to the Voronoi diagram construction (Section 3.1.2). To retain such geodesic information, the propagated windows on the edges of E_{vd} must be kept. Then, the VD-DGP problem can be solved by constantly updating the windows on edges of E_{vd} during window propagation.

Challenge: a both fast and memory-efficient Voronoi diagram construction method The geodesic computation costs the majority of the time and memory in the Voronoi diagram construction (Liu et al., 2011). Among the four state-of-the-art exact geodesic algorithms (Surazhsky et al., 2005; Xin and Wang, 2009; Xu et al., 2015), only the MMP and FWP-MMP algorithms can be used since they retain the propagated windows on edges of the mesh, which provides the necessary geodesic information for the Voronoi diagram construction. However, although these two algorithms are relatively fast, the

retained windows cost large amounts of memory (Surazhsky et al., 2005) and becomes their bottleneck. Thus, it is still challenging to propose a both fast and memory-efficient Voronoi diagram construction method.

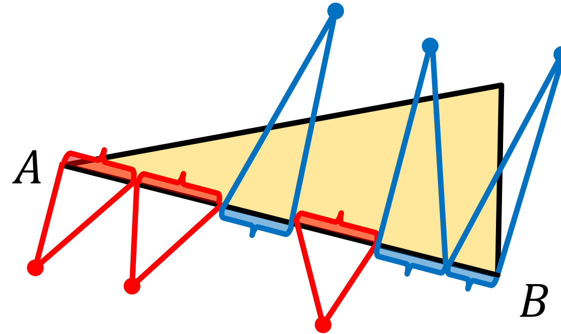


FIGURE 3.11: The reason for employing the EWG Definition 3.2 in the solution of the VD-DGP problem. The retained windows on the edge AB are optimized into non-overlapping ones for the Voronoi diagram construction.

Employed EWG Definition: Definition 3.2 In the MMP and FWP-MMP algorithms, the retained windows on edges are trimmed into non-overlapping ones and ordered according to their positions, which form an elegant structure for the Voronoi diagram construction (Liu et al., 2011). Following it, Definition 3.2 is employed to provide the necessary geodesic information for the Voronoi diagram construction (Figure 3.11).

Solution The challenge of the geodesic-based Voronoi diagram construction is addressed as follows,

- ***Fast speed*** Compared to the existing methods, the proposed Voronoi diagram construction method is fast because it has lower window management cost in geodesic computation (Section 3.3.2.3). Note that the window redundancy reduction module in the proposed method does not contribute to the speed-up because it employs the same "window trimming rule" used by the MMP and FWP-MMP algorithms to optimize the retained windows on edges for the Voronoi diagram construction.

- ***Low memory cost*** To provide the necessary geodesic information for the Voronoi diagram construction, the propagated windows must be retained on edges. However, since the Voronoi diagrams are usually more sparse than meshes, the retained windows on most of the edges are redundant can be removed. By applying EWG, such retained windows are grouped as window lists. Thus, they can be removed efficiently

in batches by identifying the redundant window lists, which makes the proposed method memory-efficient.

Based on the above ideas, a fast and memory-efficient method to construct the geodesic-based Voronoi diagram on triangle meshes is proposed in Chapter 5.

Chapter 4

Fast and Exact SS-DGP Algorithm

To solve the SS-DGP problem which computes the geodesic distances from a source vertex to all the other vertices of a triangle mesh (Section 3.1.1), an EWG-based fast and memory-efficient exact geodesic algorithm is proposed in this chapter. The high performance of this algorithm comes from its low window redundancy, low window management cost, and that it does not retain propagated windows on edges (Section 3.3.3.1). To illustrate how EWG is applied to achieve such high performance, this chapter is organized as follows:

- Section 4.1 overviews the proposed algorithm.
- Section 4.2 introduces how EWG is applied to **reduce window redundancy** in the window list propagation within a triangle. When a window list is propagated, its redundancy can be removed by performing the *pairwise window pruning*. Here, EWG helps to form more window pairs by accumulating windows in window lists and propagating them together. Thus, the redundant windows can be pruned more thoroughly.
- Section 4.3 introduces how EWG is applied to **reduce window management cost** when propagating the wavefront over a mesh. The wavefront contains the window lists to be propagated and its propagation is managed by a priority queue. Here, EWG is applied to build the connections between mesh edges and window lists so that the window lists can be propagated by sorting mesh primitives (e.g. vertices). Thus, the window management cost is dramatically reduced since the number of mesh primitives is much smaller than that of the windows.
- Section 4.4 analyses the complexity of the proposed algorithm.
- Section 4.5 shows the experimental results of the proposed algorithm.
- Section 4.6 summarizes this chapter.

4.1 Algorithm Overview

This section briefly overviews the proposed EWG-based SS-DGP algorithm and its outline is shown in Algorithm 1. To concisely refer to it, the proposed algorithm is named as **VTP** (**V**ertex-oriented **T**riangle **P**ropagation).

Initialization Given a triangle mesh M and a source vertex s, a single window is created for every opposite edge of s in its 1-ring neighbourhood and stored in the corresponding window list. All the adjacent vertices of s is pushed into a priority queue. A vector D is created to maintain the geodesic distances of the vertices of M during propagation.

Algorithm 1 Vertex-oriented Triangle Propagation (VTP) Outline

Input: M - Mesh, S - Source set;
Output: D - a vector containing the geodesic distances of M's vertices;
1: **procedure** VTP(M, S)
2: Perform initialization; ▷ **Initialization**
3: Push all adjacent vertices of S into a priority queue Q;
4: **while** $!Q.empty()$ **do** ▷ **Wavefront Propagation**
5: Pop a vertex v from Q;
6: Update the wavefront and the traversed area R;
7: Propagate the window lists on previous wavefront edges through the newly added triangles of R; ▷ **Window List Propagation**
8: Update Q and D;
9: **end while**
10: **end procedure**

Window List Propagation EWG groups windows as window lists and propagates them in batches. In this scenario, any two windows picked from a window list can form a *window pair*. To remove the redundancy of such a window pair during propagation, a set of effective rules are proposed from exhaustively studying the pairwise window pruning cases inside a triangle. Then, these rules are applied to prune redundant windows in the three stages of the window list propagation: window list splitting, window list propagation and window list merging. As a result, the proposed VTP algorithm achieves low window redundancy.

Wavefront Propagation EWG builds the connections between window lists and mesh edges. Thus, the wavefront is defined as the boundary edges of the *traversed area*, which is a single connected region containing all the visited triangles of the mesh. The wavefront is then propagated by gradually enclosing unvisited triangles abutting the traversed area in a continuous Dijkstra

style (Mitchell et al., 1987). That is, a priority queue is employed to add the unvisited triangles into the traverse area according to their distances from the source vertex. Each time the traversed area expands, the window lists on the previous wavefront edges are propagated through the newly added triangles until they reach the updated wavefront or be removed during propagation. In the proposed VTP algorithm, vertices are employed as the priority queue elements since it can add *multiple* triangles from its 1-ring neighbourhood to the traversed area every time, which yields low window management cost.

4.2 EWG in Window List Propagation Within a Triangle

The proposed VTP algorithm employs the EWG-based window propagation framework (Section 3.3.2.2) to compute the geodesic distances of mesh vertices by propagating window lists. These window lists are defined by Definition 3.1 which is more suitable for the SS-DGP problem (Section 3.3.3.1). When propagating a window list, a non-redundant window may become partially or completely redundant after the propagation. Such a change of status is caused by another overlapping window propagated through the same triangle from the same window list. These two windows form a *window pair*, which works as the basic unit for redundancy identification and removal in the proposed algorithm. Note that EWG is applied to from more window pairs by accumulating windows in window lists so that the redundancy can be removed more thoroughly. To illustrate it, this section is organized as follows:

- Section 4.2.1 enumerates the pairwise window pruning cases in a triangle exhaustively and derives the corresponding window pruning rules.
- Section 4.2.2 studies the principles of window pruning.
- Section 4.2.3 introduces how to apply the derived rules in the window list propagation.
- Section 4.2.4 justifies the algorithmic choices of the proposed VTP algorithm in terms of the window list propagation.

4.2.1 Pairwise Window Pruning Within a Triangle

This section exhaustively enumerates the pairwise window pruning cases in a triangle and derives the corresponding pruning rules. The proof of these rules will be given in the next section as Proposition 4.1, 4.2, 4.3 and 4.4. These rules are used to prune redundant windows in the EWG-based window list propagation, which will be discussed in Section 4.2.3.

First, consider the propagation of a single window inside a triangle. This window initially stays on one edge of the triangle. After one step of propagation, it either moves to another edge of the same triangle or partially cover both opposite edges (Section 3.3.1). Now, consider two windows inside the same triangle simultaneously. Even if both windows are initially non-redundant, after one step of propagation, one of them may become partially or completely redundant because the relative position between the two windows may have changed.

Consider two windows w_0 and w_1, whose respective pseudo sources are p and q. Let $\sigma_0 = w_0.\sigma$ and $\sigma_1 = w_1.\sigma$. Let ΔABC be the triangle where one-step propagation of w_0 and w_1 takes place, and w_0' and w_1' be their propagated version, respectively. Denote $a_0 = w_0.a_0$, $a_1 = w_0.a_1$, $b_0 = w_1.a_0$, $b_1 = w_1.a_1$, and $a_0' = w_0'.a_0$, $a_1' = w_0'.a_1$, $b_0' = w_1'.a_0$, $b_1' = w_1'.a_1$. Note that p does not lie inside the visible cone of w_1 since otherwise w_1 should have been split into two windows. Similarly, q does not lie inside the visible cone of w_0. Define $g(pa) = \sigma_0 + ||pa||$ and $g(pab) = \sigma_0 + ||pa|| + ||ab||$, where $p = w_0.p$, a and b are two other points on the plane determined by ΔABC. $g(qa)$ and $g(qab)$ have similar definitions. And, $g(Aa) = D(A) + ||Aa||$ where $D(A)$ is the shortest distance so far at vertex A. $g(Ba)$ and $g(Ca)$ have similar definitions. $\langle pab \rangle$ and $\langle qab \rangle$ represent the sub-windows defined by the three points. The *separating point* of a window is defined as follows.

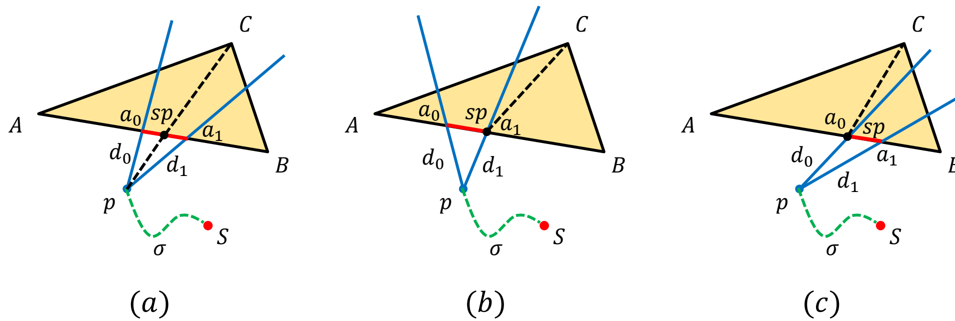


FIGURE 4.1: Three configurations of the separating point of a window.

Separating Point Definition Let w be a window to be propagated through ΔABC from edge AB. The separating point *sp* of w is the intersection between AB and the shortest path between $w.p$ and C routed through interval $[a_0, a_1]$. It can be easily verified that $w.sp = w.a_1$ if w only propagates to AC; $w.sp = w.a_0$ if w only propagates to BC. When w propagates to two edges, $w.sp$ is the intersection between line segments AB and pC. Thus, the window structure used in this chapter is extended as $w = (\Delta ABC, a_0, a_1, p, d_0, d_1, \sigma, \boldsymbol{sp})$. Figure 4.1 shows an illustration of the three possible positions of the separating point of a window.

The 15 cases which may produce redundant windows inside ΔABC are listed and classified into five situations. The corresponding windows pruning rules are discussed in each situation respectively as follows.

Situation 1: Propagating w_0 and w_1 from the same edge to another edge. Here, only the configuration where w_0 and w_1 are propagated from AB to AC is discussed, other configurations in this situation can be dealt with similarly. Without loss of generality, assume $a_0 < b_0$. This situation is refined into three cases (Case 1, Case 2 and Case 3) according to the relative position between w'_0 and w'_1 (Figure 4.2). The corresponding window pruning rules are shown as follows.

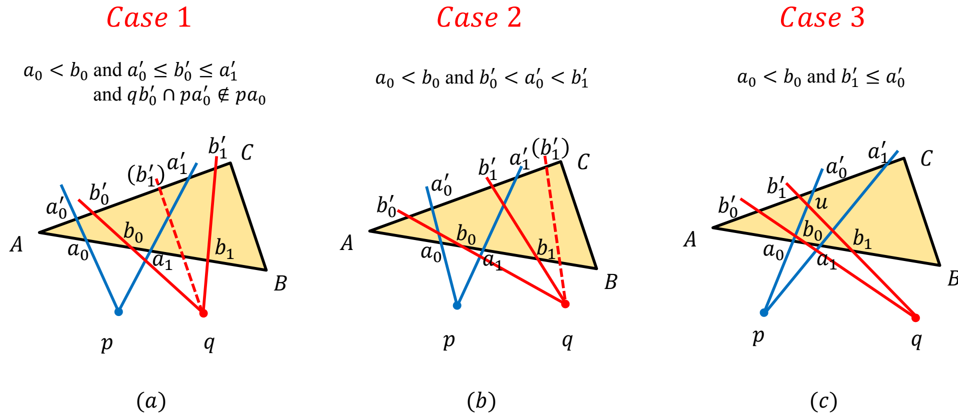


FIGURE 4.2: The three cases that w_0 and w_1 are propagated from the same edge to another edge.

- Pruning rules for **Case 1**:
  - If $g(pb'_0) > g(qb'_0)$, delete $\langle pb'_0a'_1 \rangle$.
  - If $b'_1 \le a'_1$ and $g(pb'_1) \le g(qb'_1)$, delete w'_1.
  - If $b'_1 \le a'_1$ and $g(pb'_1) > g(qb'_1)$, delete $\langle pb'_1a'_1 \rangle$.

- If $b'_1 > a'_1$ and $g(pa'_1) < g(qa'_1)$, delete $\langle qb'_0a'_1 \rangle$.

- Pruning rules for **Case 2**:
  - If $g(pa'_0) < g(qa'_0)$, delete $\langle qb'_0a'_0 \rangle$.
  - If $g(pa'_0) \geq g(qa'_0)$, delete w'_0.
  - If $b'_1 \leq a'_1$ and $g(pb'_1) \leq g(qb'_1)$, delete w'_1.
  - If $b'_1 \leq a'_1$ and $g(pb'_1) > g(qb'_1)$, delete $\langle pb'_1a'_1 \rangle$.
  - If $b'_1 > a'_1$ and $g(pa'_1) < g(qa'_1)$, delete $\langle qb'_0a'_1 \rangle$.

- Pruning rules for **Case 3**:
  - Let $u = qb'_1 \cap pa'_0$. If $g(pu) < g(qu)$, delete w'_1; else, delete w'_0.

These rules are derived using Proposition 4.1. Case 2 is actually the same as step 2 (*Delete Dominated Candidate Intervals*) of procedure *Insert-Interval (I, c)* in the MMP algorithm (Mitchell et al., 1987). And the other two cases are novel. Note that in Case 1, the cross checking is not performed when $a_0 < b_0$, $a'_0 \leq b'_0 \leq a'_1$ and $qb'_0 \cap pa'_0 \in pa_0$ as this special case rarely happens in the proposed algorithm. This is because the two windows satisfying the condition should have already crossed each other inside a triangle they passed through earlier and it is very likely that cross checking has been performed between them.

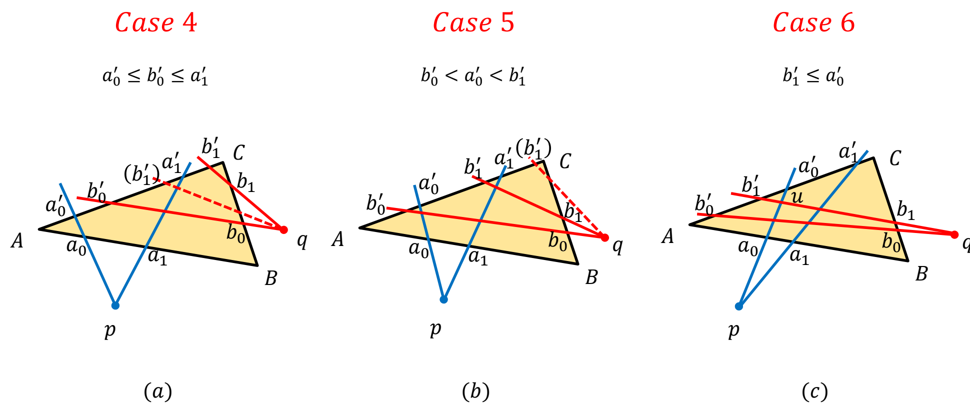


FIGURE 4.3: The three cases that w_0 and w_1 are propagated from two edges to the third edge.

Situation 2: Propagating w_0 and w_1 from two edges to the third edge. Assume w_0 is propagated from AB to AC, and w_1 is propagated from BC to AC. Then, three cases are derived (Case 4, Case 5 and Case 6) corresponding

to Cases 1-3, respectively (Figure 4.3). The window pruning rules are also the same as those in the first three cases.

Situation 3: Propagating w_0 and w_1 from the same edge to two other edges. Assume both w_0 and w_1 lie on AB, and they are respectively propagated to BC and AC. This situation is refined into Case 7, Case 8 and Case 9 shown in Figure 4.4. The corresponding window pruning rules are shown as follows.

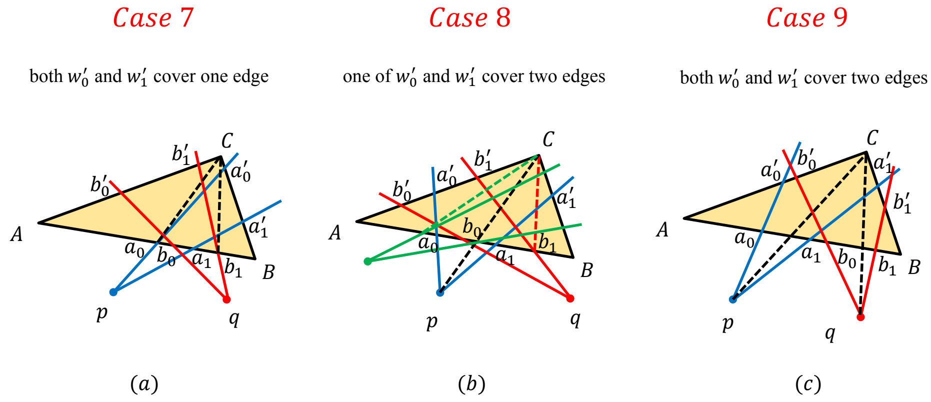


FIGURE 4.4: The three cases that w_0 and w_1 are propagated from the same edge to two other edges.

- Pruning rules for **Case 7**:
  - When $a_0 < b_1$, if $g(pa_0C) > g(qb_1C)$, delete w_0'; else delete w_1'.
- Pruning rules for **Case 8**:

  (1) For $w_1' \in AC$.
  - When $w_0.sp < b_1$, if $g(pC) < g(qb_1C)$, delete w_1'; else, delete $\langle pCa_1' \rangle$.
  - If $w_0.sp > b_1$ and $g(pC) > g(qb_1C)$, delete $\langle pa_0'C \rangle$.

  (2) For $w_1' \in BC$.
  - When $w_0.sp > b_0$, if $g(pC) < g(qb_0C)$, delete w_1'; else, delete $\langle pa_0'C \rangle$.
  - If $w_0.sp < b_0$ and $g(pC) > g(qb_0C)$, delete $\langle pCa_1' \rangle$.
- Pruning rules for **Case 9**:
  - If $g(pC) > g(qC)$, delete $\langle pCa_1' \rangle$; else, delete $\langle pb_0'C \rangle$.

Case 7 only shows the configuration where w'_0 lies on BC and w'_1 lies on AC, and other configurations can be dealt with similarly. The pruning rules in this case are based on Proposition 4.4. Similarly, in Case 8, only the configuration where w'_0 covers two edges is shown. The pruning rules in this case are based on Proposition 4.2 and 4.3. Among these rules, rule (1) is based on Proposition 4.2 (2) and 4.3 (1) while rule (2) is based on Proposition 4.2 (1) and 4.3 (2). Case 7 and Case 8 are quite useful in removing redundant windows in the proposed geodesic algorithm. Case 9 is exactly the same as the "One-Angle-One-Split" rule in the CH algorithm (Chen and Han, 1990).

Situation 4: Checking with vertices. In this situation, the "checking with vertices" rule in the ICH algorithm (Xin and Wang, 2009) is extended to the two-window scenario. The corresponding window pruning rules are shown as follows.

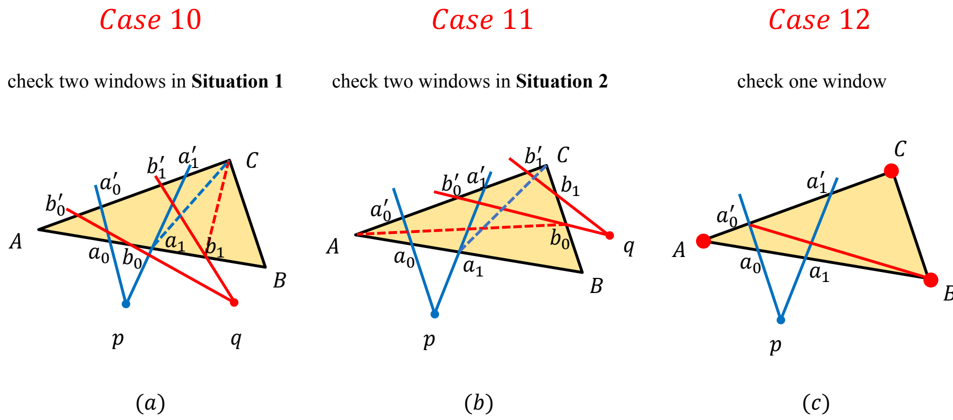


FIGURE 4.5: The three cases of checking windows with vertices.

- Pruning rules for **Case 10**:
  - If $a_1 < b_1$ and $g(pa_1C) < g(qb_1C)$, delete w'_1.
- Pruning rules for **Case 11**:
  - If $g(pa_1C) < g(qb_1C)$, delete w'_1.
  - If $g(qb_0A) < g(pa_0A)$, delete w'_0.
- Pruning rules for **Case 12**:
  - If $g(Aa'_1) < g(pa'_1)$ or $g(Ca'_0) < g(pa'_0)$ or $g(Ba'_0) < g(pa'_0)$, delete w'_0.

As shown in Figure 4.5, Case 10 illustrates the configuration where both windows are propagated from the same edge to another edge. The pruning rule is based on Proposition 4.4 (1). Case 11 illustrates the same configuration as in Situation 2. The corresponding pruning rules are derived from Proposition 4.4. Case 12 shows the same configuration as that in Theorem 3.2 of the ICH algorithm (Xin and Wang, 2009). The corresponding pruning rule can be derived from Proposition 4.1.

Situation 5: Propagating w_0 and w_1 from two edges to two edges. In addition, the three cases where the two windows are propagated from two different edges to two edges are discussed.

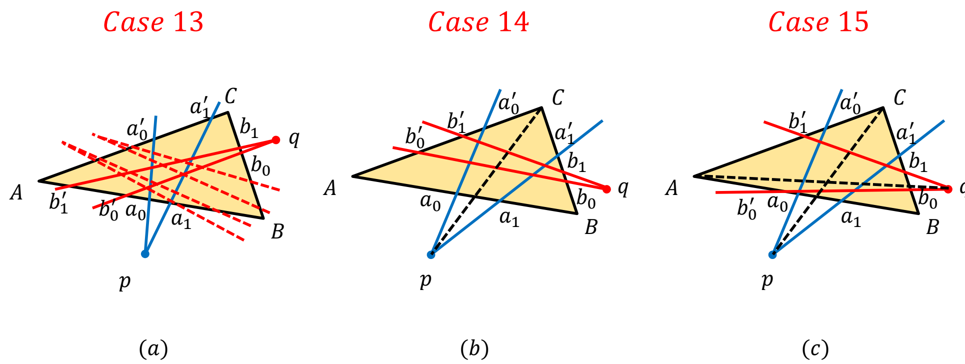


FIGURE 4.6: The three cases that w_0 and w_1 are propagated from two edges to two edges.

Figure 4.6 (a) shows Case 13 where both w_0' and w_1' cover one edge only. Assume w_0' lies on AC. Then, there are three possible ways to propagate w_1: from BC to AB, from AC to AB and from AC to BC. All these configurations induce checking between two windows propagated to the same edge from different sides. A possible solution is the strategy proposed by Liu (2013). In this section, this case is skipped since it requires solving a quadratic equation, which is of relatively high computational cost. Case 14 (only one window is propagated from one edge to two other edges) and Case 15 (both windows are propagated from one edge to two other edges) are also shown in Figure 4.6, where both windows can be split into sub-windows and these cases can be reduced to Cases 1-9 and Case 13.

4.2.2 Principles for Window Pruning

The preceding section exhaustively enumerated the 15 pairwise window pruning cases in a triangle and derived the corresponding pruning rules. In this section, the derived rules are summarized into 8 window pruning principles and proved as follows. In the following discussion, let w_0 and w_1 be two windows, p and q be their respective pseudo sources, σ_0 and σ_1 be the geodesic distances from their pseudo sources to the true source vertex s, and all geometric primitives are assumed to lie on the same plane.

Proposition 4.1.
(1) As Figure 4.7 (a) shows, suppose a line pK in w_0 intersects the upper ray of w_1 at point K. Then, w_1 is redundant if $\sigma_0 + ||pK|| < \sigma_1 + ||qK||$.
(2) As Figure 4.7 (b) shows, suppose a line qK in w_1 intersects the upper ray of w_0 at point K. Then, w_0 is redundant if $\sigma_1 + ||qK|| < \sigma_0 + ||pK||$.

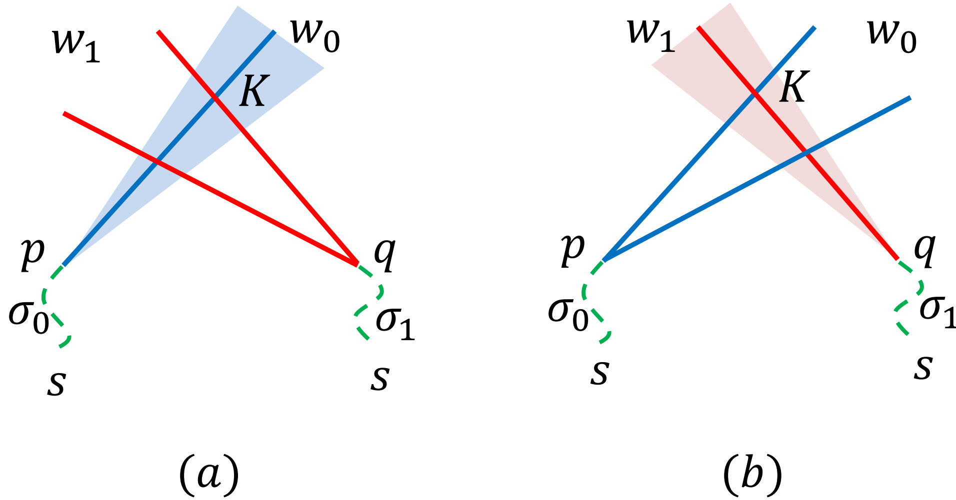


FIGURE 4.7: Window configurations for Proposition 4.1. (a) The blue shadowed area shows the visible cone of window w_0. The red rays are the boundaries of window w_1. (b) The red shadowed area shows the visible cone of window w_1. The blue rays are the boundaries of window w_0.

Proof. Here, only the case in Figure 4.7 (a) is proved since the two cases are symmetrical and can be proved in a similar way.

As shown in Figure 4.7 (a), for any point X on the segment pK, it can be derived that $\sigma_1 + ||qX|| + ||XK|| \geq \sigma_1 + ||qK|| > \sigma_0 + ||pK|| = \sigma_0 + ||pX|| + ||XK||$. Then $\sigma_1 + ||qX|| > \sigma_0 + ||pX||$. This means p can provide

a shorter path from the source to any point on pK that is also inside w_1, and w_1 becomes redundant.

□

Proposition 4.2.
(1) As Figure 4.8 (a) shows, suppose a polyline pGQ from w_0 intersects the upper ray of w_1 at point Q. Then, w_1 is redundant if $\sigma_0 + ||pG|| + ||GQ|| < \sigma_1 + ||qQ||$.
(2) As Figure 4.8 (b) shows, suppose a polyline qHR from w_1 intersects the upper ray of w_0 at point R. Then, w_0 is redundant if $\sigma_1 + ||qH|| + ||HR|| < \sigma_0 + ||pR||$.

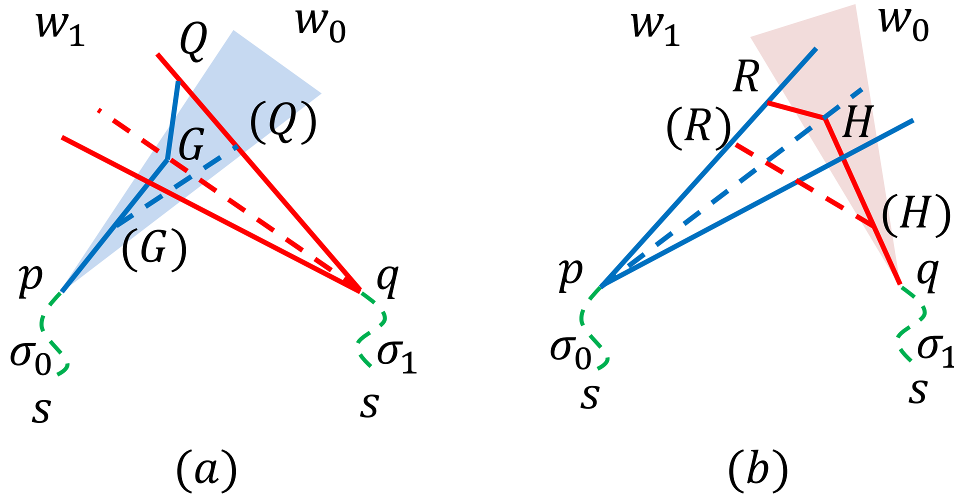


FIGURE 4.8: Window configurations for Proposition 4.2. (a) The blue shadowed area shows the visible cone of window w_0. The red rays are the boundaries of window w_1. (b) The red shadowed area shows the visible cone of window w_1. The blue rays are the boundaries of window w_0.

Proof. Here, only the case in Figure 4.8 (a) is proved since the two cases are symmetrical and can be proved in a similar way.

As shown in Figure 4.8 (a), the case where G lies outside w_1 can be reduced to the case in Figure 4.7 (a) where G is set to p. When G lies inside the visible cone of w_1, w_1 is split into two sub-windows with the dashed red line shown in Figure 4.8 (a). For the upper sub-window, the conclusion can be reached directly by taking G as p in Figure 4.7 (a); for the lower sub-window, it can be derived that $\sigma_0+||pG||+||GQ|| < \sigma_1+||qQ|| \leq \sigma_1+||qG||+||GQ||$.

Then, $\sigma_0 + ||pG|| < \sigma_1 + ||qG||$ is reached, which means w_1 is redundant according to Figure 4.7 (a) and Proposition 4.1 (1). □

Proposition 4.3.
(1) As Figure 4.9 (a) shows, considering the same scenario as in Figure 4.7 (a), let E be a point on the extended part of pK and D a point on the segment qK. Then, w_1 is redundant if $\sigma_0 + ||pE|| < \sigma_1 + ||qD|| + ||DE||$.
(2) As Figure 4.9 (b) shows, considering the same scenario as in Figure 4.7 (b), let S be a point on the extended part of qK and O a point on the segment pK. Then, w_0 is redundant is $\sigma_1 + ||qS|| < \sigma_0 + ||pO|| + ||OS||$.

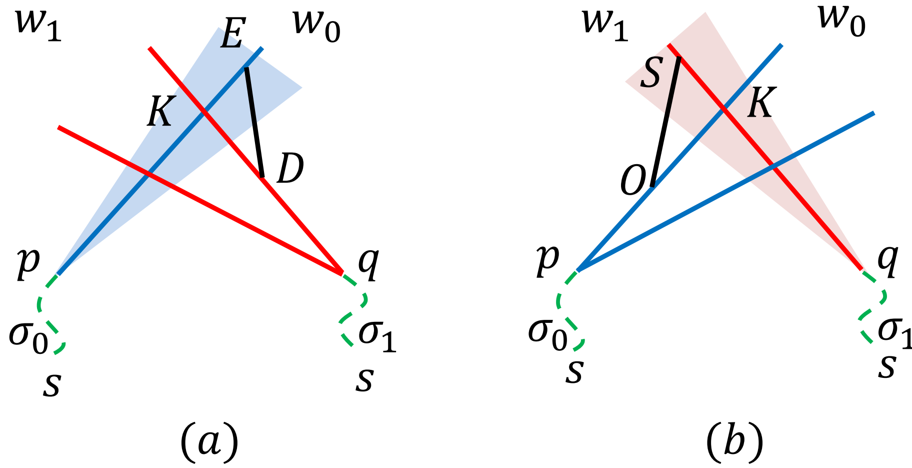


FIGURE 4.9: Window configurations for Proposition 4.3. (a) The blue shadowed area shows the visible cone of window w_0. The red rays are the boundaries of window w_1. (b) The red shadowed area shows the visible cone of window w_1. The blue rays are the boundaries of window w_0.

Proof. Here, only the case in Figure 4.9 (a) is proved since the two cases are symmetrical and can be proved in a similar way.

As shown in Figure 4.9 (a), it can be derived that $\sigma_0 + ||pK|| + ||KE|| < \sigma_1 + ||qD|| + ||DE|| \leq \sigma_1 + ||qK|| + ||KE||$. Then, $\sigma_0 + ||pK|| < \sigma_1 + ||qK||$ is reached, and the conclusion can be derived using Proposition 4.1 (1). □

Proposition 4.4.
(1) As Figure 4.10 (a) shows, considering the same scenario as in Figure 4.8 (a), let F be a point on the extended part of GQ and D a point on the segment qK. Then, w_1 is redundant if $\sigma_0 + ||pG|| + ||GF|| < \sigma_1 + ||qD|| + ||DF||$.
(2) As Figure 4.10 (b) shows, considering the same scenario as in Figure 4.8

(b), let T be a point on the extended part of HR and O a point on the segment pK. Then, w_0 is redundant if $\sigma_1 + ||qH|| + ||HT|| < \sigma_0 + ||pO|| + ||OT||$.

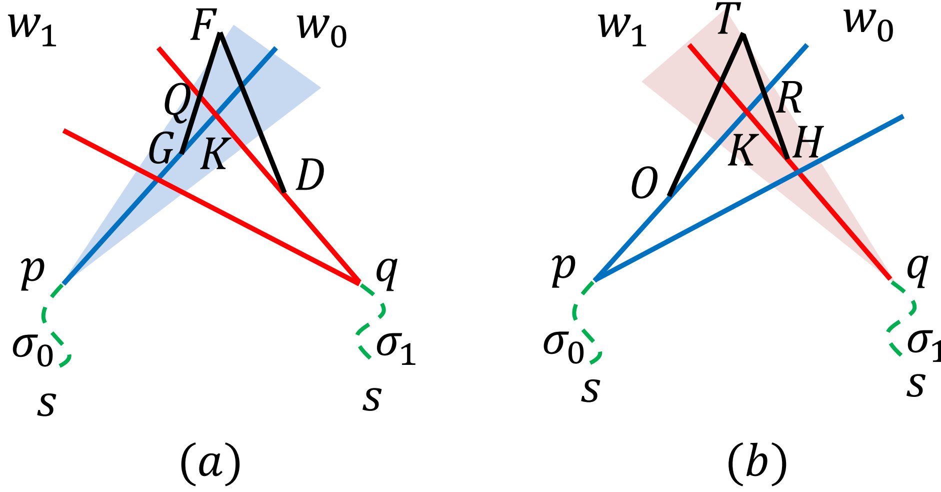


FIGURE 4.10: Window configurations for Proposition 4.4. (a) The blue shadowed area shows the visible cone of window w_0. The red rays are the boundaries of window w_1. (b) The red shadowed area shows the visible cone of window w_1. The blue rays are the boundaries of window w_0.

Proof. Here, only the case in Figure 4.10 (a) is proved since the two cases are symmetrical and can be proved in a similar way.

