

# Fast and Exact Discrete Geodesic Computation Based on Triangle-Oriented Wavefront Propagation

Yipeng Qin<sup>1,\*</sup> Xiaoguang Han<sup>2,\*</sup> Hongchuan Yu<sup>1†</sup> Yizhou Yu<sup>2</sup> Jianjun Zhang<sup>1</sup><sup>1</sup>Bournemouth University<sup>2</sup>The University of Hong Kong

## Abstract

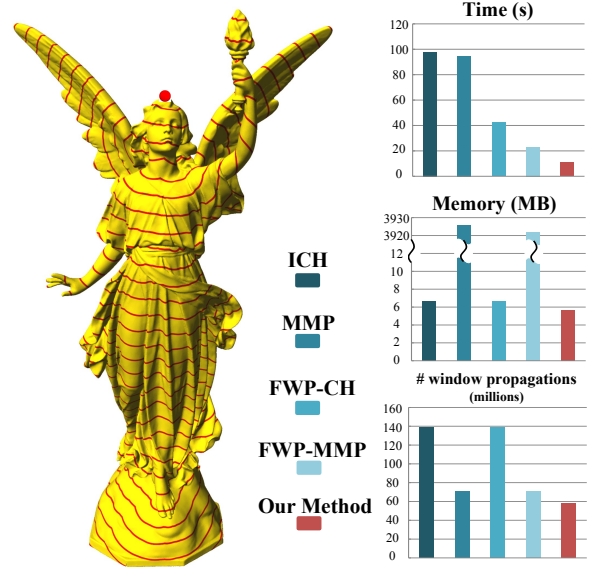
Computing discrete geodesic distance over triangle meshes is one of the fundamental problems in computational geometry and computer graphics. In this problem, an effective window pruning strategy can significantly affect the actual running time. Due to its importance, we conduct an in-depth study of window pruning operations in this paper, and produce an exhaustive list of scenarios where one window can make another window partially or completely redundant. To identify a maximal number of redundant windows using such pairwise cross checking, we propose a set of procedures to synchronize local window propagation within the same triangle by simultaneously propagating a collection of windows from one triangle edge to its two opposite edges. On the basis of such synchronized window propagation, we design a new geodesic computation algorithm based on a triangle-oriented region growing scheme. Our geodesic algorithm can remove most of the redundant windows at the earliest possible stage, thus significantly reducing computational cost and memory usage at later stages. In addition, by adopting triangles instead of windows as the primitive in propagation management, our algorithm significantly cuts down the data management overhead. As a result, it runs 4-15 times faster than MMP and ICH algorithms, 2-4 times faster than FWP-MMP and FWP-CH algorithms, and also incurs the least memory usage.

**Keywords:** Discrete Geodesics, Pairwise Window Pruning, Order Preservation, Window List Propagation, Wavefront Propagation

**Concepts:** •Theory of computation → Computational geometry; •Computing methodologies → Shape modeling;

## 1 Introduction

Computing geodesics is one of the fundamental problems in computational geometry and computer graphics since it plays an important role in many applications, including remeshing [Peyré and Cohen 2006], shape analysis [Bronstein et al. 2006] and non-rigid shape retrieval [Ye et al. 2013; Ye and Yu 2016]. In this paper, we focus on the “One Source All destinations” discrete geodesic problem over triangle meshes. As proposed in [Mitchell et al. 1987], this problem can be solved by performing window propagation where a “window” encodes the information of a cone originating from the source vertex. The geodesic distance field over a mesh surface is computed by propagating windows and updating the distance from



**Figure 1:** Our exact geodesic algorithm (VTP) outperforms all recent discrete geodesic algorithms (ICH, MMP, FWP-ICH, FWP-MMP) in terms of running time, peak memory usage and total number of window propagations on the Lucy model (1.6M faces, courtesy of Suggestive Contour Gallery at Princeton University).

the source vertex to all other vertices progressively. Note that window propagation creates a large number of new windows, many of which are redundant and do not contribute to the shortest path for any vertex. Without any pruning operations, these redundant windows significantly slow down the overall computation.

In fact, an effective window pruning strategy can be more important to geodesic computation than the asymptotic time complexity. For example, the Mitchell-Mount-Papadimitriou (MMP) algorithm [Mitchell et al. 1987] takes  $O(n^2 \log n)$  time while the algorithm proposed by Chen and Han (CH) [Chen and Han 1990] achieves a better asymptotic time complexity, which is  $O(n^2)$ . Interestingly, algorithms in the MMP family often run faster than those in the CH family in practice. An important reason is that MMP algorithms can prune many more redundant windows.

Nevertheless, algorithms in the CH family have important advantages over MMP algorithms. First, CH algorithms avoid expensive window trimming using hyperbolic curves and adopt more efficient window filtering rules. Second, CH algorithms do not bookkeep windows at every visited edge as MMP algorithms do, and thus are much more memory efficient. Therefore, we aim to fully realize the potential of CH algorithms by developing highly effective window pruning strategies as well as an overall geodesic computation algorithm based on such strategies.

Due to the aforementioned importance, we conduct an in-depth study of window pruning operations in this paper. If we focus on

\*Joint first authors

†Corresponding author: hyu@bournemouth.ac.uk

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or to publish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org. © 2016 ACM.

SIGGRAPH '16 Technical Paper., July 24-28, 2016, Anaheim, CA,

ISBN: 978-1-4503-4279-7/16/07

DOI: <http://dx.doi.org/10.1145/2897824.2925930>

a single window during one step of propagation, this window always moves from one triangle edge to another or two other edges. These hosting edges before and after the propagation always belong to the same triangle. A non-redundant window may become partially or completely redundant after this one-step propagation. Such a change of status is caused by another overlapping window passing through the same triangle. Due to this observation, the first outcome of our study is an exhaustive list of scenarios where one window can make another window partially or completely redundant after one step of propagation.

The above discussion indicates pairwise cross checking among windows passing through the same triangle is necessary in order to discover redundant windows. To identify a maximal number of redundant windows using cross checking, we need to propagate as many windows as possible in the same triangle at the same time. To make this happen, we propose to synchronize local window propagation within a triangle by simultaneously propagating a collection of windows from one triangle edge to its two opposite edges. Such a synchronized propagation consists of three substeps: i) split a collection of windows reaching the same triangle edge into two subsets; ii) propagate the two subsets to the two opposite edges respectively; iii) if a triangle edge receives more than one propagated subsets of windows from other edges of the same triangle, these propagated subsets are cross-checked and merged into a single set. Efficient algorithms are designed to perform all of these substeps.

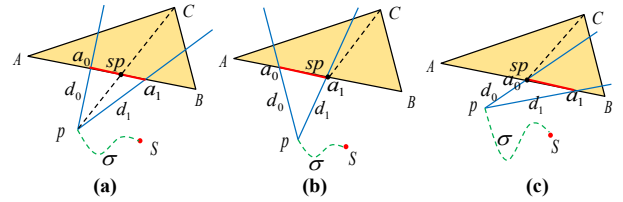
On the basis of the above triangle-oriented window propagation, we design a new geodesic computation algorithm based on a triangle-oriented region growing scheme. Starting from the source vertex, instead of considering one window at a time, our new algorithm considers one triangle at a time. All visited triangles form a single connected region, called the traversed area, over the mesh surface. Our algorithm expands this traversed area in a continuous Dijkstra style by gradually enclosing unvisited triangles abutting the traversed area and propagating windows accumulated at the previous wavefront of the traversed area through these newly added triangles. Our triangle-oriented wavefront propagation scheme can remove most of the redundant windows at the earliest possible stage, thus significantly reducing computational cost at later stages. In addition, by adopting triangles instead of windows as the primitive in propagation management, our algorithm significantly cuts down the data management overhead (e.g. priority queue maintenance cost) since the total number of windows is much larger than the number of triangles.

In summary, this paper has the following contributions.

- A new geodesic computation framework based on triangle-oriented window propagation. Our new geodesic algorithm has an  $O(n^2)$  time complexity. It runs 4-15 times faster than MMP [Surazhsky et al. 2005] and ICH [Xin and Wang 2009] algorithms, 2-4 times faster than 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 synchronized windows propagation within a triangle.

