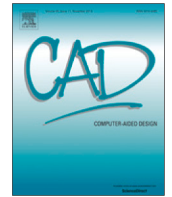
Computer-Aided Design 150 (2022) 103333

Contents lists available at ScienceDirect

Computer-Aided Design

journal homepage: www.elsevier.com/locate/cad



An Accuracy Controllable and Memory Efficient Method for Computing High-Quality Geodesic Distances on Triangle Meshes ☆

Yohanes Yudhi Adikusuma, Jie Du, Zheng Fang, Ying He *

School of Computer Science and Engineering, Nanyang Technological University, Singapore

A R T I C L E I N F O

Article history:
Received 15 April 2022
Accepted 16 May 2022

Keywords:
Discrete geodesics
Triangle meshes
Accuracy controllable window propagation
Discrete geodesic graphs
Vertex-oriented triangle propagation

A B S T R A C T

This paper presents a new method for computing approximate geodesic distances and paths on triangle meshes. Our method combines two state-of-the-art discrete geodesic methods, which are discrete geodesic graphs (DGG) and vertex-oriented triangle propagation (VTP), so that it allows the user to specify the desired accuracy using a single parameter ε. The method, called DGG-VTP, extends the conventional window propagation framework by monitoring the accuracy of the computed distances so that propagation can terminate immediately when the desired accuracy is reached. It is worth noting that for robustness consideration, tiny windows with length less than a threshold (usually, between 10^{-7} and 10^{-6}) are discarded in the implementation of the existing exact algorithms, such as the Mitchel–Mount–Papadimitriou (MMP) algorithm, the Chen–Han (CH) algorithm and their many variants. By setting the accuracy parameter $\varepsilon \in [10^{-7}, 10^{-6}]$, our method can produce results with comparable accuracy to VTP, while being 3–40 times faster and consuming much less memory. Furthermore, the performance of our method is less sensitive to mesh tessellation than what VTP does. Our method empirically produces $O\left(\frac{n}{\varepsilon^{0.23}}\right)$ windows and scales well to deal with large-scale models. Though the parameter ε in DGG-VTP is not a guaranteed error bound, it acts as an intuitive guide for the user to set the desired accuracy. Extensive evaluations demonstrate the effectiveness of our accuracy control: given a parameter $\varepsilon \in [10^{-7}, 10^{-4}]$, 99% of the computed distances have error less than the accuracy parameter. The features of predicable accuracy and computational efficiency distinguish DGG-VTP from the existing approximation methods, and make it an alternative to exact methods in computing accurate geodesic distances on large-scale mesh models. We also develop a parallel version of DGG-VTP on multi-core CPUs, which runs up to 60× faster than the existing parallel VTP algorithm with comparable accuracy under single floating point precision setting. The source code is available at https://github.com/GeodesicGraph/DGG-VTP.

© 2022 Elsevier Ltd. All rights reserved.

1. Introduction

Geodesic computation is a fundamental problem in computer graphics and computational geometry with many applications, ranging from shape analysis, sampling, to remeshing and texture mapping. To date, there are two notable exact algorithms for solving the discrete geodesic problem, namely the Mitchel–Mount–Papadimitriou (MMP) algorithm [1] and the Chen–Han (CH) algorithm [2]. Both algorithms are based on the same core data structure, called windows, which partition mesh edges into intervals so that each window encodes the geodesic paths in the same face sequence. They compute geodesic distances by propagating windows from a given source vertex.