As shown in Figure 4.10 (a), it can be derived that $\sigma_0 + ||pG|| + ||GQ|| + ||QF|| < \sigma_1 + ||qD|| + ||DF|| \leq \sigma_1 + ||qQ|| + ||QF||$. Then, $\sigma_0 + ||pG|| + ||GQ|| < \sigma_1 + ||qQ||$ is reached, which means w_1 is redundant according to Proposition 4.2 (1). □

4.2.3 EWG-based Window List Propagation

Most of the window pruning operations discussed in Section 4.2.1 perform cross checking between pairs of nearby windows. If more windows are accumulated in a triangle and propagate simultaneously, more window pairs can be formed and the window pruning can be carried out more thoroughly. To make this happen, the local window propagation within the same triangle are synchronized by employing the *EWG-based window propagation framework* (Section 3.3.2.2), which simultaneously propagates a collection of windows from one triangle edge to its two opposite edges.

In this section, three rules are proposed for the window list propagation and window pruning within a triangle. Rule 1 splits a window list into two sublists before propagation and each sublist only needs to be propagated to another single edge. Rule 2 propagates a window list from one edge to another edge and efficiently prunes those propagated windows that have just become redundant. Rule 3 merges two window lists propagated from different source edges to the same destination edge.

Although an exhaustive list of scenarios for pairwise window pruning within a triangle has been identified, the proposed three rules in this section do not perform cross checking for all window pairs. Instead, they prune redundant windows by performing pairwise checking between spatially adjacent windows only. As a result, they do not remove all redundant windows. The rationale behind this strategy is that all-pairs checking is too expensive while pairwise checking between spatially adjacent windows can already remove most of the redundant windows. This will be validated in Section 4.2.4.

4.2.3.1 Window List Splitting

Let ΔABC be the triangle where the window list propagation is performed. Consider a window list $wl = \{w_0, w_1, ..., w_k\}$ on edge AB, and this window list is going to be propagated across ΔABC. First, define the distance to C via wl as $wl.dis = \min_i\{w_i.\sigma + dist(w_i.p, w_i.sp) + dist(w_i.sp, C)\}$. In fact, this distance defines the length of the shortest path on the mesh from the source vertex s to C routed through the windows in wl. Let the window supporting the shortest distance to C be w_s. Next, define the separating point of wl as $wl.sp = w_s.sp$. Note that the separating point of a window $w_i \in wl$ can be calculated using the intersection between $PC(P = w_i.p)$ and AB. Let this intersection be t. It can be derived that $w_i.sp = w_i.a_0$ if $t < w_i.a_0$, $w_i.sp = w_i.a_1$ if $t > w_i.a_1$ and $w_i.sp = t$ otherwise.

Then, the Proposition 4.5 for splitting a window list is proposed as follows. It is based on Cases 7-9 in Figure 4.4 and Case 10 in Figure 4.5.

Proposition 4.5. ***One Angle Two Sides (Rule 1)***
For each window $w \in wl$ and $w \neq w_s$, the propagation of w to BC is redundant if $w.sp < wl.sp$, and the propagation of w to AC is redundant if $w.sp > wl.sp$.

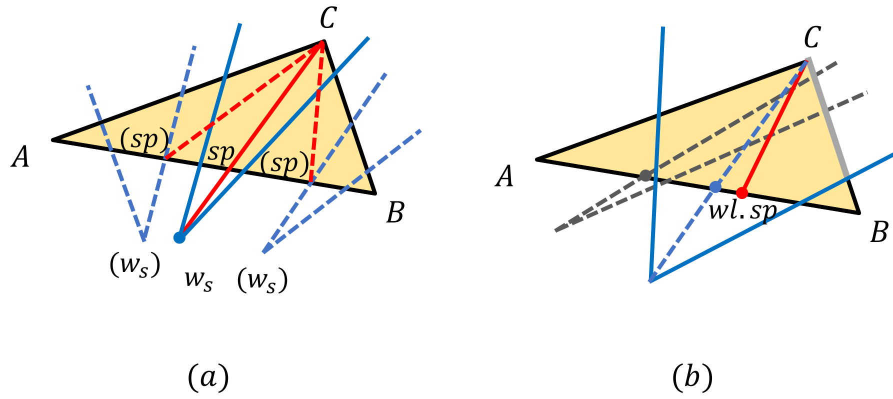


FIGURE 4.11: One Angle Two Sides (Rule 1).

Proof. Let w'_s be the propagated version of w_s within ΔABC. As shown in Figure 4.11 (a), there are three possible positions of w'_s with respect to vertex C. For each $w \in wl$, let w' be the propagated version of w. The proof proceeds by enumerating all window configurations and the corresponding window pruning rules from Section 4.2.1.

(1) If $w'_s \in AC$ and $w' \in BC$ and $w.sp < w_s.sp$, $w_s.sp = w_s.a_1$, w' is redundant according to Case 7 in Figure 4.4.

(2) If $w'_s \in AC$ and $w' \in AC$ and $w.sp > w_s.sp$, $w_s.sp = w_s.a_1$, w' is redundant according to Case 10 in Figure 4.5.

(3) If $w'_s \in AC$ and w' covers C, $w_s.sp = w_s.a_1$, according to Case 8 rule 1) in Figure 4.4, the part of w' on BC is redundant if $w.sp < w_s.sp$, and the part of w' on AC is redundant if $w.sp > w_s.sp$.

(4) If $w'_s \in BC$ and $w.sp < w_s.sp$ and $w' \in BC$, $w_s.sp = w_s.a_0$, w' is redundant according to the symmetric Case of Case 10 in Figure 4.5.

(5) If $w'_s \in BC$ and $w.sp > w_s.sp$ and $w' \in AC$, $w_s.sp = w_s.a_0$, w' is redundant according to Case 7 in Figure 4.4.

(6) If $w'_s \in BC$ and w' covers C, $w_s.sp = w_s.a_0$, the conclusion can be reached according to Case 8 rule 2) in Figure 4.4.

(7) If w'_s covers C and w' covers C, the conclusion can be reached according to Case 9 in Figure 4.4.

(8) If w'_s covers C and $w' \in AC$ and $w.sp > w_s.sp$, w' is redundant according to Case 8 rule 1) in Figure 4.4.

(9) If w'_s covers C and $w' \in BC$ and $w.sp < w_s.sp$, w' is redundant according to Case 8 rule 2) in Figure 4.4. □

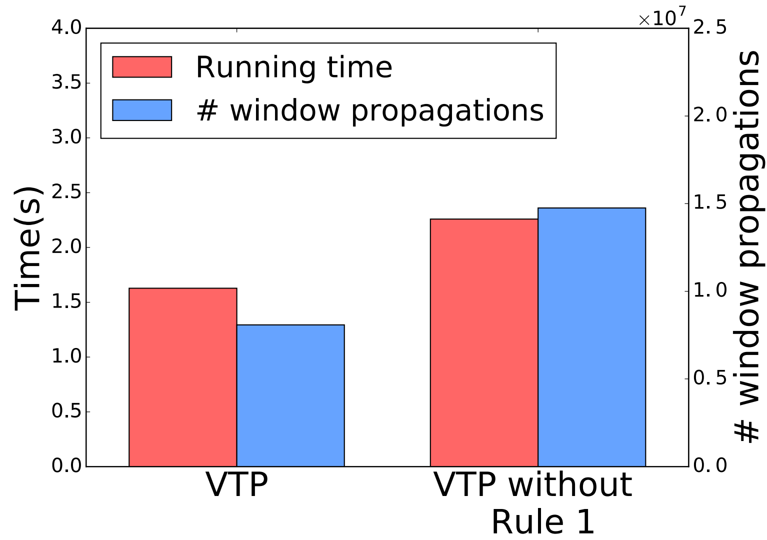
Note that this proposition is applicable to all three possible configurations

of w_s shown in Figure 4.11 (a). Intuitively, windows on AB can be split into two subsets by w_s. This rule is named as the “One Angle Two sides" since it can be considered as a generalized version of the “One-Angle-One-Split" rule proposed by Chen and Han (1990). In fact, the original “One-Angle-One-Split" rule is equivalent to performing crossing checking when both windows cover vertex C, which is exactly Case 9 in Figure 4.4. This generalized rule is actually much more powerful. It performs more thorough window pruning by taking into account three novel cases (Case 7, Case 8 and Case 10).

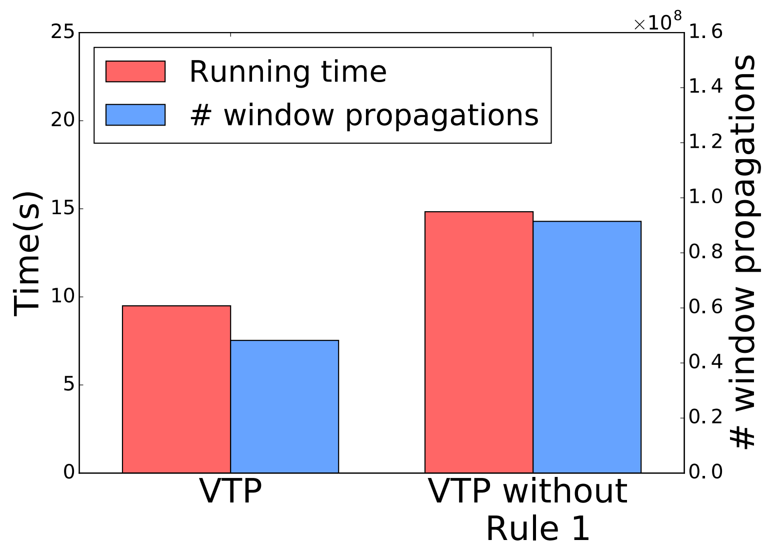
The detailed procedure of enforcing Rule 1 has the following steps: first, compute separating points $w_i.sp(i = 1, ..., k)$ for all windows and locate w_s; second, split window $w_i(i \neq s)$ into two sub-windows if w_i covers vertex C, and remove a subset of the updated windows using Proposition 4.5; third, remaining windows that should be propagated to AC form a window list wl_{left} and those windows that should be propagated to BC form another window list wl_{right}; finally, update the vertex distance by setting $D(C)$ to $wl.dis$ if $wl.dis < D(C)$. As shown in Figure 4.11 (b), the dashed gray window and the gray side of the blue window are pruned since their sp (dark point and blue point) lies on the left of $wl.sp$ (red point).

To validate the performance of Rule 1, a comparison is conducted between the VTP algorithm and its variant without Rule 1 (Figure 4.12). That is, when Rule 1 is turned off, to split a window list to two sides, two sub-windows for each window that covers two opposite edges are created and no pruning operations are performed on any windows or sub-windows. This comparison is conducted on ten models with different model sizes, and the results on two models (Armadillo and Asian Dragon) are shown in Figure 4.12. The rest of the results have been included in Appendix B. It can be observed that Rule 1 saves approximately 25% running time and 50% window propagations.

Remark Rule 1 prunes many redundant windows using the separating point of a window list before propagating them. Compared with MMP (Mitchell et al., 1987; Surazhsky et al., 2005) and ICH algorithms (Xin and Wang, 2009) which only perform window trimming or filtering after propagation, the proposed Rule 1 significantly reduces the time for propagating redundant windows.



Armadillo (F: 345K)



Asian Dragon (F: 1.4M)

FIGURE 4.12: Ablation study on Rule 1. The left y-axis represents running time and the right y-axis represents the number of window propagations.

4.2.3.2 Window List Propagation

After enforcing Rule 1, the window list wl has been cleaned and split into wl_{left} and wl_{right}. In this section, the method to perform window pruning when propagating each sublist from one edge to another edge is discussed. Extending from the method in Section 3.3.2.2, the workflow for propagating $wl_{left} = \{w_0, w_1, ..., w_m\}$ from AB to AC is given below (propagating wl_{right} from AB to CB is similar).

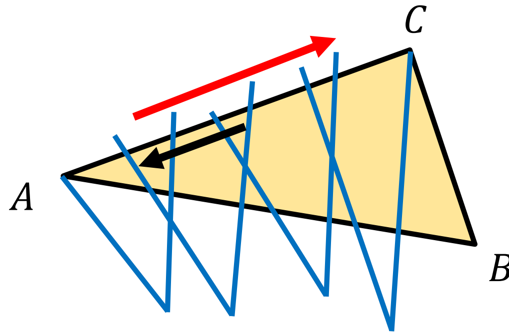


FIGURE 4.13: Window List Propagation (Rule 2).

Procedure $WindowListPropagation(wl)$ (**Rule 2**)
Step 0. Perform one step of propagation for all windows in wl. Let w_i' be the propagated version of w_i.
Step 1. Set $i = wl.head$ and $j = i + 1$.
Step 2. If $j == NULL$, finish; otherwise, perform pairwise window pruning between w_i and w_j according to Case 1, Case 2 and Case 3 in Figure 4.2.
Step 3. If w_j is removed from the list in Step 2, set $j = j + 1$ and goto Step 2. In the event that w_i is removed in Step 2, if $i == wl.head$, set $i = j, j = j+1$ and goto Step 4; otherwise, set $i = i - 1$ and goto Step 2. If neither w_i nor w_j is removed, set $i = j$, $j = j + 1$ and goto Step 4.
Step 4. If $j == NULL$, finish; otherwise, goto Step 2.

There is a double loop in the above procedure. Index j is associated with the outer loop and index i is associated with the inner loop. This procedure is illustrated in Figure 4.13, where it traverses all windows in the outer loop (red arrow) and checks each window against its preceding windows in the inner loop (black arrow). Its time complexity is $O(m)$, which is proved by Proposition 4.6 as follows.

Proposition 4.6. *Applying Rule 2 to a window list with N windows costs $O(N)$ time.*

Proof. Let $wl = \{w_0, w_1, w_2, ..., w_n\}$ be the input window list, and t_i be the number of times pairwise cross checking is performed between w_i and its preceding windows. Since pairwise cross checking between w_i and its preceding windows is terminated only when it reaches a preceding window that cannot be removed, such cross checking removes $t_i - 1$ redundant windows. In the worst case, pairwise cross checking between w_{i+1} and its preceding windows needs to be performed $i + 1$ times, and the total number of redundant windows removed before w_{i+1} is reached is $\sum_{k=1}^{i}(t_k - 1)$. It can be derived that $t_{i+1} \leq i+1-\sum_{k=1}^{i}(t_k-1)$, that is $\sum_{k=0}^{i+1} t_k \leq 2i+1$. Thus $\sum_{k=0}^{n} t_k \leq 2n+1$, which indicates linear time complexity. □

Checking with vertices During the process of enforcing Rule 2, for each propagated window on AC, Case 12 in Figure 4.5 is also applied by checking the window against the distance to vertices, which is the same as the filtering rule in ICH (Xin and Wang, 2009).

The ablation study on Rule 2 is shown together with Rule 3 in the next section since they have similar functions.

4.2.3.3 Window List Merging

Suppose a window list $wl^l = \{w_0^l, w_1^l, ..., w_m^l\}$ is given on AC, which is propagated from AB. In this section, the following procedure to propagate windows from another list $\widetilde{wl}^r = \{w_0, w_1, ..., w_n\}$ from BC to AC is presented, and the propagated windows are merged with wl^l. Meanwhile, the window pruning is performed on the merged window list using Cases 1-6 in Figure 4.2 and Figure 4.3.

Procedure $PrimeMerge(wl^l, \widetilde{wl}^r)$ (**Rule 3**) consists of the following steps. First, perform one step of propagation for all windows in $\widetilde{wl}^r$. Let w_i' be the propagated version of w_i. Then, for each window from w_0' to w_n', run the following substeps: (i) append it to wl^l; (ii) set $j = wl^l.tail$ and $i = j - 1$; (iii) perform pairwise checking and pruning on the updated wl^l using Steps 2-4 in Rule 2 except that in Step 2, instead of considering Cases 1-3 only, it is required to check where the two windows are from and use either Cases 1-3 (if both windows are propagated from the same edge) or Cases 4-6 (if the two windows are propagated from two different edges).

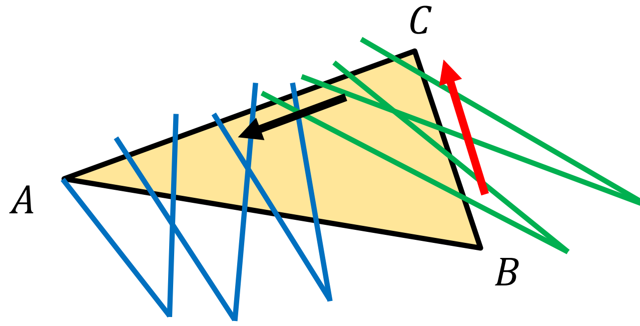


FIGURE 4.14: Window List Merging (Rule 3).

This procedure is named $PrimeMerge()$ because it will be used for merging window lists on an edge for the first time. It is complementary to procedure $SecondMerge()$ in Section 4.3.1. Figure 4.14 shows an illustration of the main loop (red arrow) and the inner loop (black arrow) of this procedure. The time complexity of $PrimeMerge()$ is $O(m+n)$, which is proved by Proposition 4.7.

Proposition 4.7. *$PrimeMerge(wl, wl')$ costs $O(M+N)$ time, where M and N are the number of windows in wl and wl', respectively.*

Proof. The complexity of $PrimeMerge()$ can be derived in a similar way as the proof of Proposition 4.6 from that it enforces a variant of Rule 2 on the appended window list of size $M+N$. Thus, performing $PrimeMerge()$ costs $O(M+N)$ time. □

Order Preservation. A window list $wl = \{w_0, w_1, ..., w_k\}$ is spatially coherent if $w_i.a_0 \leq w_{i+1}.a_0$ for all $i = 0, ..., k-1$.

Proposition 4.8. *If both wl_{AB} and wl_{BC} are spatially coherent, the window list $wl^l = wl_{AB \to AC}$ obtained after applying Rule 1 and Rule 2 is also spatially coherent. And the merged list obtained after applying Rule 3 is still spatially coherent.*

Proof. Rule 1 in Section 4.2.3.1 does not affect the order of the windows in the list. In the following, Rule 2 is first proved by induction to preserve the spatial order. Consider propagating $wl_{AB} = \{w_0, w_1, ..., w_k\}$ (already split by Rule 1) to AC. Let the propagated version of window w_i be w'_i. During the steps of Rule 2, assume the sublist from w'_0 to w'_i are already spatially coherent. Then, it is needed to prove that both Step 2 and Step 3 in Rule 2 preserve the spatial order of the sublist from w'_0 to $w'_j (j = i+1)$. Let

$a_0 = w_i.a_0$, $a_1 = w_i.a_1$, $b_0 = w_j.a_0$, $b_1 = w_j.a_1$ and $a'_0 = w'_i.a_0$, $a'_1 = w'_i.a_1$, $b'_0 = w'_j.a_0$, $b'_1 = w'_j.a_1$. The following cases are discussed where w'_j is not spatially coherent with w'_i:
(i) $b'_1 < a'_0$. This corresponds to Case 3 in Figure 4.2. One of the windows is redundant and should be removed.
(ii) $b'_0 < a'_0 < b'_1$. This corresponds to Case 2 in Figure 4.2. After being checked against the rules in this case, if both windows survive, w'_j must have been partially trimmed. The trimmed w'_j has become spatially coherent with w'_i and its preceding windows.

For all these cases, the removal of w'_j does not affect the spatial order of its preceding sublist and the removal of w'_i triggers pairwise cross checking between w'_j and w'_{i-1}. As such checking is continued until a spatially coherent window is found or all preceding windows are removed, the conclusion can be reached.

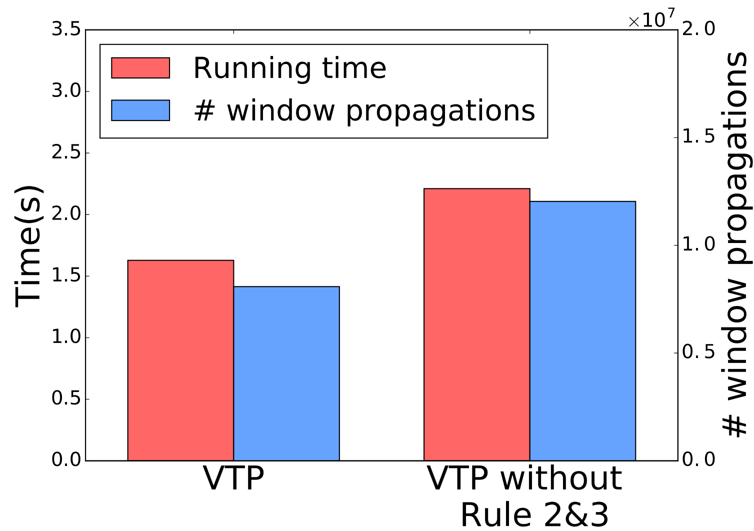
The order preservation property of Rule 3 can be reached according to Cases 1-6 in Figure 4.2 and Figure 4.3 in a similar way. □

To validate the performance of Rule 2 and 3, a comparison is conducted between the VTP algorithm and its variant without Rule 2 and 3 (Figure 4.15). Here, these two rules are not dealt with separately since they are two similar rules to maintain the spatial coherence of window lists. When both Rules 2 and 3 are turned off, no pairwise cross checking are performed after propagating a window list or merging two window lists. This comparison is also conducted on ten models, and the results on two models (Armadillo and Asian Dragon) are shown in Figure 4.15. The rest of the results are shown in Appendix B. It can be observed that Rules 2 and 3 together save approximately 15% running time and 30% window propagations.

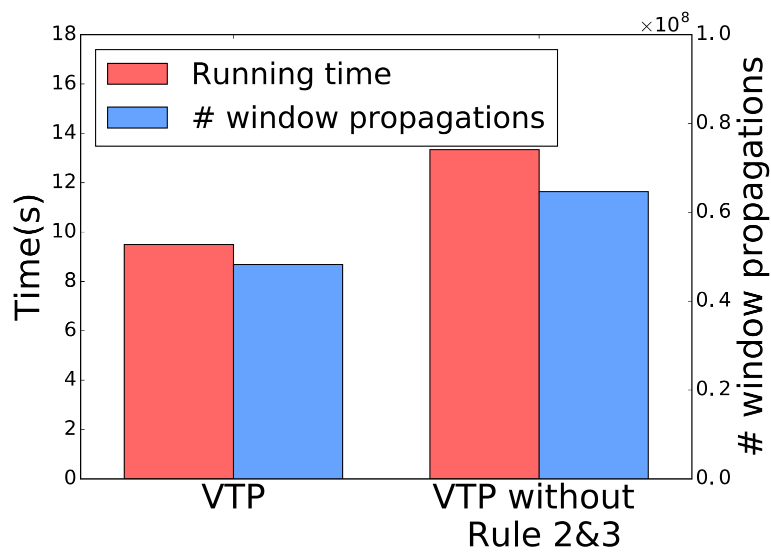
4.2.4 Algorithmic Choices Justification

To justify the algorithmic choices of VTP in terms of the window list propagation, this section compares it against four alternatives: VTP-Exhaustive, VTP-Trimming, VTP-MMP and VTP-CH.

VTP-Exhaustive *vs.* VTP To reduce window redundancy more thoroughly, EWG is employed by VTP to form more window pairs. However, VTP does not perform the pairwise window pruning between all pairs of windows in a window list but only the spatially adjacent one. Although this strategy does



Armadillo (F: 345K)



Asian Dragon (F: 1.4M)

FIGURE 4.15: Ablation study on Rule 2 and 3. The left y-axis represents running time and the right y-axis represents the number of window propagations.

not remove all redundant windows, it is time-efficient and removes most of them. To justify it, VTP is compared with VTP-Exhaustive, which is a variant of VTP that performs exhaustive pairwise window pruning in a window list. Table 4.1 shows the comparison results on five representative models. The complete results are shown in Appendix C. It can be seen that VTP-Exhaustive performs fewer window propagations but runs much slower than VTP.

Model	Performance	Algorithms	
		VTP-Exhaustive	VTP
Bunny (F:144K)	Time(s)	4.557	0.78
	# window propagations	4,801,056	4,943,670
	Peak Memory(MB)	1.22	1.24
Rocker Arm (F:482K)	Time(s)	36.586	4.13
	# window propagations	24,289,066	25,654,638
	Peak Memory(MB)	3.43	3.70
Asian Dragon (F:1,400K)	Time(s)	49.954	9.495
	# window propagations	46,926,451	48,217,896
	Peak Memory(MB)	4.036	4.373
Neptune (F:4,008K)	Time(s)	665.847	47.629
	# window propagations	239,054,124	246,364,008
	Peak Memory(MB)	15.62	16.38
Lucy (F:14,464K)	Time(s)	16559	549.934
	# window propagations	2,703,707,866	2,808,823,718
	Peak Memory(MB)	66.096	69.42

TABLE 4.1: Performance comparison between VTP-Exhaustive and VTP on running time, peak memory usage and total number of window propagations. F: means the number of faces on a model.

VTP-Trimming *vs.* VTP In VTP, the window redundancy is removed by performing the proposed window pruning rules based on simple distance comparisons. An alternative to these rules is the "window trimming" rule used by the MMP algorithm (Surazhsky et al., 2005). However, it requires solving quadratic equations and is of high time cost. To justify the choice of the proposed window pruning rules, VTP is compared with VTP-Trimming, which is a variant of VTP that performs window trimming instead of window pruning following the proposed Rule 2 and Rule 3 when two windows overlap on an edge. Table 4.2 shows the comparison results on five representative models. The complete results are shown in Appendix C. It can be seen that VTP-Trimming runs slower than VTP though it performs fewer window propagations.

VTP-MMP/CH *vs.* VTP To evaluate the overall performance of the proposed Rules 1,2 and 3, a comparison is conducted between VTP and VTP-MMP/CH, which are two variants of VTP that replace the proposed three

Model	Performance	Algorithms	
		VTP-Trimming	VTP
Bunny (F:144K)	Time(s)	0.872	0.78
	# window propagations	4,686,252	4,943,670
	Peak Memory(MB)	1.146	1.24
Rocker Arm (F:482K)	Time(s)	4.655	4.13
	# window propagations	24,380,006	25,654,638
	Peak Memory(MB)	3.49	3.70
Asian Dragon (F:1,400K)	Time(s)	13.763	9.495
	# window propagations	46,316,630	48,217,896
	Peak Memory(MB)	4.017	4.373
Neptune (F:4,008K)	Time(s)	58.49	47.629
	# window propagations	239,375,390	246,364,008
	Peak Memory(MB)	15.962	16.38
Lucy (F:14,464K)	Time(s)	615.215	549.934
	# window propagations	2,733,324,263	2,808,823,718
	Peak Memory(MB)	66.848	69.42

TABLE 4.2: Performance comparison between VTP-Trimming and VTP on running time, peak memory usage and total number of window propagations. F: means the number of faces on a model.

rules with the window redundancy reduction rules used in the MMP and ICH algorithms (Surazhsky et al., 2005; Xin and Wang, 2009) respectively. Table 4.3 shows the comparison results on five representative models. The complete results are shown in Appendix C. It can be seen that VTP runs the fastest and performs the least window propagations.

Model	Performance	Algorithms		
		VTP-CH	VTP-MMP	VTP
Bunny (F:144K)	Time(s)	2.672	1.304	0.78
	# window propagations	12,491,178	6,454,800	4,943,670
	Peak Memory(MB)	1.71	340.45	1.24
Rocker Arm (F:482K)	Time(s)	15.449	6.954	4.13
	# window propagations	69,208,037	33,947,674	25,654,638
	Peak Memory(MB)	5.32	1797.18	3.70
Asian Dragon (F:1,400K)	Time(s)	29.492	15.388	9.495
	# window propagations	109,311,094	61,995,300	48,217,896
	Peak Memory(MB)	5.253	3354.04	4.373
Neptune (F:4,008K)	Time(s)	158.912	60.297	47.629
	# window propagations	606,937,112	278,925,270	246,364,008
	Peak Memory(MB)	17.65	14221.35	16.38
Lucy (F:14,464K)	Time(s)	1809.91	Out of memory	549.934
	# window propagations	6,859,484,793		2,808,823,718
	Peak Memory(MB)	78.31		69.42

TABLE 4.3: Performance comparison between VTP-MMP, VTP-CH and VTP on running time, peak memory usage and total number of window propagations. F: means the number of faces on a model.

4.3 EWG in Wavefront Propagation Over a Mesh

To remove the redundant windows at the earliest stage, VTP employs the continuous Dijkstra technique (Mitchell et al., 1987) to propagate the wavefront from near to far using a priority queue. To avoid the high computational costs of such priority queues in previous methods (Surazhsky et al., 2005; Xin and Wang, 2009; Xu et al., 2015), EWG is employed to change the sorting elements from windows to mesh primitives (e.g. vertices). This is achieved by the connections between EWG window lists and mesh edges. Since the number of mesh primitives is much smaller than that of windows, the window management cost of VTP is significantly cut down.

The wavefront propagation of VTP is implemented by a triangle-oriented region growing scheme. In this scheme, all visited triangles form a single connected region, called the traversed area, over the mesh surface. The boundary of this traversed area is defined as the propagation wavefront. The proposed algorithm expands this traversed area in a continuous Dijkstra style by gradually enclosing unvisited triangles abutting the traversed area. During each iteration, the proposed algorithm adds one or more unvisited triangles to the traversed area, and the wavefront is also updated. Let R and R' be the existing and expanded traversed area respectively. The region outside R but inside R' is denoted as ΔR, which consists of the newly added triangles. This section is organized as follows:

- Section 4.3.1 presents a basic face-sorted propagation algorithm.
- Section 4.3.2 extends the propagation algorithm from face-sorted to vertex-sorted, which achieves improved performance.
- Section 4.3.3 justifies the algorithmic choices of the proposed VTP algorithm in terms of the wavefront propagation.

4.3.1 Face-Sorted Wavefront Propagation

As shown in Figure 4.16 (a) and (b), the proposed face-sorted geodesic algorithm expands the traversed area one triangle face at a time. Its outline is given below.

Initialization. Create a single window for every opposite edge of S in its 1-ring neighborhood (bold blue lines around S in Figure 4.16 (a)), and push all

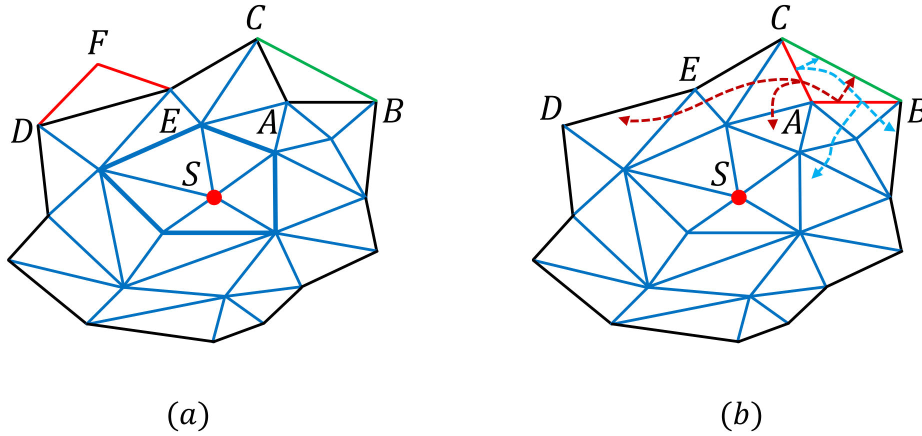


FIGURE 4.16: Face-sorted wavefront propagation.

triangles that are outside the 1-ring neighborhood of S and share at least one opposite edge of S to Q. Set $D(S) = 0$, $D(P) = dist(S, P)$ if P is a 1-ring neighbor of S, and $D(V) = \infty$ for all other vertices.

Wavefront Propagation.
Step 1. Pop the triangle with the highest priority from Q and add it to R. This single triangle forms ΔR.
Step 2. If this triangle has only one edge on the previous wavefront (ΔDEF in Figure 4.16 (a)), propagate the window list on DE to both DF and FE using Rule 1 and Rule 2. Push adjacent triangles sharing either DF or FE with ΔDEF into Q and calculate their priority; if any of these adjacent triangles is already in Q, simply update its priority.
Step 3. If the popped triangle has two edges on the previous wavefront (ΔABC in Figure 4.16 (a)), run procedure $GeodesicUpdate()$ on each of the window lists residing on CA and AB, respectively. $GeodesicUpdate()$ updates geodesic distances at vertices in the expanded traversed area R' while propagating the given window list inside both R and ΔR. Push the adjacent triangle sharing BC with ΔABC into Q and calculate its priority.
Step 4. If Q is empty, finish; otherwise, goto Step 1.

The priority of a triangle is defined as follows. For a triangle which shares only one edge with the wavefront, such as ΔDEF in Figure 4.16 (a), its priority is defined as the negative distance from S to F via wl_{DE}, which is $-wl_{DE}.dis$, where $wl_{DE}.dis$ is defined in Section 4.2.3.1. For a triangle which shares two edges with the wavefront, such as ΔABC in Figure 4.16 (a), its priority is defined as the larger one between $-wl_{AB}.dis$ and $-wl_{CA}.dis$.

Geodesic Update. Once a new triangle, such as ΔABC in Figure 4.16 (a),

has been added to the traversed area, windows inside ΔR are propagated from previous wavefront edges (such as CA and AB) to new wavefront edges in ΔR (such as BC). In addition, windows inside R need to be propagated again along previously unexplored paths, such as $AB \rightarrow AC \rightarrow$ the interior of R and $AC \rightarrow AB \rightarrow$ the interior of R, because these paths might give rise to smaller geodesic distances from S to some vertices in R. Therefore, window lists are propagaed along these paths, and the geodesic distances at vertices in R are updated along the way until all the propagated windows have been either removed by windows pruning rules or merged into window lists residing on edges of the wavefront in R.

Note that when there are two window lists on the same wavefront edge, they need to be merged. The window list mergers at edges on the wavefront are classified into two categories. Mergers taking place at new wavefront edges in ΔR are called *prime mergers* while mergers taking place at wavefront edges in R are called *secondary mergers*. Prime mergers are handled by Rule 3 in Section 4.2.3.3, and secondary mergers will be discussed later in this section.

Procedure $GeodesicUpdate(wlist)$

Step 1 Push $wlist$ to an FIFO queue W.

Step 2 Pop a window list wl from W. If wl is on an internal edge e of the expanded traversed area R' and the propagation tries to enter a triangle f from e, propagate wl to the two opposite edges of e in the triangle f using Rule 1 and Rule 2. Meanwhile, update the distances at vertices if needed, and push the non-empty propagated window lists on the opposite edges into W.

Step 3 If wl resides on a wavefront edge (e_w), save wl on e_w. If $e_w \in \Delta R$ and e_w already has another window list wl_{e_w}, run $PrimeMerge(wl, wl_{e_w})$; otherwise, if $e_w \in R$ and e_w already has another window list wl_{e_w}, run $SecondMerge(wl, wl_{e_w})$.

Step 4 If W is empty or all propagated windows have been pruned, finish; otherwise, goto Step 2.

During geodesic update, window lists are propagated not only towards the wavefront, but also towards the interior of the previously traversed area to make sure none of the paths is overlooked. When multiple window lists reach the same edge on the wavefront, they are merged; but when they reach the same edge inside the traversed area, they move forward independently without any interaction. There are multiple reasons for this strategy.

- First, whether merging window lists reaching the same edge or not only affects efficiency, but does not affect the overall correctness of the proposed geodesic algorithm.

- Second, merging window lists reaching the same edge on the wavefront is useful because it removes redundant windows at an early stage and only propagates a compact set of windows towards the unvisited area of the mesh, thus reducing the computational cost in later stages.

- Third, such a merger at an internal edge of the traversed area is not as important because near optimal distance values have been computed at most vertices in this area and these distance values can prune windows very effectively. Therefore, all windows entering the traversed area would be eventually pruned after being propagated a small number of steps. This statement is verified by the experiment on the distribution of window propagations (Section 4.3.3), which show that on average 96.25% propagations are prime propagations.

Order-Preserving Secondary Merger. When a secondary merger is performed at an edge on the wavefront, the merged window list can be made spatially coherent to ensure its further propagation compatible with the proposed three rules. Let an existing window list at an edge e_w on the wavefront be wl_{e_w}, and an incoming window list propagated from another edge to e_w during geodesic update be $wl^g = \{w_0^g, w_1^g, ..., w_k^g\}$. The procedure $OPSecondMerge(wl_{e_w}, wl^g)$ incrementally inserts each window from wl^g into wl_{e_w} by performing a binary search in the ordered list of the first endpoints of all windows in wl_{e_w}. Its time complexity is analysed as follows.

Proposition 4.9. *$OPSecondMerge(wl, wl')$ costs $O(N \log(M + N))$ time, where M and N are the number of windows in wl and wl', respectively.*

Proof. The complexity of $OPSecondMerge()$ can be derived easily using the fact that a binary search has logarithmic complexity. □

Order-Free Secondary Merger. Since the frequency of secondary mergers is relatively low in the proposed algorithm, the following simple secondary merging scheme is also designed, which does not strictly maintain spatial coherence in the merged window list.