## 2 Background and Related Work

In this section, we give a brief introduction to the window-based solution to the “One source All destinations” geodesic distance problem. Refer to [Mitchell et al. 1987] for further details. We also discuss related work on this topic. Let the triangle mesh we pro-



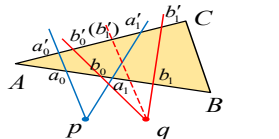
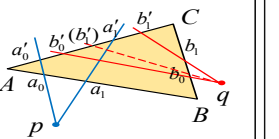
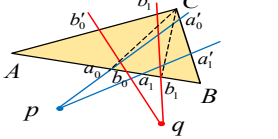
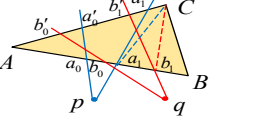
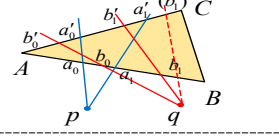
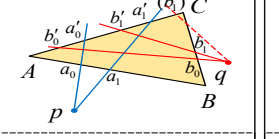
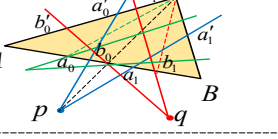
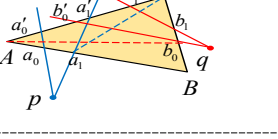
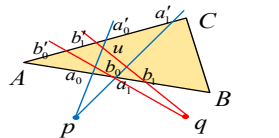
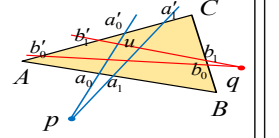
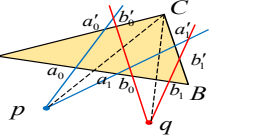
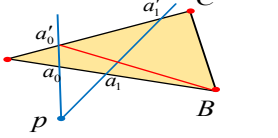
**Figure 2:** Three configurations of a window and its separating point.

cess be  $M = (V, E, F)$ , where  $V, E, F$  are respectively the sets of vertices, edges and faces. Let  $S$  be the source vertex.

**Window** In geodesic algorithms, a window is defined as  $w = (\Delta ABC, a_0, a_1, p, d_0, d_1, \sigma, sp)$ , where  $\Delta ABC$  stands for the triangle it enters from  $AB$  (this is equivalent to the half edge used in [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  $\Delta ABC$ , and  $d_0, d_1$  are the distances from  $a_0, a_1$  to  $p$  respectively.  $\sigma$  denotes the geodesic distance from the pseudo source to the source vertex  $S$ . Later in this paper, we will need a concept called the separating point ( $sp$ ), which 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  $pC$ . Fig 2 shows an illustration of a window and the three possible positions of a separating point.

**Window Propagation and Management** The geodesic distances from the source to all vertices are maintained using a vector  $D = (d_1, d_2, \dots, d_n) (n = |V|)$ . At the beginning, every opposite edge in the 1-ring neighborhood of  $S$  is defined as a window. Each window is iteratively propagated from edge to edge. The distance vector  $D$  is constantly updated during window propagation. Since the number of windows could be large, a common strategy pushes windows into a priority queue and processes windows in an increasing order of their distance to  $S$  [Mitchell et al. 1987; Surazhsky et al. 2005; Xin and Wang 2009]. This can prevent propagating windows from farther edges to closer ones. Recently, Xu et al. [2015] replaced the priority queue with bucket sorting, and their algorithms (FWP-MMP/CH) propagate multiple windows from the same bucket during each iteration. This strategy effectively reduces the time spent on window management. All these methods only propagate a single or multiple windows during each iteration while our algorithm propagates all windows accumulated on a single or multiple triangle edges simultaneously during each iteration. In addition, our algorithm organizes vertices instead of a much larger number of windows. These strategies prune redundant windows more thoroughly and further significantly reduce the cost of window management.

**Window Pruning** Many windows created during propagation do not contribute to the shortest path for any vertex. Window pruning reduces the cost of window propagation by stopping redundant windows from participating in propagation. The key idea is to detect redundant windows or redundant parts of windows according to certain rules and remove them. A window  $w$  becomes redundant if for every point  $q \in (w.a_0, w.a_1)$ , there exists one path on the

<b>Situation 1</b> From the same edge to another edge	<b>Situation 2</b> From two edges to the third edge	<b>Situation 3</b> From the same edge to two other edges	<b>Situation 4</b> Checking with vertices
<b>Case 1</b> <sup>†</sup> $a_0 < b_0$ & $a'_0 \leq b'_0 \leq a'_1$ & $qb'_0 \cap pa'_0 \notin \overline{pa_0}$ 	<b>Case 4</b> $a'_0 \leq b'_0 \leq a'_1$ 	<b>Case 7</b> both $w'_0$ and $w'_1$ cover one edge 	<b>Case 10</b> check two windows in <b>Situation 1</b> 
The following rules are shared by Case 1 and Case 4: 1) If $g(pb'_0) > g(qb'_0)$ , $\text{del}(pb'_0a'_1)$ . 2) If $b'_1 \leq a'_1$ & $g(pb'_1) > g(qb'_1)$ , $\text{del}(pb'_1a'_1)$ . 3) If $b'_1 \leq a'_1$ & $g(pb'_1) \leq g(qb'_1)$ , $\text{del}(w'_1)$ . 4) If $b'_1 > a'_1$ & $g(pa'_1) < g(qa'_1)$ , $\text{del}(qb'_1a'_1)$ .		When $a_0 < b_1$ , If $g(pa_0C) > g(qb_1C)$ , $\text{del}(w'_0)$ ; else, $\text{del}(w'_1)$ .	If $a_1 < b_1$ & $g(pa_1C) < g(qb_1C)$ , $\text{del}(w'_1)$ .
<b>Case 2</b> $a_0 < b_0$ and $b'_0 < a'_0 < b'_1$ 	<b>Case 5</b> $b'_0 < a'_0 < b'_1$ 	<b>Case 8</b> one of $w'_0$ and $w'_1$ covers two edges 	<b>Case 11</b> check two windows in <b>Situation 2</b> 
The following rules are shared by Case 2 and Case 5: 1) If $g(pa'_0) < g(qa'_0)$ , $\text{del}(qb'_0a'_0)$ . 2) If $g(pa'_0) \geq g(qa'_0)$ , $\text{del}(w'_0)$ . 3) If $b'_1 \leq a'_1$ & $g(pb'_1) \leq g(qb'_1)$ , $\text{del}(w'_1)$ . 4) If $b'_1 \leq a'_1$ & $g(pb'_1) > g(qb'_1)$ , $\text{del}(pb'_1a'_1)$ . 5) If $b'_1 > a'_1$ & $g(pa'_1) < g(qa'_1)$ , $\text{del}(qb'_0a'_1)$ .		1) For $w'_1 \in AC$ . When $w_0.sp < b_1$ , - If $g(pC) < g(qb_1C)$ , $\text{del}(w'_1)$ ; else, $\text{del}(pCa'_1)$ . - If $w_0.sp > b_1$ & $g(pC) > g(qb_1C)$ , $\text{del}(pa'_0C)$ . 2) For $w'_1 \in BC$ . When $w_0.sp > b_0$ , - If $g(pC) < g(qb_0C)$ , $\text{del}(w'_1)$ ; else, $\text{del}(pC)$ . - If $w_0.sp < b_0$ & $g(pC) > g(qb_0C)$ , $\text{del}(pCa'_1)$ .	1) If $g(pa_1C) < g(qb_1C)$ , $\text{del}(w'_1)$ . 2) If $g(qb_0A) < g(pa_0A)$ , $\text{del}(w'_0)$ .
<b>Case 3</b> $a_0 < b_0$ and $b'_1 \leq a'_0$ 	<b>Case 6</b> $b'_1 \leq a'_0$ 	<b>Case 9</b> both $w'_0$ and $w'_1$ cover two edges 	<b>Case 12</b> check one window 
The following rule is shared by Case 3 and Case 6: If $g(pu) < g(qu)$ , $\text{del}(w'_1)$ . else, $\text{del}(w'_0)$ . ( $u = qb'_1 \cap pa'_0$ ).		If $g(pC) > g(qC)$ , $\text{del}(pCa'_1)$ ; else, $\text{del}(pb'_0C)$ .	If $g(Aa'_1) < g(pa'_1)$ or $g(Ca'_0) < g(pa'_0)$ or $g(Ba'_0) < g(pa'_0)$ , $\text{del}(w'_0)$ .