There are two central problems in window propagation based algorithms: one is to detect as many redundant windows as possible and prevent them from propagating, and the other is to effectively organize windows in an increasing order of distances to the source. Xin and Wang [3] proposed a set of distance-based window filters, which can eliminate 99% of the useless windows created by the CH algorithm, and used priority queue to sort windows. The resulting algorithm, called improved CH algorithm or ICH, runs two orders of magnitude faster than the original CH algorithm. Xu et al. [4] proposed fast wavefront propagation (FWP), which organizes windows using a bucket queue. Although bucket queue does not sort windows in an exact way, it has better practical performance than priority queue since the common operations, such as enqueue and dequeue, take constant time $O(1)$. Computational results show that FWP improves ICH by a factor of 3 to 10. Qin et al. [5] thoroughly examined the situations of propagating windows across a triangle, and developed 12 filtering rules to detect useless windows. They also grouped windows

☆ This paper has been recommended for acceptance by Professor George-Pierre Bonneau, Dr. Morad Behandish & Professor Jianmin Zheng.

* Corresponding author.
E-mail address: yhe@ntu.edu.sg (Y. He).

https://doi.org/10.1016/j.cad.2022.103333
0010-4485/© 2022 Elsevier Ltd. All rights reserved.

using edges and organized them using a vertex-oriented priority queue, which is much more efficient than a window-oriented priority queue due to the significantly fewer number of elements in the queue. Their algorithm, called VTP, runs up to 4 times faster than FWP.

In this paper, we are interested in computing high-quality approximate geodesics with user controllable accuracy. There are roughly two types of approaches for computing high-accuracy geodesics on triangle meshes. The graph-based methods, including saddle vertex graphs (SVG) [6] and discrete geodesic graphs (DGG) [7,8], are pre-computation methods, which construct source-independent sparse graphs whose complexity depends on the desired accuracy. Then given a source vertex, SVG and DGG can compute geodesic distances in empirically linear time. The pre-computation methods are highly desired in applications that require *frequent* geodesic computations. However, they are not suited for other applications that compute geodesics once or only a few times due to its high computational cost for graph construction and storage. The other type of approaches [9–11] modifies the existing exact algorithms to trade accuracy for window complexity. So far each of the popular exact algorithms, MMP, CH, and VTP, has an approximate counterpart. Although these approximate algorithms allow the user to control accuracy via some local parameter, none of them supports direct accuracy control. For example, the approximate VTP (A-VTP) algorithm [11], the state-of-the-art approximate algorithm, controls accuracy by a parameter λ, which is not directly linked to accuracy. During window propagation, whenever the wavefront radius is a multiple of λh (where h is the average edge length), A-VTP clears all the existing windows and uses vertices on the wavefront as sources to continue propagation. This window resetting strategy can effectively reduce window complexity, thereby improving the runtime performance. However, as the parameter λ varies among models, tuning them requires the user's experience and is often done in a trial-and-error manner. From the perspective of applications, it is highly desired to develop efficient approximate algorithms that allow the user to control accuracy directly.

Towards this goal, we develop a new method for computing approximate geodesic distances with high accuracy. Our method does not require any pre-computation and it allows direct accuracy control with expected results. The key ideas of our algorithm can be explained as follows. Since windows split after hitting vertices during window propagation, they become smaller and smaller when travelling further away from the source, and there are more and more windows produced. To obtain highly accurate distances with minimal computational cost, we need to control the travelling distance of each window. Specifically, we only allow a window to travel a necessary distance that can guarantee the expected accuracy and then stop propagating it after exceeding the distance limit.