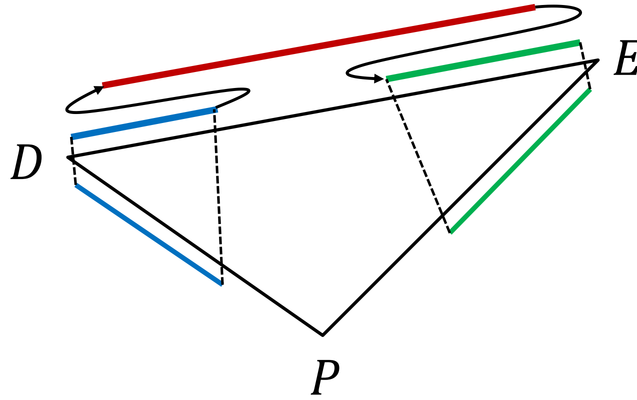


FIGURE 4.17: Order-free secondary merger.

Procedure $SecondMerge(wl_{DE}, wl^g)$ (Figure 4.17): If wl^g is propagated from PE, append it to the tail of wl_{DE}; if wl^g is propagated from DP, append the entire wl_{DE} to the tail of wl^g .

The order-free secondary merger is finally employed by the proposed algorithm since it runs faster than the order-preserving version, which does not remove many more redundant windows and sometimes even keeps slightly more redundant windows (verified by the experiment in Section 4.3.3). The reason is threefold:

- First, the merged window list from an order-free secondary merger is still piecewise ordered. Windows in the list propagated from DP are likely located to the left of the windows in wl_{DE}, and windows in the list propagated from PE are likely located to the right of the windows in wl_{DE}.
- Second, the procedure for order-preserving secondary merger only enforces spatial coherence, and leaves window pruning to the next iteration.
- Third, the order-free version does not incur the cost of binary search, and thus reaches a better trade-off.

The face-oriented propagation algorithm with order-free secondary merger is named as FTP, and the version with order-preserving secondary merger is named as OPFTP.

4.3.2 Vertex-Sorted Wavefront Propagation

Instead of expanding one triangle at a time, the wavefront could expand multiple triangles every time. A natural choice for the latter case adds all unvisited triangles in the 1-ring neighborhood of a vertex on the wavefront to the traversed area during every expansion. A variant of the proposed geodesic algorithm based on this expansion scheme is given below. The priority queue in this variant holds all vertices on the wavefront, and the priority of a vertex in the queue is simply defined as the negative most recently updated distance at the vertex.

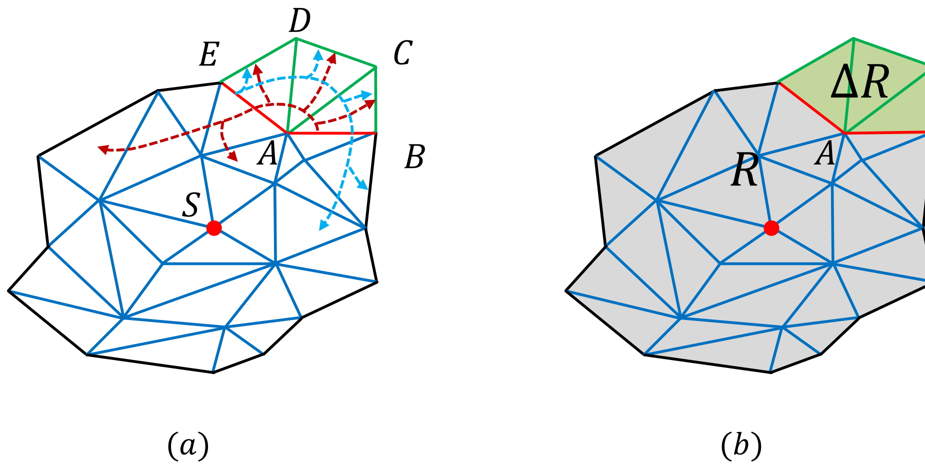


FIGURE 4.18: Vertex-sorted wavefront propagation.

The revised geodesic algorithm also proceeds in a continuous Dijkstra style. As shown in Figure 4.18 (a), a vertex A with the highest priority is chosen from the priority queue Q in each iteration. Unvisited triangles in the 1-ring neighborhood of A are added to the traversed area. And run procedure $GeodesicUpdate()$ on each of wl_{AB} and wl_{AE} respectively (AB and AE are the two edges on the previous wavefront incident to A). The geodesic update process is constrained within the updated traversed area R'.

This variant of the proposed algorithm is finally employed since it runs faster than FTP, which is verified in Section 4.3.3. The reason for its high performance is that it propagates windows on the wavefront through multiple newly added triangles during each iteration and, thus, reduces the overall data management overhead. Although this variant processes multiple triangles every time, it only runs procedure $GeodesicUpdate()$ twice since the triangles are connected.

This vextex-sorted propagation algorithm with order-free secondary merger is named as VTP, and the version with order-preserving secondary merger is named as OPVTP. The pseudo code of VTP is shown in Algorithm 2, which is the final version of the proposed algorithm.

Algorithm 2 EWG-based SS-DGP Algorithm (VTP)

Input: M - Mesh, S - Source set;
Output: D - a vector containing the geodesic distances of M's vertices;
1: **procedure** VTP(M, S)
2: Denote the wavefront as wf and the area it encloses as R;
3: Define a priority queue Q and a FIFO queue W;
4: Perform initialization as in Section 4.3.1;
5: Push all adjacent vertices of S into Q;
6: **while** Q is not empty **do**
7: Pop a vertex v from Q;
8: Let $E(v)$ be the subset of nonincident 1-ring edges of v;
9: Push the edges on the wavefront incident to v into W;
10: **while** W is not empty **do**
11: Pop an edge e from W;
12: Suppose the opposite edges of e are e_0 and e_1
13: Propagate wl_e to e_0 and e_1 using Rules 1 and 2;
14: Update distance vector D and priority queue Q;
15: Let the propagated lists be $wl_{e \to e_0}$ and $wl_{e \to e_1}$
16: **for** each edge e_i **do**
17: **if** $wl_{e \to e_i}$ is not empty **then**
18: Let wl_{e_i} be the existing window list on e_i
19: **if** $e_i \notin R$ and $e_i \in E(v)$ **then**
20: PrimeMerge($wl_{e \to e_i}$, wl_{e_i});
21: **else if** $e_i \in wf$ **then**
22: SecondMerge($wl_{e \to e_i}$, wl_{e_i});
23: **else**
24: Push e_i to W;
25: **end if**
26: **end if**
27: **end for**
28: **end while**
29: Update wavefront wf and R.
30: **end while**
31: **end procedure**

Saddle Vertices. Finally, this part discusses how to handle the following scenarios where a saddle vertex (pseudo source) is visited. The first scenario applies to FTP only. When a triangle ΔABC sharing two edges with the previous wavefront joins the traversed area as shown in Figure 4.16 (a), and A is a saddle vertex. The second scenario applies to VTP only. When a vertex

A with the highest priority is chosen from the previous wavefront, and A is a saddle vertex. Third, when the distance at v, a vertex not on the wavefront, is updated during the $GeodesicUpdate()$ procedure, v is a saddle vertex. The following special procedure is performed in all these scenarios: for each opposite edge in the 1-ring neighborhood of the saddle vertex, create a new window as in the ICH algorithm (Xin and Wang, 2009). If this window lies on an edge of the new wavefront, add it to the beginning of the existing window list on this edge; otherwise, call $GeodesicUpdate()$ to propagate this single window inside R. In addition, the distance at every 1-ring neighbor of the saddle vertex is updated as in (Xin and Wang, 2009).

4.3.3 Algorithmic Choices Justification

To justify the algorithmic choices of VTP in terms of wavefront propagation, this section compares VTP against two alternatives: FTP and OPVTP, and counts the the distribution of window propagations.

Model	Performance	Algorithms	
		FTP	VTP
Bunny (F:144K)	Time(s)	1.044	0.78
	# window propagations	4,755,872	4,943,670
	Peak Memory(MB)	1.20	1.24
Rocker Arm (F:482K)	Time(s)	4.84	4.13
	# window propagations	25,013,422	25,654,638
	Peak Memory(MB)	3.68	3.70
Asian Dragon (F:1,400K)	Time(s)	13.223	9.495
	# window propagations	46,525,313	48,217,896
	Peak Memory(MB)	4.10	4.373
Neptune (F:4,008K)	Time(s)	64.113	47.629
	# window propagations	243,102,435	246,364,008
	Peak Memory(MB)	15.90	16.38
Lucy (F:14,464K)	Time(s)	617.343	549.934
	# window propagations	2,668,122,127	2,808,823,718
	Peak Memory(MB)	67.81	69.42

TABLE 4.4: Performance comparison between FTP and VTP on running time, peak memory usage and total number of window propagations. F: means the number of faces on a model.

FTP *vs.* VTP The difference between VTP and FTP is that VTP adds more faces to the traversed area than FTP during each iteration and lets windows on the previous wavefront propagate through these faces to reach the new wavefront. This strategy pushes the wavefront forward faster. To justify the choice of VTP over FTP as the final algorithm, a comparison is performed between them. Table 4.4 shows the comparison results on five representative models.

The complete results are shown in Appendix C. It can be seen that VTP runs faster than FTP even though it usually performs more window propagations.

OPVTP *vs.* VTP This part compares VTP with OPVTP by applying them to all the 55 testing models (Appendix A). Table 4.5 shows the comparison results on five representative models. The complete results are shown in Appendix C. It can be seen that enforcing spatial coherence increases the overall running time but does not remove many more redundant windows. Therefore, VTP strikes a better balance between the overall speed and the thoroughness in window pruning.

Model	Performance	Algorithms	
		OPVTP	VTP
Bunny (F:144K)	Time(s)	0.908	0.78
	# window propagations	4,875,712	4,943,670
	Peak Memory(MB)	1.22	1.24
Rocker Arm (F:482K)	Time(s)	5.173	4.13
	# window propagations	25,723,669	25,654,638
	Peak Memory(MB)	3.71	3.70
Asian Dragon (F:1,400K)	Time(s)	11.42	9.495
	# window propagations	47,573,341	48,217,896
	Peak Memory(MB)	4.20	4.373
Neptune (F:4,008K)	Time(s)	60.192	47.629
	# window propagations	244,586,129	246,364,008
	Peak Memory(MB)	16.19	16.38
Lucy (F:14,464K)	Time(s)	608.414	549.934
	# window propagations	2,734,517,299	2,808,823,718
	Peak Memory(MB)	68.01	69.42

TABLE 4.5: Performance comparison between OPVTP and VTP on running time, peak memory usage and total number of window propagations. F: means the number of faces on a model.

Distribution of Window Propagations As mentioned in Section 4.3.2, in an iteration of the proposed VTP algorithm, R and R' stand for the existing and expanded traversed areas, respectively, and ΔR stands for the region outside R but inside R' (Figure 4.18). Denote the window propagations taking place inside ΔR prime propagations, and those inside R secondary propagations. Then, these two types of propagations are counted on all 55 testing models (Appendix A) and the results are given in Appendix D. The results show that on average 96.25% propagations are prime propagations. This finding confirms that it is important to perform window propagation and pruning efficiently and thoroughly inside ΔR and the proposed algorithmic designs satisfy this demand. On the other hand, there are few secondary propagations and the overall performance would not be much affected by the thoroughness of window pruning during secondary propagations and mergers. Thus, the decision to perform order-free secondary mergers is reasonable.

4.4 Complexity Analysis

Let n be the number of vertices on the mesh. First, the complexities of FTP and OPFTP algorithms are discussed. It is easy to verify that these algorithms are improved versions of the original CH algorithm (Chen and Han, 1990), in the worst case, the number of created windows are still $O(n^2)$. It is obvious that it takes linear time to execute Rule 1 (Section 4.2.3.1), and $SecondMerge()$ (Section 4.3.1) costs $O(1)$ time. According to Proposition 4.6 and 4.7, all window list propagation and pruning operations in FTP have linear complexity with respect to the windows involved. Therefore, their total cost is $O(n^2)$, and the same tasks in OPFTP cost $O(n^2 \log n)$, where $\log n$ is due to the binary search in $OPSecondMerge()$ (Proposition 4.9). For window management, since FTP organizes triangle faces instead of windows, the time complexity of this part is $O(n \log n)$. In summary, the time complexity of FTP is $O(n^2 + n \log n) = O(n^2)$, and the time complexity of OPFTP is $O(n^2 \log n + n \log n) = O(n^2 \log n)$.

Likewise, the time complexity of VTP is $O(n^2)$, and the time complexity of OPVTP is $O(n^2 \log n)$. Similar to all existing algorithms, the space complexity of the proposed algorithms is also $O(n^2)$.

4.5 Experimental Results

To validate the performance of the proposed VTP algorithm, experiments have been conducted to compare it against existing state-of-the-art algorithms: MMP, ICH, FWP-MMP and FWP-CH (Surazhsky et al., 2005; Xin and Wang, 2009; Xu et al., 2015), on a variety of models. Specifically, 55 models are collected, including sculptures, animals and manmade objects, for testing (Appendix A). The resolution of these models (number of faces) ranges from 10K to 14M. To evaluate overall performance, three measures, i.e. running time, total number of window propagations and peak memory usage, are employed. All the algorithms are tested using a PC with an Intel Core i7-3770 3.40GHz CPU and 32GB memory. For better reading experience, the results on some of the testing models are shown in this section and all the remaining results are given in the Appendix. In the experiments, the first vertex on all meshes is chosen as the source vertex. To make fair comparisons, a fixed threshold $\epsilon = 10^{-6}$ is adopted for distance comparisons throughout the following experiments as Xu et al. (2015). Prior to performance comparisons,

the correctness of the VTP algorithm is conformed by comparing the results from VTP against those from the exact implementation of the MMP algorithm (Surazhsky et al., 2005).

4.5.1 Overall Performance

All algorithms in the comparison have been tested on all 55 models and the detailed results are given in Appendix E. The mean and standard deviation of performance ratios are calcuated between other algorithms and the proposed VTP algorithm. The details are shown in Table 4.6.

	ICH vs. VTP	*MMP vs. VTP*	*FWP-CH vs. VTP*	*FWP-MMP vs. VTP*
Time	6.21/1.88	6.11/1.90	3.37/0.57	2.03/0.33
# window propagations	2.26/0.23	1.24/0.11	2.28/0.30	1.25/0.11
Memory	1.21/0.13	377.78/300.635	1.22/0.13	377.78/300.651

TABLE 4.6: The mean and standard deviation of performance ratios between other algorithms and the proposed VTP algorithm on running time, the number of window propagations and peak memory usage.

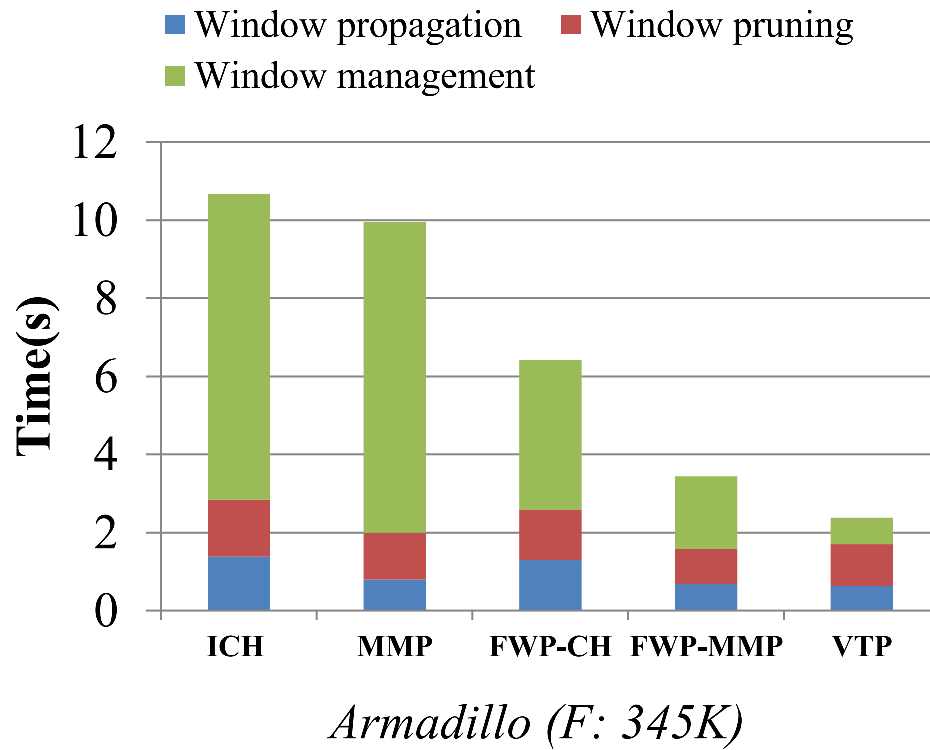
It can be seen from the table that the proposed VTP algorithm on average runs 6 times as fast as both ICH and MMP, more than 3 times as fast as FWP-CH, and twice as fast as FWP-MMP. The VTP algorithm has 56% fewer window propagations than ICH and FWP-CH, and 20% fewer window propagations than MMP and FWP-MMP. Furthermore, the VTP algorithm on average uses 17% less memory than ICH and FWP-CH, and 99.7% less memory than MMP and FWP-MMP. Note that MMP algorithms are fast but memory intensive while existing CH algorithms are memory efficient but relatively slow. The proposed VTP algorithm is impressive in the sense that it achieves the best performance in both aspects. For example, it uses 99.7% less memory than FWP-MMP while still being twice as fast. Detailed results on 5 representative testing models are shown in Table 4.7.

4.5.2 Performance Profiling

As mentioned in Section 3.3.2.2, all geodesic algorithms based on window propagation have three primary components: window propagation, window pruning (window redundancy reduction) and window management. The running times of these individual components in all participating algorithms are profiled on ten models. Here, the results on two models (Armadillo and Asian

Model	Performance	Algorithms				
		ICH	**MMP**	**FWP-CH**	**FWP-MMP**	**VTP**
Bunny (F:144K)	Time(s)	5.034	4.612	3.056	1.737	0.78
	# window propagations	12,305,579	6,485,320	12,327,991	6,451,352	4,943,670
	Peak Memory(MB)	1.69	340.45	1.70	340.46	1.24
Rocker Arm (F:482K)	Time(s)	36.577	33.286	19.536	11.867	4.13
	# window propagations	68,553,846	33,989,638	70,513,186	35,940,386	25,654,638
	Peak Memory(MB)	5.29	1797.16	5.42	1797.19	3.70
Asian Dragon (F:1,400K)	Time(s)	73.204	73.092	35.637	23.674	9.495
	# window propagations	107,742,094	62,161,583	108,122,218	62,025,717	48,217,896
	Peak Memory(MB)	5.184	3354.04	5.207	3354.05	4.373
Neptune (F:4,008K)	Time(s)	455.271	424.331	193.945	120.012	47.629
	# window propagations	585,784,159	270,930,198	602,587,831	284,581,696	246,364,008
	Peak Memory(MB)	16.96	14225.26	17.14	14219.76	16.38
Lucy (F:14,464K)	Time(s)	8894.87	Out of memory	2415.88	Out of memory	549.934
	# window propagations	6,837,670,602		6,841,729,337		2,808,823,718
	Peak Memory(MB)	78.29		78.28		69.42

TABLE 4.7: Performance comparison with state-of-the-art geodesic algorithms on running time, peak memory usage and total number of window propagations. F : means the number of faces on a model.



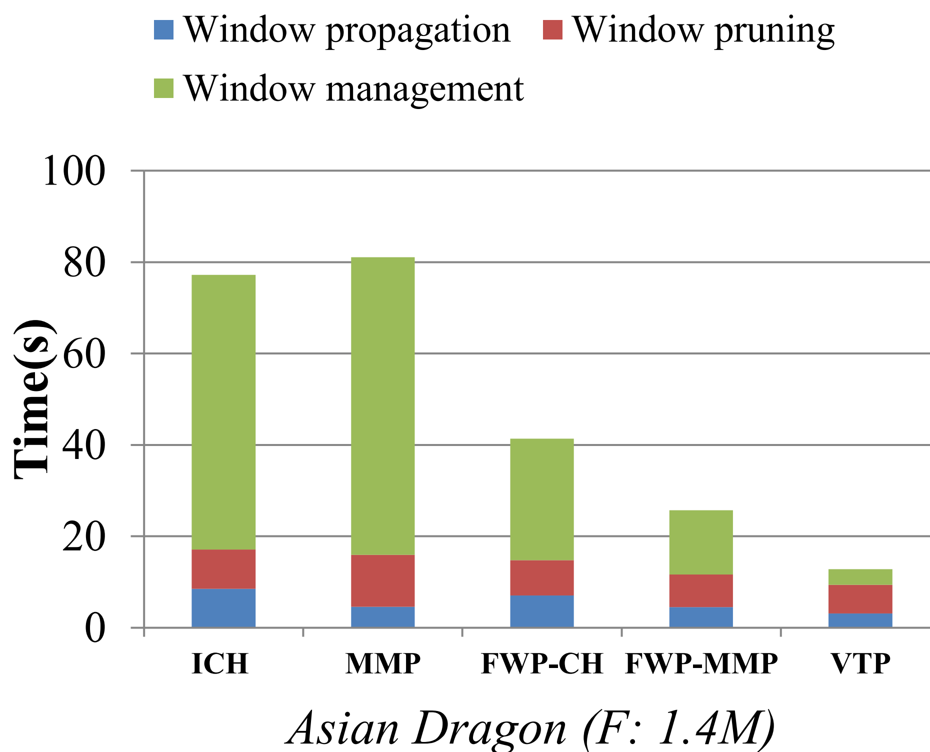


FIGURE 4.19: Comparison of running times of three common components on two models.

Dragon) are shown in Figure 4.19 and the rest of the results have been included in Appendix F. It can be seen that the proposed VTP algorithm outperforms other existing algorithms in the efficiency of all three components. Thanks to the triangle-based propagation strategy, the proposed algorithm cuts down the time spent on priority queue management at the same time performs more thorough window pruning. Without solving any quadratic equations as MMP algorithms, all the proposed pruning rules only require comparisons between two distances. This also reduces the computational cost of window pruning itself.

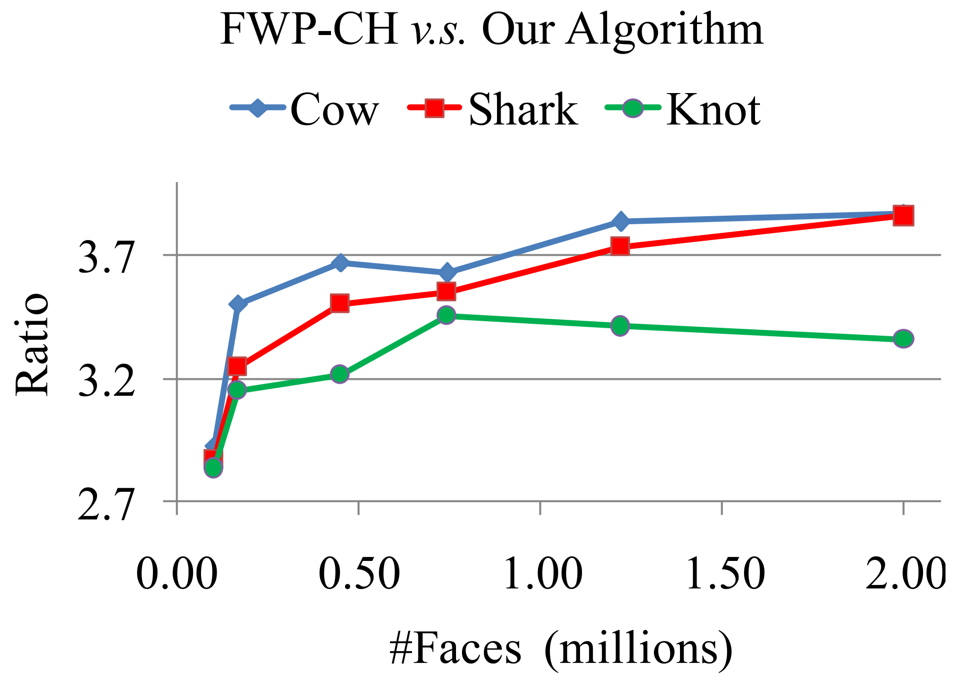
4.5.3 Scalability

The scalability of the proposed VTP algorithm, i.e. how the performance of VTP varies with increasing mesh resolution, is further studied. First, three test models (Cow, Shark and Knot) are chosen. Let each of them have six different resolutions through subdivision. The number of faces ranges from 0.1M to 2M in these subdivided models. For each model, its ratios between the running times of both FWP-CH and FWP-MMP and that of VTP on all six resolutions is calculated. The experiments are designed to show how the ratios change with the changing resolution. As illustrated in Figure 4.20, the timing ratios increase with an increasing resolution. That is, the proposed VTP algorithm achieves more significant performance gain when dealing with larger models.

4.5.4 Robustness

This section further validates that the proposed VTP algorithm is robust to mesh triangulation quality. As in FWP (Xu et al., 2015), a sequence of meshes (eight) with different degrees of anisotropy but a fixed resolution on two testing models (Fertility with 800K faces and Hand with 200K faces) are created respectively. Here, $g(M) = \frac{\Sigma_{f \in F} g'(f)}{|F|}$ is also used to measure the degree of anisotropy of a mesh M, where $g'(f) = \frac{PH}{2\sqrt{3}S}$ and P, H, S are the half-perimeter, longest edge length and area of f respectively. All these meshes with varied degrees of anisotropy are generated using the method in (Zhong et al., 2013).

The curves in Figure 4.21 show how the running times change with increasing anisotropy (g). The proposed VTP algorithm is the most robust



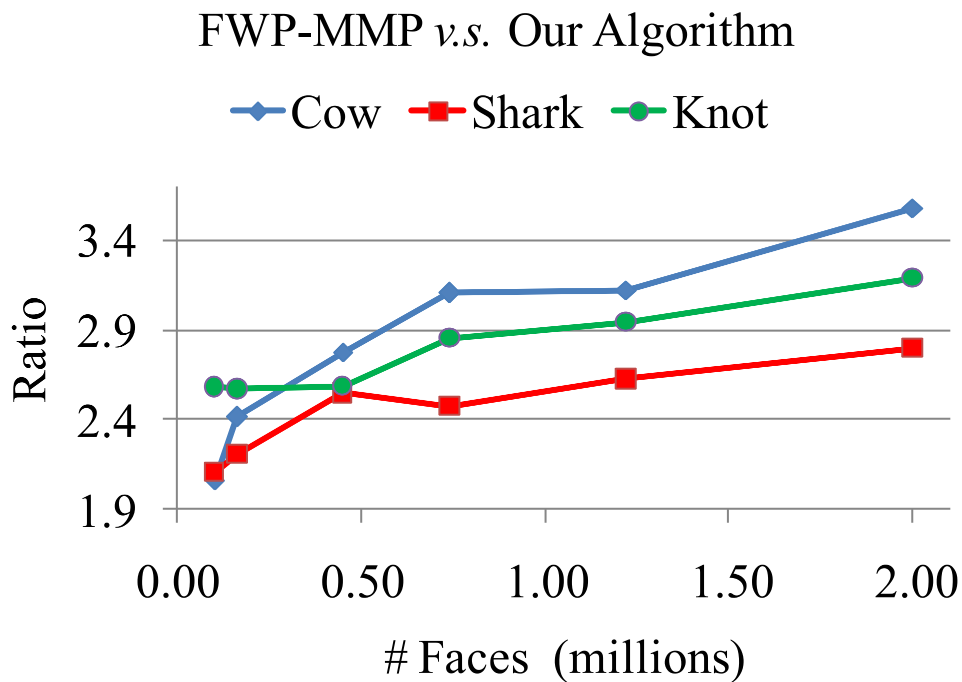


FIGURE 4.20: Comparison of scalability against recent geodesic algorithms. The x-axis represents mesh resolution, and the y-axis represents performance ratio.

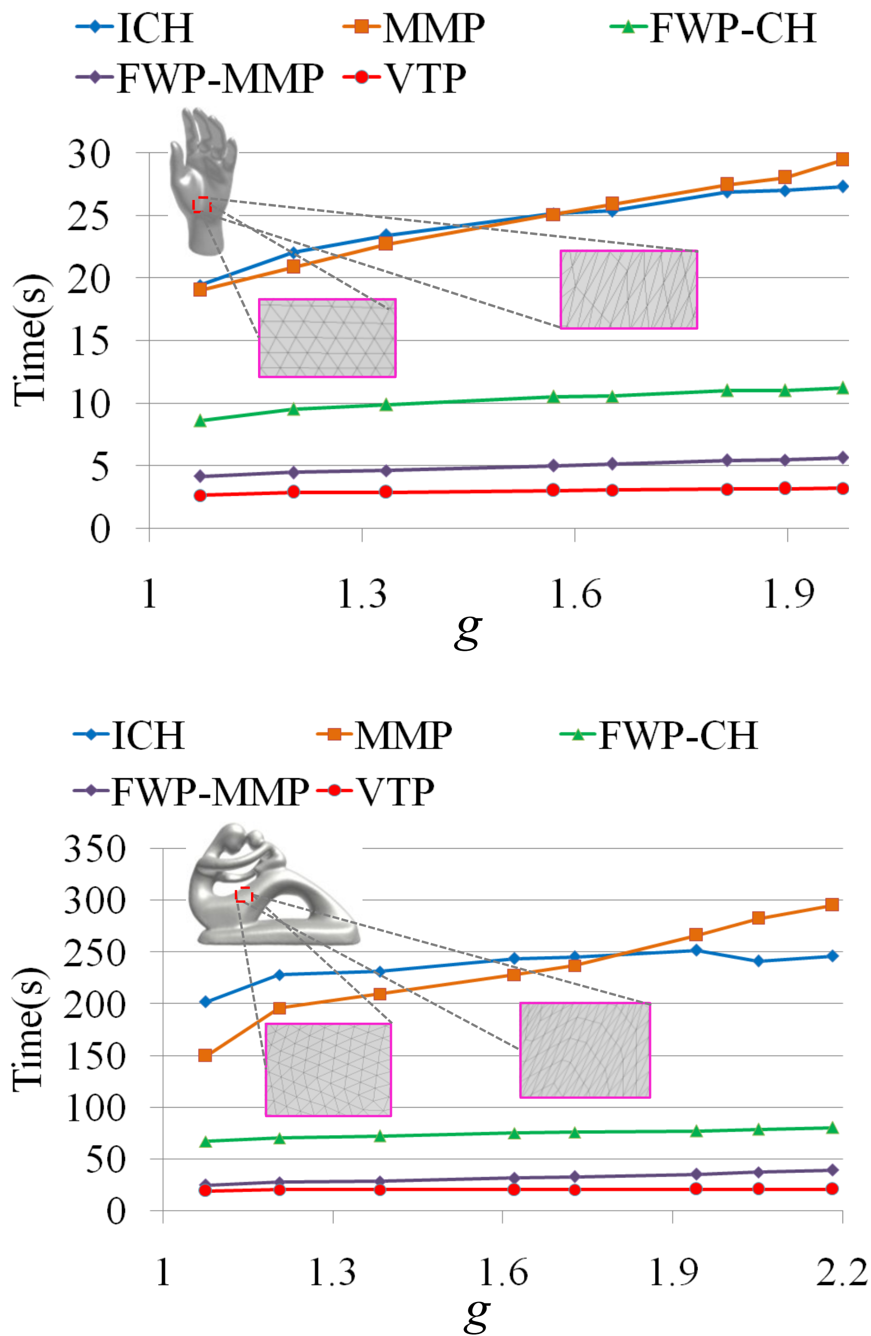


FIGURE 4.21: Comparison of robustness against anisotropic triangulation. The x-axis represents the degree of anisotropy, and the y-axis represents running time.

among all algorithms since its running time does not obviously increase when the input mesh has a much larger anisotropy. As all related algorithms have the same three components (window propagation, window pruning and window management), Figure 4.22, 4.23 and 4.24 further shows how the running times of each component changes with increasing g on the Armadillo model. In general, a larger value of g generates more windows. Without efficient window pruning, ICH and FWP-CH spend more time on window propagation when g is increasing, as shown in Figure 4.22. For window pruning, MMP and FWP-MMP require binary search and solving quadratic equations, which also entails longer running times, as shown in Figure 4.23. As MMP and ICH algorithms use a priority queue for managing windows, they need more time to process more windows, as shown in Figure 4.24. Though FWP-based methods significantly reduce the cost of window management, they still require extra cost to process an increased number of windows, which gives rise to a minor increase in their running times, as shown in Figure 4.24. In contrast, the proposed VTP algorithm is insensitive to the number of windows since it manages triangle vertices instead of windows. As a result, the running times of VTP's components change the least with increasing anisotropy.

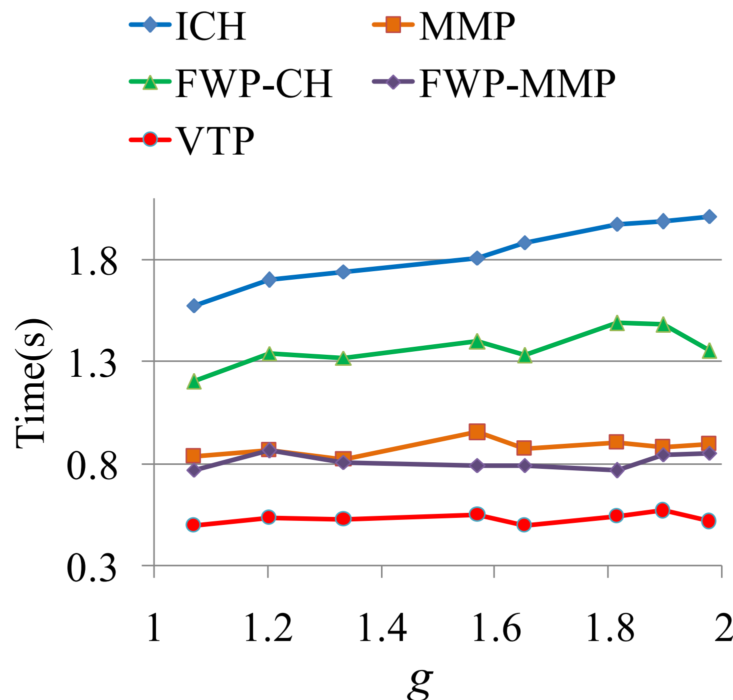


FIGURE 4.22: Robustness of individual components against anisotropic triangulation (window propagation component).

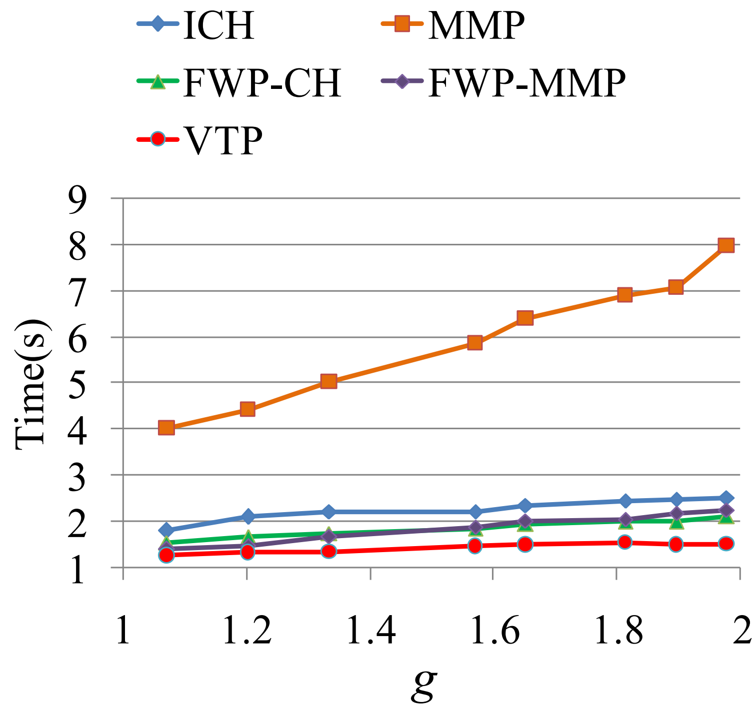


FIGURE 4.23: Robustness of individual components against anisotropic triangulation (window pruning component).

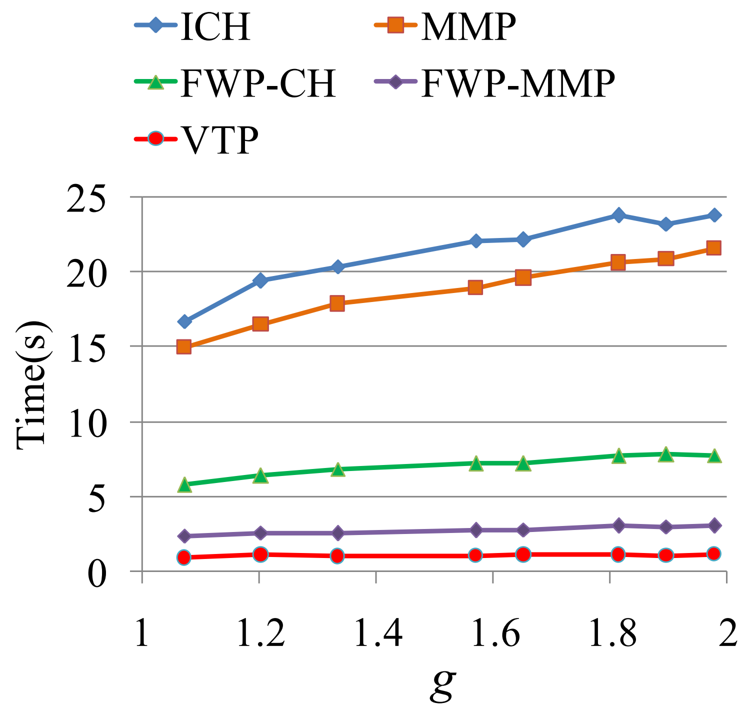


FIGURE 4.24: Robustness of individual components against anisotropic triangulation (window management component).