**Figure 3:** Pairwise window cross checking and pruning within a triangle.  $w'_0$  and  $w'_1$  are the resulting windows after two original windows,  $w_0$  and  $w_1$ , are propagated from edge to edge inside a triangle. We 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  $\Delta ABC$ .  $g(qa)$  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. ‘del’ stands for the operator to delete the window or sub-window.

mesh from  $S$  to  $q$  whose length is smaller than  $w.\sigma + \text{dist}(w.p, q)$ , where  $\text{dist}()$  stands for the Euclidean distance between two points on a plane.

Classical MMP algorithms [Mitchell et al. 1987; Surazhsky et al. 2005]) only consider cases where a window  $w$  partially overlaps with another window  $w'$  and they share the same propagation direction. To determine which window gives rise to the shortest path to each point in  $w \cap w'$ , they solve a quadratic equation to find the tie point (see Section 3.3 of [Surazhsky et al. 2005]). Liu [2013] improved the algorithm in [Surazhsky et al. 2005] by further considering the cases where two windows overlap on the same edge but have different propagation directions. Binary search is used to find such overlapping windows on an edge, which causes the  $O(n^2 \log n)$  complexity.

If a window covers a vertex, the CH algorithm [Chen and Han 1990] trims the window using the ‘One Angle One Split’ rule. The ICH algorithm [Xin and Wang 2009] uses an additional ‘checking with vertices’ rule to filter out more redundant windows. These filtering rules avoid solving quadratic equations and deliver a performance comparable to that of MMP algorithms. A parallel version of the ICH algorithm has been developed in [Ying et al. 2014]. In this

paper, we propose a complete set of pairwise pruning rules that do not need to solve quadratic equations.

**Other Methods** For convex polyhedral surfaces, an  $O(n \log n)$ -time algorithm is presented in [Schreiber and Sharir 2009], which reaches the theoretical lower bound. Balasubramanian et al. [2009] proposed a method to compute the exact minimal geodesic distance between every pair of vertices on a polyhedral surface.

In addition to the aforementioned exact geodesic algorithms, there are approximation algorithms aimed at faster geodesic computation. Refer to [Bose et al. 2011] for a detailed survey. Surazhsky et al. [2005] presented an approximate MMP algorithm with bounded errors and the optimal  $O(n \log n)$  time complexity. Xin et al. [2012] proposed a two-phase algorithm which first computes

<sup>†</sup>In Situation 1 Case 1, we do not perform cross checking when  $a_0 < b_0$ ,  $a'_0 \leq b'_0 \leq a'_1$  and  $qb'_0 \cap pa'_0 \in \overline{pa_0}$  as this special case rarely happens in our 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.



distance fields by taking sample points as sources. On the basis of “geodesic unfolding”, the approximate geodesic distance between any pair of points can be obtained in constant time. The Saddle Vertex Graph (SVG), presented in [Ying et al. 2013], encodes the geodesic information on triangle meshes in a sparse graph. After precomputing an approximate SVG based on window propagation, this method can handle both “One source All destinations” and “One source One destination” in an efficient manner.

There are also methods based on finite element approximation. Kimmel and Sethian [1998] extended the Fast Marching Method (FMM) [Sethian 1996] to triangulated domains to compute geodesic distances using a discrete version of the Eikonal equation. Yatziv et al. [2005] accelerated their method from  $O(n \log n)$  to  $O(n)$  by using an untidy priority queue. This FMM method has been further extended to various geometric domains, such as implicit surfaces [Mémoli and Sapiro 2001] and point clouds [Mémoli and Sapiro 2005]. Recently, Crane *et al.* [2013] proposed the heat method for computing geodesic distance fields which runs in near-linear time. However, these methods have their own drawbacks: unbounded errors (for example, FMM might introduce significant errors over a triangle mesh by locally unfolding obtuse triangles), violation of metric properties (symmetry and triangle inequality) [Crane et al. 2013], inconsistency of geodesic paths generated by back-tracing (that is, a generated path might have a different length from the calculated distance), and sensitivity to mesh triangulation [Liu et al. 2015]. Both exact and approximation algorithms based on window propagation perform better in these aspects. Window propagation has also been successfully applied to the computation of geodesic Voronoi diagrams [Liu et al. 2011].

### 3 Pairwise Window Pruning within a Triangle

Let us 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 covers both opposite edges (Fig 2(a)). Now let us consider the propagation of two windows entering the same triangle. 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. In this section, we aim to produce an exhaustive list of scenarios for pairwise window pruning inside a triangle under the presumption that hyperbolic curves (see Section 4 of [Mitchell et al. 1987]) are not used to trim windows due to the relatively high computational cost of solving quadratic equations.

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  $\triangle ABC$  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  $w_1$  (i.e.,  $qb_0b_1$ ) since otherwise  $w_1$  should have been split into two windows. Similarly,  $q$  does not lie inside the visible cone of  $w_0$ .

On the basis of Proposition A.1 in Appendix A, we list 15 cases which may produce redundant windows inside  $\triangle ABC$ , and design corresponding pruning rules in the following five situations. An illustration of the first 12 cases and their corresponding pruning rules is shown in Fig 3.

**Situation 1: Propagating  $w_0$  and  $w_1$  from the same edge to another edge.** Here we only discuss the configuration where  $w_0$  and  $w_1$  are propagated from  $AB$  to  $AC$ , and other configurations in this situation can be dealt with similarly. Without loss of gen-

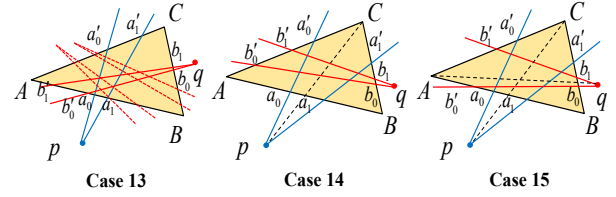


Figure 4: All the cases in Situation 5.

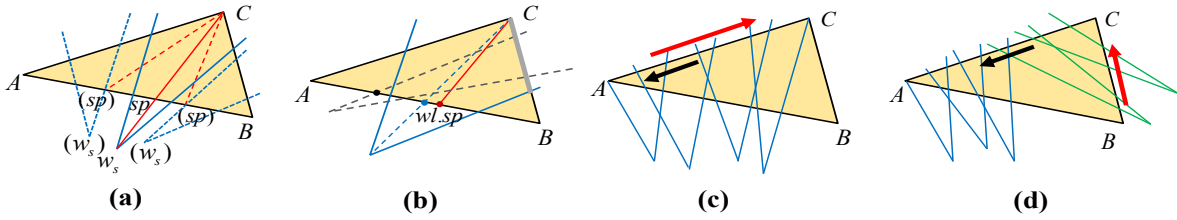
erality, we assume  $a_0 < b_0$ . We refine this situation into three cases (Case 1, Case 2 and Case 3) according to the relative position between  $w'_0$  and  $w'_1$ . The corresponding window pruning rules are shown in the dashed boxes in Fig 3. These rules are derived using Observations (a) and (e) in Proposition A.1. Case 2 is the same as step 2 (*Delete Dominated Candidate Intervals*) of procedure *Insert - Interval*( $I, c$ ) in [Mitchell et al. 1987]. The other two cases are novel.

**Situation 2: Propagating  $w_0$  and  $w_1$  from two edges to the third edge.** We assume  $w_0$  is propagated from  $AB$  to  $AC$ , and  $w_1$  is propagated from  $BC$  to  $AC$ . Then, we also have three cases (Case 4, Case 5 and Case 6) corresponding to Cases 1-3, respectively. 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.** We assume both  $w_0$  and  $w_1$  lie on  $AB$ , and they are respectively propagated to  $BC$  and  $AC$ . We refine this situation into Case 7, Case 8 and Case 9 shown in Fig 3. In Case 7, we only show 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 Observations (d) and (h) in Proposition A.1. Similarly, in Case 8, we only show the configuration where  $w'_0$  covers two edges. The pruning rules in this case are based on Observations (b), (c), (f) and (g) in Proposition A.1. Among these rules, rule 1) is based on Observations (c) and (f) while rule 2) is based on Observations (b) and (g). Case 7 and Case 8 are quite useful in removing redundant windows in our geodesic algorithm. Case 9 is exactly the same as the “One-Angle-One-Split” rule in [Chen and Han 1990].

**Situation 4: Checking with vertices.** We also extend the “checking with vertices” rule in the ICH algorithm [Xin and Wang 2009] to our two-window scenario. As shown in Fig 3, Case 10 illustrates the configuration where both windows are propagated from the same edge to another edge. The pruning rule is based on Observation (d) in Proposition A.1. Case 11 illustrates the same configuration as in Situation 2. The corresponding pruning rules are derived from Observations (d) and (h) in Proposition A.1. Case 12 shows the same configuration as that in Theorem 3.2 of [Xin and Wang 2009]. The corresponding pruning rule can be derived from Observations (a) and (e) in Proposition A.1.