Our algorithm essentially transforms the core ideas of DGG, which is a pre-computation method, into a direct approach of computing single-source-all-destinations (SSAD) distances. In contrast to DGG that builds *short* graph edges for *every* vertex, our algorithm constructs only the needed DGG edges from the given source. For efficient window propagation, we adopt VTP's window organization scheme [5], i.e., organizing windows using edge-based window lists and propagating them using vertex-oriented priority queue. Due to the nature of combining DGG and VTP, we call our method DGG-VTP. Our method has a single parameter ε, by which the user can directly control the desired accuracy of the results. Through extensive evaluation on both real-world and synthetic models, we show that our algorithm can compute highly accurate geodesic distances on large-scale mesh models efficiently. It is worth noting that due to robustness consideration, tiny windows with length less than a threshold (usually, between 10^{-7} and 10^{-6}) are discarded in the implementations of the existing exact algorithms, such as MMP [1], CH [2], FWP [4], VTP [5] and their variants. Our method with parameter $\varepsilon \in [10^{-7}, 10^{-6}]$ effectively produces results comparable to exact algorithms, such as VTP, for which single-precision floating point is used.[1] At the same time, our method is up to 35$\times$ faster than VTP. Thanks to the accuracy aware window propagation strategy, our method produces only a tiny fraction of windows as VTP does, thereby consuming much less memory than VTP. We also observe that VTP's runtime performance is sensitive to geometry, and it runs significantly slower on smoother meshes where there are less jaggedness and fine details. In contrast, DGG-VTP's performance is less sensitive to model smoothness than VTP. See Fig. 1 for two examples of our method. We also develop a parallel version of DGG-VTP on multi-core CPUs, which runs up to 60$\times$ faster than the parallel VTP algorithm [11] with comparable accuracy under single floating point precision setting.

Note that the parameter ε in DGG-VTP is not a guaranteed error bound. Instead, it acts as an intuitive guide for the user to set the desired accuracy. Extensive evaluations demonstrate the effectiveness of our accuracy control: given a parameter $\varepsilon \in [10^{-7}, 10^{-4}]$, 99% of the distances computed by our algorithm have error less than ε. Our method empirically produces $O\left(\frac{n}{\varepsilon^{0.23}}\right)$ windows and scales well to deal with large-scale models. The features of predicable accuracy and memory efficiency distinguish DGG-VTP from the existing approximation methods, and make it an alternative to exact methods in computing accurate geodesic distances on large-scale mesh models.

We organize the remaining of the paper as follows. Section 2 briefly reviews the relevant work on discrete geodesics and Section 3 presents the background knowledge. Section 4 documents the technical details of accuracy-aware window propagation for solving the SSAD geodesic problem, followed by experimental results and comparisons in Section 5. Finally, Section 6 concludes the paper and discusses future directions.

2. Related work

Computing geodesics on surfaces has been studied extensively in both computational geometry and computer graphics communities, resulting in a large body of literature. Due to space limit, we review only the most relevant works, which are window propagation-based methods and graph-based methods.

2.1. Window propagation methods

MMP is the first practical algorithm for computing exact geodesic distances on arbitrary polyhedral surfaces [1]. Given a triangle mesh with n faces, Mitchell et al. proved that there are $O(n)$ windows on each edge, hence $O(n^2)$ windows in total. Since MMP propagates windows using a priority queue, it runs in $O(n^2 \log n)$ time. Surazhsky et al. [9] observed that for triangle meshes with regular tessellations, each mesh edge has $O(\sqrt{n})$ windows on average, which is much smaller than the theoretical bound $O(n)$. Therefore, the MMP algorithm has an empirical $O(n^{1.5} \log n)$ time complexity. since MMP keeps all windows it produces in the entire window propagation procedure, it can compute exact geodesic distances and paths from the source vertex to *arbitrary* points on the surface (not necessarily vertices). However, the drawback is the quadratic space complexity $O(n^2)$, which diminishes its usage to large-scale models.

[1] Throughout the paper, we use the implementation of VTP by Qin et al. [5] with double precision floating points to compute ground truth.