4.6 Summary

In this chapter, the EWG-based window propagation framework is employed to design a novel exact geodesic algorithm (VTP) in solving the SS-DGP problem. In VTP, EWG is applied to both the window list propagation and the wavefront propagation. For the window list propagation, EWG helps to form more window pairs so that the pairwise windows pruning can be performed more thoroughly. For the wavefront propagation, EWG helps to reduce the window management cost by building connections between mesh edges and window lists. Thus, the sorting can be performed on mesh vertices instead of windows, which significantly cuts down the time spent on window management. As a result, VTP outperforms all recent methods in terms of running time, peak memory usage and the total number of window propagations. According to experiments, the VTP geodesic algorithm runs 4-15 times faster than MMP and ICH algorithms and 2-4 times faster than FWP-MMP and FWP-CH algorithms.

Chapter 5

Fast and Memory-Efficient Voronoi Diagram Construction

Computing geodesic-metric-based Voronoi diagrams on triangle meshes works as a foundation for various applications in computer graphics, including remeshing (Peyré and Cohen, 2006; Liu et al., 2011), surface reconstruction (Peethambaran and Muthuganapathy, 2015) and point pattern analysis (Liu et al., 2011), etc. In these applications, geodesics are used as the distance metric because they reflect the intrinsic properties of surfaces and are invariant to isometric deformations. To construct accurate Voronoi diagrams, Liu et al. (2011) employed the MMP algorithm (Surazhsky et al., 2005) to it. Compared to the VTP algorithm proposed in the previous chapter, the MMP algorithm has a unique feature: all the propagated windows are stored and trimmed on edges. The distinct advantage is to bring necessary geodesic information to edges for Voronoi diagram construction, and it is summarized as the VD-DGP problem in this thesis (Section 3.1.2). However, as the MMP algorithm always consumes lots of time and memory, it has become the bottleneck of constructing geodesic based Voronoi diagrams. Recently, Xu et al. (2015) proposed the FWP-MMP algorithm as an accelerated version of the MMP algorithm. But it still occupies too much memory to be applied to large scale models.

To speed up geodesic computation and save memory, the VTP algorithm proposed in the previous chapter is employed. In addition, it is revised as the *window*-VTP algorithm to provide necessary geodesic information for Voronoi diagram construction as the MMP algorithm do. Moreover for the Voronoi diagram over a mesh, the boundaries of Voronoi cells only occupy a small number of triangles on it. Thus, most of the windows are redundant in constructing Voronoi diagrams.

This chapter aims to reduce redundant computation so as to save time and memory. To this end, the Redundant Window Removal (RWR) process is proposed to remove redundant windows during the construction of a Voronoi diagram, and is involved in the proposed *window*-VTP algorithm by selectively retaining windows on edges. The key point is to detect and remove redundant windows simultaneously with the geodesic wavefront propagation. This chapter is organized as follows:

- Section 5.1 proposes a novel **Redundant Window Removal** (RWR) method to remove redundant windows during the Voronoi diagram construction. In this method, the EWG technique (Chapter 3) is employed to identify and remove the redundant windows efficiently in batches.
- Section 5.2 proposes the *window*-VTP algorithm by revising the VTP algorithm (Chapter 4) to provide the necessary geodesic information for Voronoi diagram construction, and shows how to apply RWR in it.
- Section 5.3 analyses the complexity of the proposed algorithm.
- Section 5.4 shows the experimental results of the proposed algorithm.
- Section 5.5 summarizes this chapter.

5.1 Redundant Window Removal (RWR)

Since the boundaries of Voronoi cells only cross a minority of the meshes' triangles, most of the windows stored on edges are redundant. By applying EWG, these redundant windows are grouped as window lists on edges. Thus, this section aims to identify and remove such windows in batches which occupy a large amount of memory during the Voronoi diagram construction.

5.1.1 Preliminaries

For a triangle mesh M, its Voronoi diagram is a set of Voronoi cells partitioning M. As Figure 5.1 shows, the boundaries separating Voronoi cells are closed curves spread over a small number of triangles. The definitions of Voronoi cells and their boundaries are presented as follows:

Voronoi Cell Definition (Liu et al., 2011). For a given set of source points $s_0, s_1, ..., s_n$ on mesh M, let $D_{s_i}(p)$ be the geodesic distance from source s_i

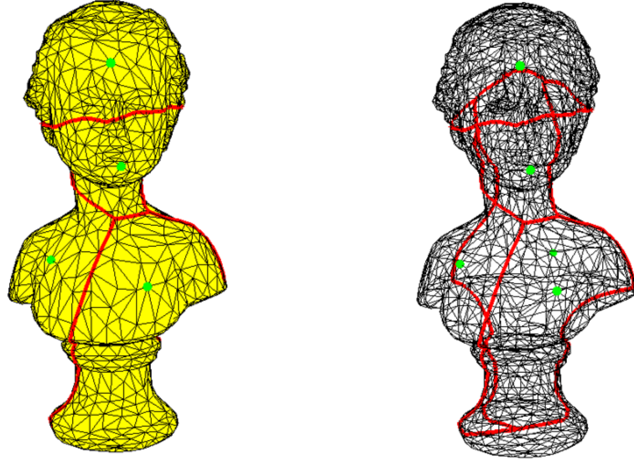


FIGURE 5.1: Voronoi diagram on the Buste model (3K faces). Left: Voronoi diagram on the rendered model. Right: Voronoi diagram on the wireframe model. The green points are sources. The red curves are the boundaries of Voronoi cells.

to point p on M. Consequently, the Voronoi cell (VC) of each source point is defined as:

$$VC(s_i) = \{p | D_{s_i}(p) \leq D_{s_j}(p), i \neq j, p \in M\}$$

Voronoi Boundary Definition. With the Voronoi cell definition above, the boundaries of Voronoi cells are formed by the collection of points q satisfying:

$$\exists i, j \text{ and } \forall k \text{ such that } D_{s_i}(q) = D_{s_j}(q) \leq D_{s_k}(q), i \neq j \neq k \tag{5.1}$$

Window Definition. Following the window definition in Section 3.3.1, a window w is defined as $w = (\Delta ABC, a_0, a_1, p, d_0, d_1, \sigma, s_i)$. Note that a new parameter s_i is added to the definition to record the source vertex from which the window is propagated.

Redundant Window Definition. As Figure 5.2 shows, suppose q is the intersection point of an edge and a Voronoi boundary. Then, q must satisfy the condition Equation 5.1 and is shared by two adjacent windows originating from two different sources respectively. The triangles occupied by the Voronoi boundaries always contain such intersection points. That is, a *valid* triangle contains windows propagated from *different* sources. Otherwise, this triangle is *invalid*. In terms of windows, the redundant primitives on a mesh

are defined as below.

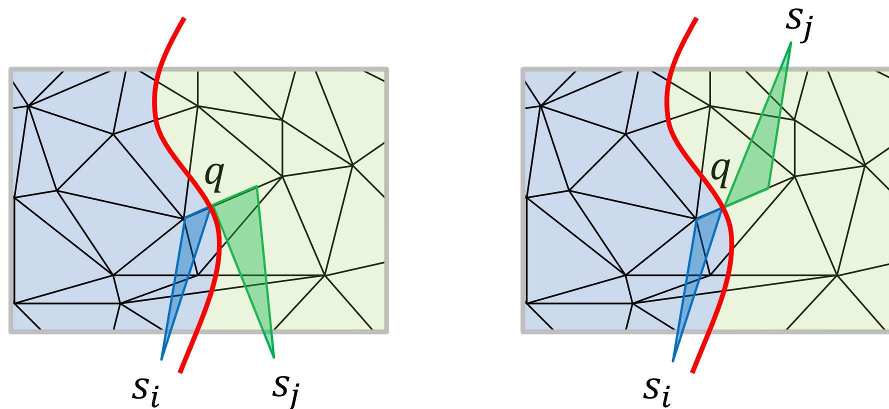


FIGURE 5.2: Illustration of intersections between mesh edges and Voronoi cell boundaries. The left Voronoi cell (in blue) is from source s_i and the right one (in green) is from source s_j. The red curve denotes the boundary between them. Point q is the intersection shared by two windows from s_i and s_j respectively that $D_{s_i}(q) = D_{s_j}(q)$. The two figures show two configurations of the source positions.

Definition 5.1. *Given a mesh M with computed geodesics, the redundant primitives on M are (Figure 5.3):*

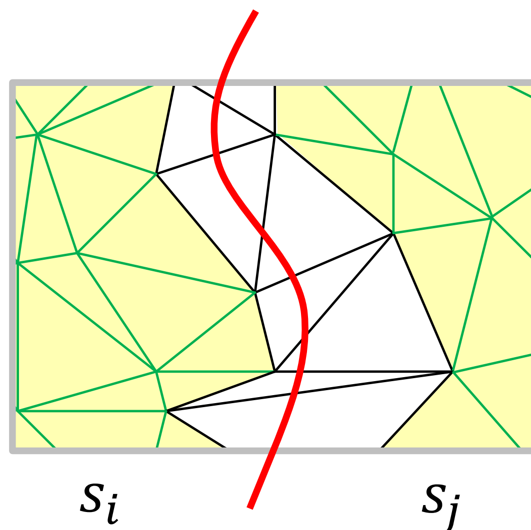


FIGURE 5.3: Illustration of redundant primitives, including redundant triangles (yellow) and redundant edges (green).

- ***Redundant triangle.*** *A triangle is redundant if all the windows on its three edges are from the same source.*
- ***Redundant edge.*** *An edge is redundant if both adjacent triangles are redundant triangles.*

- ***Redundant window.*** *A window is redundant if it resides on a redundant edge.*

5.1.2 Redundant Windows Removal (RWR)

Definition 5.1 can be directly used to identify redundant windows after the termination of geodesic computation on a mesh. However, too much memory have been consumed. To avoid it, the redundant windows must be identified and removed as early as possible *during* the geodesic computation. To this end, the **inactive region** is defined as follows:

Definition 5.2. *An **inactive region** is a region behind the geodesic wavefront, in which all the windows will be no longer updated.*

In other words, the geodesic distances of points in some inactive region have already determined. To depict the inactive region, it is necessary to first briefly address the monotonicity of window propagations.

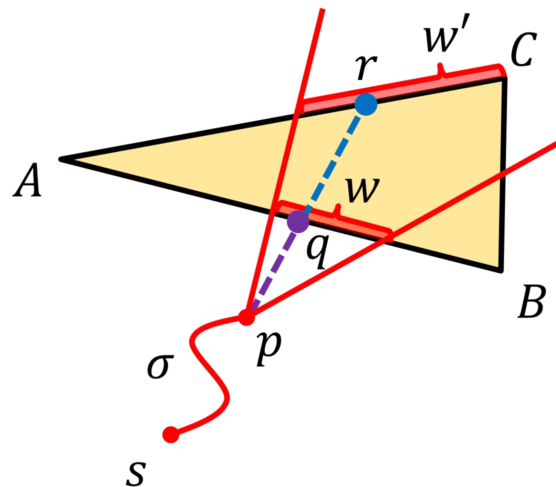


FIGURE 5.4: Illustration of the monotonicity for window propagations. Point r (blue) resides in the window w' propagated from w, segment $\overline{pr}$ intersects edge AB at point q (purple).

Monotonicity. Mitchell et al. (1987) proposed the "continuous Dijkstra" technique to organize geodesic wavefront propagation from near to far *monotonically*. Herein, the wavefront consists of all the windows to be propagated and these windows are managed by a priority queue. In the priority queue, the priority of a window w is defined as $-d_{\min}(w)$, i.e. the negative minimum distance of a window. As Figure 5.4 shows, if w' is a child window propagated

from w, it can be derived that:

$$d_{\min}(w') = \min\left(\sigma + \|\overline{pr}\|\right) \geq \min\left(\sigma + \|\overline{pq}\|\right) \geq d_{\min}(w)$$

That is, the minimum distances of windows popped from the priority queue are *monotonously* increasing.

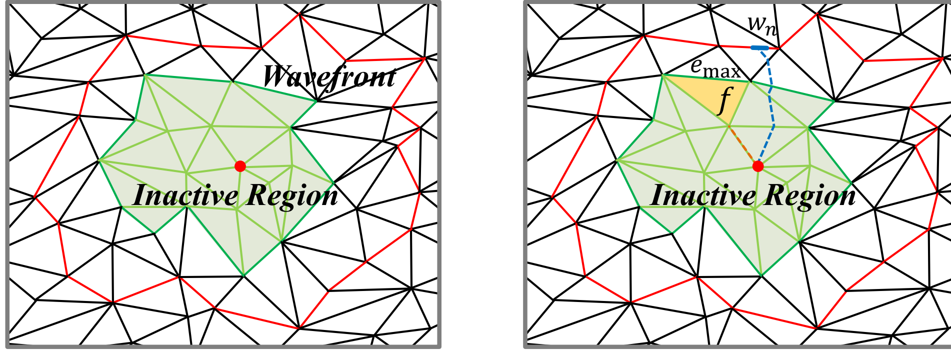


FIGURE 5.5: Illustration of an inactive region. Left: the segments in red denote the propagation wavefront wf and the green shadowed area is the Inactive Region. Right: $d_{\min}(f)$ is the length of the orange path, $e_{\max}$ is the longest edge of face f, $d_{\min}(w_n)$ is the length of the blue path.

Inactive Region Formation. To compute geodesics, windows are organized as the wavefront and propagated from near to far. Let w_n be the nearest window on the wavefront. It can be inferred with the *monotonicity* that the geodesic distance of a point p is determined if it is shorter than $d_{\min}(w_n)$. To apply this to forming the inactive region, the upper bound of points' distances within a triangle is estimated as $d_{\min}(f) + \|e_{\max}\|$, where $d_{\min}(f)$ is the minimum distance of face f, $e_{\max}$ is f's longest edge. Then, all the triangles f satisfying $d_{\min}(f) + \|e_{\max}\| \leq d_{\min}(w_n)$ form the inactive region (Figure 5.5). This process is summarized as Proposition 5.1 and shown as follows.

Proposition 5.1. *The inactive region is formed by all triangles satisfying* $d_{\min}(f) + \|e_{\max}\| \leq d_{\min}(w_n)$ *and none of the windows in it can be updated by later window propagations.*

Proof. Let f be a face satisfying $d_{\min}(f) + \|e_{\max}\| \leq d_{\min}(w_n)$ and q is the point determining $d_{\min}(f)$, i.e. $d_{\min}(f) = \delta + \|pq\| = d(q)$ (Figure 5.6).

Let r be an arbitrary point in any window on the edges of f, construct a path to r by linking q and r with a line segment. Then, the geodesic distance $d(r)$ of r must not be larger than the length of the constructed path, i.e. $d(r) \leq$

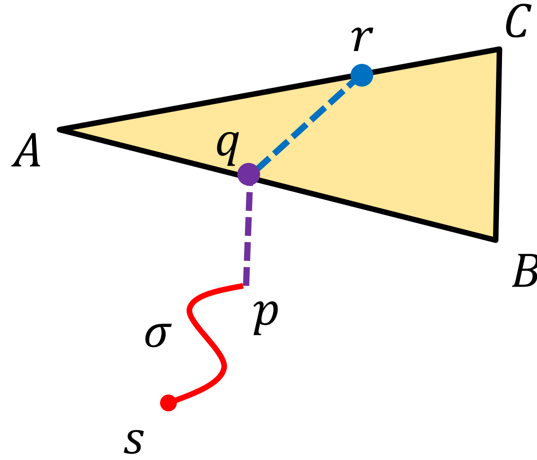


FIGURE 5.6: Illustration of Proposition 5.1.

$d_{\min}(f) + \|qr\|$. Since $\|qr\| \leq \|e_{\max}\|$ (see Lemma G.1 in Appendix G),

$$d(r) \leq d_{\min}(f) + \|qr\| \leq d_{\min}(f) + \|e_{\max}\|$$

Knowing that f satisfies $d_{\min}(f) + \|e_{\max}\| \leq d_{\min}(w_n)$, then

$$d(r) \leq d_{\min}(w_n).$$

Thus, $d(r)$ cannot be updated by w_n since w_n cannot provide a shorter distance to r. Let w_o be any other window on the propagation wavefront that $d_{\min}(w_n) \leq d_{\min}(w_o)$. Then, according to the *monotonicity* of window propagations,

$$d_{\min}(w_n) \leq d_{\min}(w'_n)$$
$$d_{\min}(w_o) \leq d_{\min}(w'_o)$$

where w'_n and w'_o are child windows propagated from w_n and w_o respectively. Then, it can be derived that,

$$d(r) \leq d_{\min}(w_n) \leq d_{\min}(w'_n)$$
$$d(r) \leq d_{\min}(w_n) \leq d_{\min}(w_o) \leq d_{\min}(w'_o)$$

Thus, $d(r)$ cannot be updated by all later window propagations. Since r is arbitrarily selected, all windows on f's edges will not be updated. □

Redundant Windows Removal (RWR) Redundant windows always appear within inactive regions. Thus, RWR works on inactive regions. Let f be a redundant triangle for removal, $d = d_{\min}(w_n)$ be the distance of the nearest window on the propagation wavefront. Then, RWR is performed in two steps:

Step 1. Judge if f is in the inactive region with Proposition 5.1. If so, continue to Step 2; else, finish.
Step 2. Check f's redundancy with Definition 5.1. If f is redundant, also check if its edges are redundant and remove all windows on the redundant edges.

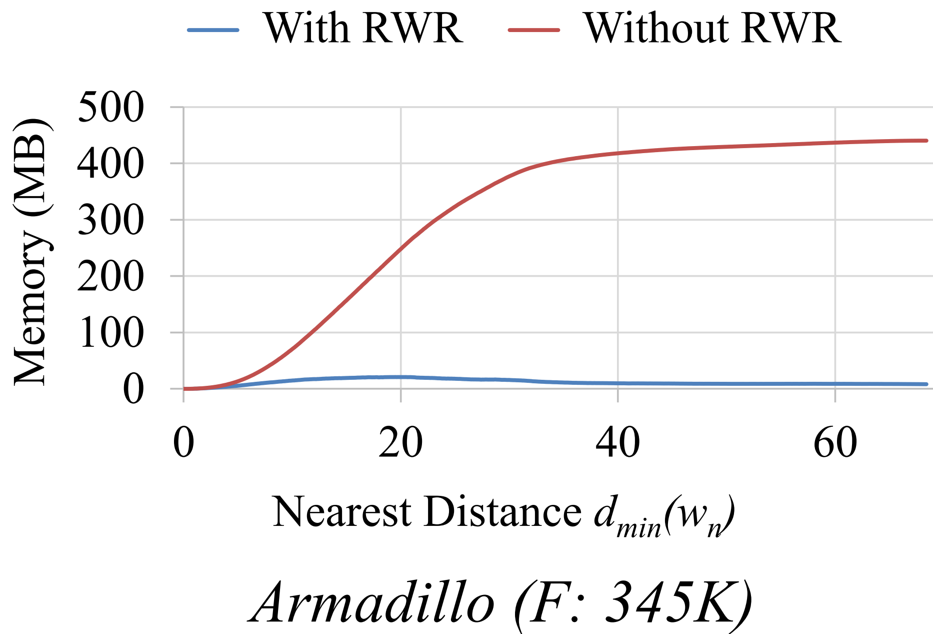
This process is summarized in Procedure 3.

Procedure 3 Redundant Windows Removal (RWR)

Input: f - Face;
d - Distance of the nearest window on the wavefront;
Output: f' - The face after redundancy removal;
1: **procedure** RWR(f, d)
2: Let $e_{\max}$ be the longest edge of f;
3: **if** $d_{\min}(f) + \|e_{\max}\| \leq d$ **then**
4: Check f's redundancy;
5: **if** f is redundant **then**
6: **for** each edge $e_i \in f$ **do**
7: Let f_i be the face sharing edge e_i with f;
8: **if** f_i is redundant **then**
9: Empty the windows on e_i;
10: **end if**
11: **end for**
12: **end if**
13: **end if**
14: **end procedure**

5.1.3 Performance Verification

To verify that the proposed RWR procedure effectively reduces memory cost, this section compares memory costs against nearest distance $d_{\min}(w_n)$ of the wavefront between two scenarios of Voronoi diagram construction: with and without RWR. The tests are performed on ten models selected from the model set (Appendix A). Figure 5.7 shows the results on two models (Armadillo and Asian Dragon) and the rest of the results have been included in Appendix H. It can be seen that applying RWR dramatically reduces the memory cost of Voronoi diagram construction. Specifically, methods without RWR, e.g.



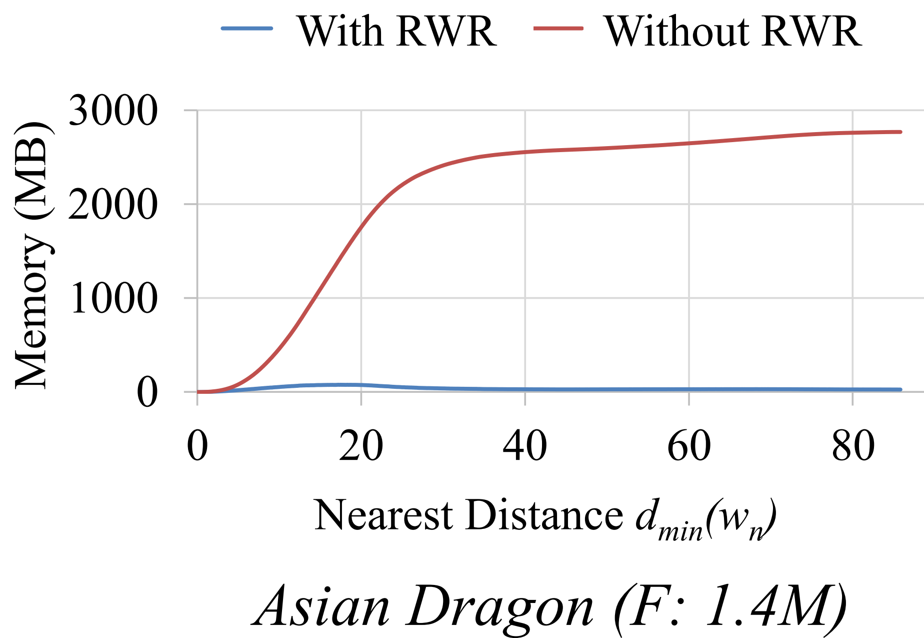


FIGURE 5.7: Performance verification on RWR. The x-axis represents the distance of the nearest window on the wavefront during propagation, i.e. $d_{\min}(w_n)$. The y-axis represents real-time memory cost during propagation.

Liu et al. (2011), store all propagated windows on edges of the mesh and their memory costs are *cumulative*. On the contrary, RWR removes redundant windows in time with geodesic wavefront propagations. Thus, the memory cost is effectively reduced.

5.2 Applying RWR in Geodesic Computation

To construct geodesic-metric-based Voronoi diagrams, the *window*-VTP algorithm is proposed by revising the original VTP algorithm (Chapter 4). The *window*-VTP algorithm is essentially a multi-source geodesic algorithm and takes triangles as the primitive for distance propagation. For each source, all visited triangles form its own traversed area. The boundary of the traversed area is defined as the propagation wavefront.

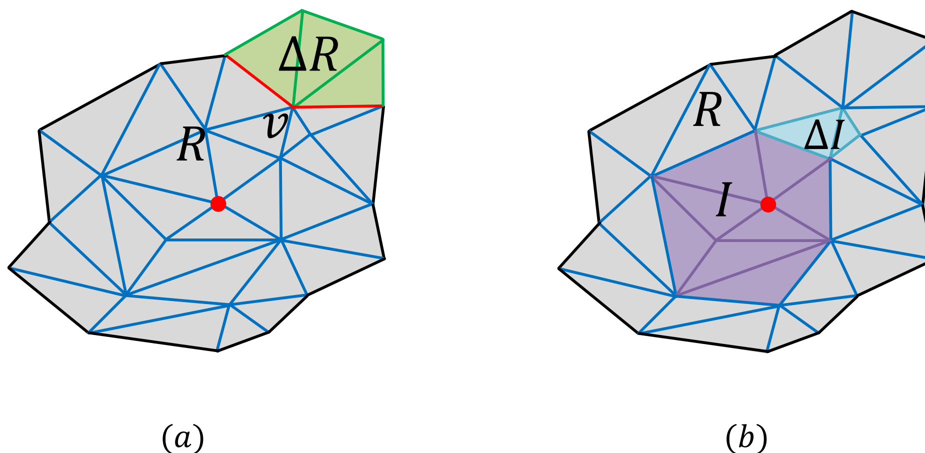


FIGURE 5.8: Illustration of the triangle-oriented region expansion scheme. (a) Expansion of the traversed area R. (b) Expansion of the inactive region I.

For simplicity, consider the one source scenario here. The proposed algorithm expands its traversed area R and inactive region I at the same time (Figure 5.8). Note that the **inactive region** I is a *proper subset* of the **traversed area** R, i.e. $I \subset R$, and the windows in I will not be updated. Both R and I are expanded in continuous Dijkstra style, and gradually involving unvisited triangles abutting the wavefront. First, the proposed algorithm creates the initial windows of each source within its 1-ring neighbourhood and pushes all the adjacent vertices of each source into a priority queue Q. Note that only *one* priority queue Q is defined for all traversed areas since every vertex is involved in Q in terms of the propagation distance of the wavefront.

When a vertex is popped from the priority queue Q, the proposed *window*-VTP algorithm performs the following:

- **Expanding traversed area R.** As Figure 5.8 (a) shows, let ΔR be the unvisited triangles in v's 1-ring neighbourhood. Then, R is expanded by involving ΔR into R, and the wavefront is also updated accordingly. Then, the windows on the previous wavefront (e.g. vE and vB in Figure 5.9) are propagated through ΔR and R either till they reach the wavefront, or are eliminated during propagation. To manage windows

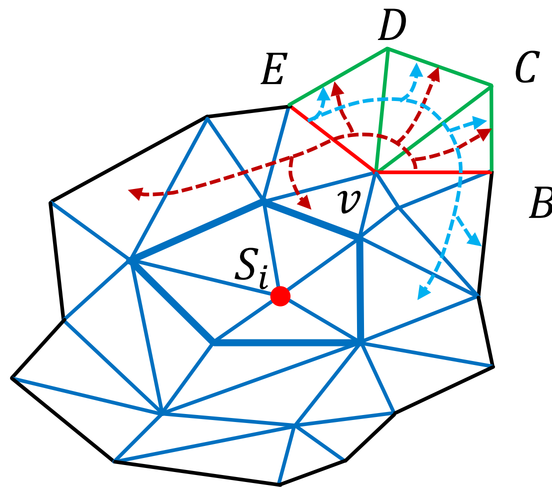


FIGURE 5.9: Vertex-sorted Triangle Propagation Qin et al. (2016).

on the wavefront for the Voronoi diagram construction, the propagated windows are trimmed on edges using the windows trimming rule (Figure 5.10) and binary insertion scheme proposed by the MMP algorithm (Surazhsky et al., 2005).

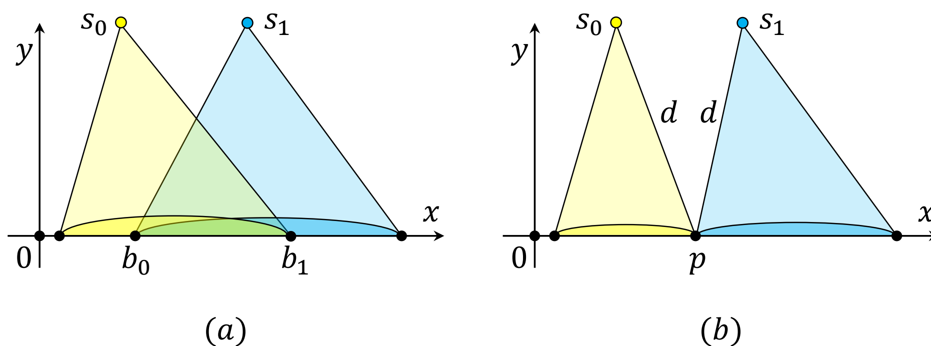


FIGURE 5.10: The window trimming rule in the MMP algorithm (Surazhsky et al., 2005). This rule trims two overlapping windows into non-overlapping ones.

- **Expanding Inactive region I.** As Figure 5.8 (b) shows, the expansion of I is limited inside R. In the region between I and R, let ΔI be the triangles satisfying Proposition 5.1. Then, I is expanded by involving ΔI in I. When a triangle is added into I, the windows on it are removed by performing procedure $RWR()$.

Algorithm 4 *window*-VTP algorithm

Input: M - Mesh;
S - Source set;
Output: M' - Mesh with sufficient geodesic information for Voronoi diagram constructions;

1: **procedure** *window*-VTP(M, S)
2: ***Step 0***. Perform Initialization.
• For each source S_i, create a window for every opposite edge of S_i in its 1-ring neighborhood (bold blue lines around S_i in Figure 5.9).
• Push all adjacent vertices of S_i into a priority queue Q.
• Define a priority queue Q_i, which is used to organize the expansion of the inactive regions;
3: **while** $!Q.empty()$ **do**
4: ***Step 1***. Pop a vertex v from Q;
5: ***Step 2***. Update the wavefront and traversed areas;
6: ***Step 3***. Expanding the traversed areas.
• Push the windows on edges of the wavefront incident to v into FIFO queue W;
7: **while** $!W.empty()$ **do**
8: • Pop a window w from W;
• Propagate w across a triangle;
• Retain and trim the propagated windows;
• Push the propagate windows into W if they survives the trimming and haven't reached the wavefront;
9: **end while**
10: ***Step 4***. Expanding the inactive regions.
11: **while** $!Q_i.empty()$ **do**
12: • Let f be $Q_i.front()$;
• Perform $RWR()$ on f to check if f is in the inactive regions; If so, remove the redundant windows on it; else, break the loop;
13: **end while**
14: ***Step 5***. Update vertices' and triangles' priorities;
15: ***Step 6***. Push the faces newly added to the traversed areas into Q_i;
16: **end while**
17: **end procedure**

The outline of the proposed algorithm is shown in Algorithm 4 and two challenges are rising as below.

(1) How to deal with the collision of the wavefronts? Note that it may be a self-intersection of one wavefront or meeting of two wavefronts.

(2) How to define the priorities for triangles and vertices in Q_i and Q properly (in ***Step 4, 5***)?

5.2.1 Wavefront Collision

Proposition 5.2. *The proposed* window-*VTP algorithm automatically handles the wavefront collisions and requires no extra operations.*

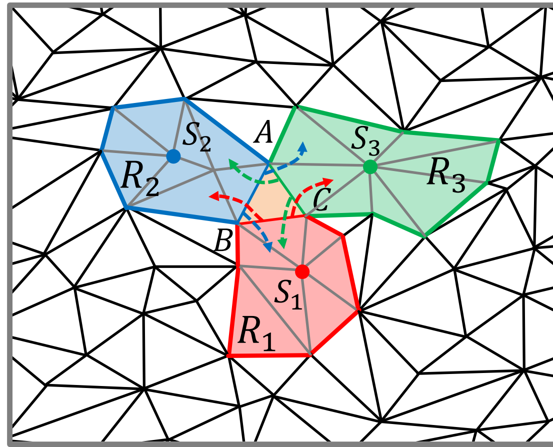


FIGURE 5.11: The collision of the propagation wavefront. The wavefront consists of three parts from three different sources, S_1, S_2 and S_3 (red, blue and green line segments).

As Figure 5.11 shows, the propagation wavefront consists of different parts corresponding to different sources. When different parts of the wavefront collide with each other, simply let the windows propagate through the wavefront and enter the interior of the traversed areas. The propagations of these windows will stop when they reach the updated wavefront or be eliminated by the retained windows on edges in the traversed areas using the windows trimming rule (Surazhsky et al., 2005). Thus, no extra operation is required. For example in Figure 5.11, the wavefront collides when ΔABC is added to the traversed areas. Then, the windows on edges AB, AC, BC are propagated into the interior of R_1, R_2 and R_3 (the dashed arrows in Figure 5.11). These propagations will stop upon reaching the updated wavefront (the bold red, green, blue line segments in Figure 5.11) or be eliminated on the interior edges (the grey line segments in Figure 5.11).

5.2.2 Priorities Definition

The key point of performing the procedure $RWR()$ during wavefront propagation is to form the **inactive region**, which resort to two priorities: the face's priority and the vertex's. Recall that the inequality of $d_{\min}(f) + \|e_{\max}\| \leq d_{\min}(w_n)$ is used to identify whether a face f is in the inactive region (Proposition 5.1). In the proposed algorithm, the priorities are defined as follows:

Face's Priority. A face f's priority in the priority queue Q_i is defined as $-(d_{\min}(f) + \|e_{\max}\|)$.

Vertex's Priority. A vertex v's priority in the priority queue Q is defined as the negative minimum of the current shortest distances to v's incident edges on the wavefront. For example in Figure 5.12,

$$-d_{\min}(A) = -\min\{d_{\min}(AB), d_{\min}(AC)\}$$

In addition, if w_n is on AB or AC, $-d_{\min}(A) = -d_{\min}(w_n)$.

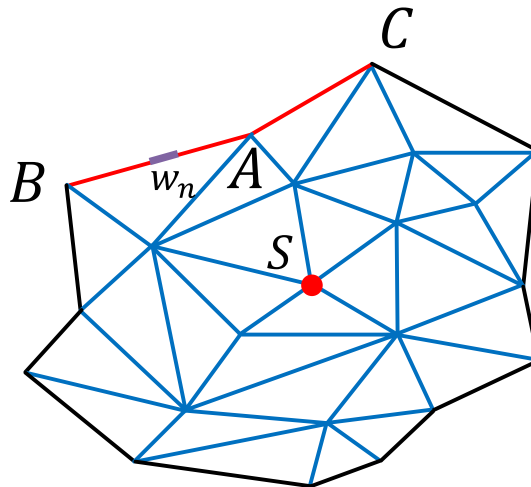


FIGURE 5.12: Illustration of the vertex's priority definition. The propagation wavefront are the black and red line segments. w_n is the nearest window on the wavefront.

Note that the two defined priorities are just the left and right sides of inequality $d_{\min}(f) + \|e_{\max}\| \leq d_{\min}(w_n)$ (Proposition 5.1), and thus they can be directly used when performing procedure $RWR()$.

5.3 Complexity Analysis

This section focuses on the complexity of geodesic computation since it is the dominant part of the Voronoi diagram construction (Liu et al., 2011).

Let n be the number of vertices on a mesh. It is easy to verify that the proposed *window*-VTP algorithm is an improved version of the original MMP algorithm (Mitchell et al., 1987). In the worst case, the number of windows generated in the geodesic computation part is $O(n^2)$ and the time complexity of geodesic computation is $O(n^2 \log n)$. For the redundant windows removal (RWR) part, the checking and deletion processes are performed on each window and thus accounts for $O(n^2)$ time. In addition, the expansion of the inactive region is triangle-oriented and thus costs $O(n \log n)$ time for $O(n)$ triangles.

In summary, the time complexity of *window*-VTP is $O(n^2 \log n + n^2 + n \log n) = O(n^2 \log n)$. Since the redundant windows removal process does not consume extra memory, the space complexity of the proposed algorithms is $O(n^2)$.

5.4 Experimental Results

To evaluate the performance of the proposed algorithm, experiments have been conducted on a variety of models. Specifically, the test models are selected from the model set (Appendix A), including sculptures, animals and manmade objects. The resolution of these models (number of faces) ranges from 10K to 14M. All the algorithms are tested using a HP Z420 Workstation with an Intel Xeon E5-1650 3.20GHz CPU and 32GB memory. Unless specified, the experiments randomly select 30 vertices as the sources on meshes, as shown by Liu et al. (2011). Figure 5.13 shows the constructed Voronoi diagrams on some example models.

5.4.1 Comparison with Liu et al. (2011)

Overall Performance According to Liu et al. (2011), constructing the geodesic-metric-based Voronoi diagram consists of two stages,

- **Stage 1.** Compute geodesic distance fields on edges of mesh M.