**Situation 5: Propagating  $w_0$  and  $w_1$  from two edges to two edges.** In addition, we discuss three cases where the two windows are propagated from two different edges to two edges. The left figure in Fig 4 shows Case 13 where both  $w'_0$  and  $w'_1$  cover one edge only. We 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 in [Liu 2013]. In this



**Figure 5:** Three rules governing window list propagation inside a triangle. (a) and (b) correspond to Rule 1, (c) illustrates Rule 2, and (d) illustrates Rule 3.

paper, we skip this case since it requires solving a quadratic equation. Note that skipping this case only results in some redundant windows rather than missing any shortest paths. 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 Fig. 4, where both windows can be split into sub-windows and these cases can be reduced to Cases 1-9 and Case 13.

## 4 Synchronized Window Propagation within a Triangle

Most of the window pruning operations discussed in the previous section perform cross checking between pairs of nearby windows. If we accumulate more windows in a triangle and propagate them simultaneously, more window pairs can be formed and window pruning can be carried out more thoroughly. To make this happen, we synchronize local window propagation within the same triangle by simultaneously propagating a collection of windows from one triangle edge to its two opposite edges.

In this section, we present three rules for 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 we have identified an exhaustive list of scenarios for pairwise window pruning within a triangle, our 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 our 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 6.2.

### 4.1 Window List Splitting

Let  $\triangle ABC$  be the triangle where we perform window list propagation. Consider a window list  $wl = \{w_0, w_1, \dots, w_k\}$  on edge  $AB$ , and this window list is going to be propagated across  $\triangle ABC$ . We 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 source vertex  $S$  to  $C$  routed through the windows in  $wl$ . Let the window supporting the shortest distance to  $C$  be  $w_s$ . Next, we 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, we have the following Proposition for splitting a window list. It is based on Cases 7-10 in Fig 3.

#### Proposition 4.1. 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  $w.sp > wl.sp$ .

Note that this proposition is applicable to all three possible configurations of  $w_s$  shown in Fig 5(a). The proof is given in Appendix B. Intuitively, windows on  $AB$  can be split into two subsets by  $w_s$ . We name this rule “One Angle Two sides” since it can be considered as a generalized version of the “One-Angle-One-Split” rule in [Chen and Han 1990]. In fact, the original “One-Angle-One-Split” rule is equivalent to performing cross checking when both windows cover vertex  $C$ , which is exactly Case 9 in Fig 3. Our generalized rule is 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  $w_i.sp$  ( $i = 1, \dots, k$ ) 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 4.1; 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 Fig 5(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).

**Remark** Rule 1 prunes many redundant windows using the separating point of a window list before propagating them. Compared with MMP and ICH algorithms which only perform window trimming or filtering after propagation, our Rule 1 significantly reduces the time for propagating redundant windows.

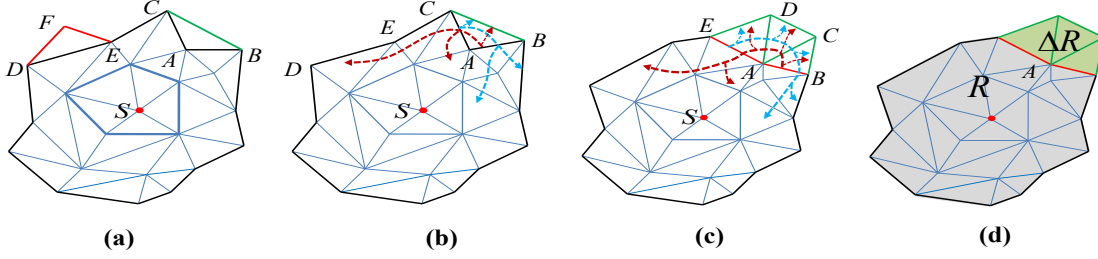
### 4.2 Window List Propagation

After enforcing Rule 1, the window list  $wl$  has been cleaned and split into  $wl_{left}$  and  $wl_{right}$ . In this subsection, we discuss how to perform window pruning when propagating each sublist from one edge to another edge. The workflow for propagating  $wl_{left} = \{w_0, w_1, \dots, w_m\}$  from  $AB$  to  $AC$  is given below (propagating  $wl_{right}$  from  $AB$  to  $CB$  is similar).

#### 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  $j = i + 1$ .



**Figure 6:** Triangle-oriented wavefront propagation over a mesh. (a)-(b) Face-sorted wavefront propagation, (c)-(d) Vertex-sorted wavefront propagation.

**Step 2.** If  $j == NULL$ , finish; otherwise, perform pairwise window pruning between  $w_i$  and  $w_j$  using Case 1, Case 2 and Case 3 in Section 3.

**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 Fig 5(c), 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 will be discussed in the next section.

**Checking with vertices** During the process of enforcing Rule 2, for each propagated window on  $AC$ , we also apply Case 12 in Fig 3 by checking the window against the distance to vertices (the same as the filtering rule in ICH [Xin and Wang 2009]).

### 4.3 Window List Merging

Suppose we already have a window list  $wl^l = \{w_0^l, w_1^l, \dots, w_m^l\}$  on  $AC$ , which is propagated from  $AB$ . In this subsection, we present the following procedure to propagate windows from another list  $wl^r = \{w_0^r, w_1^r, \dots, w_n^r\}$  from  $BC$  to  $AC$ , and merge the propagated windows with  $wl^l$ . Meanwhile, we perform window pruning using Cases 1-6 in Fig 3.

Procedure  $PrimeMerge(wl^l, wl^r)$  (**Rule 3**) consists of the following steps. First, perform one step of propagation for all windows in  $wl^r$ . Let  $w_i^r$  be the propagated version of  $w_i$ . Then, for each window from  $w_0^r$  to  $w_n^r$ , run the following substeps: (i) append it to  $wl^l$ ; (ii) set  $j = wl^l.tail$  and  $i = j - 1$ ; (ii) perform pairwise checking and window 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, we need 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).

We name this procedure  $PrimeMerge()$  because it will be used for merging window lists on an edge for the first time. It is complementary to  $SecondMerge()$  in Section 5.1. The time complexity of  $PrimeMerge()$  is  $O(m + n)$ , which will be discussed in the next section. Fig 5(d) shows an illustration of the outer loop (red arrow) and the inner loop (black arrow) of this procedure.

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

**Proposition 4.2.** If both  $wl_{AB}$  and  $wl_{BC}$  are spatially coherent, the window list  $wl^l = wl_{AB \rightarrow AC}$  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.

The proof of this Proposition is given in Appendix C.

## 5 Triangle-Oriented Wavefront Propagation over a Mesh

Our geodesic algorithm takes triangles as the primitive for synchronizing window and distance propagation. 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. Our algorithm expands this traversed area in a continuous Dijkstra style by gradually enclosing unvisited triangles abutting the traversed area. During each iteration, our 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. We denote the region outside  $R$  but inside  $R'$  as  $\Delta R$ , which consists of the newly added triangles. In this section, we first present a basic face-sorted propagation algorithm, and then extend it to vertex-sorted propagation, which achieves improved performance.

### 5.1 Face-Sorted Wavefront Propagation

As shown in Fig 6(a) and (b), our 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 Fig 6(a)), and push all triangles that are outside the 1-ring neighborhood of  $S$  and that 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  $\Delta R$ .

**Step 2.** If this triangle has only one edge on the previous wavefront ( $\Delta DEF$  in Fig 6(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  $\Delta DEF$  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 wave-

front ( $\Delta ABC$  in Fig 6(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  $\Delta R$ . Push the adjacent triangle sharing  $BC$  with  $\Delta ABC$  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  $\Delta DEF$  in Fig 6(a), we define its priority as the negative distance from  $S$  to  $F$  via  $w_{DE}$ , which is  $-w_{DE}.dis$ , where  $w_{DE}.dis$  is defined in Section 4.1. For a triangle which shares two edges with the wavefront, such as  $\Delta ABC$  in Fig 6(a), we define its priority as the larger of  $-w_{AB}.dis$  and  $-w_{CA}.dis$ .