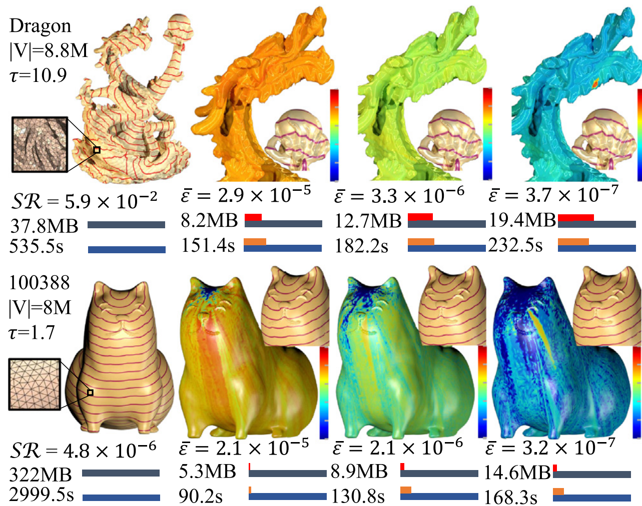


Fig. 1. Our method can compute high quality geodesic distances on triangle meshes. From left to right: exact VTP, our method with accuracy parameter $\varepsilon = 1 \times 10^{-4}, 1.4 \times 10^{-5}$ and 1.8×10^{-6}, respectively. The values below each figure are the mean error $\bar{\varepsilon}$, the peak memory and the running time. We visualize the errors using heat colour map with range from 10^{-8} (blue) to 10^{-3} (red). Our method with $\varepsilon = 10^{-6}$ yields results with accuracy comparable to the exact VTP, which discards windows with length less than 10^{-6}, but runs much faster and consumes much less memory. Furthermore, VTP's runtime performance highly depends on the smoothness of the input model. $\mathcal{SR}$ is the surface roughness measure, i.e., the lower the value of $\mathcal{SR}$, the smoother the surface is. For example, though both Model 100388 and Dragon have similar combinatorial complexity and degree of anisotropy, VTP runs almost 5 times slower on the former than the latter, which is smooth and lacks of fine details. In contrast, our method is far less sensitive to the smoothness of the input model. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

The CH algorithm [2], which is based on a different strategy, organizes windows in a hierarchical structure and propagates them in breadth-first-search order. To avoid exponential explosion, Chen and Han proposed a window filtering rule, called "one angle, one split": if two windows cover a vertex, at most one of them is allowed to generate two children. The CH algorithm has an $O(n^2)$ time complexity, and an $O(n)$ space complexity since it keeps only the windows that cover vertices. In spite of its theoretically better time complexity, the CH algorithm runs much slower than MMP on real-world models. Xin and Wang pointed out that the poor performance of CH is because the majority of windows it produces do not carry shortest distances [3], therefore are redundant. To improve the performance of the CH algorithm, they proposed a simple yet effective filter that prunes most of the redundant windows by checking distances with vertices. The improved CH algorithm (or ICH) adopts a priority queue to organize windows, hence runs in $O(n^2 \log n)$ time. Experimental results show that ICH runs much faster than CH on real-world meshes due to significantly fewer windows generated.

Since sorting windows by a priority queue is computationally expensive, Xu et al. [4] proposed fast wavefront propagation (FWP) that adopts bucket queue to organize windows in terms of their distances to source vertex. At one iteration, FWP removes the top bucket from the queue, and propagates all windows stored in it in a first-in-first-out order. Although FWP generates more windows than MMP and ICH, it runs 3 to 10 times faster than MMP and ICH on real-world meshes, due to constant-time cost of enqueue and dequeue, the most frequently used operations.

Qin et al. [5] proposed vertex-oriented triangle propagation (VTP). VTP distinguishes itself from the other window propagation algorithms in two aspects: first, it takes vertices instead of windows as elements of the priority queue; second, it groups windows on a half-edge into a list, and propagates window lists across triangles while pruning redundant windows using a set of 12 pruning rules. Since the vertex-oriented priority queue has only a size $O(n)$, which is significantly less than the window-oriented queue with size $O(n^2)$, VTP can effectively reduce the cost for queue operation without compromising the wavefront quality too much. Computational results show that VTP runs up to 2–4 times faster than FWP. Moreover, VTP stores only the windows on the wavefront, hence it takes less space than the other exact algorithms. Recently, Du et al. [11] developed approximate and parallel variants of VTP using a window resetting strategy.

The above mentioned algorithms are all running on either one or several CPU cores. There are also a few parallel algorithms [12, 13], which support parallel window propagation using GPU cores.