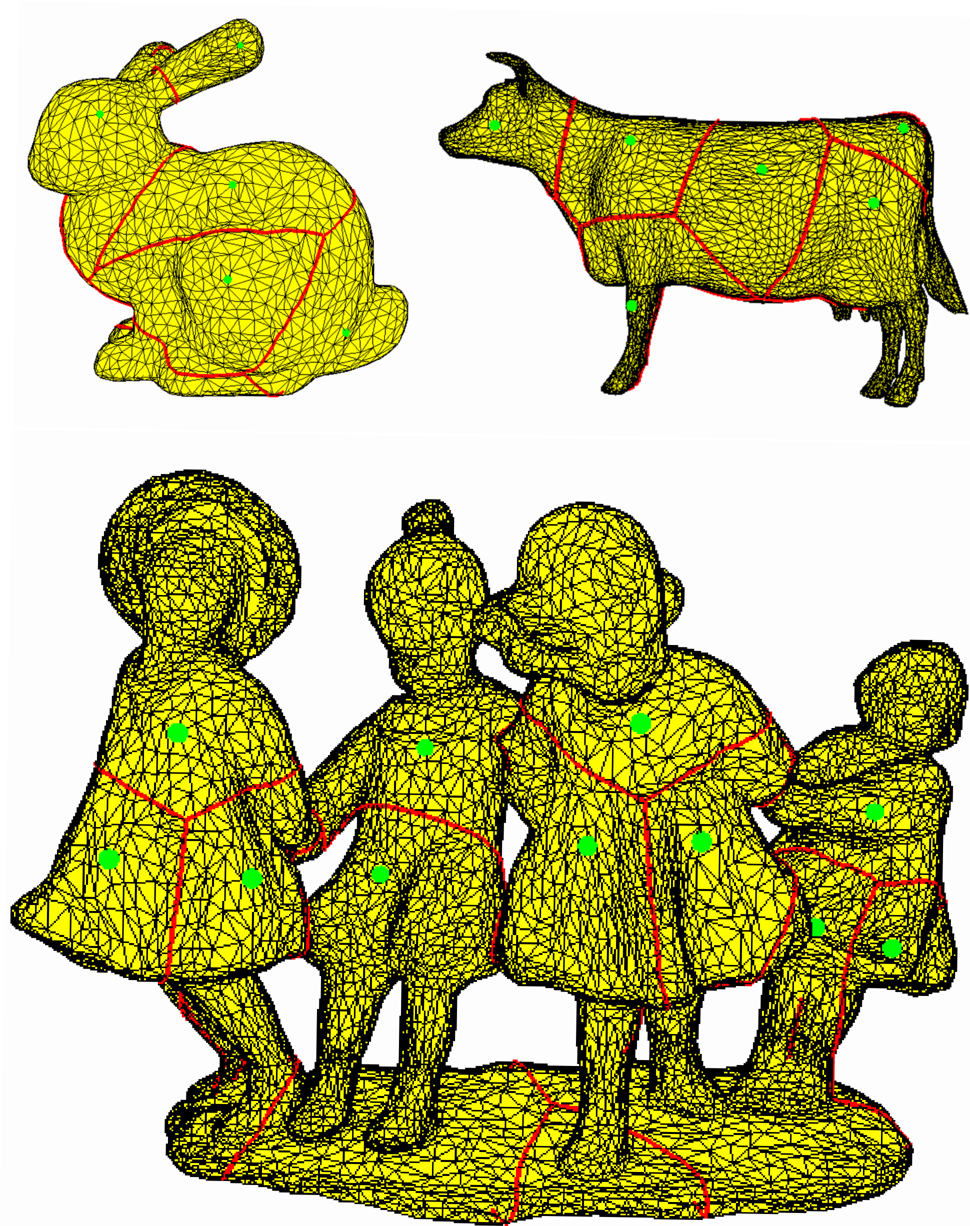


FIGURE 5.13: Examples of Voronoi diagrams on meshes. The faces of the models are: Bunny (5K faces), Cow (10K faces), Dancingchildren (20K faces).

- **Stage 2.** Extract the valid triangles which contain Voronoi cells' boundaries. March them to track and reconstruct the boundaries of Voronoi cells' by linking the intersections between them and edges of M.

The overall performance of the proposed algorithm is evaluated by two measures on the two stages: running time and peak memory usage respectively. As Table 5.1 shows, the geodesic computation part consumes the majority of time and memory in both Liu et al.'s method and ours. However, when replacing the MMP algorithm used in Liu et al. (2011) by the proposed *window*-VTP algorithm for geodesic computation, the Voronoi diagram construction runs 3-8 times faster and uses 10-70 times less memory.

Model	Performance	Liu et al. (2011)	Ours	Ratio
Horse (F: 96K)	Time(s)	1.966 + 0.015	0.66 + 0.015	2.93
	Peak memory(MB)	109.40 + 0.035	9.98 + 0.035	10.93
Bunny (F: 144K)	Time(s)	3.637 + 0.028	1.07 + 0.028	3.34
	Peak memory(MB)	187.00 + 0.046	14.86 + 0.046	12.55
Igea (F: 268K)	Time(s)	10.916 + 0.048	3.019 + 0.048	3.57
	Peak memory(MB)	478.06 + 0.065	26.50 + 0.065	18.00
Armadillo (F: 345K)	Time(s)	9.863 + 0.046	2.982 + 0.046	3.27
	Peak memory(MB)	440.33 + 0.066	21.09 + 0.066	20.81
Pulley (F: 392K)	Time(s)	23.917 + 0.115	5.345 + 0.115	4.40
	Peak memory(MB)	792.08 + 0.086	39.69 + 0.086	19.91
Rocker arm (F: 482K)	Time(s)	32.012 + 0.091	6.985 + 0.091	4.54
	Peak memory(MB)	1013.34 + 0.099	41.50 + 0.099	24.36
Asian dragon (F: 1,400K)	Time(s)	110.083 + 0.255	20.281 + 0.255	5.37
	Peak memory(MB)	2770.81 + 0.143	76.75 + 0.144	36.04
IsidoreHorse (F: 2,209K)	Time(s)	89.538 + 0.211	21.229 + 0.211	4.17
	Peak memory(MB)	2574.06 + 0.189	46.79 + 0.189	54.79
Happy buddha (F: 2,583K)	Time(s)	482.715 + 1.291	58.946 + 1.291	8.04
	Peak memory(MB)	8218.60 + 0.406	161.98 + 0.406	50.61
Neptune (F: 4,008K)	Time(s)	832.83 + 0.784	96.843 + 0.784	8.54
	Peak memory(MB)	13070.70 + 0.262	176.30 + 0.262	74.03

TABLE 5.1: Performance comparison with Liu et al. (2011). The results are shown in an addition manner as: "geodesic computation" + "Voronoi diagram construction".

Since the geodesic computation part is the bottleneck of Voronoi diagram construction, a more comprehensive comparison on it is performed as follows.

Performance Comparison on Geodesic Computation To evaluate the performance of the geodesic part, three measures are used: running time, total

number of windows stored after propagation and peak memory usage. Algorithms in this comparison have been tested on all 55 models in the model set. For better reading experience, some of the testing results are shown here and the others are given in Appendix I.

	MMP vs. window-VTP	*FWP-MMP vs. window-VTP*
Time	3.98/1.55	1.21/0.18
# windows stored	48.96/38.98	48.96/38.99
Peak Memory	21.24/15.16	21.24/15.16

TABLE 5.2: The mean and standard deviation of the performance ratios between other algorithms and the proposed *window*-VTP algorithm on running time, the number of windows stored and peak memory usage.

The mean and standard deviation of performance ratios are calculated between MMP, FWP-MMP and the proposed *window*-VTP algorithm. The details are shown in Table 5.2. It can be seen that *window*-VTP on average runs 4 times as fast as MMP and comparable to FWP-MMP (1.2 times faster). The *window*-VTP algorithm on average uses 95.29% less memory than MMP and FWP-MMP. Furthermore, the *window*-VTP algorithm stores 97.96% less windows than MMP and FWP-MMP algorithms after propagation, which shows that it removes redundant windows effectively. Note that

Model	**Performance**	**Algorithms**		
		MMP	**FWP-MMP**	***window*-VTP**
Bunny (F:144K)	Time(s)	3.637	1.27	1.07
	# windows stored	2,451,104	2,451,105	85,959
	Peak Memory(MB)	187.00	187.00	14.86
Rocker Arm (F:482K)	Time(s)	32.012	9.088	6.985
	# windows stored	13,282,080	13,282,139	271,040
	Peak Memory(MB)	1013.34	1013.35	41.50
Asian Dragon (F:1,400K)	Time(s)	110.083	28.247	20.281
	# windows stored	36,317,620	36,317,847	346,142
	Peak Memory(MB)	2770.81	2770.83	76.75
Neptune (F:4,008K)	Time(s)	832.83	173.055	96.843
	# windows stored	171,319,703	171,374,203	857,068
	Peak Memory(MB)	13070.70	13074.80	176.30
Lucy (F:14,464K)	Time(s)	Out of memory	Out of memory	806.118
	# windows stored			12,071,796
	Peak Memory(MB)			921.005

TABLE 5.3: Performance comparison between MMP, FWP-MMP and the proposed *window*-VTP on five representative models.

the proposed *window*-VTP algorithm is impressive since it resolves the memory bottleneck of Voronoi diagram oriented computation of geodesics, whilst not sacrificing the speed. For example, it uses 95.29% less memory than

FWP-MMP while still being 1.2 times as fast. Detailed results on 5 representative testing models are shown in Table 5.3.

Number of Sources This part studies how the proposed algorithm performs with varying number of sources. First, three test models (Maxplanck, Angel, RedCircularBox) are chosen. For each model, eleven sets of sources are chosen randomly whose sizes range from 1 to 1000. Then, the ratios between the running time, peak memory of FWP-MMP based Voronoi diagram construction algorithm and that of ours on all source sets are calculated. The experiments are designed to show how the ratios change with changing number of sources.

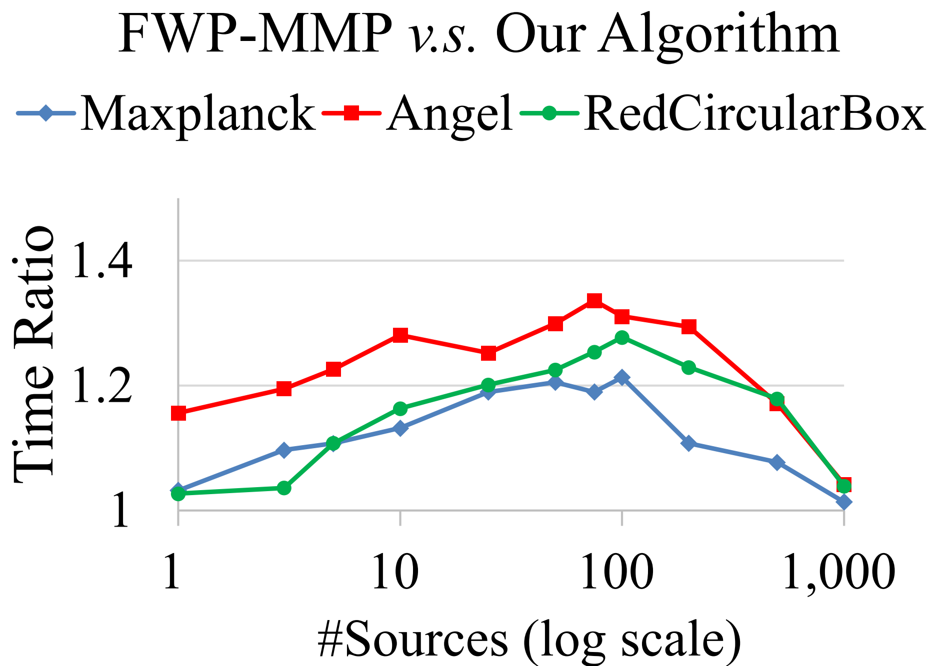
As illustrated in Figure 5.14, the time ratios increase within the range of source number at [1,100] and drop within the range at (100,1000]. This inconsistency is caused by RWR and the VTP wavefront propagation. When the number of sources increases,

- RWR is invoked less times. This is because the more triangles the Voronoi boundary occupies, the fewer the redundant windows.
- The performance of VTP wavefront propagation depends on the scale of the models, i.e. VTP performs better than the others on large scale meshes. Herein, the size of Voronoi cells becomes smaller when the number of sources increases. VTP has to work within each cell, that is, the models' size becomes smaller for VTP.

The time ratio in Figure 5.14 shows that in the range of [1,100], reducing RWR dominantly causes the time ratio increasing. In the range of (100,1000], the size of Voronoi cells becomes smaller, which leads to the performance of VTP decreasing. The low performance of VTP dominantly causes the time ratio decreasing at that time.

However, the memory ratio in Figure 5.14 shows that the memory cost is close to that of FWP-MMP with an increasing number of sources. Nevertheless, the proposed algorithm still runs faster than the FWP-MMP based Voronoi diagram construction algorithm and uses more than 3 times less memory for 1000 sources.

Performance Profiling This section profiles the running time of different components in the Voronoi diagram construction, showing how it is accelerated. As proposed in Liu et al. (2011), the Voronoi diagram construction contains two components: the computation of geodesics and the construction



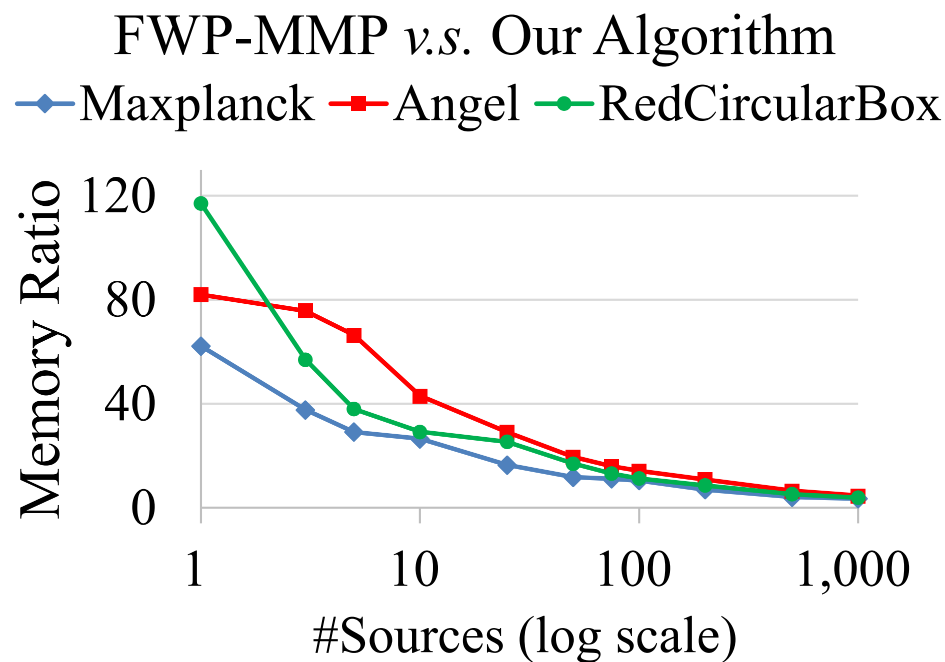
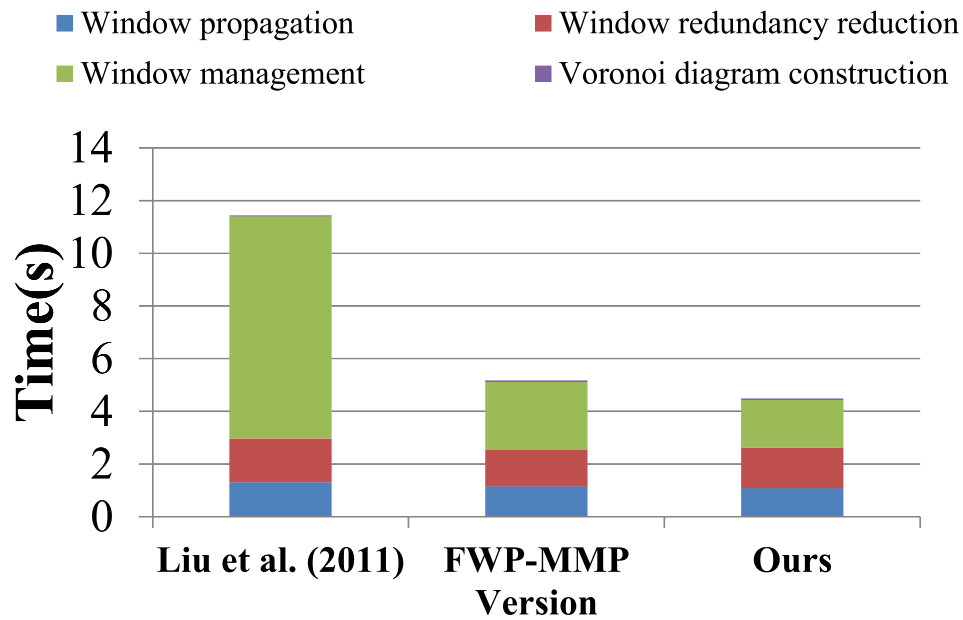


FIGURE 5.14: Performance comparison between FWP-MMP based Voronoi diagram construction algorithm and ours on the number of sources. The x-axis represents the number of sources in logarithmic scale, and the y-axis represents the performance (time, memory) ratio.



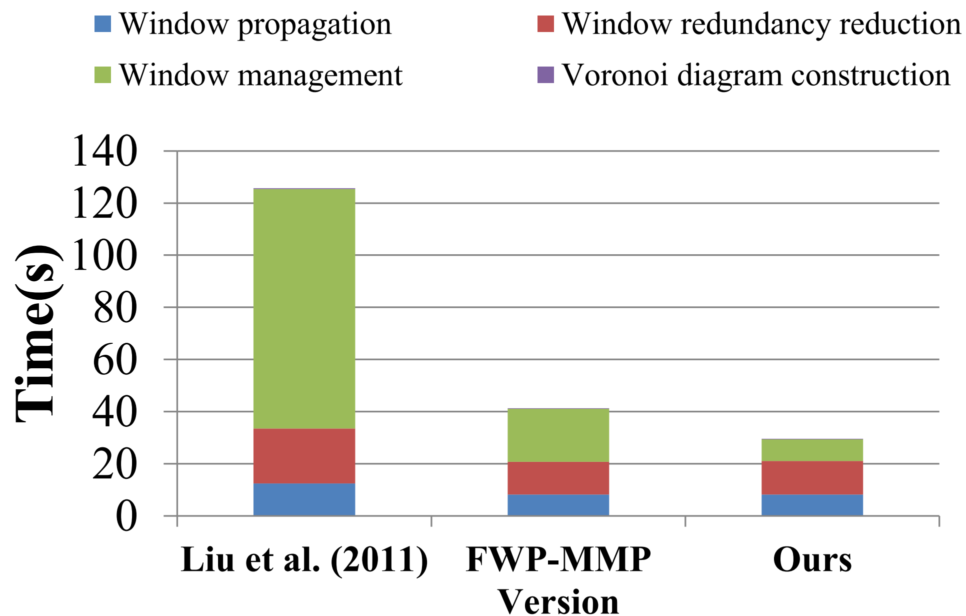
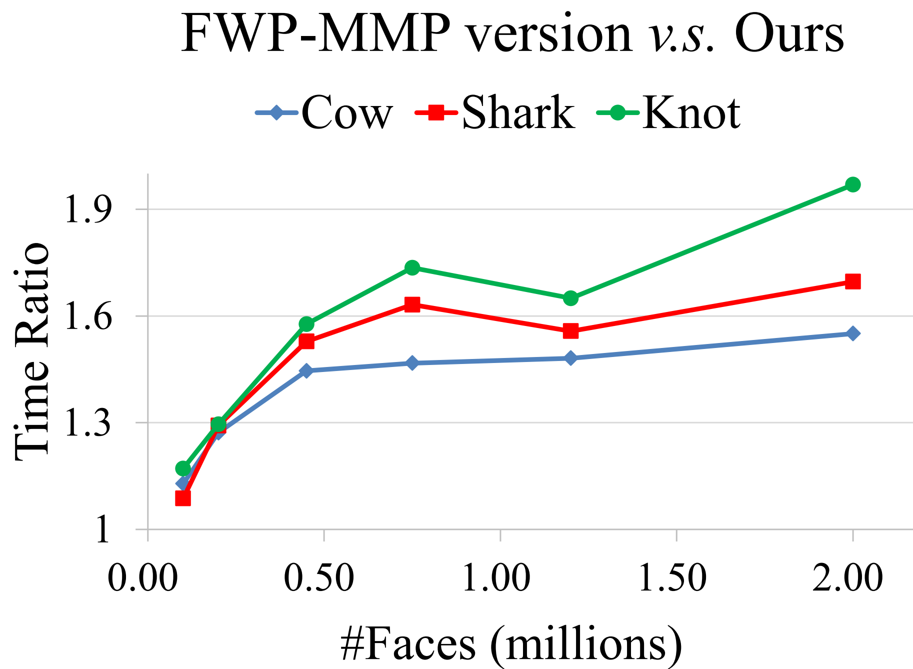


FIGURE 5.15: Comparison of running times of four common components in Voronoi diagram construction on two models. The comparison is performed on three versions of the solution: (1) the original method in Liu et al. (2011); (2) the FWP-MMP version which replaces the MMP algorithm used by Liu et al. (2011) with the FWP-MMP algorithm Xu et al. (2015); (3) The proposed version which replaces the MMP algorithm used by Liu et al. (2011) with the proposed window-VTP algorithm.

of a Voronoi diagram. In addition, the *geodesic computation* component can be further subdivided into three components: *window propagation*, *window redundancy reduction* and *window management* (Section 3.3.2.2). Thus, the running times of these four individual components in all participating algorithms are profiled on ten models selected from the model set. Figure 5.15 shows the results on two models, Armadillo and Asian Dragon (the rest of the results have been included in Appendix J). Compared to the *geodesic computation* components, the time cost of *Voronoi diagram construction* is extremely small and can be neglected. For *geodesic computation* components, it can be seen that the VTP framework effectively reduces the window management cost of the Voronoi diagram construction. Furthermore, although an extra RWR process is added in the proposed method, the running time of the window redundancy reduction component is not dramatically increased as its time cost is small compared to other computations (e.g. binary insertion and windows trimming).

Scalability First, three test models (Cow, Shark and Knot) are chosen. Let each of them have six different resolutions through subdivision. The number of faces ranges from 0.1M to 2M in these subdivided models. For each model, its ratios between the running time, peak memory of FWP-MMP based Voronoi diagram construction algorithm and that of ours on all six resolutions is calculated. The experiments are designed to show how the ratios change with the changing resolution. As illustrated in Figure 5.16, both the timing ratios and memory cost ratios increase with an increasing resolution. As shown, the rate of increase in performance for the proposed algorithm is proportional to the size of the models.

Robustness This section further validates that the proposed algorithm is robust to mesh triangulation quality. As in FWP Xu et al. (2015), a sequence of meshes (eight) with different degrees of anisotropy but a fixed resolution on two testing models (Fertility with 800K faces and Hand with 200K faces) are created respectively. Here, $g(M) = \frac{\Sigma_{f \in F} g'(f)}{|F|}$ is also used to measure the degree of anisotropy of a mesh M, where $g'(f) = \frac{PH}{2\sqrt{3}S}$ and P, H, S are the half-perimeter, longest edge length and area of f respectively. All these meshes with varied degrees of anisotropy are generated using the method in Zhong et al. (2013). The curves in Figure 5.17 and Figure 5.18 show how the running times and peak memories change with increasing anisotropy (g)



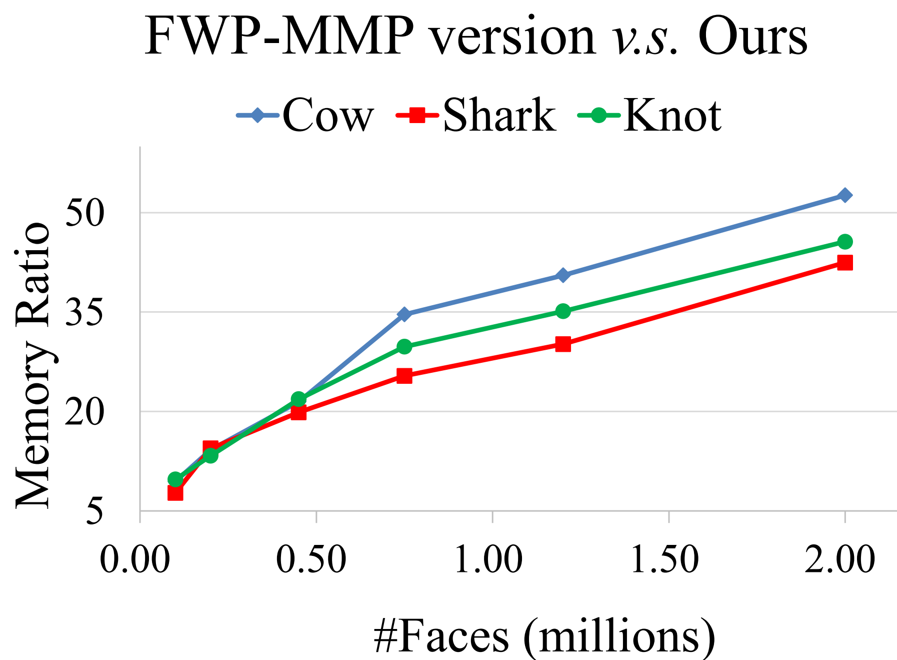
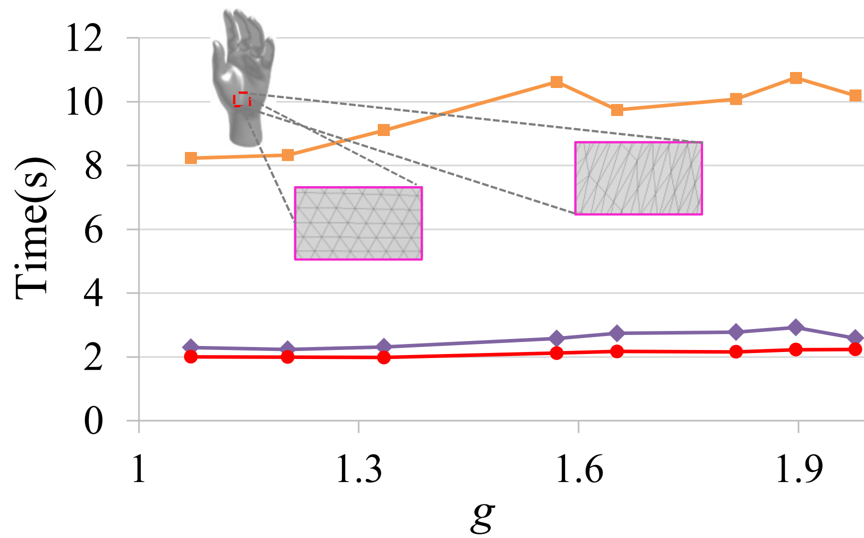


FIGURE 5.16: Comparison of scalability against FWP-MMP based Voronoi diagram construction algorithm. The x-axis represents the mesh resolution, and the y-axis represents running time ratio or memory cost ratio.



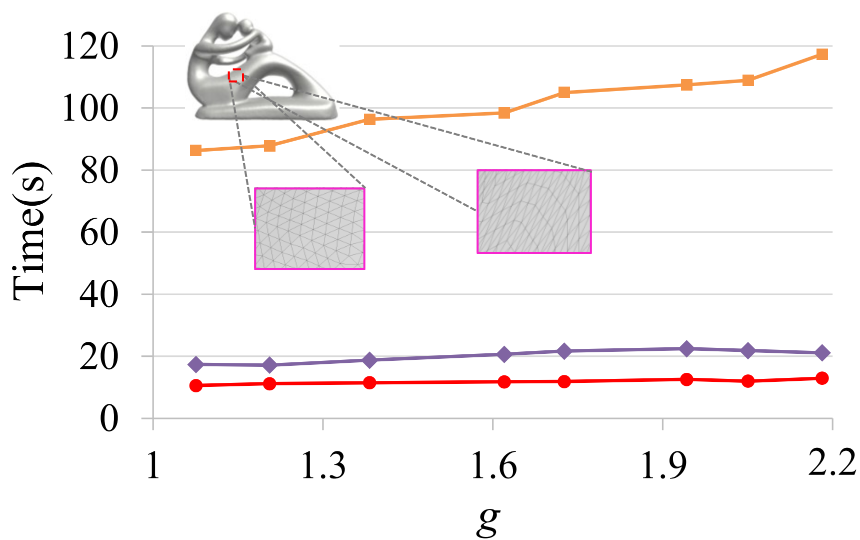
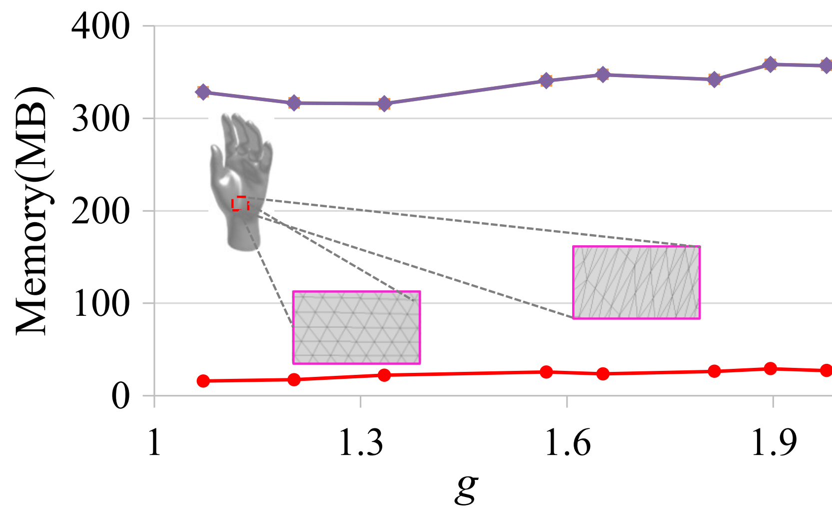


FIGURE 5.17: Comparison of robustness against anisotropic triangulation (Time). The x-axis represents the degree of anisotropy, and the y-axis represents running time.



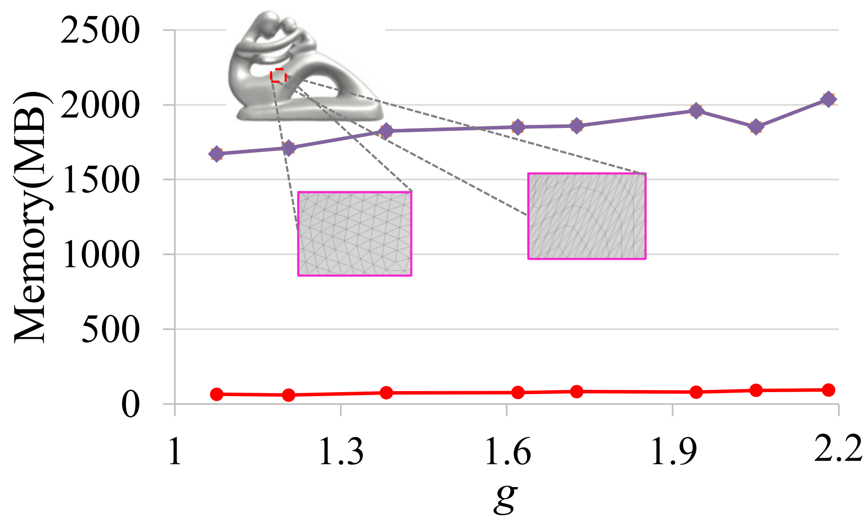


FIGURE 5.18: Comparison of robustness against anisotropic triangulation (Memory). The x-axis represents the degree of anisotropy, and the y-axis represents peak memory.

respectively. Note that the peak memories of Liu et al.'s method and its FWP-MMP based version are almost the same since both of them store all propagated windows on edges. The proposed *window*-VTP algorithm is the most robust among all algorithms since its running time and peak memory does not obviously increase when the input mesh has a much larger anisotropy.

5.4.2 Comparison with Xu et al. (2014)

As Xu et al. (2014) have used the MMP algorithm to compute geodesics , its performance has already been compared in the preceding section and thus not discussed here.

Model	Performance	*window*-VTP + Xu et al. (2014) $(\mathbf{c = 1})$	Ours
Horse (F: 96K)	Time(s)	1.16	0.68
	Peak memory(MB)	13.38	10.01
Bunny (F: 144K)	Time(s)	1.93	1.10
	Peak memory(MB)	19.95	14.90
Igea (F: 268K)	Time(s)	5.42	3.07
	Peak memory(MB)	35.97	26.56
Armadillo (F: 345K)	Time(s)	5.04	3.03
	Peak memory(MB)	33.75	21.16
Pulley (F: 392K)	Time(s)	12.60	5.46
	Peak memory(MB)	58.17	39.78
Rocker arm (F: 482K)	Time(s)	12.41	7.08
	Peak memory(MB)	63.53	41.60
Asian dragon (F: 1,400K)	Time(s)	42.17	20.54
	Peak memory(MB)	132.99	76.90
IsidoreHorse (F: 2,209K)	Time(s)	29.73	21.51
	Peak memory(MB)	128.62	46.98
Happy buddha (F: 2,583K)	Time(s)	160.47	60.24
	Peak memory(MB)	493.70	162.39
Neptune (F: 4,008K)	Time(s)	195.45	97.63
	Peak memory(MB)	514.98	176.56

TABLE 5.4: Performance comparison with Xu et al. (2014).

Xu et al. (2014) proposed another method to reduce the memory cost of Voronoi diagram construction rather than the proposed RWR technique. The main deficiency of their method is the *inefficiency* of the redundancy check. In their method, the redundancy check is performed on all unlabelled triangles rather than just the ones in the *inactive region* (Proposition 5.1). Thus, windows on many triangles are repeatedly checked since they are not inactive and will be updated by later propagated windows. In addition, since

the cost of their redundancy check is large, performing it frequently is time-consuming. Thus, their method suffers from the *trade-off* between running time and memory-cost. In more details, they perform one redundancy check with every cn window propagations, where n is the face number of the mesh and c is a *user-defined* parameter to balance the performance. A smaller c means that the redundancy check is performed more frequently, reducing memory cost but sacrificing the running time.

On the contrary, the proposed RWR technique performs the redundancy check efficiently in the *inactive region* every time a vertex is popped from the priority queue. To make a fair comparison, the proposed algorithm is compared with an improved version of Xu et al. (2014) which uses the proposed *window*-VTP for geodesic computation but still employs their redundancy reduction method rather than the proposed RWR (Table 5.4). In the experiments, the parameter c is set as 1 for a balanced performance. It can be seen that the proposed algorithm outperforms theirs in both running time and peak memory.

5.4.3 Comparison with VTP

The original VTP algorithm does not retain windows, while the revised version keeps partial windows. Compared to the original VTP, this experiment shows how the change influences the performance.

In this experiment, the performance are compared using the proposed *window*-VTP with the original VTP to solve the SS-DGP problem, with the first vertex set as the source on the mesh. As Table 5.5 shows, *window*-VTP runs approximately two times slower than VTP. The main reason is that the *window*-VTP has to strictly sort windows on edges by binary insertion. However, Voronoi diagrams are usually more sparse than meshes and there is no distinct decline in performance.

5.4.4 Application to Remeshing

Due to that the Delaunay triangulation of a point set S is the dual of its Voronoi diagram, the proposed algorithm can be applied to remesh the dense models reconstructed from range data. In this context, the number of sources is usually fairly large and reaches the order of hundreds. Figure 5.19 shows the remeshing result of the Neptune model with 4K randomly selected sources.

Model	Performance	VTP	*window*-VTP
Horse (F: 96K)	Time(s)	0.64	1.13
	Peak memory(MB)	1.25	5.67
Bunny (F: 144K)	Time(s)	0.88	1.50
	Peak memory(MB)	1.08	4.56
Igea (F: 268K)	Time(s)	2.04	4.11
	Peak memory(MB)	2.00	9.10
Armadillo (F: 345K)	Time(s)	1.68	2.68
	Peak memory(MB)	1.31	5.62
Pulley (F: 392K)	Time(s)	3.97	8.71
	Peak memory(MB)	4.53	18.58
Rocker arm (F: 482K)	Time(s)	4.26	9.32
	Peak memory(MB)	3.26	14.32
Asian dragon (F: 1,400K)	Time(s)	9.74	20.95
	Peak memory(MB)	3.72	16.77
IsidoreHorse (F: 2,209K)	Time(s)	10.41	17.72
	Peak memory(MB)	2.76	12.19
Happy buddha (F: 2,583K)	Time(s)	31.44	68.75
	Peak memory(MB)	8.44	40.46
Neptune (F: 4,008K)	Time(s)	51.62	91.14
	Peak memory(MB)	14.42	37.26

TABLE 5.5: Performance comparison with VTP.