**Geodesic Update** Once a new triangle, such as  $\Delta ABC$  in Fig 6(a), has been added to the traversed area, we propagate windows inside  $\Delta R$  from previous wavefront edges (such as  $CA$  and  $AB$ ) to new wavefront edges in  $\Delta R$  (such as  $BC$ ). In addition, we also need to propagate windows inside  $R$  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, we propagate window lists along these paths, and update geodesic distances at vertices in  $R$  along the way until all the propagated windows have been either removed by our 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, we need to merge them. We classify window list mergers at edges on the wavefront into two categories. Mergers taking place at new wavefront edges in  $\Delta R$  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 4.3, and secondary mergers will be discussed later in this section.

Procedure *GeodesicUpdate*(*wlist*)

**Step 1** Push *wlist* to a 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*( $wl, wl_{e_w}$ ); 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, we propagate window lists 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 our 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 dis-

---

**Algorithm 1** Vertex-Oriented Triangle Propagation
 

---

```

1: procedure VTP( $M, S$ )      ▷  $M$  - Mesh,  $S$  - Source Vertex
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 Subsection 5.1;
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  $D$ ;
15:      Let the propagated lists be  $wl_{e \rightarrow e_0}$  and  $wl_{e \rightarrow e_1}$ ;
16:      for each edge  $e_i$  do
17:        if  $wl_{e \rightarrow e_i}$  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( $wl_{e \rightarrow e_i}, wl_{e_i}$ );
21:          else if  $e_i \in wf$  then
22:            SecondMerge( $wl_{e \rightarrow e_i}, wl_{e_i}$ );
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

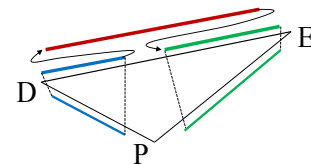
```

---

tance 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.

**Order-Preserving Secondary Merger** When a secondary merger is performed at an edge on the wavefront, we can make the merged window list spatially coherent to ensure its further propagation compatible with our 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  $wl^g = \{w_0^g, w_1^g, \dots, w_k^g\}$ . 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  $wl_{e_w}$ .

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



**Figure 7:** Order-free secondary merger.



Procedure  $SecondMerge(wl_{DE}, wl^g)$  (Fig 7): 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  $wl^g$ .

In our experiments (Section 6.2), we have found that this simple scheme runs faster than the order-preserving version, which does not remove many more redundant windows and sometimes even keeps slightly more redundant windows. 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, our 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 tradeoff.

We name the face-oriented propagation algorithm with order-free secondary merger FTP, and the version with order-preserving secondary merger OPFTP.

## 5.2 Vertex-Sorted Wavefront Propagation

Instead of expanding the wavefront one triangle at a time, we 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 our 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.

The revised geodesic algorithm also proceeds in a continuous Dijkstra style. As shown in Fig 6(c), 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 we 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  $R'$ .

As reported in Table 1 in Section 6, this variant of our algorithm runs faster than FTP since 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  $GeodesicUpdate()$  twice since the triangles are connected. We call this vertex-sorted propagation algorithm with order-free secondary merger VTP, and the version with order-preserving secondary merger OPVTP. The pseudo code of VTP is shown in Algorithm 1, which is the final version of our algorithm.

**Saddle Vertices** Finally let us discuss how to handle the following scenarios where a saddle vertex (pseudo source) is visited. The first scenario applies to FTP only. When a triangle  $\Delta ABC$  sharing two edges with the previous wavefront joins the traversed area as shown in Fig 6(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. We perform the following special procedure 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,

	<i>ICH vs. Ours</i>	<i>MMP vs. Ours</i>	<i>FWP-CH vs. Ours</i>	<i>FWP-MMP vs. Ours</i>
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 2:** The mean and standard deviation of performance ratios between other algorithms and our VTP algorithm on running time, the number of window propagations and peak memory usage.

call  $GeodesicUpdate()$  to propagate this single window inside  $R$ . In addition, we update the distance at every 1-ring neighbor of the saddle vertex as in [Xin and Wang 2009].

## 5.3 Complexity Analysis

**Proposition 5.1.** Applying Rule 2 to a window list with  $N$  windows costs  $O(N)$  time,  $PrimeMerge(wl, wl')$  costs  $O(M + N)$  time, and  $OPSecondMerge(wl, wl')$  costs  $O(N \log(M + N))$  time, where  $M$  and  $N$  are the number of windows in  $wl$  and  $wl'$ , respectively.

The proof of this Proposition is given in Appendix D.

Let  $n$  be the number of vertices on the mesh. We first discuss the complexity of FTP and OPFTP. 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 our Rule 1, and  $SecondMerge()$  costs  $O(1)$  time. According to Proposition 5.1, 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  $OPSecondMerge()$ . 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 our algorithms is also  $O(n^2)$ .

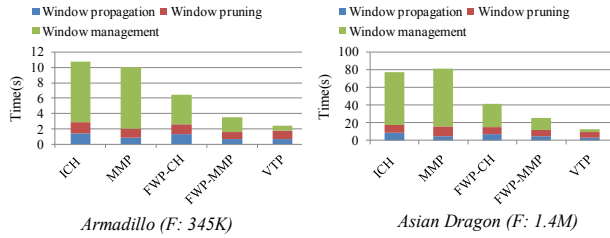
## 6 Experimental Results

To validate the performance of our algorithms, we have tested them on a variety of models and compared them against existing state-of-the-art algorithms (MMP, ICH, FWP-MMP, FWP-CH). Specifically, we collected 55 models, including sculptures, animals and man-made objects, for testing. The resolution of these models (number of faces) ranges from 10k to 14M. To evaluate overall performance, we measure running time, total number of window propagations and peak memory usage. All the algorithms are tested using a PC with an Intel Core i7-3770 3.40GHz CPU and 32GB memory. Due to space limitations, the results on some of the testing models are shown in the paper and all the remaining results are given in the supplemental materials. In our experiments, we choose the first vertex on all meshes as the source vertex. To make fair comparisons, we adopt a fixed threshold  $\epsilon = 10^{-6}$  for distance comparisons throughout our experiments as in [Xu et al. 2015].



Model	Performance	Algorithms						
		ICH	MMP	FWP-CH	FWP-MMP	VTP-CH	VTP-MMP	VTP
Bunny (F: 144 K)	Time(s)	5.034	4.612	3.056	1.737	2.672	1.304	0.78
	# window propagations	12,305,579	6,485,320	12,327,991	6,451,352	12,491,178	6,454,800	4,943,670
	Peak memory(MB)	1.69	340.45	1.70	340.46	1.71	340.45	1.24
Rocker Arm (F: 482 K)	Time(s)	36.577	33.286	19.536	11.867	15.449	6.954	4.13
	# window propagations	68,553,846	33,989,638	70,513,186	35,940,386	69,208,037	33,947,674	25,654,638
	Peak memory(MB)	5.29	1797.16	5.42	1797.19	5.35	1797.16	3.70
Asian Dragon (F: 1,400 K)	Time(s)	73.204	73.092	35.637	23.674	29.492	15.388	9.495
	# window propagations	107,742,094	62,161,583	108,122,218	62,025,717	109,311,094	61,995,300	48,217,896
	Peak memory(MB)	5.184	3354.04	5.207	3354.05	5.23	3354.03	4.373
Neptune (F: 4,008 K)	Time(s)	455.271	424.331	193.945	120.012	158.912	60.297	47.629
	# window propagations	585,784,159	270,930,198	602,587,831	284,581,696	606,937,112	278,925,270	246,364,008
	Peak memory(MB)	16.96	14225.26	17.14	14219.76	17.26	14224.78	16.38
Lucy (F: 14,464 K)	Time(s)	8894.87	Out of memory	2415.88	Out of memory	1853.66	Out of memory	549.934
	# window propagations	6,837,670,602	Out of memory	6,841,729,337	Out of memory	6,859,484,793	Out of memory	2,808,823,718
	Peak memory(MB)	78.29	Out of memory	78.28	Out of memory	79.32	Out of memory	69.42

**Table 1:** 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 8:** Comparison of running times of three common components on two models.

## 6.1 Comparison with Existing Algorithms

In this section, we compare the performance of our final algorithm (VTP) with existing state-of-the-art algorithms (ICH, MMP, FWP-CH, and FWP-MMP) in four aspects. Prior to performance comparisons, we have confirmed the correctness of our algorithm by comparing the results from VTP against those from the exact implementation of MMP [Surazhsky et al. 2005].