2.2. Graph-based methods

Graph-based methods convert the mesh-based geodesic problem into a graph-based shortest path problem, which can be solved using Dijkstra's algorithm [14]. The geodesic triangle unfolding method [15] constructs a dense graph, whereas saddle vertex graphs [6] and discrete geodesic graphs [7,8], are sparse graphs. The edges of SVG and DGG are direct geodesic paths that terminate at but not passing through mesh vertices. SVG and DGG differ in graph complexity and the strategy in selecting the relay vertices. Graph-based methods are closely related to window propagation methods, since the graph edges can be computed by applying any exact geodesic algorithm *locally*.

2.3. Other methods

Besides window propagation and graph-based methods, there are other techniques for computing discrete geodesics, such as PDE-based methods [16–21], optimization-based methods [22–26], curve shortening methods [27,28] and Steiner points based methods [29,30]. The PDE and optimization methods are flexible in that they work for a wide range of discrete and continuous domains, and can also deal with constraints and anisotropic metric. However, their accuracy are not comparable to window propagation methods and graph-based methods. The curve shortening methods can compute geodesic paths and loops in an exact manner, however they are not efficient when applied to computing single-source, all-destinations distances.

Recently, Trettner et al. [31] proposed geodesic source propagation (GSP) for computing approximate geodesic distances. In contrast to window propagation methods that assign each window a pseudo source, GSP stores a virtual source for each triangular face and adopts data-driven heuristic to determine visibility in propagation. It also uses dual queue and pre-computed cache coherent mesh layout to improve propagation speed.

3. Preliminaries

Since our method is built upon VTP [5] and DGG [8], we present their essential ideas in this section.

3.1. Sequential VTP

VTP [5] is the state-of-the-art algorithm for computing exact geodesic distances on triangle meshes. Similar to other exact methods, such as MMP [1,9], CH [2], ICH [10], FWP [4], it adopts a window data structure to encode geodesic paths from the source vertex to other vertices of the mesh. Imagine we place a light source at source vertex s, from which it shines lights to all directions. The light rays are restricted on the faces of the

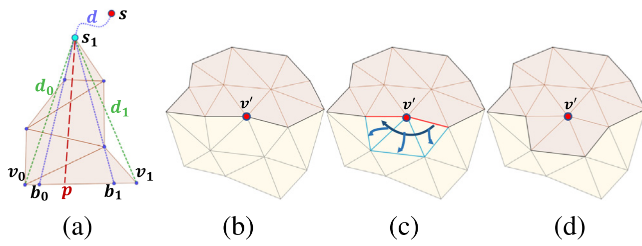


Fig. 2. Illustration of the window data structure and the enclosed propagation area in VTP. (a) The window data structure allows us to parameterize mesh edges for computing distances. (b) shows the current enclosed propagation area in brown and the region outside the wavefront in yellow. (c)–(d) Propagating windows from v' results in adding the triangles incident to v' to the enclosed area. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

mesh. Intuitively, a window groups the set of light rays that pass through the same sequence of triangles of the mesh.

Mitchel et al. [1] showed that a geodesic path is a sequence of connected line segments on the mesh that only turn upon meeting a saddle vertex (i.e. vertex with the sum of incident angles greater than 2π). Saddle vertices along a geodesic path are also called pseudo-sources. Mathematically speaking, a window w on an oriented half-edge (v_0, v_1) is an interval $(b_0, b_1) \in [v_0, v_1]$ that is associated with a saddle vertex s_1 (i.e., pseudo-source), geodesic distance d_0 (resp. d_1) from v_0 (resp. v_1) to s_1, and geodesic distance between pseudo-source s_1 and source s. If a window has no pseudo-source, d_0 (resp. d_1) is the distance between s and v_0 (resp. v_1). Using the window data structure, we can parameterize the half edge (v_0, v_1) and compute the geodesic distance between s and an arbitrary point $p \in [b_0, b_1]$ as

$$d(s, p) = d(s, s_1) + \|s_1 p\|,$$

where $\|s_1 p\|$ denotes the Euclidean distance between s_1 and p. See Fig. 2(a).