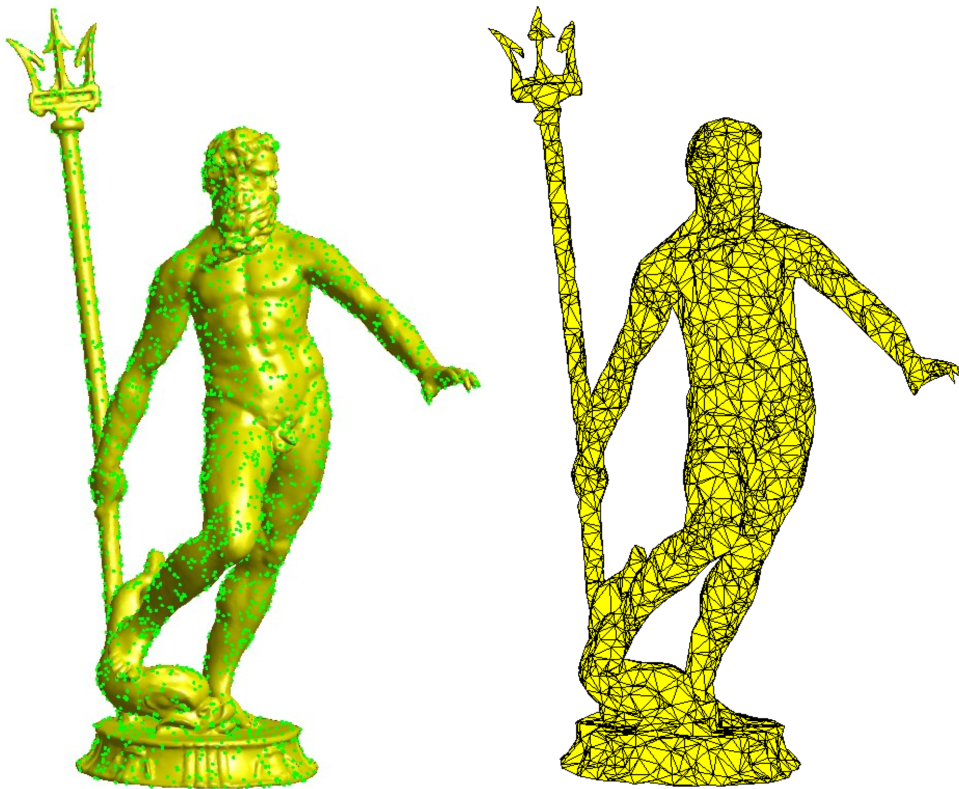


FIGURE 5.19: Illustration of remeshing with the proposed algorithm.

To show the performance of our method, we compare it with the FWP-MMP version of Liu et al.'s (2011) method on six dense models selected from the proposed dataset, whose numbers of faces range from 1.4M to 6.4M. For each model, 2K sources are randomly selected if its number of faces is less than 2M; otherwise, 4K sources are selected. As Table 5.6 shows, our method runs faster and uses much less memory than the FWP-MMP version of Liu et al.'s (2011) method in the remeshing problem.

# Samples: 2000			
Model	**Performance**	**FWP-MMP version**	**Ours**
Asian dragon (F: 1,400K)	Time(s)	14.07	11.18
	Peak memory(MB)	863.93	170.65
Pensatore (F: 1,996K)	Time(s)	25.02	17.24
	Peak memory(MB)	1503.96	251.48
Seahorse (F: 2,014K)	Time(s)	23.24	17.26
	Peak memory(MB)	1455.77	230.63

# Samples: 4000			
Model	**Performance**	**FWP-MMP version**	**Ours**
Happy buddha (F: 2,583K)	Time(s)	28.21	23.48
	Peak memory(MB)	1690.61	310.59
Neptune (F: 4,008K)	Time(s)	52.26	39.07
	Peak memory(MB)	2925.16	422.98
Vase lion (F: 6,370K)	Time(s)	111.381	72.22
	Peak memory(MB)	5567.37	673.80

TABLE 5.6: Performance comparison with the FWP-MMP version of Liu et al.'s (2011) method on remeshing.

5.5 Summary

In this chapter, the RWR procedure is presented to reduce the memory cost of constructing the geodesic-metric-based Voronoi diagrams, in which windows on edges are grouped by EWG within the inactive regions so that they can be removed together in time. The proposed *window*-VTP algorithm incorporates the RWR procedure in the vertex-oriented wavefront propagation framework. As a result, the *window*-VTP algorithm effectively resolves the memory bottleneck of the Voronoi diagram construction while not sacrificing the speed. In terms of experiments, the proposed algorithm runs 3-8 times faster than

Liu et al.'s (2011) method, 1.2 times faster than its FWP-MMP variant and more importantly uses 10-70 times less memory than both of them.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis, the *Edge-based Windows Grouping* (EWG) technique is proposed to improve the performance of exact geodesic computation on triangle meshes, which is ideal for large scale models. Specifically, EWG groups “windows” into “window lists” on edges and builds the connections between them. Compared to previous methods which organize the geodesic wavefront propagation by individual windows, the EWG-based window propagation framework propagates nearby windows on edges in batches. Then, the inter-window geodesic information among windows in each window list can be used to reduce window redundancy and window management cost during propagation. As a result, both the running time and memory cost of exact geodesic computations can be reduced.

Based on EWG, a fast and memory-efficient exact geodesic algorithm, namely VTP, is first proposed to solve the single-source discrete geodesic problem (SS-DGP). On one hand, VTP employs EWG to group windows into window lists on edges. Hence, many window pairs can be formed in a window list. In each window pair, the redundancy is identified and removed effectively by simple distance comparisons. Thus, VTP achieves low window redundancy and runs fast. In addition, due to that such redundancy reduction process do not require retaining propagated windows on edges, VTP's memory cost is low. On the other hand, EWG builds the connections between mesh edges and window lists. Thus, the wavefront can be propagated by sorting vertices instead of windows in a priority queue. Since the number of mesh vertices is much smaller than that of windows, the window management cost is dramatically reduced and this further accelerates VTP. As a result, the proposed VTP algorithm is the fastest and most memory-efficient exact geodesic

algorithm so far.

Then, the proposed VTP algorithm is revised and applied to construct the geodesic-metric-based Voronoi diagram on triangle meshes. In this application, the geodesic computation part consumes the majority of the time and memory and is the performance bottleneck. To speed up the computation, the EWG-based window propagation framework in VTP is employed to organize the wavefront propagation by sorting vertices. Thus, the window management cost is dramatically reduced. To reduce the memory cost, the redundant windows which do not contribute to the Voronoi diagram construction are grouped on edges by EWG and removed in batches efficiently. As a result, the proposed algorithm resolves the memory bottleneck of the Voronoi diagram construction without sacrificing its speed.

6.2 Future Work

The work in this thesis leads to the following directions for future work:

Weighted SS-DGP Algorithm The VTP algorithm proposed in this thesis focuses on the geodesic computation under the Euclidean cost metric, i.e. the difficulty of the paths passing through all faces of the mesh is the same. However, to obtain a more realistic model, the faces of the mesh are usually weighted in some applications, e.g. route planning tasks in computer games. Thus, extending the proposed VTP algorithm to solve the weighted SS-DGP problem is an interesting future direction and may result in some useful applications. The main challenge in this direction is that the shortest paths on the unfolded triangle strips are no longer straight line segments. Thus, the window propagation should be redesigned into a weighted version. However, the proposed EWG may also be applied to accelerate the computation by grouping the weighted windows on edges and processing them in batches.

VTP Parallelization To accelerate the computation, parallelizing a traditional CPU-based serial algorithm to run on *Graphics Processing Unit* (GPU) has become an increasing trend in the past decade. Such parallelization usually requires to divide the original CPU-based algorithm into many similar but independent sub-tasks, which form a parallel structure. Following this trend, Ying et al. (2014) first proposed the PCH algorithm for exact geodesic computations, which is a parallelized version of the CH algorithm (Chen and

Han, 1990). Their experiments show that PCH runs a magnitude faster than the ICH algorithm (Xin and Wang, 2009). However, since their method is based on the ICH algorithm whose redundancy reduction rules are not effective, large amounts of redundant windows are propagated. Compared to the ICH algorithm, the proposed VTP algorithm uses EWG to group windows into window lists on edges and several effective windows pruning rules are proposed to remove windows redundancy. Thus, it is expected that parallelizing the VTP algorithm can achieve better performance than PCH. In addition, VTP processes the windows in window lists in *batches*, which can be potentially implemented as independent and similar sub-tasks. This may further improve the performance of the parallelized VTP algorithm.

All-Pair Exact Geodesic Computation The VTP algorithm proposed in this thesis focuses on solving the *Single-Source Discrete Geodesic Problem* (SS-DGP). However, for applications like generating bending invariant signatures (Elad and Kimmel, 2003) and near-isometric surface parametrization (Balasubramanian et al., 2010), the *All-Pair Discrete Geodesic Problem* (AP-DGP) is involved and needs to be solved. Although the AP-DGP problem can be simply solved by performing the SS-DGP algorithm on all vertices in turn, it may cause redundancies and worsen the performance. Since the EWG technique proposed in this thesis groups windows on edges as window lists, the windows redundancy can be reduced using the inter-window information. Thus, it is expected that applying EWG and utilizing such inter-window information can improve the performance of all-pair exact geodesic computations.

Shape Processing Recently, Xin et al. (2016) showed that the distances of geodesic loops can be used to design intrinsic and discriminative shape descriptors, which are used in shape retrieval tasks. The computation of geodesic loops requires finding a path linking two windows on the opposite sides of an edge. Since the proposed EWG technique is essentially based on edges, it is expected that applying EWG helps to find valid window pairs containing such paths efficiently. In addition, Xin et al. (2016) used the ICH algorithm (Xin and Wang, 2009) to compute exact geodesics in their implementation. Since our VTP algorithm outperforms ICH in both running time and memory cost, it is expected that using our VTP for exact geodesic computations can further improve the performance of the applications proposed in (Xin et al., 2016).

Geodesic Remeshing With the development of 3D scanning technology, large quantity of models are obtained via 3D reconstruction algorithms using scanned range data. However, the reconstructed meshes are usually with nonuniform triangle aspect ratio, which is not desired in applications that involves solving partial differential equations on meshes. Thus, a mesh is often required to be uniformly remeshed so that the triangles in the resulting mesh are as close as possible to equilateral triangles. To solve this problem, Peyré and Cohen (2006) proposed an algorithm based on the geodesic farthest point sampling (Eldar et al., 1997). However, since they used the Fast Marching Method (FMM) (Kimmel and Sethian, 1998) for approximate geodesic distance computations, large errors may be produced on the meshes with sliver triangles. To make the remeshing robust and accurate, Liu et al. (2011) proposed a similar algorithm based on exact geodesics computed by the MMP algorithm (Surazhsky et al., 2005). Nevertheless, the MMP algorithm is slow and consumes too much memory. Since the proposed VTP algorithm outperforms MMP in both speed and memory cost, the work on implementing geodesic remeshing algorithms using the proposed VTP algorithm (with possible modifications) is worth pursuing.

Bibliography

K. P. Agarwal, S. Har-Peled, and M. Karia. Computing approximate shortest paths on convex polytopes. *Algorithmica*, 33(2):227–242, 2002. 4, 13

P. K. Agarwal, S. Har-Peled, M. Sharir, and K. R. Varadarajan. Approximating shortest paths on a convex polytope in three dimensions. *J. ACM*, 44 (4):567–584, July 1997. 13, 18

L. Aleksandrov, M. Lanthier, A. Maheshwari, and J. R. Sack. *Algorithm Theory — SWAT'98: 6th Scandinavian Workshop on Algorithm Theory Stockholm, Sweden, July 8–10, 1998 Proceedings*, chapter An ϵ — Approximation algorithm for weighted shortest paths on polyhedral surfaces, pages 11–22. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998. 22

L. Aleksandrov, A. Maheshwari, and J.-R. Sack. Approximation algorithms for geometric shortest path problems. In *Proceedings of the Thirty-second Annual ACM Symposium on Theory of Computing*, STOC '00, pages 286–295, New York, NY, USA, 2000. ACM. 22, 23

L. Aleksandrov, A. Maheshwari, and J.-R. Sack. Determining approximate shortest paths on weighted polyhedral surfaces. *J. ACM*, 52(1):25–53, Jan. 2005. 23

L. Aleksandrov, H. N. Djidjev, H. Guo, A. Maheshwari, D. Nussbaum, and J.-R. Sack. Algorithms for approximate shortest path queries on weighted polyhedral surfaces. *Discrete & Computational Geometry*, 44(4):762–801, 2010. 5, 23

F. Aurenhammer. Voronoi diagrams—a survey of a fundamental geometric data structure. *ACM Comput. Surv.*, 23(3):345–405, Sept. 1991. 28

M. Balasubramanian, J. Polimeni, and E. Schwartz. Exact geodesics and shortest paths on polyhedral surfaces. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 31(6):1006–1016, June 2009. 17

M. Balasubramanian, J. R. Polimeni, and E. L. Schwartz. Near-isometric flattening of brain surfaces. *NeuroImage*, 51(2):694–703, 2010. 125

L. Bertelli, B. Sumengen, and B. Manjunath. Redundancy in all pairs fast marching method. In *Image Processing, 2006 IEEE International Conference on*, pages 3033–3036, Oct 2006. 25

J. Canny and J. Reif. New lower bound techniques for robot motion planning problems. In *Foundations of Computer Science, 1987., 28th Annual Symposium on*, pages 49–60, Oct 1987. 1, 3

B. Chazelle. Triangulating a simple polygon in linear time. *Discrete & Computational Geometry*, 6(3):485–524, 1991. 33

J. Chen and Y. 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. 4, 16, 19, 57, 65, 83, 124

K. Clarkson. Approximation algorithms for shortest path motion planning. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 56–65, New York, NY, USA, 1987. ACM. 21

K. Crane, C. Weischedel, and M. Wardetzky. Geodesics in heat: A new approach to computing distance based on heat flow. *ACM Trans. Graph.*, 32 (5):152:1–152:11, Oct. 2013. 3, 6, 18, 27

E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959. 1, 11, 14, 18, 20, 21, 22, 25

R. Dudley. Metric entropy of some classes of sets with differentiable boundaries. *Journal of Approximation Theory*, 10(3):227 – 236, 1974. 13

A. Elad and R. Kimmel. On bending invariant signatures for surfaces. *IEEE Trans. Pattern Anal. Mach. Intell.*, 25(10):1285–1295, Oct. 2003. 125

Y. Eldar, M. Lindenbaum, M. Porat, and Y. Y. Zeevi. The farthest point strategy for progressive image sampling. *IEEE Transactions on Image Processing*, 6(9):1305–1315, Sep 1997. 126

S. Fortune. A sweepline algorithm for voronoi diagrams. In *Proceedings of the Second Annual Symposium on Computational Geometry*, SCG '86, pages 313–322, New York, NY, USA, 1986. ACM. 30

P. J. Green and R. Sibson. Computing dirichlet tessellations in the plane. *The Computer Journal*, 21(2):168, 1978. 28, 29

S. Har-Peled. Approximate shortest paths and geodesic diameter on a convex polytope in three dimensions. *Discrete & Computational Geometry*, 21(2): 217–231, 1999a. 13, 19

S. Har-Peled. Constructing approximate shortest path maps in three dimensions. *SIAM Journal on Computing*, 28(4):1182–1197, 1999b. 19

S. Har-Peled, M. Sharir, and K. R. Varadarajan. Approximating shortest paths on a convex polytope in three dimensions. In *Proceedings of the Twelfth Annual Symposium on Computational Geometry*, SCG '96, pages 329–338, New York, NY, USA, 1996. ACM. 13

J. Hershberger and S. Suri. Practical methods for approximating shortest paths on a convex polytope in {R3}. *Computational Geometry*, 10(1):31 – 46, 1998. 12, 13

J. Hershberger and S. Suri. An optimal algorithm for euclidean shortest paths in the plane. *SIAM Journal on Computing*, 28(6):2215–2256, 1999. 14

T. Kanai and H. Suzuki. Approximate shortest path on a polyhedral surface and its applications. *Computer-Aided Design*, 33(11):801 – 811, 2001. 4, 18

B. Kaneva and J. O'Rourke. An implementation of chen & han's shortest paths algorithm. In *Proceedings of the 12th Canadian Conference on Computational Geometry, Fredericton, New Brunswick, Canada, August 16-19, 2000*, 2000. 16

S. Kapoor. Efficient computation of geodesic shortest paths. In *Proceedings of the Thirty-first Annual ACM Symposium on Theory of Computing*, STOC '99, pages 770–779, New York, NY, USA, 1999. ACM. 4, 19

S. Kiazyk, S. Loriot, and É. C. de Verdière. Triangulated surface mesh shortest paths. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.9 edition, 2016. 5

T.-Y. Kim, N. Chentanez, and M. Müller-Fischer. Long range attachments - a method to simulate inextensible clothing in computer games. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '12, pages 305–310, Aire-la-Ville, Switzerland, Switzerland, 2012. Eurographics Association. 5

R. Kimmel and J. A. Sethian. Computing geodesic paths on manifolds. *Proceedings of the National Academy of Sciences*, 95(15):8431–8435, 1998. 3, 6, 15, 18, 25, 27, 32, 126

R. Kimmel and J. A. Sethian. Fast voronoi diagrams and offsets on triangulated surfaces. In *Proc. of AFA Conf. on Curves and Surfaces*, pages 193–202. University Press, 1999. 32

M. Kline. *Mathematical thought from ancient to modern times*, volume 3. Oxford University Press, 1990. 2

Y.-K. Lai, Q.-Y. Zhou, S.-M. Hu, and R. R. Martin. Feature sensitive mesh segmentation. In *Proceedings of the 2006 ACM Symposium on Solid and Physical Modeling*, SPM '06, pages 17–25, New York, NY, USA, 2006. ACM. 5

M. Lanthier, A. Maheshwari, and J.-R. Sack. Approximating weighted shortest paths on polyhedral surfaces. In *Proceedings of the Thirteenth Annual Symposium on Computational Geometry*, SCG '97, pages 274–283, New York, NY, USA, 1997. ACM. 21, 22

M. Lanthier, A. Maheshwari, and J. R. Sack. Approximating shortest paths on weighted polyhedral surfaces. *Algorithmica*, 30(4):527–562, 2001. 21, 22

M. Lanthier, D. Nussbaum, and J.-R. Sack. Parallel implementation of geometric shortest path algorithms. *Parallel Computing*, 29(10):1445 – 1479, 2003. High Performance Computing with geographical data. 22

R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979. 24

S. Liu, X. Jin, C. C. L. Wang, and J. X. Chen. Water wave animation on mesh surfaces. *Computing in Science and Engg.*, 8(5):81–87, Sept. 2006. 5

Y. Liu. Semi-continuity of skeletons in two-manifold and discrete voronoi approximation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 37(9):1938–1944, Sept 2015. 5

Y. Liu, Z. Chen, and K. Tang. Construction of iso-contours, bisectors, and voronoi diagrams on triangulated surfaces. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 33(8):1502–1517, Aug 2011. vi, x, 5, 7, 32, 34, 47, 48, 93, 94, 102, 107, 109, 111, 113, 118, 121, 122, 126, 164

Y.-J. Liu. Exact geodesic metric in 2-manifold triangle meshes using edge-based data structures. *Computer-Aided Design*, 45(3):695 – 704, 2013. 15, 58

Y.-J. Liu, Q.-Y. Zhou, and S.-M. Hu. Handling degenerate cases in exact geodesic computation on triangle meshes. *The Visual Computer*, 23(9): 661–668, 2007. 15

T. Lozano-Pérez and M. A. Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *Commun. ACM*, 22(10):560–570, Oct. 1979. 1

D. Martínez, L. Velho, and P. C. Carvalho. Computing geodesics on triangular meshes. *Computers & Graphics*, 29(5):667 – 675, 2005. 26

C. S. Mata and J. S. B. Mitchell. A new algorithm for computing shortest paths in weighted planar subdivisions (extended abstract). In *Proceedings of the Thirteenth Annual Symposium on Computational Geometry*, SCG '97, pages 264–273, New York, NY, USA, 1997. ACM. 20, 21

J. S. B. Mitchell and C. H. Papadimitriou. The weighted region problem: Finding shortest paths through a weighted planar subdivision. *J. ACM*, 38 (1):18–73, Jan. 1991. 5, 20

J. S. B. Mitchell, D. M. Mount, and C. H. Papadimitriou. The discrete geodesic problem. *SIAM Journal on Computing*, 16(4):647–668, 1987. 2, 4, 10, 14, 15, 16, 19, 20, 26, 31, 32, 33, 34, 37, 38, 39, 43, 44, 45, 52, 55, 65, 74, 97, 107

D. M. Mount. On finding shortest paths on convex polyhedra. Technical report, DTIC Document, 1985a. 1, 4, 11, 12

D. M. Mount. Voronoi diagrams on the surface of a polyhedron. Technical report, DTIC Document, 1985b. 31, 32

D. M. Mount. Storing the subdivision of a polyhedral surface. *Discrete & Computational Geometry*, 2(2):153–174, 1987. 12

T. Ohya, M. Iri, and K. Murota. Improvements of the incremental method for the voronoi diagram with computational comparison of various algorithms. *J. OPER. RES. SOC. JAPAN.*, 27(4):306–336, 1984. 29

J. O'Rourke. Computational geometry column 35. *SIGACT News*, 30(2): 31–32, June 1999. 4, 19

J. O'Rourke, S. Suri, and H. Booth. *STACS 85: 2nd Annual Symposium on Theoretical Aspects of Computer Science Saarbrücken, January 3–5, 1985*, chapter Shortest paths on polyhedral surfaces, pages 243–254. Springer Berlin Heidelberg, Berlin, Heidelberg, 1985. 1, 4, 17

J. Peethambaran and R. Muthuganapathy. Reconstruction of water-tight surfaces through delaunay sculpting. *Computer-Aided Design*, 58:62 – 72, 2015. Solid and Physical Modeling 2014. 93

G. Peyré and L. D. Cohen. Geodesic remeshing using front propagation. *International Journal of Computer Vision*, 69(1):145–156, 2006. 5, 93, 126

K. Polthier and M. Schmies. *Mathematical Visualization: Algorithms, Applications and Numerics*, chapter Straightest Geodesics on Polyhedral Surfaces, pages 135–150. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998. 3, 26

Y. Qin, X. Han, H. Yu, Y. Yu, and J. Zhang. Fast and exact discrete geodesic computation based on triangle-oriented wavefront propagation. *ACM Trans. Graph.*, 35(4):125:1–125:13, July 2016. 103

Y. Schreiber. An optimal-time algorithm for shortest paths on realistic polyhedra. *Discrete & Computational Geometry*, 43(1):21–53, 2009. 14

Y. Schreiber and M. Sharir. An optimal-time algorithm for shortest paths on a convex polytope in three dimensions. *Discrete & Computational Geometry*, 39(1):500–579, 2007. 14

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. 24, 25

M. I. Shamos and D. Hoey. Closest-point problems. In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, SFCS '75, pages 151–162, Washington, DC, USA, 1975. IEEE Computer Society. 29, 31

M. Sharir and A. Schorr. On shortest paths in polyhedral spaces. *SIAM Journal on Computing*, 15(1):193–215, 1986. 1, 3, 4, 10, 11, 15, 16, 17

C. Sommer. Shortest-path queries in static networks. *ACM Comput. Surv.*, 46 (4):45:1–45:31, Mar. 2014. 1

Z. Sun and J. H. Reif. On finding approximate optimal paths in weighted regions. *Journal of Algorithms*, 58(1):1 – 32, 2006. 5, 23

V. Surazhsky, T. Surazhsky, D. Kirsanov, S. J. Gortler, and H. Hoppe. Fast exact and approximate geodesics on meshes. *ACM Trans. Graph.*, 24(3): 553–560, July 2005. 4, 5, 6, 15, 16, 17, 18, 19, 40, 43, 45, 46, 47, 48, 65, 72, 73, 74, 83, 84, 93, 103, 105, 126, 148, 159

K. R. Varadarajan and P. K. Agarwal. Approximating shortest paths on a nonconvex polyhedron. *SIAM Journal on Computing*, 30(4):1321–1340, 2000. 18

S.-Q. Xin and G.-J. Wang. Efficiently determining a locally exact shortest path on polyhedral surfaces. *Computer-Aided Design*, 39(12):1081 – 1090, 2007. 26

S.-Q. Xin and G.-J. Wang. Improving chen and han's algorithm on the discrete geodesic problem. *ACM Trans. Graph.*, 28(4):104:1–104:8, Sept. 2009. 4, 5, 6, 16, 18, 43, 45, 46, 47, 57, 58, 65, 68, 73, 74, 81, 83, 125, 148

S.-Q. Xin, D. T. P. Quynh, X. Ying, and Y. He. A global algorithm to compute defect-tolerant geodesic distance. In *SIGGRAPH Asia 2012 Technical Briefs*, SA '12, pages 23:1–23:4, New York, NY, USA, 2012. ACM. 26, 27

S.-Q. Xin, W. Wang, S. Chen, J. Zhao, and Z. Shu. Intrinsic girth function for shape processing. *ACM Trans. Graph.*, 35(3):25:1–25:14, Apr. 2016. 125

C. Xu, Y.-J. Liu, Q. Sun, J. Li, and Y. He. Polyline-sourced geodesic voronoi diagrams on triangle meshes. *Computer Graphics Forum*, 33(7):161–170, 2014. vi, 5, 32, 118, 119

C. Xu, T. Wang, Y.-J. Liu, L. Liu, and Y. He. Fast wavefront propagation (fwp) for computing exact geodesic distances on meshes. *Visualization and Computer Graphics, IEEE Transactions on*, 21(7):822–834, July 2015. 4, 7, 16, 17, 43, 45, 46, 47, 74, 83, 87, 93, 113, 114, 148, 159, 164

L. Yatziv, A. Bartesaghi, and G. Sapiro. O(n) implementation of the fast marching algorithm. *Journal of Computational Physics*, 212(2):393 – 399, 2006. 25

X. Ying, X. Wang, and Y. He. Saddle vertex graph (svg): A novel solution to the discrete geodesic problem. *ACM Trans. Graph.*, 32(6):170:1–170:12, Nov. 2013. 4, 17

X. Ying, S.-Q. Xin, and Y. He. Parallel chen-han (pch) algorithm for discrete geodesics. *ACM Trans. Graph.*, 33(1):9:1–9:11, Feb. 2014. 4, 17, 124

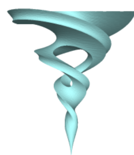
Z. Zhong, X. Guo, W. Wang, B. Lévy, F. Sun, Y. Liu, and W. Mao. Particle-based anisotropic surface meshing. *ACM Trans. Graph.*, 32(4):99:1–99:14, July 2013. 87, 114

N. Zink and A. Hardy. Cloth simulation and collision detection using geometry images. In *Proceedings of the 5th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa*, AFRIGRAPH '07, pages 187–195, New York, NY, USA, 2007. ACM. 5

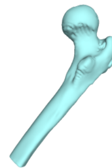
Appendix A

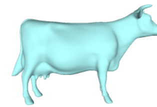
Model Collection

All the 55 models used in the thesis for performance testing are listed as follows, which are from the AIM@SHAPE shape repository (A), Large Geometric Models Archive at Georgia Institute of Technology (B), Suggestive Contour Gallery provided by Princeton University (C) and Stanford scanning repository (D).


Twirl © A

Sword © A


Femur © A


Cow © C

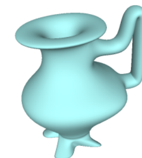
Venus © A

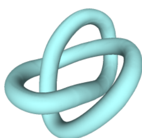
Foot © A

Camel © A

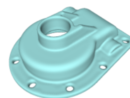
Homer Simpson © A

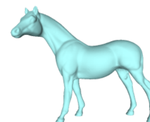
Dilo © A

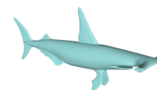

Sketched Vase © A


Knot © A

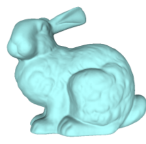
Buste © A


Casting © A


Horse © C


Shark © A

Pegasus © A


Bunny © C

Bimba © A

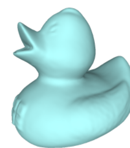
Elephant © A

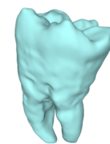
Hand © A

Filigree © A

Woodfish © A

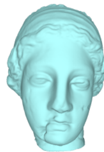
Maxplanck © C

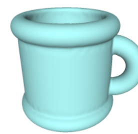

Duck © A

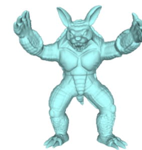

Tooth © A

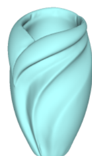
Moai © A

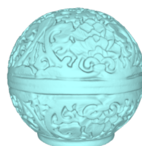
Dancing Children © A


Igea © C

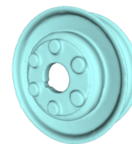

Cup © A


Armadillo © C

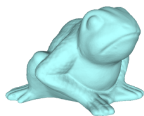

Vase © A


Red Circular Box © A

Julius Caesar © A


Pulley © A

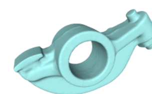
Eros © A


Frog © A

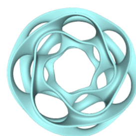
Magalie's Hand © A

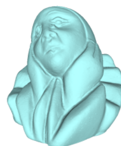
Wooden Chair © A

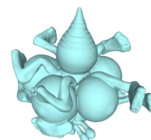
Angel © B


Rocker Arm © A

Fertility © A

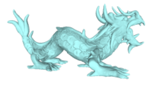

Heptoroid © C

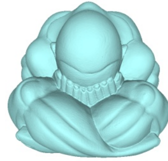

Pierrot © A


Bozbezbozzel © A

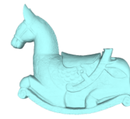
Chinese Dragon © A

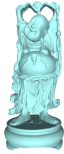
Ramesses © A

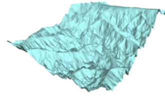

Asian Dragon © D


Pensatore © A

Seahorse © A


IsidoreHorse © A


Happy Buddha © B


Cervino Terrain © A

Neptune © A

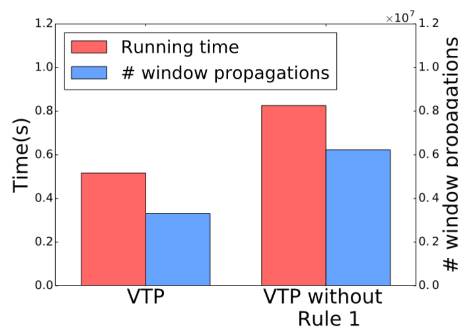
Vase Lion © A

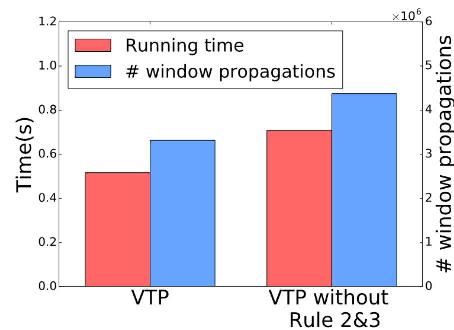
Lucy © C

Appendix B

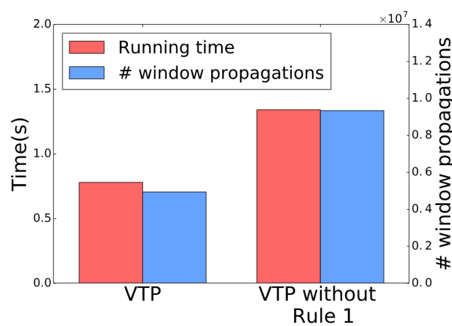
VTP Ablation Study

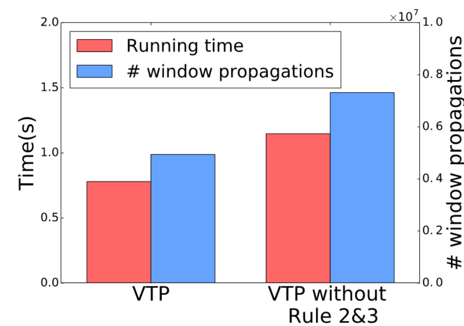
This part shows more performance comparisons of the proposed VTP algorithm with and without Rule 1 and comparisons with and without Rules 2&3. In addition to the two models (Armadillo and Asian Dragon) used in the thesis, this part shows the results on 8 other models with various resolutions.



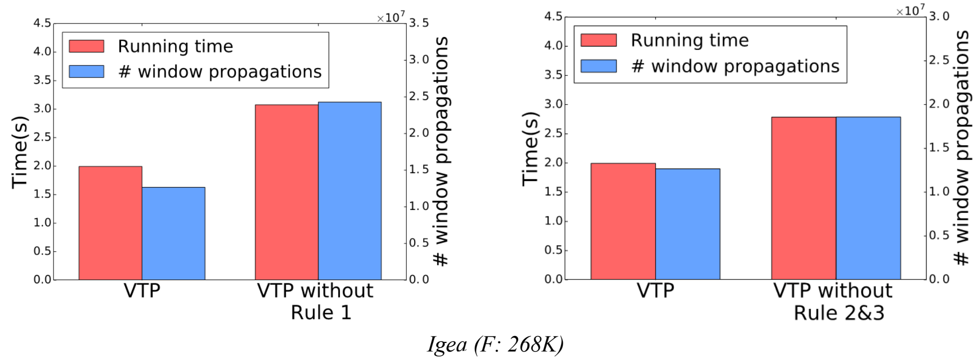


Horse (F: 96K)

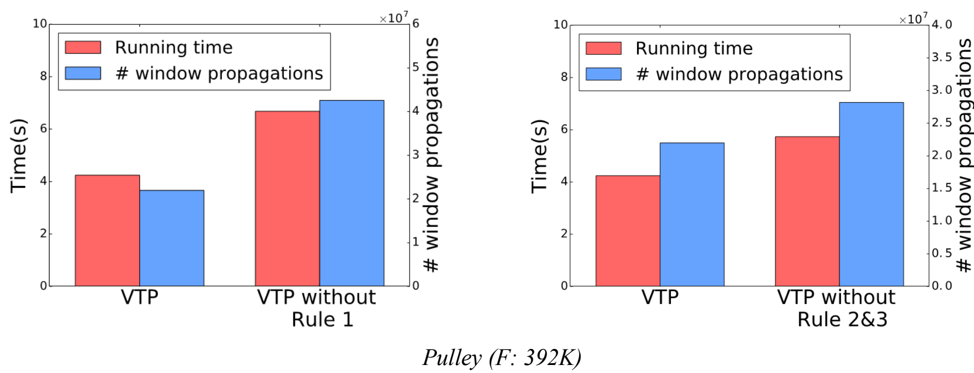




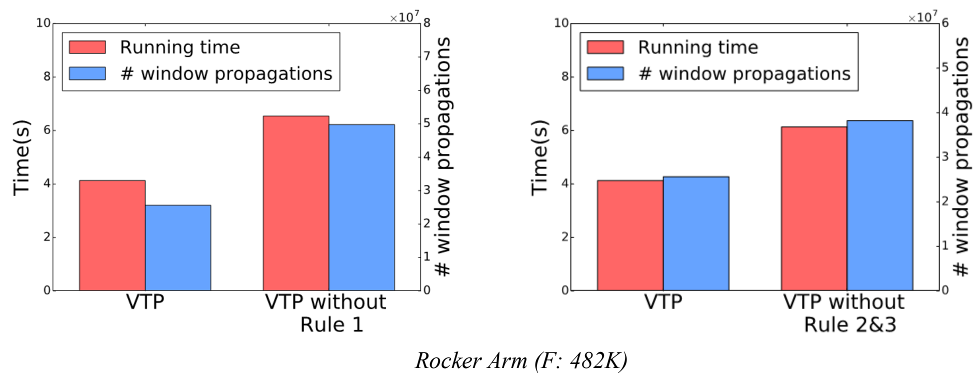
Bunny (F: 144K)



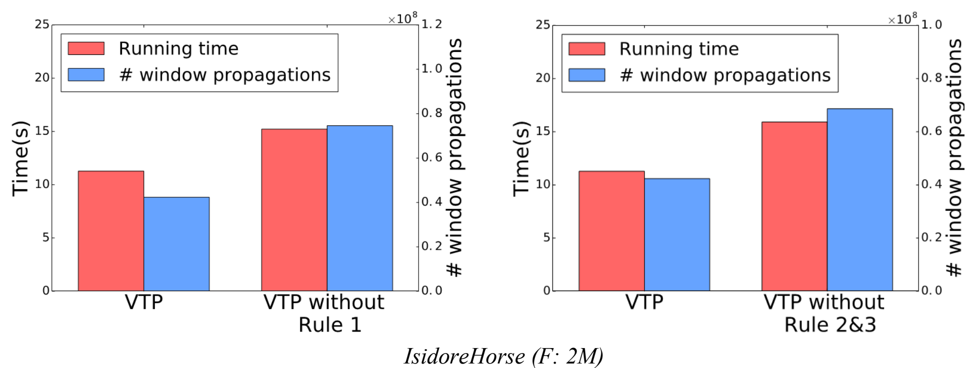
Igea (F: 268K)



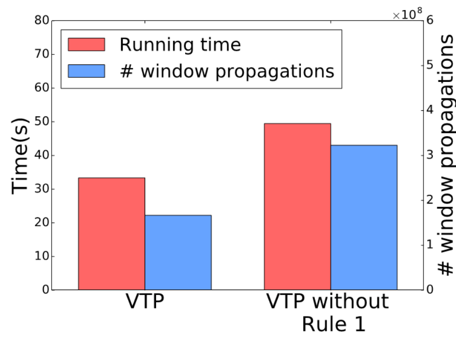
Pulley (F: 392K)

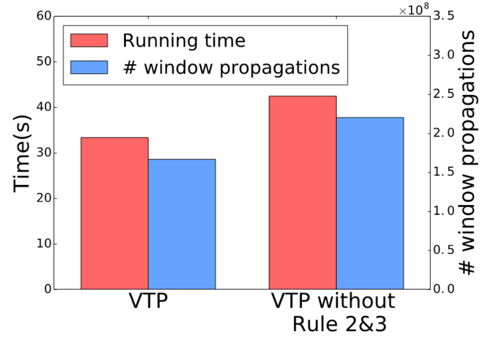


Rocker Arm (F: 482K)

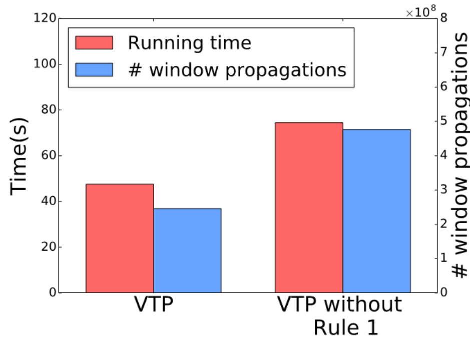


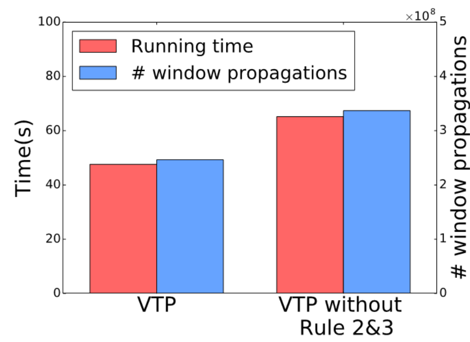
IsidoreHorse (F: 2M)





Happy Buddha (F: 2.6M)





Neptune (F: 4M)

Appendix C

VTP Variants Comparison

This part compares the performance among the variants of the proposed VTP algorithm, including VTP-Exhaustive, VTP-Trimming, VTP-CH, VTP-MMP, FTP and OPVTP, on the 55 models shown in Appendix A. F: means the number of faces of the model.