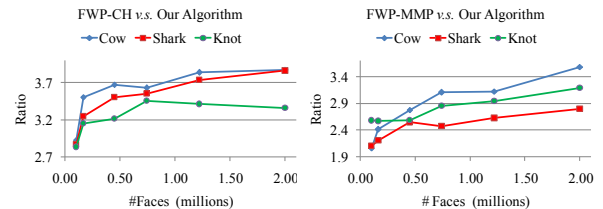
**Overall Performance** All algorithms in our comparison have been tested on all 55 models and the detailed results are given in the supplemental materials. We calculate the mean and standard deviation of performance ratios between other algorithms and our VTP algorithm. The details are shown in Table 2. It can be seen from the table that our 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. Our algorithm has 56% fewer window propagations than ICH and FWP-CH, and 20% fewer window propagations than MMP and FWP-MMP. Furthermore, our 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. Our 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 1.

**Performance Profiling** As mentioned, all geodesic algorithms based on window propagation have three primary components:

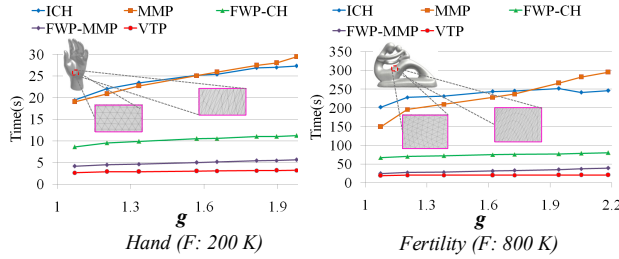
window propagation, window pruning and window management. We profile the running times of these individual components in all participating algorithms on ten models. We only show the results on two models (Armadillo and Asian Dragon) in Fig 8 and the rest of the results have been included in the supplemental materials. We can see that our algorithm outperforms other existing algorithms in the efficiency of all three components. Thanks to the triangle-based propagation strategy, our 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 our pruning rules only require comparisons between two distances. This also reduces the computational cost of window pruning itself.

**Scalability** We further study how the performance of our algorithm varies with increasing mesh resolution. We first choose three test models (Cow, Shark and Knot) and 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, we calculate ratios between the running times of both FWP-CH and FWP-MMP and that of VTP on all six resolutions, and show how they change with the changing resolution. As illustrated in Fig 9, the timing ratios increase with an increasing resolution. That is, our algorithm achieves more significant performance gain when dealing with larger models.

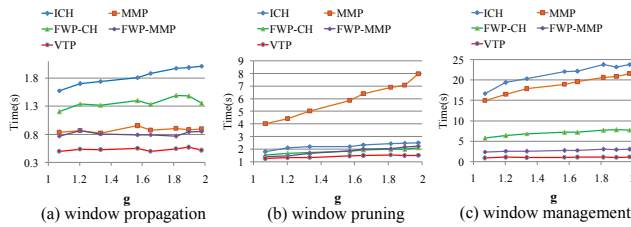
**Robustness** In this section, we further validate that our algorithm is robust to mesh triangulation quality. As in FWP [Xu et al. 2015], we first create a sequence of meshes (eight) with different degrees of anisotropy but a fixed resolution on two testing models



**Figure 9:** Comparison of scalability against recent geodesic algorithms. The x-axis represents mesh resolution, and the y-axis represents performance ratio.



**Figure 10:** Comparison of robustness against anisotropic triangulation. The  $x$ -axis represents the degree of anisotropy, and the  $y$ -axis represents running time.



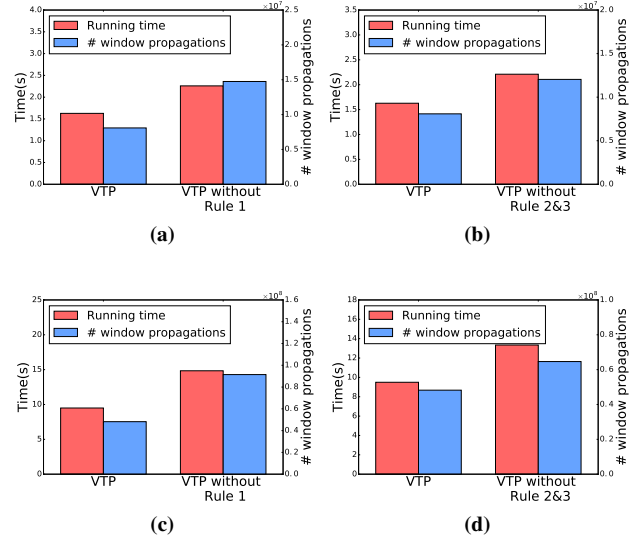
**Figure 11:** Robustness of individual components against anisotropic triangulation.

(Fertility with 800K faces and Hand with 200K faces) respectively. Here, we also use  $g(M) = \frac{\sum_{f \in F} g'(f)}{|F|}$  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 Fig 10 show how the running times change with increasing anisotropy ( $g$ ). Our algorithm is the most robust 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), we further show how the running times of each component changes with increasing  $g$  on the Armadillo model in Fig 11. 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 Fig 11(a). For window pruning, MMP and FWP-MMP require binary search and solving quadratic equations, which also entails longer running times, as shown in Fig 11(b). As MMP and ICH algorithms use a priority queue for managing windows, they need more time to process more windows, as shown in Fig 11(c). 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 Fig 11(c). In contrast, our algorithm is insensitive to the number of windows since it manages triangle vertices instead of windows. As a result, the running times of our components change the least with increasing anisotropy.

## 6.2 Comparison among Algorithmic Choices

We first justify the choice of VTP as our final algorithm by comparing it against other variants, including FTP, OPVTP, VTP-



**Figure 12:** Ablation study on the three rules. (a) and (b) are results on model Armadillo ( $F: 345K$ ), while (c) and (d) are results on model Asian Dragon ( $F: 1.4M$ ). The left  $y$ -axis represents running time and the right  $y$ -axis represents the number of window propagations.

Exhaustive, VTP-Trimming, and VTP-MMP/CH. These comparisons were conducted on all 55 models. We show results on 5 representative models in Table 1 and Table 3, and the complete results are given in the supplemental materials.

**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. From the results in Table 3, we find that VTP runs faster than FTP even though it usually performs more window propagations.

**OPVTP vs. VTP** We have also compared VTP with OPVTP by applying them to all 55 testing models. The results in Table 3 indicate that enforcing spatial coherence increases the overall running time but does not remove many more redundant windows. VTP strikes a better balance between the overall speed and the thoroughness in window pruning.

**VTP-Exhaustive vs. VTP** We have implemented a variant of the VTP algorithm by performing redundancy checking on all possible window pairs within the same window list as well as across different window lists inside the same triangle. This scheme essentially performs exhaustive pairwise window checking, hence, the name VTP-Exhaustive. As seen in Table 3, this variant performs fewer window propagations but runs much slower than VTP.

**VTP-Trimming vs. VTP** We have also implemented a variant named VTP-Trimming, which performs window trimming as in [Surazhsky et al. 2005] instead of window pruning following our Rule 2 and Rule 3 when two windows overlap on an edge. As shown in Table 3, this method runs slower than VTP though it performs fewer window propagations. This is because window trimming by solving quadratic equations is expensive.

Model	Performance	Algorithms				
		VTP-Exhaustive	VTP-Trimming	FTP	OPVTP	VTP
Bunny (F: 144K)	Time(s)	4.557	0.872	1.044	0.908	0.78
	# window propagations	4,801,056	4,686,252	4,755,872	4,875,712	4,943,670
	Peak memory(MB)	1.22	1.146	1.20	1.22	1.24
Rocker Arm (F: 482K)	Time(s)	36.586	4.655	4.84	5.173	4.13
	# window propagations	24,289,066	24,380,006	25,013,422	25,723,669	25,654,638
	Peak memory(MB)	3.43	3.49	3.68	3.71	3.70
Asian Dragon (F: 1,400K)	Time(s)	49.954	13.763	13.223	11.42	9.495
	# window propagations	46,926,451	46,316,630	46,525,313	47,573,341	48,217,896
	Peak memory(MB)	4.036	4.017	4.10	4.20	4.373
Neptune (F: 4,008K)	Time(s)	665.847	58.49	64.113	60.192	47.629
	# window propagations	239,054,124	239,375,390	243,102,435	244,586,129	246,364,008
	Peak memory(MB)	15.62	15.962	15.9	16.19	16.38
Lucy (F:14,464K)	Time(s)	16559	615.215	617.343	608.414	549.934
	# window propagations	2,703,707,866	2,733,324,263	2,668,122,127	2,734,517,299	2,808,823,718
	Peak memory(MB)	66.096	66.848	67.81	68.01	69.42

**Table 3:** Performance comparison with variants of our VTP algorithm on running time, peak memory usage and total number of window propagations.  $F$  : means the number of faces on a model.