The existing exact methods differ in the ways of window organization and propagation. The VTP algorithm organizes windows in a vertex-oriented manner. Specially, VTP creates an *enclosed propagation area*, which is the region inside the wavefront and has been already swept by the windows created. At the beginning, VTP initializes the enclosed propagation area as the triangles surrounding the source vertex s and creates a window for each edge opposite s. At each iteration, VTP takes the nearest vertex to s, say v', and extends its propagation area by adding the triangles incident to v' (see Fig. 2(b)–(d)). If v' is a saddle vertex, it creates new windows to the surrounding area of v'. It then propagates all the windows that lie in the newly-added triangles until they leave the enclosed propagation area iteratively.

VTP adopts a set of rules to detect redundant windows and trim windows. The ICH pruning rule [3] determines whether a window w, when entering a triangular face f, is useful by checking distances from source s to the three vertices of f. The "one-angle, one-split" rule [2] is to prevent multiple windows splitting upon meeting a vertex. It states that among the windows coming from a triangle edge to a vertex opposite the edge, only one can split into two child windows. The "one-angle, two-sides" rule [5] is a generalization of "one-angle, one-split" and it can prune more windows based on their positions on the edge by looking into the window that gives the shortest distance to a vertex. We observe that the ICH rule and the "one-angle, two-sides" rule are more effective than the rest of the rules. Therefore, we only adopt these two rules in our algorithm.

VTP also groups windows associated to a triangle edge into two lists, depending on their directions upon hitting that edge. The window lists provide a rough order for the windows based on their positions on the triangle edge, and enable VTP to prune windows efficiently. At each iteration, VTP propagates windows from 1 window list and puts the resulting windows to the half-edge window list where they lie on by using the rules in [5] to ensure the windows have some ordering along the half-edge.

3.2. Parallel VTP

The VTP algorithm processes only one vertex at a time by expanding the propagation area associated to that vertex. The parallel VTP (PVTP) algorithm, developed by Du et al. [11], can handle multiple vertices simultaneously in each iteration. Since priority queue is intrinsically sequential, PVTP replaces it by a bucket data structure $\mathcal{B}$ [4]. It also utilizes a pair of bucket queues, $\mathcal{A}_{\text{from}}$ and $\mathcal{A}_{\text{to}}$, to facilitate parallel propagation of window lists. PVTP propagates K window lists in an iteration, with K being an adaptive constant.

At each iteration, PVTP selects the top k_v vertices from buckets $\mathcal{B}$ and extends the wavefront using the one-ring neighbouring triangles incident to the chosen vertices. In detail, it proceeds as follows: First, select K window lists around the k_v vertices and push them to $\mathcal{A}_{\text{from}}$. Second, propagate and prune the selected K window lists using all VTP pruning rules in a parallel manner. It creates two buffers for each window list and stores their child window lists into their own temporary buffers. Third, merge the updated window lists in their buffers using the rules in [5], and push them into the other array $\mathcal{A}_{\text{to}}$. Push also the updated vertices into the bucket queue $\mathcal{B}$ sequentially. Fourth, swap arrays of $\mathcal{A}_{\text{from}}$ and $\mathcal{A}_{\text{to}}$. If $\mathcal{A}_{\text{from}}$ is not empty, go to the second step; otherwise go to the first step. Repeat the above steps until all vertices are processed.

3.3. Discrete geodesic graphs

DGG is a graph-based approximate algorithm for the discrete geodesic problem. Its key idea is to partition a long geodesic path γ into a sequence of short and direct geodesic paths that are nearly parallel to γ. In this way, DGG can be viewed as a divide and conquer method. The DGG method consists of two steps. In the pre-computation step, it constructs a sparse graph that covers all mesh vertices and each graph edge is a short and direct geodesic path. In the query step, given a source vertex, it applies a graph-based shortest path algorithm (e.g., Dijkstra's algorithm) to compute shortest paths and distances on the fly.