Model	Performance	Algorithms						
		VTP-Exhaustive	**VTP-Trimming**	**VTP-CH**	**VTP-MMP**	**FTP**	**OPVTP**	**VTP**
Twirl (F: 10K)	Time(s)	0.112	0.041	0.091	0.052	0.060	0.043	0.04
	#window propagations	197,217	192,691	361,272	260,380	191,053	192,235	201,107
	Peak memory(MB)	0.251	0.252	0.30	13.07	0.243	0.25	0.254
Sword (F: 29K)	Time(s)	3.639	0.24	0.55	0.336	0.315	0.261	0.209
	#window propagations	1,468,797	1,451,652	3,009,123	1,559,334	1,512,011	1,459,627	1.499,968
	Peak memory(MB)	0.576	0.573	0.66	82.18	0.61	0.57	0.581
Femur (F: 30K)	Time(s)	0.75	0.17	0.36	0.222	0.175	0.159	0.144
	#window propagations	832,253	808,575	1,825,281	1,084,096	812,559	830,433	848,057
	Peak memory(MB)	0.305	0.296	0.38	58.64	0.30	0.31	0.309
Cow (F: 36K)	Time(s)	0.76	0.23	0.42	0.263	0.211	0.181	0.176
	#window propagations	977,109	944,698	2,084,408	1,255,395	952,731	970,775	993,393
	Peak memory(MB)	0.594	0.584	0.69	68.46	0.59	0.59	0.598
Venus (F: 43K)	Time(s)	1.908	0.31	0.757	0.479	0.302	0.292	0.27
	#window propagations	1,732,229	1,676,392	3,906,934	2,302,895	1,705,248	1,744,113	1,760,343
	Peak memory(MB)	0.66	0.658	0.87	125.68	0.659	0.661	0.672
Foot (F: 44K)	Time(s)	2.2	0.42	0.792	0.534	0.361	0.374	0.303
	#window propagations	1,935,202	1,885,004	4,095,235	2,550,736	1,905,123	1,942,112	1,963,132
	Peak memory(MB)	1.621	1.594	1.61	140.87	1.45	1.47	1.627
Camel (F: 48K)	Time(s)	0.799	0.26	0.501	0.361	0.261	0.222	0.21
	#window propagations	1,145,645	1,101,856	2,588,770	1,433,425	1,112,089	1,133,446	1,166,711
	Peak memory(MB)	0.437	0.419	0.57	78.57	0.44	0.45	0.441
HomerSimpson (F: 48K)	Time(s)	1.316	0.341	0.66	0.381	0.304	0.279	0.248
	#window propagations	1,455,144	1,421,497	3,101,274	1,850,498	1,424,124	1,458,000	1,482,477
	Peak memory(MB)	0.798	0.779	0.97	103.48	0.79	0.791	0.808
Dilo (F: 54K)	Time(s)	0.794	0.272	0.554	0.323	0.305	0.251	0.238
	#window propagations	1,178,516	1,138,802	2,510,230	1,566,420	1,147,055	1,166,106	1,198,835
	Peak memory(MB)	0.578	0.570	0.79	80.08	0.575	0.581	0.587

Continue from previous table

Model	Performance	Algorithms						
		VTP-Exhaustive	VTP-Trimming	VTP-CH	VTP-MMP	FTP	OPVTP	VTP
SketchedVase (F: 54K)	Time(s)	2.695	0.354	0.876	0.629	0.451	0.349	0.314
	#window propagations	2,082,535	2,042,643	4,548,918	2,926,886	2,082,301	2,080,132	2,119,200
	Peak memory(MB)	0.642	0.645	0.77	147.74	0.65	0.64	0.651
Knot (F: 56K)	Time(s)	1.906	0.376	0.77	0.499	0.399	0.350	0.302
	#window propagations	1,998,630	1,926,644	4,191,262	2,557,102	2,000,353	2,003,573	2,015,140
	Peak memory(MB)	0.359	0.347	0.46	144.12	0.358	0.359	0.36
Buste (F: 60K)	Time(s)	1.376	0.371	0.76	0.527	0.362	0.363	0.299
	#window propagations	1,684,666	1,655,014	3,967,893	2,110,539	1,659,081	1,695,365	1,739,991
	Peak memory(MB)	0.646	0.631	0.82	117.89	0.640	0.651	0.658
Casting (F: 90K)	Time(s)	2.965	0.761	1.311	0.743	0.62	0.6	0.516
	#window propagations	2,948,477	2,861,050	6,363,310	3,474,086	2,896,291	2,931,236	2,995,809
	Peak memory(MB)	1.089	1.072	1.36	188.33	1.088	1.09	1.105
Horse (F: 96K)	Time(s)	3.184	0.67	1.567	0.863	0.651	0.641	0.517
	#window propagations	3,262,520	3,157,608	8,106,166	4,410,197	3,240,048	3,283,168	3,317,318
	Peak memory(MB)	1.368	1.344	1.80	231.53	1.350	1.369	1.384
Shark (F: 107K)	Time(s)	6.821	0.855	2.081	1.312	0.908	0.919	0.758
	#window propagations	5,032,274	4,886,346	10,962,701	6,213,042	4,983,023	5,058,816	5,112,354
	Peak memory(MB)	1.011	1.0	1.20	352.92	1.011	1.012	1.015
Pegasus (F: 127K)	Time(s)	3.06	0.767	1.814	1.015	0.861	0.873	0.693
	#window propagations	3,559,660	3,470,668	7,761,300	4,767,038	3,491,952	3,558,053	3,631,004
	Peak memory(MB)	1.671	1.652	2.06	249.17	1.65	1.68	1.696
Bunny (F: 144K)	Time(s)	4.557	0.872	2.672	1.304	1.044	0.908	0.78
	#window propagations	4,801,056	4,686,252	12,491,178	6,454,800	4,755,872	4,875,712	4,943,670
	Peak memory(MB)	1.22	1.146	1.71	340.45	1.20	1.22	1.24
Bimba (F: 149K)	Time(s)	7.005	1.094	3.04	1.634	1.428	1.27	0.982
	#window propagations	5,655,097	5,464,752	12,933,807	7,655,990	5,595,011	5,700,385	5,749,138
	Peak memory(MB)	2.239	2.171	2.81	407.11	2.20	2.249	2.258
Elephant (F: 160K)	Time(s)	7.686	1.282	3.452	1.927	1.579	1.558	1.179
	#window propagations	6,679,878	6,521,389	14,795,030	9,187,716	6,607,611	6,728,077	6,779,937
	Peak memory(MB)	3.116	3.066	3.92	487.31	3.109	3.149	3.157
Hand (F: 176K)	Time(s)	14.187	2.196	4.451	3.129	1.955	2.003	1.638
	#window propagations	9,975,156	9,654,198	22,295,976	13,895,643	9,924,021	10,134,178	10,071,080
	Peak memory(MB)	2.680	2.623	3.05	733.33	2.652	2.721	2.689
Filigree (F: 186K)	Time(s)	3.031	0.88	2.178	1.12	1.255	1.0	0.821
	#window propagations	3,977,864	3,877,297	8,294,639	4,852,485	3,870,035	3,928,044	4,066,106
	Peak memory(MB)	1.467	1.443	1.81	257.83	1.439	1.451	1.49
Woodfish (F: 191K)	Time(s)	12.921	1.832	5.094	2.756	2.01	2.054	1.542
	#window propagations	9,535,507	9,326,365	22,434,888	12,619,027	9,453,888	9,641,025	9,722,818
	Peak memory(MB)	2.873	2.796	3.60	693.69	2.80	2.889	2.899
Maxplanck (F: 210K)	Time(s)	20.913	2.426	7.169	3.541	2.569	2.663	1.891
	#window propagations	12,250,767	12,035,550	30,642,469	16,150,075	12,194,999	12,578,888	12,597,188
	Peak memory(MB)	3.334	3.285	4.31	887.99	3.30	3.399	3.405

Continue from previous table

Model	Performance	Algorithms						
		VTP-Exhaustive	VTP-Trimming	VTP-CH	VTP-MMP	FTP	OPVTP	VTP
Duck (F: 219K)	Time(s)	30.999	3.74	9.209	4.538	2.919	3.54	2.53
	#window propagations	16,323,333	15,828,827	42,977,191	21,059,883	16,297,522	16,887,512	16,821,923
	Peak memory(MB)	3.655	3.490	5.25	1160.22	3.650	3.79	3.743
Tooth (F: 220K)	Time(s)	24.13	2.74	7.543	3.972	3.05	3.11	2.339
	#window propagations	14,130,211	13,841,486	33,695,533	18,534,740	14,035,011	14,397,995	14,458,399
	Peak memory(MB)	4.741	4.658	6.13	1037.37	4.71	4.80	4.833
Moai (F: 238K)	Time(s)	25.846	2.965	8.818	4.286	3.64	3.276	2.483
	#window propagations	14,748,182	14,421,835	36,615,050	19,858,675	14,676,991	15,097,323	15,085,009
	Peak memory(MB)	4.548	4.467	6.10	1076.09	4.49	4.74	4.631
DancingChildren (F: 265K)	Time(s)	8.352	1.837	4.736	2.309	2.285	2.143	1.707
	#window propagations	8,344,456	8,143,451	18,226,736	10,416,158	8,195,406	8,360,313	8,501,699
	Peak memory(MB)	3.524	3.49	4.25	566.14	3.501	3.533	3.569
Igea (F: 268K)	Time(s)	16.214	2.124	7.087	3.369	2.377	2.41	1.995
	#window propagations	12,310,461	12,114,442	32,998,672	17,106,678	12,533,960	12,664,898	12,673,530
	Peak memory(MB)	2.168	2.151	3.11	890.92	2.212	2.210	2.218
Cup (F: 316K)	Time(s)	91.437	6.241	16.825	8.34	6.788	6.753	5.465
	#window propagations	32,361,352	29,710,676	69,021,313	35,266,895	32,080,266	32,702,102	32,580,614
	Peak memory(MB)	9.782	8.793	11.06	1880.35	9.770	9.98	9.835
Armadillo (F: 345K)	Time(s)	5.196	1.829	4.692	2.206	2.141	1.975	1.628
	#window propagations	19,193,615	7,596,092	19,584,534	10,258,285	7,745,552	7,884,201	8,084,456
	Peak memory(MB)	2.03	1.396	2.01	538.32	1.399	1.401	1.458
Vase (F: 354K)	Time(s)	47.964	4.324	13.569	7.304	4.650	5.112	4.056
	#window propagations	25,461,130	24,848,861	60,607,873	33,891,566	25,400,117	25,978,005	26,004,542
	Peak memory(MB)	5.609	5.438	7.00	1860.06	5.80	5.82	5.743
Red Circular Box (F: 360K)	Time(s)	7.203	1.882	4.991	2.195	2.471	2.577	1.763
	#window propagations	7,583,520	7,384,425	17,247,883	9,275,636	7,426,105	7,509,066	7,781,762
	Peak memory(MB)	1.518	1.521	1.95	479.66	1.525	1.53	1.554
Julius Caesar (F: 386K)	Time(s)	10.683	2.9	9.383	3.605	3.119	3.03	2.372
	#window propagations	12,417,028	12,044,577	35,093,981	17,022,542	12,322,351	12,708,557	12,744,572
	Peak memory(MB)	2.116	2.073	2.52	877.05	2.149	2.152	2.153
Pulley (F: 392K)	Time(s)	32.359	4.44	13.325	6.252	5.501	5.599	4.242
	#window propagations	21,472,248	21,090,529	49,453,462	28,510,288	21,518,838	22,007,199	21,989,818
	Peak memory(MB)	4.911	4.882	6.12	1537.16	5.01	5.12	5.030
Eros (F: 394K)	Time(s)	11.146	3.199	7.91	3.347	3.59	3.265	2.623
	#window propagations	11,493,579	11,249,885	28,928,294	14,327,257	11,456,266	11,665,873	11,856,262
	Peak memory(MB)	2.525	2.430	3.63	752.46	2.521	2.598	2.602
Frog (F: 394K)	Time(s)	24.242	4.179	12.728	5.719	4.559	4.388	3.175
	#window propagations	19,324,968	19,356,277	57,143,731	30,435,416	20,838,259	20,221,979	20,134,270
	Peak memory(MB)	3.479	3.518	5.11	1393.93	3.69	3.62	3.610
Magalie'sHand (F: 396K)	Time(s)	8.802	2.712	8.501	3.503	3.508	2.958	2.387
	#window propagations	11,095,197	10,626,126	30,829,525	15,577,298	10,948,457	11,111,123	11,324,832
	Peak memory(MB)	2.861	2.775	4.31	764.68	2.875	2.881	2.895

Continue from previous table

Model	Performance	Algorithms						
		VTP-Exhaustive	VTP-Trimming	VTP-CH	VTP-MMP	FTP	OPVTP	VTP
WoodenChair (F: 408K)	Time(s)	34.168	4.273	12.511	5.958	5.019	4.961	4.071
	#window propagations	21,544,323	20,858,337	51,026,008	29,503,171	21,488,002	21,881,208	21,937,266
	Peak memory(MB)	4.866	4.747	5.67	1541.42	4.856	4.928	4.935
Angel (F: 474K)	Time(s)	14.23	3.175	8.354	4.443	3.812	3.705	2.877
	#window propagations	15,525,223	15,126,178	36,360,538	20,982,691	15,397,863	15,718,815	15,858,241
	Peak memory(MB)	2.239	2.202	2.82	1105.82	2.212	2.251	2.272
Rocker Arm (F: 482K)	Time(s)	36.586	4.655	15.449	6.954	4.83	5.169	4.13
	#window propagations	24,289,066	24,380,006	69,208,037	33,947,674	25,013,422	25,723,699	25,654,638
	Peak memory(MB)	3.43	3.49	5.32	1797.18	3.68	3.71	3.70
Fertility (F: 483K)	Time(s)	30.432	4.961	14.837	6.719	5.905	5.502	4.133
	#window propagations	23,589,123	23,873,376	63,236,514	32,461,272	24,995,619	24,523,539	24,686,942
	Peak memory(MB)	4.345	4.359	5.68	1588.54	4.64	4.311	4.465
Heptoroid (F: 573K)	Time(s)	47.982	7.359	17.634	10.691	8.425	7.46	5.556
	#window propagations	33,930,378	33,605,904	72,113,378	52,419,840	33,900,519	34,370,470	34,287,452
	Peak memory(MB)	5.254	5.238	6.07	2520.55	5.24	5.39	5.284
Pierrot (F: 887K)	Time(s)	74.345	10.325	36.26	15.232	13.149	11.713	9.136
	#window propagations	49,306,554	49,909,353	153,799,315	74,822,570	54,026,799	51,796,977	51,644,860
	Peak memory(MB)	5.326	3569.28	9.42	3569.23	5.88	5.73	5.614
Bozbezbozzel (F: 911K)	Time(s)	45.808	5.521	22.211	11.952	9.601	8.768	7.005
	#window propagations	38,123,582	36,572,880	92,447,464	56,200,036	37,691,146	38,404,143	38,660,645
	Peak memory(MB)	6.171	5.942	7.89	2731.02	6.058	6.209	6.229
Chinese dragon (F: 1,222K)	Time(s)	133.271	17.996	47.582	21.553	19.245	18.052	16.435
	#window propagations	72,048,774	69,750,625	169,903,995	99,337,629	71,568,095	72,635,813	72,865,547
	Peak memory(MB)	9.842	9.611	11.82	5284.42	9.789	9.802	9.918
Asian Dragon (F: 1,400K)	Time(s)	49.954	13.763	29.492	15.388	13.011	11.415	9.495
	#window propagations	46,926,451	46,316,630	109,311,094	61,995,300	46,523,513	47,572,141	48,217,896
	Peak memory(MB)	4.036	4.017	5.253	3354.04	4.10	4.20	4.373
Ramesses (F: 1,653K)	Time(s)	36.667	9.834	22.466	9.691	13.125	10.669	8.938
	#window propagations	32,930,864	32,144,860	79,642,701	38,664,117	32,478,771	33,036,958	34,128,397
	Peak memory(MB)	2.853	2.879	3.92	2014.17	2.842	2.901	2.948
Pensatore (F: 1,996K)	Time(s)	97.4	21.606	67.756	24.9	25.879	23.201	20.328
	#window propagations	84,606,295	85,631,449	256,922,732	130,392,192	88,479,353	88,527,448	88,716,654
	Peak memory(MB)	5.728	5.75	9.62	6051.53	5.79	5.792	5.797
Seahorse (F: 2,014K)	Time(s)	206.269	24.674	83.202	35.234	29.061	28.843	21.869
	#window propagations	126,848,230	125,823,913	324,094,284	172,033,093	126,316,839	129,482,659	130,860,212
	Peak memory(MB)	5.596	5.589	7.65	9295.01	5.593	5.75	5.767
IsidoreHorse (F: 2,209K)	Time(s)	25.143	12.622	35.209	16.547	17.449	13.428	11.28
	#window propagations	41,341,363	39,453,625	134,929,706	68,234,985	40,457,511	40,760,043	42,350,025
	Peak memory(MB)	3.002	2.919	5.23	2725.09	2.985	2.998	3.062
Happy Buddha (F: 2,583K)	Time(s)	381.02	34.973	103.901	49.307	41.205	42.453	33.373
	#window propagations	164,137,340	159,542,907	410,199,299	218,302,329	165,473,466	167,592,235	166,942,683
	Peak memory(MB)	9.160	9.084	12.82	11408.55	9.32	9.59	9.337

Continue from previous table

Model	**Performance**	**Algorithms**						
		VTP-Exhaustive	**VTP-Trimming**	**VTP-CH**	**VTP-MMP**	**FTP**	**OPVTP**	**VTP**
Cervino Terrain (F: 3,146K)	Time(s)	124.557	31.251	72.083	32.869	42.257	36.542	28.287
	#window propagations	121,895,449	132,331,239	279,286,307	108,256,256	125,640,081	125,847,948	135,012,449
	Peak memory(MB)	4.95	5.77	7.07	5627.34	5.76	5.75	5.85
Neptune (F: 4,008K)	Time(s)	665.847	58.49	158.912	60.297	64.011	60.187	47.629
	#window propagations	239,054,124	239,375,390	606,937,112	278,925,270	243,102,352	244,573,921	246,364,008
	Peak memory(MB)	15.62	15.962	17.65	14221.35	16.1	16.19	16.38
VaseLion (F: 6,370K)	Time(s)	2560.92	160.497	461.345	Out of memory	174.235	162.705	145.455
	#window propagations	687,375,867	681,210,620	1,729,700,093		686,216,897	702,119,736	704,638,382
	Peak memory(MB)	38.481	38.385	52.21		38.45	39.22	39.257
Lucy (F: 14,464K)	Time(s)	16559	615.215	1809.91	Out of memory	617.343	608.414	549.934
	#window propagations	2,703,707,866	2,733,324,263	6,859,484,793		2,668,122,127	2,734,517,299	2,808,823,718
	Peak memory(MB)	66.096	66.848	78.31		67.81	68.01	69.42

Appendix D

Distribution of Window Propagations

This part counts prime propagations and secondary propagations (which are defined in the thesis) when applying the proposed VTP algorithm to all the 55 testing models in Appendix A and the results are shown as follows. F: means the number of faces of the model.

Model	Prime Propagation	Secondary Propagation	Model	Prime Propagation	Secondary Propagation
Twirl (F: 10K)	181,258 (90.13%)	19,849 (9.87%)	Cup (F: 316K)	31,974,614 (98.14%)	606,000 (1.86%)
Sword (F: 29K)	1,278,123 (85.21%)	221,845 (14.79%)	Armadillo (F: 345K)	7,589,687 (96.88%)	494,769 (3.12%)
Femur (F: 30K)	811,336 (95.67%)	36,721 (4.33%)	Vase (F: 354K)	25,656,081 (98.66%)	348,461 (1.34%)
Cow (F: 36K)	948,889 (95.52%)	44,504 (4.48%)	Red Circular Box (F: 360K)	7,562,316 (97.18%)	219,446 (2.82%)
Venus (F: 43K)	1,712,462 (97.28%)	47,881 (2.72%)	Julius Caesar (F: 386K)	12,487,132 (97.98%)	257,440 (2.02%)
Foot (F: 44K)	1,892,852 (96.42%)	70,280 (3.58%)	Pulley (F: 392K)	21,670,966 (98.55%)	318,852 (1.45%)
Camel (F: 48K)	1,116,892 (95.73%)	49,819 (4.27%)	Eros (F: 394K)	11,625,065 (98.05%)	231,197 (1.95%)
HomerSimpson (F: 48K)	1,413,690 (95.36%)	68,787 (4.64%)	Frog (F: 394K)	19,820,175 (98.44%)	314,095 (1.56%)

Continue from previous table

Model	Prime Propagation	Secondary Propagation	Model	Prime Propagation	Secondary Propagation
Dilo (F: 54K)	1,134,338 (94.62%)	63,278 (5.47%)	Magalie's Hand (F: 396K)	11,123,250 (98.22%)	201,882 (1.78%)
SketchedVase (F: 54K)	1,954,114 (92.21%)	165,086 (7.79%)	WoodenChair (F: 408K)	21,509,489 (98.05%)	427,777 (1.95%)
Knot (F: 56K)	1,945,416 (96.54%)	69,724 (3.46%)	Angel (F: 474K)	15,334,919 (96.7%)	523,322 (3.3%)
Buste (F: 60K)	1,659,429 (95.37%)	80,562 (4.63%)	Rocker Arm (F: 482K)	25,031,230 (97.57%)	623,408 (2.43%)
Casting (F: 90K)	2,894,850 (96.63%)	100,959 (3.37%)	Fertility (F: 483K)	23,588,373 (95.55%)	1,098,569 (4.45%)
Horse (F: 96K)	3,228,746 (97.33%)	88,572 (2.67%)	Heptoroid (F: 573K)	32,713,658 (95.41%)	1,573,794 (4.59%)
Shark (F: 107K)	4,836,798 (94.61%)	275,556 (5.39%)	Pierrot (F: 887K)	49,770,152 (96.37%)	1,874,708 (3.63%)
Pegasus (F: 127K)	3,512,633 (96.74%)	118,371 (3.26%)	Bozbezbozzel (F: 911K)	37,829,441 (97.85%)	831,204 (2.15%)
Bunny (F: 144K)	4,821,561 (97.53%)	122,109 (2.47%)	Chinese Dragon (F: 1,222K)	72,158,751 (99.03%)	706,796 (0.97%)
Bimba (F: 149K)	5,655,427 (98.37%)	93,711 (1.63%)	Ramesses (F: 1,653K)	32,179,666 (94.29%)	1,948,713 (5.71%)
Elephant (F: 160K)	6,615,185 (97.57%)	164,752 (2.43)	Asian dragon (F: 1,400K)	46,086,665 (95.58%)	2,131,231 (4.42%)
Hand (F: 176K)	9,750,820 (96.82%)	320,260 (3.18%)	Pensatore (F: 1,996K)	85,513,983 (96.39%)	3,202,671 (3.61%)
Filigree (F: 186K)	3,780,259 (92.97%)	285,847 (7.03%)	Seahorse (F: 2,014K)	125,193,965 (95.67%)	5,666,247 (4.33%)
Woodfish (F: 191K)	9,424,327 (96.93%)	298,491 (3.07%)	IsidoreHorse (F: 2,209K)	41,155,754 (97.18%)	1,194,271 (2.82%)
Maxplanck (F: 210K)	12,262,103 (97.34%)	335,085 (2.66%)	Happy Buddha (F: 2,583K)	161,199,855 (96.56%)	5,742,828 (3.44%)
Duck (F: 219K)	16,525,857 (98.24%)	296,066 (1.76%)	Cervino Terrain (F: 3,146K)	117,649,848 (87.41%)	17,362,601 (12.59%)
Tooth (F: 220K)	13,955,247 (96.52%)	503,152 (3.48%)	Neptune (F: 4,008K)	240,451,272 (97.6%)	5,912,736 (2.4%)
Moai (F: 238K)	14,674,697 (97.28%)	410,312 (2.72%)	Vase Lion (F: 6,370K)	685,613,146 (97.3%)	19,025,236 (2.7%)
DancingChildren (F: 265K)	8,179,485 (96.21%)	322,214 (3.79%)	Lucy (F: 14,464K)	2,756,860,479 (98.15%)	51,963,239 (1.85%)
Igea (F: 268K)	12,444,139 (98.19%)	229,391 (1.81%)	***Mean***	**96.25%**	**3.75%**
			Standard Deviation	**0.026**	**0.026**

Appendix E

Performance Comparison between VTP and Others

This part compares the performance between VTP and the state-of-the-art exact geodesic algorithms: ICH, MMP, FWP-CH and FWP-MMP (Surazhsky et al., 2005; Xin and Wang, 2009; Xu et al., 2015) on all the 55 models shown in Appendix A.

Model	Performance	Algorithms				
		ICH	MMP	FWP-CH	FWP-MMP	VTP
Twirl (F: 10K)	Time(s)	0.116	0.134	0.085	0.062	0.04
	#window propagations	346,691	260,339	355,482	262,748	201,107
	Peak memory(MB)	0.28	13.07	0.29	13.07	0.254
Sword (F: 29K)	Time(s)	0.996	1.002	0.748	0.437	0.209
	#window propagations	2,973,951	1,537,986	3,019,022	1,634,507	1.499,968
	Peak memory(MB)	0.65	82.18	0.66	82.02	0.581
Femur (F: 30K)	Time(s)	0.594	0.604	0.387	0.264	0.144
	#window propagations	1,799,106	1,089,745	1,803,536	1,084,237	848,057
	Peak memory(MB)	0.37	58.64	0.38	58.64	0.309
Cow (F: 36K)	Time(s)	0.756	0.814	0.453	0.328	0.176
	#window propagations	2,047,472	1,257,463	2,049,564	1,250,713	993,393
	Peak memory(MB)	0.69	68.46	0.69	68.46	0.598
Venus (F: 43K)	Time(s)	1.379	1.525	0.813	0.579	0.27
	#window propagations	3,868,758	2,309,841	3,871,332	2,301,511	1,760,343
	Peak memory(MB)	0.86	125.68	0.86	125.68	0.672
Foot (F: 44K)	Time(s)	1.65	1.941	1.096	0.679	0.303
	#window propagations	4,056,466	2,558,311	4,057,698	2,548,552	1,963,132
	Peak memory(MB)	1.60	140.86	1.59	140.87	1.627
Camel (F: 48K)	Time(s)	0.888	0.877	0.564	0.376	0.21
	#window propagations	2,547,675	1,434,054	2,551,711	1,429,049	1,166,711
	Peak memory(MB)	0.57	78.57	0.57	78.57	0.441
HomerSimpson (F: 48K)	Time(s)	1.166	1.304	0.815	0.49	0.248
	#window propagations	3,052,442	1,853,265	3,054,551	1,845,828	1,482,477
	Peak memory(MB)	0.97	103.48	0.97	103.48	0.808

Continue from previous table

Model	Performance	Algorithms				
		ICH	MMP	FWP-CH	FWP-MMP	VTP
Dilo (F: 54K)	Time(s)	0.9	0.985	0.581	0.4	0.238
	#window propagations	2,437,808	1,567,105	2,466,709	1,568,832	1,198,835
	Peak memory(MB)	0.78	80.08	0.79	80.09	0.587
SketchedVase (F: 54K)	Time(s)	1.51	1.959	0.881	0.766	0.314
	#window propagations	4,299,781	2,788,262	4,319,563	2,798,371	2,119,200
	Peak memory(MB)	0.75	147.74	0.75	147.74	0.651
Knot (F: 56K)	Time(s)	1.412	1.576	0.875	0.63	0.302
	#window propagations	4,158,394	2,556,797	4,148,550	2,543,995	2,015,140
	Peak memory(MB)	0.45	144.12	0.45	144.12	0.36
Buste (F: 60K)	Time(s)	1.444	1.412	0.885	0.558	0.299
	#window propagations	3,914,969	2,119,762	3,920,578	2,108,560	1,739,991
	Peak memory(MB)	0.81	117.89	0.81	117.89	0.658
Casting (F: 90K)	Time(s)	2.547	2.586	1.499	0.943	0.516
	#window propagations	6,288,906	3,485,917	6,307,579	3,468,718	2,995,809
	Peak memory(MB)	1.35	188.33	1.35	188.33	1.105
Horse (F: 96K)	Time(s)	3.187	3.129	2.25	1.146	0.517
	#window propagations	7,944,765	4,404,199	7,963,743	4,384,422	3,317,318
	Peak memory(MB)	1.81	231.51	1.78	231.53	1.384
Shark (F: 107K)	Time(s)	4.222	4.827	2.345	1.685	0.758
	#window propagations	10,980,857	6,265,563	11,116,740	6,307,618	5,112,354
	Peak memory(MB)	1.21	352.92	1.22	352.94	1.015
Pegasus (F: 127K)	Time(s)	3.392	3.64	2.184	1.361	0.693
	#window propagations	7,598,676	4,782,987	7,646,452	4,751,645	3,631,004
	Peak memory(MB)	2.03	249.17	2.06	249.17	1.696
Bunny (F: 144K)	Time(s)	5.034	4.612	3.056	1.737	0.78
	#window propagations	12,305,579	6,485,320	12,327,991	6,451,352	4,943,670
	Peak memory(MB)	1.69	340.45	1.70	340.46	1.24
Bimba (F: 149K)	Time(s)	6.018	6.542	3.547	2.248	0.982
	#window propagations	12,730,540	7,685,795	12,771,938	7,650,043	5,749,138
	Peak memory(MB)	2.78	407.11	2.79	407.11	2.258
Elephant (F: 160K)	Time(s)	7.383	8.019	4.021	2.667	1.179
	#window propagations	14,594,865	9,215,918	14,619,409	9,175,973	6,779,937
	Peak memory(MB)	3.89	487.31	3.89	487.31	3.157
Hand (F: 176K)	Time(s)	10.385	13.322	5.632	4.214	1.638
	#window propagations	21,915,233	13,848,650	22,049,873	13,963,632	10,071,080
	Peak memory(MB)	3.02	733.33	3.06	733.33	2.689
Filigree (F: 186K)	Time(s)	3.507	3.719	2.177	1.495	0.821
	#window propagations	8,002,484	4,836,918	8,228,318	4,957,377	4,066,106
	Peak memory(MB)	1.80	257.83	1.81	257.85	1.49
Woodfish (F: 191K)	Time(s)	11.183	12.116	6.064	3.742	1.542
	#window propagations	22,234,036	12,638,238	22,245,833	12,592,783	9,722,818
	Peak memory(MB)	3.60	693.69	3.58	693.69	2.899

Continue from previous table

Model	Performance	Algorithms				
		ICH	MMP	FWP-CH	FWP-MMP	VTP
Maxplanck (F: 210K)	Time(s)	15.405	15.342	8.312	4.914	1.891
	#window propagations	30,447,971	16,184,715	30,458,413	16,141,584	12,597,188
	Peak memory(MB)	4.31	887.99	4.31	887.99	3.405
Duck (F: 219K)	Time(s)	22.84	21.496	11.411	6.463	2.53
	#window propagations	42,760,744	21,077,986	42,790,802	21,728,282	16,821,923
	Peak memory(MB)	5.25	1160.22	5.25	1160.25	3.743
Tooth (F: 220K)	Time(s)	19.403	21.129	9.638	5.898	2.339
	#window propagations	33,512,612	18,564,126	33,502,205	18,518,680	14,458,399
	Peak memory(MB)	6.15	1037.37	6.12	1037.37	4.833
Moai (F: 238K)	Time(s)	20.632	21.707	10.616	6.495	2.483
	#window propagations	36,344,587	19,904,387	36,363,511	19,849,151	15,085,009
	Peak memory(MB)	6.10	1076.09	6.09	1076.09	4.631
DancingChildren (F: 265K)	Time(s)	9.846	9.634	5.444	3.426	1.707
	#window propagations	17,977,851	10,463,582	18,047,429	10,403,108	8,501,699
	Peak memory(MB)	4.21	566.13	4.26	566.14	3.569
Igea (F: 268K)	Time(s)	15.211	14.232	8.532	4.586	1.995
	#window propagations	32,561,369	17,137,879	32,522,412	17,066,447	12,673,530
	Peak memory(MB)	3.11	890.91	3.10	891.01	2.218
Cup (F: 316K)	Time(s)	46.262	50.203	20.175	11.499	5.465
	#window propagations	68,373,922	35,226,751	68,454,171	35,150,391	32,580,614
	Peak memory(MB)	11.05	1880.35	11.04	1880.43	9.835
Armadillo (F: 345K)	Time(s)	8.878	7.858	5.196	2.976	1.628
	#window propagations	19,132,785	10,298,238	19,193,615	10,215,363	8,084,456
	Peak memory(MB)	2.00	538.32	2.03	538.32	1.458
Vase (F: 354K)	Time(s)	35.388	42.203	19.614	10.912	4.056
	#window propagations	60,147,611	33,770,683	60,908,787	33,733,987	26,004,542
	Peak memory(MB)	7.00	1860.04	7.01	1860.04	5.743
Red Circular Box (F: 360K)	Time(s)	7.812	7.025	5.375	2.83	1.763
	#window propagations	16,771,179	9,368,831	16,875,650	9,267,397	7,781,762
	Peak memory(MB)	1.89	479.66	1.92	479.66	1.554
Julius Caesar (F: 386K)	Time(s)	16.331	13.234	9.963	4.861	2.372
	#window propagations	34,460,870	17,112,887	34,602,443	17,006,880	12,744,572
	Peak memory(MB)	2.46	877.05	2.49	877.05	2.153
Pulley (F: 392K)	Time(s)	29.392	33.032	15.508	9.497	4.242
	#window propagations	48,803,681	28,566,263	48,880,682	28,455,748	21,989,818
	Peak memory(MB)	6.03	1537.16	6.05	1537.18	5.030
Eros (F: 394K)	Time(s)	15.341	12.538	9.259	4.567	2.623
	#window propagations	28,320,100	14,437,016	28,452,631	14,322,675	11,856,262
	Peak memory(MB)	3.56	752.46	3.61	752.46	2.602
Frog (F: 394K)	Time(s)	29.975	26.453	15.663	8.305	3.175
	#window propagations	55,709,004	29,663,952	55,703,101	29,603,551	20,134,270
	Peak memory(MB)	5.08	1393.92	5.08	1393.93	3.610