**VTP-MMP/CH vs. VTP** VTP has two ingredients making it outperform all existing algorithms: a vertex-sorted window management scheme and three window pruning rules. To evaluate the contribution of individual components, we have implemented VTP-MMP/CH, which uses our vertex-sorted priority queue to propagate all windows around a vertex every time while performing redundant window checking using the same rules as in the MMP and ICH algorithms respectively. We have conducted comparisons between VTP and these variants. As seen in Table 1, vertex-sorted window management is more efficient than all existing management schemes, including priority queues used in the MMP/ICH algorithms and buckets used in the FWP-MMP/CH algorithms. Nonetheless, this window management scheme alone does not reduce the total number of window propagations.

Since our algorithm is built upon three window propagation and pruning rules, we also perform an ablation study to verify the effectiveness of individual rules.

**With and Without Rule 1** We first compare the performance of our VTP algorithm with and without using Rule 1 introduced in Section 4.1. When Rule 1 is turned off, to split a window list to two sides, we simply create two sub-windows for each window that covers two opposite edges and do not perform any pruning operations on any windows or sub-windows. We have tested this revised algorithm on ten models with different model sizes, and the results on two models (Armadillo and Asian Dragon) are shown in Fig 12. The rest of the results have been included in the supplemental materials. We find that Rule 1 saves approximately 25% running time and 50% window propagations.

**With and Without Rules 2 and 3** We also compare the performance of our VTP algorithm with and without using Rules 2 and 3 introduced in Sections 4.2 and 4.3. Here, we do not deal with these two rules separately since they are two similar rules to maintain the spatial coherence of window lists. When both Rules 2 and 3 are turned off, we do not perform any pairwise cross checking after propagating a window list or merging two window lists. We have also tested this revised algorithm on the same ten models, and show the results on two models (Armadillo and Asian Dragon) in Fig 12. The rest of the results are shown in the supplemental materials. We find that Rules 2 and 3 together save approximately 15% running time and 30% window propagations.

**Distribution of Window Propagations** As mentioned earlier, in an iteration of our VTP algorithm,  $R$  and  $R'$  stand for the existing and expanded traversed areas, respectively, and  $\Delta R$  stands for the region outside  $R$  but inside  $R'$  (Fig 6(d)). Let us call window propagations taking place inside  $\Delta R$  prime propagations, and those inside  $R$  secondary propagations. Then, we count these two types of propagations on all 55 testing models and the results are given in the supplemental materials. We find 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  $\Delta R$  and our 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, our decision to perform order-free secondary mergers is reasonable.

## 7 Conclusions and Discussions

In this paper, a novel exact geodesic computational framework has been presented. It outperforms all recent methods in terms of running time, peak memory usage and total number of window propagations. To perform thorough window pruning, an exhaustive list of scenarios have been proposed to trim or filter windows by performing pairwise cross checking. Further, a triangle-oriented region growing scheme has been developed for efficient window propagation and pruning. In addition, by organizing triangles instead of windows using a priority queue, our algorithm also significantly cuts down the time spent on window management. According to our experiments, our geodesic algorithm runs 4-15 times faster than MMP and ICH algorithms and 2-4 times faster than FWP-MMP and FWP-CH algorithms.

**Limitations** Although our algorithm can remove more redundant windows than existing methods, it is still challenging to remove all redundant windows. Another limitation of our current algorithm is that it is only suited for triangle domains and cannot be applied to other domains, such as quad meshes and point clouds.

There exist a few directions for future work. Applying the FWP strategy (i.e., using buckets instead of a priority queue to sort the vertices) can potentially further improve the performance of our algorithm. A parallel implementation of our algorithm would be

an interesting approach for developing faster exact geodesic algorithms. In addition, adapting our triangle-oriented window propagation to the all-pairs geodesic problem and geodesic Voronoi diagram computation is also worth pursuing.

## Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments. This project was supported by EU H2020 project AniAge (No.691215) and Hong Kong Research Grants Council under General Research Funds (HKU718712). All our testing models are from the AIM@SHAPE shape repository, Large Geometric Models Archive at Georgia Institute of Technology, Suggestive Contour Gallery provided by Princeton University and Stanford scanning repository.

## References

- BALASUBRAMANIAN, M., POLIMENI, J., AND SCHWARTZ, E. L. 2009. Exact geodesics and shortest paths on polyhedral surfaces. *IEEE Trans. Pattern Anal. Mach. Intell.* 31, 6, 1006–1016.
- BOSE, P., MAHESHWARI, A., SHU, C., AND WUHRER, S. 2011. A survey of geodesic paths on 3d surfaces. *Comput. Geom. Theory Appl.* 44, 9 (Nov.), 486–498.
- BRONSTEIN, A. M., BRONSTEIN, M. M., AND KIMMEL, R. 2006. Generalized multidimensional scaling: a framework for isometry-invariant partial surface matching. *Proceedings of the National Academy of Sciences* 103, 5, 1168–1172.
- CHEN, J., AND HAN, Y. 1990. Shortest paths on a polyhedron. In *Proceedings of the Sixth Annual Symposium on Computational Geometry*, ACM, New York, NY, USA, SCG '90, 360–369.
- CRANE, K., WEISCHEDEL, C., AND WARDETZKY, M. 2013. Geodesics in heat: A new approach to computing distance based on heat flow. *ACM Trans. Graph.* 32, 5 (Oct.), 152:1–152:11.
- KIMMEL, R., AND SETHIAN, J. A. 1998. Computing geodesic paths on manifolds. In *Proc. Natl. Acad. Sci. USA*, 8431–8435.
- LIU, Y. J., CHEN, Z., AND TANG, K. 2011. Construction of isocontours, bisectors, and voronoi diagrams on triangulated surfaces. *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 33, 8, 1502–1517.
- LIU, Y.-J., XU, C.-X., FAN, D., AND HE, Y. 2015. Efficient construction and simplification of delaunay meshes. *ACM Transactions on Graphics (TOG)* 34, 6, 174.
- LIU, Y.-J. 2013. Exact geodesic metric in 2-manifold triangle meshes using edge-based data structures. *Computer-Aided Design* 45, 3, 695–704.
- MÉMOLI, F., AND SAPIRO, G. 2001. Fast computation of weighted distance functions and geodesics on implicit hypersurfaces. *Journal of computational Physics* 173, 2, 730–764.
- MÉMOLI, F., AND SAPIRO, G. 2005. Distance functions and geodesics on submanifolds of  $r^d$  and point clouds. *SIAM Journal on Applied Mathematics* 65, 4, 1227–1260.
- MITCHELL, J. S. B., MOUNT, D. M., AND PAPADIMITRIOU, C. H. 1987. The discrete geodesic problem. *SIAM J. Comput.* 16, 4 (Aug.), 647–668.
- PEYRÉ, G., AND COHEN, L. D. 2006. Geodesic remeshing using front propagation. *International Journal of Computer Vision* 69, 1, 145–156.
- SCHREIBER, Y., AND SHARIR, M. 2009. An optimal-time algorithm for shortest paths on a convex polytope in three dimensions. In *Twentieth Anniversary Volume*. Springer, 1–80.
- SETHIAN, J. A. 1996. A fast marching level set method for monotonically advancing fronts. *Proceedings of the National Academy of Sciences* 93, 4, 1591–1595.
- SURAZHISKY, V., SURAZHISKY, T., KIRSANOV, D., GORTLER, S. J., AND HOPPE, H. 2005. Fast exact and approximate geodesics on meshes. *ACM Trans. Graph.* 24, 3 (July), 553–560.
- XIN, S.-Q., AND WANG, G.-J. 2009. Improving chen and han’s algorithm on the discrete geodesic problem. *ACM Trans. Graph.* 28, 4 (Sept.), 104:1–104:8.
- XIN, S.-Q., YING, X., AND HE, Y. 2012. Constant-time all-pairs geodesic distance query on triangle meshes. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D '12, 31–38.
- XU, C., WANG, T. Y., LIU, Y., LIU, L., AND HE, Y. 2015. Fast wavefront propagation (FWP) for computing exact geodesic distances on meshes. *IEEE Trans. Vis. Comput. Graph.* 21, 7, 822–834.
- YATZIV, L., BARTESAGHI, A., AND SAPIRO, G. 2005. O(n) implementation of the fast marching algorithm. *Journal of Computational Physics* 212, 393–399.
- YE, J., AND YU, Y. 2016. A fast modal space transform for robust nonrigid shape retrieval. *The Visual Computer* 32, 5 (May), 553–568.
- YE, J., YAN, Z., AND YU, Y. 2013. Fast nonrigid 3d retrieval using modal space transform. In *3rd ACM International Conference on Multimedia Retrieval (ICMR)*, 121–126.
- YING, X., WANG, X., AND HE, Y. 2013. Saddle vertex graph (svg): A novel solution to the discrete geodesic problem. *ACM Trans. Graph.* 32, 6 (Nov.), 170:1–170:12.
- YING, X., XIN, S.-Q., AND HE, Y. 2014. Parallel chen-han (pch) algorithm for discrete geodesics. *ACM Trans. Graph.* 33, 1 (Feb.), 9:1–9:11.
- ZHONG, Z., GUO, X., WANG, W., LVY, B., SUN, F., LIU, Y., AND MAO, W. 2013. Particle-based anisotropic surface meshing. *ACM Transactions on Graphics (SIGGRAPH conference proceedings)*.

## A Principles for Window Pruning

The following 8 observations serve as our basic principles for window pruning. We assume all geometric primitives in each observation lie on the same plane.

**Proposition A.1.** Let  $w_0$  and  $w_1$  be two windows,  $p$  and  $q$  be their respective pseudo sources, and  $\sigma_0$  and  $\sigma_1$  be the geodesic distances from their pseudo sources to the true source vertex.

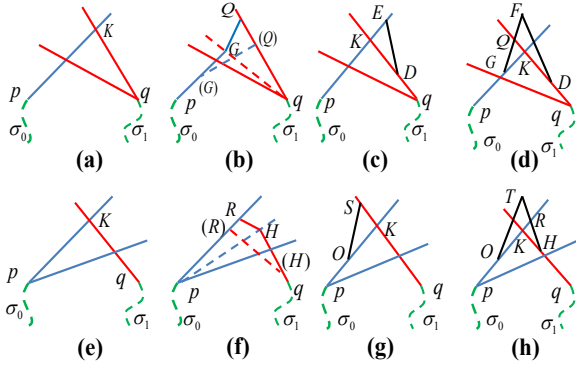
—(a) Suppose a line  $pK$  intersects the upper ray of  $w_1$  at point  $K$ . Then,  $w_1$  is redundant if  $\sigma_0 + \|pK\| < \sigma_1 + \|qK\|$ .

—(b) Suppose a polyline  $pGQ$  intersects the upper ray of  $w_1$  at point  $Q$ . Then,  $w_1$  is redundant if  $\sigma_0 + \|pG\| + \|GQ\| < \sigma_1 + \|qQ\|$ .

—(c) Considering the same scenario as in 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  $\sigma_0 + \|pE\| < \sigma_1 + \|qD\| + \|DE\|$ .

—(d) Considering the same scenario as in b), 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  $\sigma_0 + \|pG\| + \|GF\| < \sigma_1 + \|qD\| + \|DF\|$ .




**Figure 13: Window configurations for Proposition A.1**

—(e) Suppose a line  $qK$  intersects the upper ray of  $w_0$  at point  $K$ . Then,  $w_0$  is redundant if  $\sigma_1 + \|qK\| < \sigma_0 + \|pK\|$ .

—(f) Suppose a polyline  $qHR$  intersects the upper ray of  $w_0$  at point  $R$ . Then,  $w_0$  is redundant if  $\sigma_1 + \|qH\| + \|HR\| < \sigma_0 + \|pR\|$ .

—(g) Considering the same scenario as in (e), let  $S$  be a point on the extended part of  $qK$  and  $O$  a point on the segment  $pK$ . Then,  $w_0$  is redundant if  $\sigma_1 + \|qS\| < \sigma_0 + \|pO\| + \|OS\|$ .

—(h) Considering the same scenario as in (f), 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  $\sigma_1 + \|qH\| + \|HT\| < \sigma_0 + \|pO\| + \|OT\|$ .

*Proof.* Here we only prove the first 4 observations and the remaining ones can be proved in a similar way.

(a), as shown in Fig 13(a), for any point  $X$  on the segment  $pK$ , we have  $\sigma_1 + \|qX\| + \|XK\| \geq \sigma_1 + \|qK\| > \sigma_0 + \|pK\| = \sigma_0 + \|pX\| + \|XK\|$ . 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.

(b), the case where  $G$  lies outside  $w_1$  can be reduced to observation (a) where  $G$  is set to  $p$ . When  $G$  lies inside the visible cone of  $w_1$ , we can split  $w_1$  into two sub-windows with the dashed red line shown in Fig 13(b). For the upper sub-window, the conclusion can be reached directly by taking  $G$  as  $p$  in observation (a); for the lower sub-window, we have  $\sigma_0 + \|pG\| + \|GQ\| < \sigma_1 + \|qQ\| \leq \sigma_1 + \|qG\| + \|GQ\|$ . Then, we reach  $\sigma_0 + \|pG\| < \sigma_1 + \|qG\|$ , which means  $w_1$  is redundant according to observation (a).

(c), we have  $\sigma_0 + \|pK\| + \|KE\| < \sigma_1 + \|qD\| + \|DE\| \leq \sigma_1 + \|qK\| + \|KE\|$ . Then, we reach  $\sigma_0 + \|pK\| < \sigma_1 + \|qK\|$ , and the conclusion can be derived using observation (a).

(d), we have  $\sigma_0 + \|pG\| + \|GQ\| + \|QF\| < \sigma_1 + \|qD\| + \|DF\| \leq \sigma_1 + \|qQ\| + \|QF\|$ . Then, we reach  $\sigma_0 + \|pG\| + \|GQ\| < \sigma_1 + \|qQ\|$ , which means  $w_1$  is redundant according to observation (b).  $\square$

## B Proof of Proposition 4.1

*Proof.* Let  $w'_s$  be the propagated version of  $w_s$  within  $\triangle ABC$ . As shown in Fig 5(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 3.

(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 Fig 3.

(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 Fig 3.

(3) If  $w'_s \in AC$  and  $w'$  covers  $C$ ,  $w_s.sp = w_s.a_1$ , according to Case 8 rule 1) in Fig 3, 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 Fig 3.

(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 Fig 3.

(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 Fig 3.

(7) If  $w'_s$  covers  $C$  and  $w'$  covers  $C$ , the conclusion can be reached according to Case 9 in Fig 3.

(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 Fig 3.

(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 Fig 3.  $\square$

## C Proof of Proposition 4.2

*Proof.* Our Rule 1 in Section 4 does not affect the order of the windows in the list. In the following, we first prove by induction that Rule 2 preserves the spatial order. Consider propagating  $w_{LAB} = \{w_0, w_1, \dots, w_k\}$  (already split by Rule 1) to  $AC$ . Let the propagated version of window  $w_i$  be  $w'_i$ . During the steps of Rule 2, let us assume the sublist from  $w'_0$  to  $w'_i$  are already spatially coherent. Then, we need 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  $a'_0 = w'_i.a_0$ ,  $a'_1 = w'_i.a_1$ ,  $b'_0 = w'_j.a_0$ ,  $b'_1 = w'_j.a_1$ . We discuss the following cases where  $w'_j$  is not spatially coherent with  $w'_i$ :

(i)  $b'_1 < a'_0$ . This corresponds to Case 3 in Fig 3. One of the windows is redundant and should be removed.

(ii)  $b'_0 < a'_0 < b'_1$ . This corresponds to Case 2 in Fig 3. 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, our conclusion can be reached.

The order preservation property of our Rule 3 can be reached according to Cases 1-6 in Fig 3 in a similar way.  $\square$

## D Proof of Proposition 5.1

*Proof.* We first prove that our Rule 2 has linear time complexity. Let  $wl = \{w_0, w_1, w_2, \dots, w_n\}$  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  $\sum_{k=1}^i (t_k - 1)$ . We have  $t_{i+1} \leq i + 1 - \sum_{k=1}^i (t_k - 1)$ , 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. The complexity of *PrimeMerge*() can be derived in a similar way as Rule 2. The complexity of *OPSecondMerge*() can be derived easily using the fact that a binary search has logarithmic complexity.  $\square$