Continue from previous table

Model	Performance	Algorithms				
		ICH	MMP	FWP-CH	FWP-MMP	VTP
Magalie'sHand (F: 396K)	Time(s)	15.725	13.059	9.658	4.537	2.387
	#window propagations	28,775,730	14,885,683	29,864,392	14,824,480	11,324,832
	Peak memory(MB)	4.08	764.60	4.23	764.63	2.895
WoodenChair (F: 408K)	Time(s)	28.429	29.839	14.405	8.977	4.071
	#window propagations	50,355,149	29,553,762	50,435,328	29,450,039	21,937,266
	Peak memory(MB)	5.65	1541.40	5.67	1541.42	4.935
Angel (F: 474K)	Time(s)	16.42	17.172	9.695	6.366	2.877
	#window propagations	35,750,531	21,046,492	36,277,286	21,441,579	15,858,241
	Peak memory(MB)	2.81	1105.82	2.82	1105.82	2.272
Rocker Arm (F: 482K)	Time(s)	36.577	33.286	19.536	11.867	4.13
	#window propagations	68,553,846	33,989,638	70,513,186	35,940,386	25,654,638
	Peak memory(MB)	5.29	1797.16	5.42	1797.19	3.70
Fertility (F: 483K)	Time(s)	34.202	30.594	18.306	9.576	4.133
	#window propagations	60,924,913	31,420,922	60,920,074	31,321,993	24,686,942
	Peak memory(MB)	5.70	1588.50	5.68	1588.84	4.465
Heptoroid (F: 573K)	Time(s)	42.112	71.431	21.288	17.357	5.556
	#window propagations	66,227,523	48,580,064	67,386,876	49,942,806	34,287,452
	Peak memory(MB)	5.78	2520.55	5.92	2520.60	5.284
Pierrot (F: 887K)	Time(s)	102.416	88.012	45.395	24.695	9.136
	#window propagations	150,271,829	73,707,582	150,295,550	73,740,134	51,644,860
	Peak memory(MB)	9.17	3569.21	9.21	3569.28	5.614
Bozbezbozzel (F: 911K)	Time(s)	57.35	64.402	28.083	18.558	7.005
	#window propagations	90,769,891	55,949,635	91,001,989	55,767,105	38,660,645
	Peak memory(MB)	7.85	2731.02	7.88	2731.02	6.229
Chinese dragon (F: 1,222K)	Time(s)	126.488	160.468	53.518	39.839	16.435
	#window propagations	167,453,594	99,261,429	167,627,646	98,941,538	72,865,547
	Peak memory(MB)	11.75	5284.42	11.73	5284.43	9.918
Asian Dragon (F: 1,400K)	Time(s)	73.204	73.092	35.637	23.674	9.495
	#window propagations	107,742,094	62,161,583	108,122,218	62,025,717	48,217,896
	Peak memory(MB)	5.184	3354.04	5.207	3354.05	4.373
Ramesses (F: 1,653K)	Time(s)	42.246	34.253	26.193	14.708	8.938
	#window propagations	77,345,899	38,983,510	78,805,085	40,122,773	34,128,397
	Peak memory(MB)	3.59	2014.17	3.73	2014.17	2.948
Pensatore (F: 1,996K)	Time(s)	189.217	158.819	81.927	49.049	20.328
	#window propagations	252,500,639	129,966,322	253,345,289	129,968,435	88,716,654
	Peak memory(MB)	9.56	6051.50	9.67	6051.53	5.797
Seahorse (F: 2,014K)	Time(s)	200.439	203.342	97.221	58.201	21.869
	#window propagations	322,605,156	172,324,693	322,460,340	171,858,409	130,860,212
	Peak memory(MB)	7.55	9294.98	7.53	9295.03	5.767
IsidoreHorse (F: 2,209K)	Time(s)	63.183	49.591	43.972	23.295	11.28
	#window propagations	103,331,867	52,406,687	110,210,607	59,846,982	42,350,025
	Peak memory(MB)	4.48	2725.09	4.74	2725.09	3.062

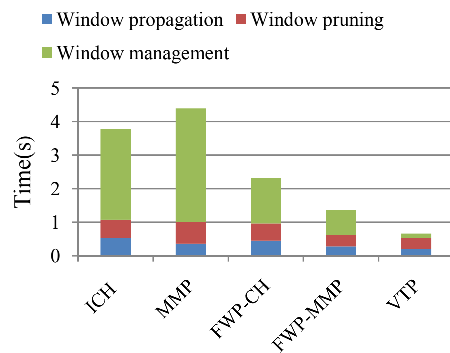
Continue from previous table

Model	Performance	Algorithms				
		ICH	MMP	FWP-CH	FWP-MMP	VTP
Happy Buddha (F: 2,583K)	Time(s)	320.813	386.681	135.018	99.984	33.373
	#window propagations	394,485,853	215,162,089	410,662,438	230,681,707	166,942,683
	Peak memory(MB)	12.25	11408.39	12.88	11410.54	9.337
Cervino Terrain (F: 3,146K)	Time(s)	179.285	117.187	92.968	46.331	28.287
	#window propagations	267,838,521	107,191,823	273,131,668	112,447,283	135,012,449
	Peak memory(MB)	6.94	5627.32	7.04	5630.01	5.85
Neptune (F: 4,008K)	Time(s)	455.271	424.331	193.945	120.012	47.629
	#window propagations	585,784,159	270,930,198	602,587,831	284,581,696	246,364,008
	Peak memory(MB)	16.96	14225.26	17.14	14219.76	16.38
VaseLion (F: 6,370K)	Time(s)	2012.72	Out of memory	604.662	Out of memory	145.455
	#window propagations	1,721,300,347		1,729,607,080		704,638,382
	Peak memory(MB)	52.17		52.24		39.257
Lucy (F: 14,464K)	Time(s)	8894.87	Out of memory	2415.88	Out of memory	549.934
	#window propagations	6,837,670,602		6,841,729,337		2,808,823,718
	Peak memory(MB)	78.29		78.28		69.42

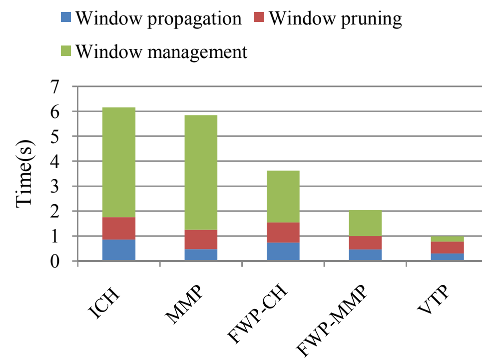
Appendix F

VTP Performance Profiling

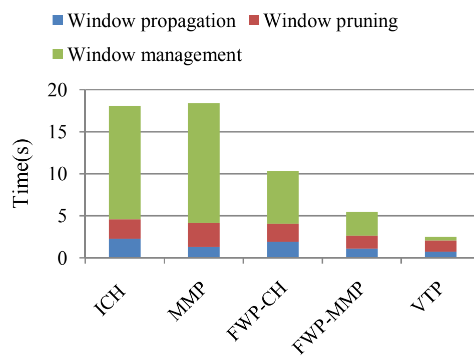
To profile the running times of the three individual components (window propagation, window pruning and window management) in state-of-the-art algorithms and the proposed VTP algorithm, in addition to the two models (Armadillo and Asian Dragon) used in the thesis, this part also show the results on 8 other models with various resolutions.



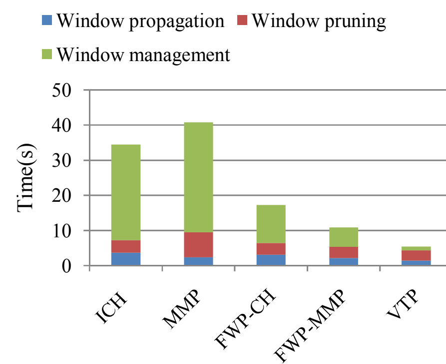
Horse (F: 96K)



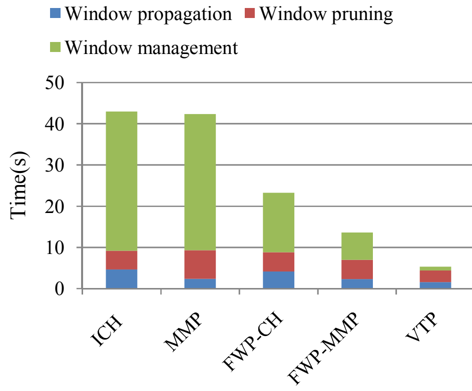
Bunny (F: 144K)



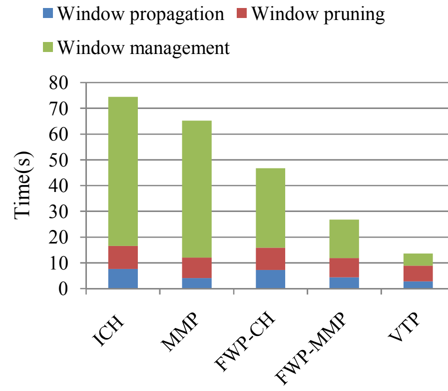
Igea (F: 268K)



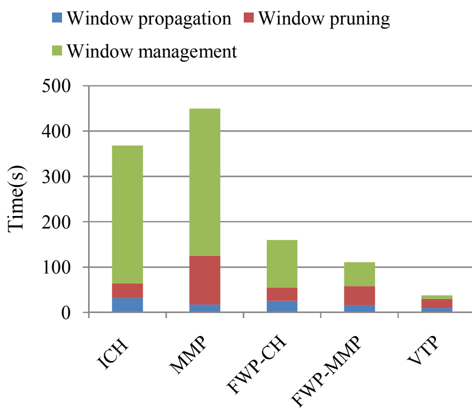
Pulley (F: 392K)



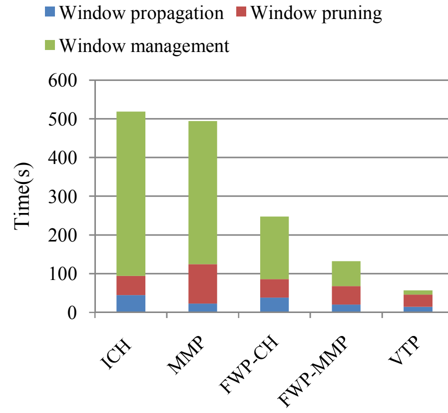
Rocker Arm (F: 482K)



IsidoreHorse (F: 2M)



Happy Buddha (F: 2.6M)



Neptune (F: 4M)

Appendix G

Longest Length in a Triangle

Lemma G.1. *Given a triangle whose three edges' lengths are a, b and c respectively. Let l be the length of a line segment whose endpoints are on the triangle's edges. Then, $l \leq \max(a, b, c)$.*

Proof. As Figure G.1 shows, assume G is the line segment GH's endpoint on edge DE, $\|DF\| = a$, $\|EF\| = b$, $\|DE\| = c$, $\|DG\| = \lambda c$ that $0 \leq \lambda \leq 1$. Construct a circle whose centre is G and radius $r = \max(a, b, c)$. Lemma G.1 is proved via showing that ΔDEF is in the circle so that the length l of any line segment in ΔDEF starting from G is less or equal to r. This is equivalent to proving that $\|DG\| \leq r = \max(a, b, c)$, $\|EG\| \leq \max(a, b, c)$ and $\|GF\| \leq \max(a, b, c)$.

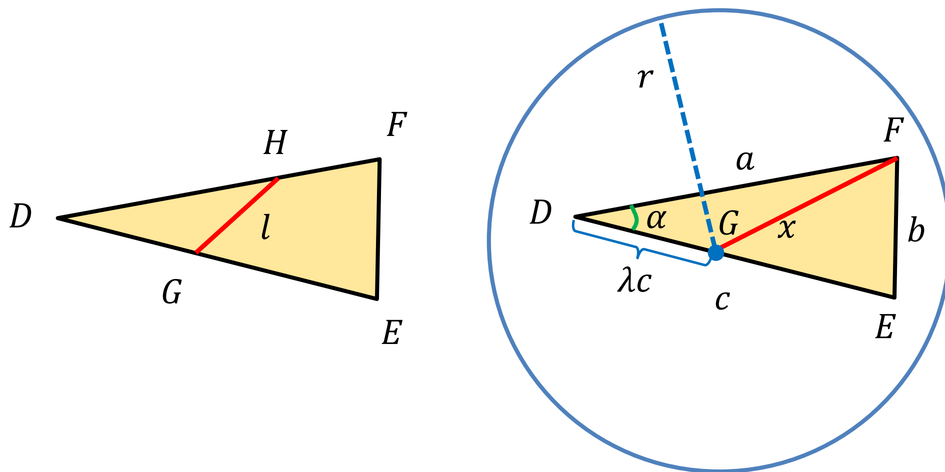


FIGURE G.1: Illustration of Lemma G.1.

Obviously, $\|DG\| \leq \max(a, b, c)$ and $\|EG\| \leq \max(a, b, c)$ since they are parts of edge DE that $\|DE\| = c \leq \max(a, b, c)$. Let $x = \|GF\|$, $\alpha = \angle FDE$. Then, $\|GF\| \leq \max(a, b, c)$ is proved as follows.

From the *Law of cosines*,

$$\cos\alpha = \frac{a^2 + c^2 - b^2}{2ac} = \frac{a^2 + (\lambda c)^2 - x^2}{2a(\lambda c)}$$

Therefore,

$$\begin{aligned} x^2 &= a^2 + (\lambda c)^2 - 2a(\lambda c)\cos\alpha \\ &= a^2 + (\lambda c)^2 - \lambda(a^2 + c^2 - b^2) \\ &= (1-\lambda)a^2 + \lambda b^2 + \lambda c^2(\lambda - 1) \end{aligned}$$

Then, according to which is the maximum among a, b and c, the proof of $\|GF\| \leq \max(a, b, c)$ is divided into three cases as follows:

(1) If $\max(a, b, c) = a$,

$$\begin{aligned} x^2 - a^2 &= -\lambda a^2 + \lambda b^2 + \lambda c^2(\lambda - 1) \\ &= \lambda(b^2 - a^2) + \lambda c^2(\lambda - 1) \end{aligned}$$

Since $\lambda \geq 0$, $\lambda - 1 \leq 0$ and $b^2 - a^2 \leq 0$, it can be derived that $x^2 - a^2 \leq 0$.

(2) If $\max(a, b, c) = b$,

$$\begin{aligned} x^2 - b^2 &= (1-\lambda)a^2 + (\lambda - 1)b^2 + \lambda c^2(\lambda - 1) \\ &= (\lambda - 1)(b^2 - a^2) + \lambda c^2(\lambda - 1) \end{aligned}$$

Since $\lambda \geq 0$, $\lambda - 1 \leq 0$ and $b^2 - a^2 \geq 0$, it can be derived that $x^2 - b^2 \leq 0$.

(3) If $\max(a, b, c) = c$,

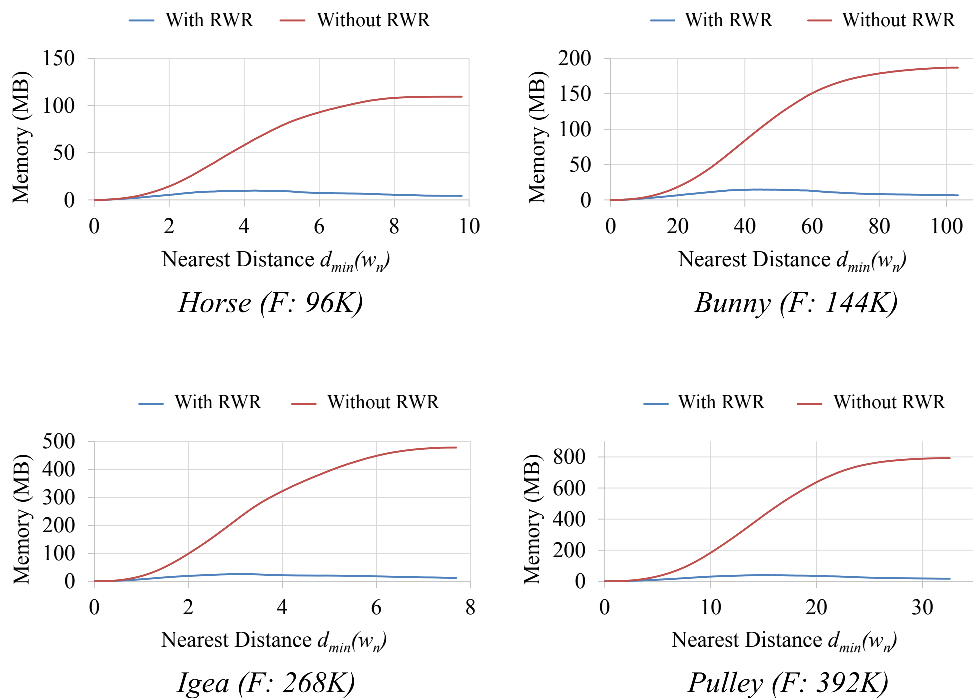
$$\begin{aligned} x^2 - c^2 &= (1-\lambda)a^2 + \lambda b^2 + \lambda c^2(\lambda - 1) - c^2 \\ &\leq \max(a^2, b^2) \cdot (1 - \lambda + \lambda) + \lambda c^2(\lambda - 1) - c^2 \\ &= \max(a^2, b^2) - c^2 + \lambda c^2(\lambda - 1) \leq 0 \end{aligned}$$

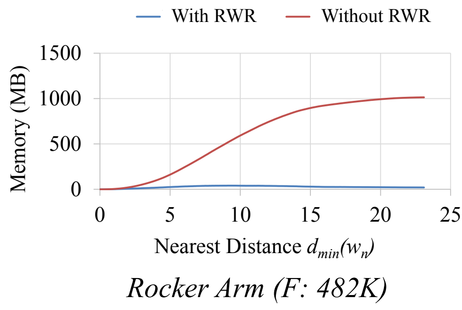
Since $\lambda \geq 0$, $\lambda - 1 \leq 0$ and $\max(a^2, b^2) - c^2 \leq 0$, it can be derived that $x^2 - c^2 \leq 0$. $\square$

Appendix H

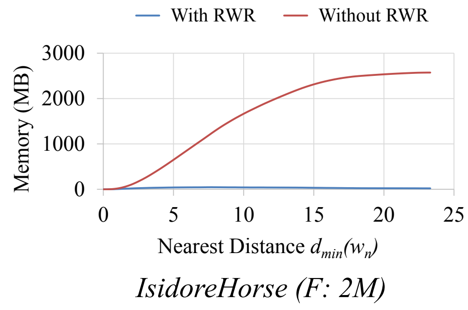
RWR Performance Verification

To verify that the RWR procedure proposed in the thesis effectively reduces memory cost, this section compares memory costs against nearest distance $d_{\min}(w_n)$ of the wavefront between two variants of the Voronoi diagram construction: with and without RWR. In this part, the results on 8 other models with various resolutions are shown in addition to the two models (Armadillo and Asian Dragon) used in the thesis.

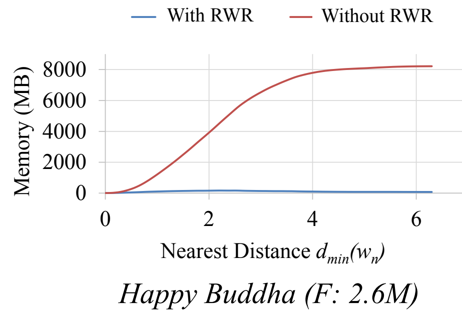




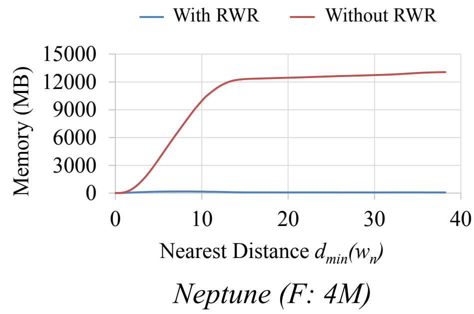
Rocker Arm (F: 482K)



IsidoreHorse (F: 2M)



Happy Buddha (F: 2.6M)



Neptune (F: 4M)

Appendix I

Performance Comparison among VD-DGP Algorithms

This part compares the performance among state-of-the-art Voronoi diagram oriented geodesic algorithms, MMP (Surazhsky et al., 2005), FWP-MMP (Xu et al., 2015) and the proposed *window*-VTP on the model set proposed in Appendix A.

Model	Performance	Algorithms		
		MMP	**FWP-MMP**	***window*-VTP**
Twirl (F: 10K)	Time(s)	0.073	0.032	0.031
	#windows stored	71,918	71,918	10,835
	Peak memory(MB)	5.49	5.49	1.77
Sword (F: 29K)	Time(s)	0.374	0.164	0.147
	#windows stored	342,133	342,332	33,806
	Peak memory(MB)	26.10	26.12	4.94
Femur (F: 30K)	Time(s)	0.292	0.124	0.119
	#windows stored	263,839	263,839	22,463
	Peak memory(MB)	20.13	20.13	3.36
Cow (F: 36K)	Time(s)	0.408	0.16	0.14
	#windows stored	360,289	360,289	27,630
	Peak memory(MB)	27.49	27.49	3.41
Venus (F: 43K)	Time(s)	0.57	0.267	0.22
	#windows stored	518,629	518,629	35,210
	Peak memory(MB)	39.57	39.57	4.52
Foot (F: 44K)	Time(s)	0.581	0.235	0.226
	#windows stored	495,620	495,624	32,785
	Peak memory(MB)	37.81	37.81	5.45
Camel (F: 48K)	Time(s)	0.626	0.303	0.28
	#windows stored	533,141	533,141	28,674
	Peak memory(MB)	40.68	40.68	4.34
HomerSimpson (F: 48K)	Time(s)	0.651	0.256	0.241
	#windows stored	533,641	533,649	39,277
	Peak memory(MB)	40.71	40.71	5.74

Continue from previous table

Model	Performance	Algorithms		
		MMP	**FWP-MMP**	***window*-VTP**
Dilo (F: 54K)	Time(s)	0.789	0.355	0.294
	#windows stored	684,702	684,720	37,512
	Peak memory(MB)	52.24	52.25	5.04
SketchedVase (F: 54K)	Time(s)	1.234	0.42	0.392
	#windows stored	945,438	945,438	80,584
	Peak memory(MB)	72.13	72.13	10.85
Knot (F: 56K)	Time(s)	1.073	0.345	0.328
	#windows stored	738,428	738,428	41,106
	Peak memory(MB)	56.34	56.34	6.73
Buste (F: 60K)	Time(s)	0.957	0.369	0.342
	#windows stored	654,859	654,859	40,265
	Peak memory(MB)	49.96	49.96	6.84
Casting (F: 90K)	Time(s)	1.566	0.66	0.609
	#windows stored	1,121,043	1,121,043	53,522
	Peak memory(MB)	85.53	85.53	9.68
Horse (F: 96K)	Time(s)	1.966	0.685	0.66
	#windows stored	1,433,970	1,433,990	59,009
	Peak memory(MB)	109.40	109.41	9.98
Shark (F: 107K)	Time(s)	2.345	0.868	0.817
	#windows stored	1,748,695	1,748,791	74,777
	Peak memory(MB)	133.42	133.42	11.82
Pegasus (F: 127K)	Time(s)	2.749	0.977	0.932
	#windows stored	1,849,764	1,849,768	67,633
	Peak memory(MB)	141.13	141.13	12.97
Bunny (F: 144K)	Time(s)	3.637	1.27	1.07
	#windows stored	2,451,104	2,451,105	85,959
	Peak memory(MB)	187.00	187.00	14.86
Bimba (F: 149K)	Time(s)	3.576	1.358	1.218
	#windows stored	2,461,523	2,461,523	83,382
	Peak memory(MB)	187.80	187.80	13.40
Hand (F: 176K)	Time(s)	7.345	2.296	2.076
	#windows stored	4,517,908	4,517,982	175,080
	Peak memory(MB)	344.69	344.70	22.73
Filigree (F: 186K)	Time(s)	3.518	1.36	1.284
	#windows stored	2,363,812	2,363,913	84,379
	Peak memory(MB)	180.35	180.35	17.05
Woodfish (F: 191K)	Time(s)	6.458	2.028	1.821
	#windows stored	3,875,027	3,875,066	112,070
	Peak memory(MB)	295.64	295.64	19.47
Maxplanck (F: 210K)	Time(s)	7.105	2.346	2.121
	#windows stored	4,384,798	4,384,805	129,240
	Peak memory(MB)	334.53	334.53	20.64

Continue from previous table

Model	Performance	Algorithms		
		MMP	**FWP-MMP**	***window*-VTP**
Duck (F: 219K)	Time(s)	7.969	2.615	2.406
	#windows stored	4,739,873	4,740,044	136,862
	Peak memory(MB)	364.62	361.64	22.47
Tooth (F: 220K)	Time(s)	8.479	2.709	2.365
	#windows stored	4,717,835	4,717,834	137,556
	Peak memory(MB)	359.94	359.94	22.66
Moai (F: 238K)	Time(s)	9.165	2.963	2.603
	#windows stored	5,171,473	5,171,567	137,861
	Peak memory(MB)	394.55	394.56	25.11
Dancing Children (F: 265K)	Time(s)	8.526	2.79	2.501
	#windows stored	4,891,962	4,891,983	130,293
	Peak memory(MB)	373.23	373.23	22.04
Igea (F: 268K)	Time(s)	10.916	3.362	3.019
	#windows stored	6,266,009	6,266,232	161,435
	Peak memory(MB)	478.06	478.08	26.50
Cup (F: 316K)	Time(s)	15.955	4.317	3.827
	#windows stored	7,130,333	7,130,438	172,881
	Peak memory(MB)	544.00	544.01	32.86
Armadillo (F: 345K)	Time(s)	9.863	3.304	2.982
	#windows stored	5,771,551	5,771,552	109,213
	Peak memory(MB)	440.33	440.33	21.09
Vase (F: 354K)	Time(s)	17.61	5.07	4.6
	#windows stored	9,000,357	9,000,338	204,435
	Peak memory(MB)	686.67	686.67	28.35
Red circular box (F: 360K)	Time(s)	8.67	3.053	2.763
	#windows stored	5,058,283	5,058,359	95,243
	Peak memory(MB)	385.92	385.92	16.65
Julius Caesar (F: 386K)	Time(s)	17.102	5.198	4.435
	#windows stored	8,302,679	8,302,680	163,670
	Peak memory(MB)	633.44	633.44	29.08
Pulley (F: 392K)	Time(s)	23.917	6.622	5.345
	#windows stored	10,381,917	10,382,287	218,368
	Peak memory(MB)	792.08	792.11	39.69
Eros (F: 394K)	Time(s)	15.908	4.892	4.393
	#windows stored	7,977,843	7,977,925	146,482
	Peak memory(MB)	608.66	608.67	27.93
Frog (F: 394K)	Time(s)	25.859	6.93	5.366
	#windows stored	10,599,549	10,599,799	264,306
	Peak memory(MB)	808.68	808.70	46.46
Magalie's hand (F: 396K)	Time(s)	16.947	4.923	4.237
	#windows stored	7,828,886	7,828,888	137,411
	Peak memory(MB)	597.30	597.30	32.20

Continue from previous table

Model	Performance	Algorithms		
		MMP	**FWP-MMP**	***window*-VTP**
WoodenChair (F: 408K)	Time(s)	20.081	5.995	5.232
	#windows stored	10,611,023	10,611,108	166,678
	Peak memory(MB)	809.56	809.56	29.84
Angel (F: 474K)	Time(s)	21.463	6.764	5.67
	#windows stored	10,797,339	10,797,405	191,428
	Peak memory(MB)	823.77	823.78	29.56
Rocker arm (F: 482K)	Time(s)	32.012	9.088	6.985
	#windows stored	13,282,080	13,282,139	271,040
	Peak memory(MB)	1013.34	1013.35	41.50
Fertility (F: 483K)	Time(s)	35.412	9.235	6.986
	#windows stored	14,209,376	14,213,322	343,115
	Peak memory(MB)	1084.09	1084.39	49.44
Heptoroid (F: 573K)	Time(s)	88.438	16.295	11.854
	#windows stored	25,684,650	25,686,036	680,891
	Peak memory(MB)	1959.58	1959.69	110.13
Pierrot (F: 887K)	Time(s)	102.134	22.831	15.986
	#windows stored	30,719,649	30,720,244	471,137
	Peak memory(MB)	2343.72	2343.77	82.37
Bozbezbozzel (F: 911K)	Time(s)	67.579	16.903	13.444
	#windows stored	25,725,566	25,725,665	297,927
	Peak memory(MB)	1962.70	1962.71	54.70
Chinese dragon (F: 1,222K)	Time(s)	137.814	32.257	22.836
	#windows stored	42,905,252	42,905,259	461,210
	Peak memory(MB)	3273.41	3273.41	89.39
Ramesses (F: 1,653K)	Time(s)	55.345	21.233	15.4
	#windows stored	22,398,278	22,398,569	232,074
	Peak memory(MB)	1708.85	1708.88	36.93
Asian dragon (F: 1,400K)	Time(s)	110.083	28.247	20.281
	#windows stored	36,317,620	36,317,847	346,142
	Peak memory(MB)	2770.81	2770.83	76.75
Pensatore (F: 1,996K)	Time(s)	258.372	60.713	37.665
	#windows stored	67,448,762	67,448,895	592,650
	Peak memory(MB)	5145.93	5145.94	111.73
Seahorse (F: 2,014K)	Time(s)	366.545	72.9	47.885
	#windows stored	90,781,860	90,782,014	716,661
	Peak memory(MB)	6926.11	6926.12	145.24
IsidoreHorse (F: 2,209K)	Time(s)	89.538	31.107	21.299
	#windows stored	33,738,770	33,738,791	283,255
	Peak memory(MB)	2574.06	2574.07	46.79
Happy buddha (F: 2,583K)	Time(s)	482.715	105.009	58.946
	#windows stored	107,722,874	107,732,048	912,637
	Peak memory(MB)	8218.60	8219.30	161.98

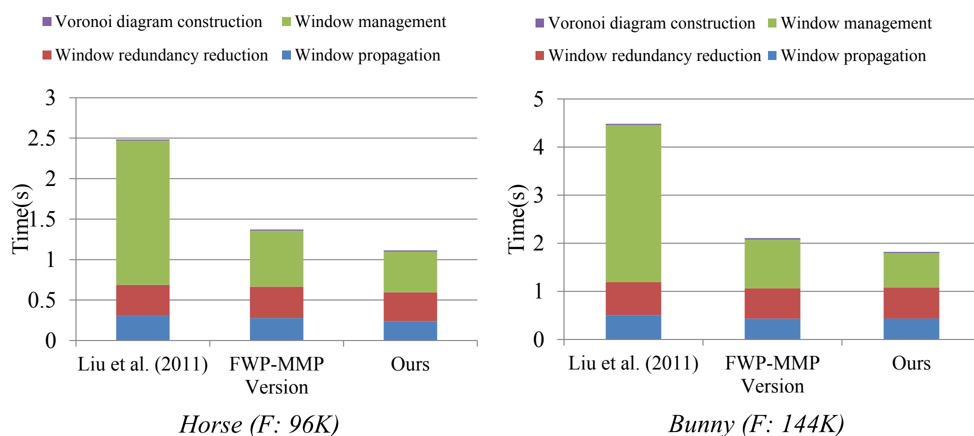
Continue from previous table

Model	Performance	Algorithms		
		MMP	**FWP-MMP**	***window*-VTP**
Cervino Terrain (F: 3,146K)	Time(s)	259.395	66.332	45.838
	#windows stored	72,176,116	72,195,898	575,108
	Peak memory(MB)	5506.60	5508.11	104.44
Neptune (F: 4,008K)	Time(s)	832.83	173.055	96.843
	#windows stored	171,319,703	171,374,203	857,068
	Peak memory(MB)	13070.70	13074.80	176.30
Vase lion (F: 6,370K)	Time(s)	Out of Memory	Out of Memory	238.298
	#windows stored			2,299,918
	Peak memory(MB)			333.65
Lucy (F: 14,464K)	Time(s)	Out of Memory	Out of Memory	806.118
	#windows stored			12,071,796
	Peak memory(MB)			921.005

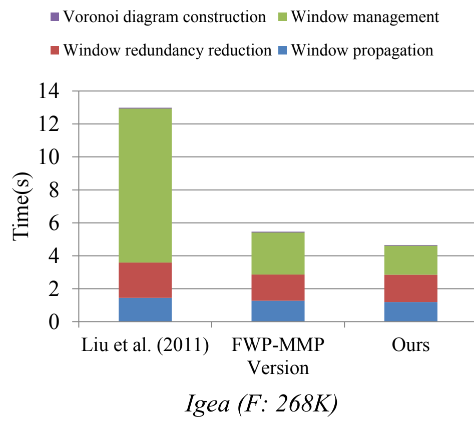
Appendix J

Voronoi Diagram Construction Performance Profiling

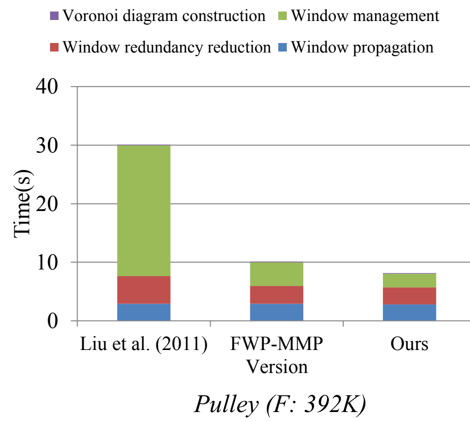
This section profiles the running times of the four individual components (Voronoi diagram construction, window propagation, window redundancy reduction and window management) in the Voronoi diagram construction. The comparison is performed on three versions of the solution, (1) the original method in (Liu et al., 2011); (2) the FWP-MMP version which replaces the MMP algorithm used by (Liu et al., 2011) with the FWP-MMP algorithm (Xu et al., 2015); (3) The proposed version which replaces the MMP algorithm used by (Liu et al., 2011) with the proposed *window*-VTP algorithm. In addition to the two models (Armadillo and Asian Dragon) used in the thesis, the results on 8 other models with various resolutions are also shown in this part.



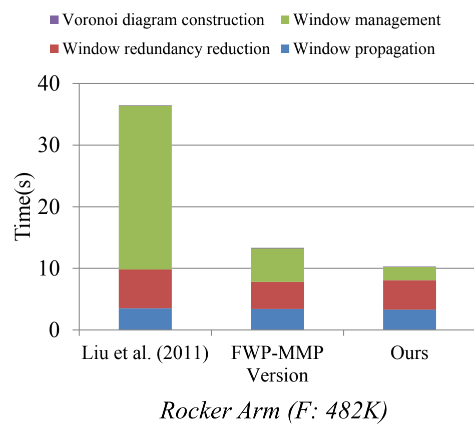
Horse (F: 96K) *Bunny (F: 144K)*



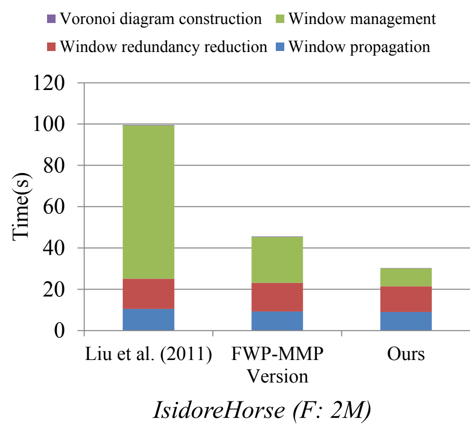
Igea (F: 268K)



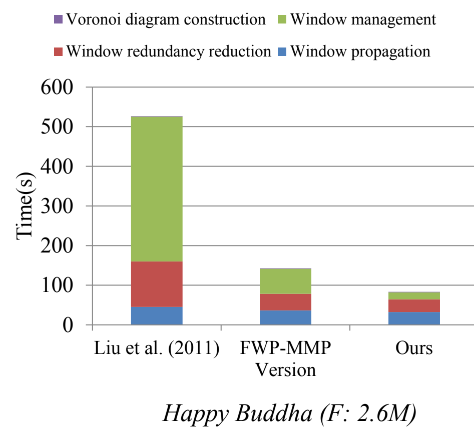
Pulley (F: 392K)



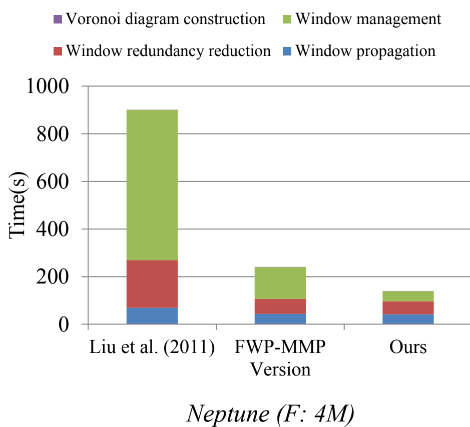
Rocker Arm (F: 482K)



IsidoreHorse (F: 2M)



Happy Buddha (F: 2.6M)



Neptune (F: 4M)