So far, there are two methods [7,8] for constructing DGGs and answering distance query. In this paper, we adopt Adikusuma et al.'s algorithm [8], which is more computationally efficient than Wang et al.'s method [7]. There are two key components in Adikusuma et al.'s algorithm [8], which are CH-relay vertices and a distance bound assigned to each window.

Theorem 1 ([8]). *Assume only the CH's one-angle-one-split rule is used to prune windows during window propagation. For any direct window $w = (e_{ij}, b_0, b_1, d_0, d_1, 0)$ associated with the half edge e_{ij}, the direct geodesic path $\gamma(s, b_i)$, $i \in 0, 1$, crosses at least one vertex, which is called a CH-relay vertex (see Fig. 3).*

Using CH-relay vertices, one can bound the distance that a window needs to travel using the following theorem.

Theorem 2 ([8]). *Let s be the source point and w a direct window with CH-relay vertices c_0 and c_1, and $\angle c_0 s c_1 < 90°$. Let $l_i = d_g(s, c_i)$, $i \in 0, 1$. Without loss of generality, assume $l_0 \geq l_1$. Extend sc_1 to sc_2 so that $\|sc_2\| = l_0$. Denote by $d = \|c_0 c_2\|$, $\alpha = \frac{l_1}{l_0}$ and*

$$\mu = \frac{1}{1-\alpha} - \sqrt{\frac{1}{(1-\alpha)^2} - \frac{1}{1-\alpha} - \frac{2\varepsilon l_0 (l_0 - d)}{d^2}}.$$

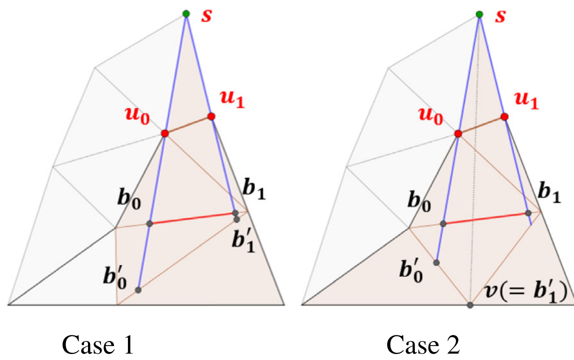


Fig. 3. Each direct window w has two CH-relay vertices on its boundary lines $\gamma(s, b_0)$ and $\gamma(s, b_1)$. Initially, when a window $[u_0, u_1]$ is first created from the source s, it is easy to see the two endpoints u_0 and u_1 are the CH-relay vertices. We consider two cases during window propagation. In Case 1, the window does not light any vertex and thereby does not split. u_0 and u_1 are the CH-relay vertices for the child windows. In Case 2, the window lights a vertex v and splits into two windows. v is the CH-relay vertex for the child windows.

Set

$$\beta = \begin{cases} 0.5, & \text{if } \alpha = 1; \\ \mu, & \text{if } \alpha < 1 \text{ and } \mu \in (0, 1); \\ 1.0, & \text{otherwise.} \end{cases}$$

For any geodesic path $\gamma(s, t)$ passing through w with length

$$d_g(s, t) \geq l_0 + \frac{((1-\beta)d)^2}{2\varepsilon(l_0 - d)} (\triangleq d_{max}), \tag{1}$$

it can be approximated by a relay vertex $u \in F(\gamma(s, t))$ with approximation ratio $1 + \varepsilon$.

In Adikusuma et al.'s algorithm, windows are propagated from each mesh vertex in order to construct graph edges. A window stops propagating if it either hits a vertex or its travelling distance exceeds the bound $d_{\max}$. Therefore, all windows in their algorithm are *direct* windows in the sense that they do not use saddle vertices as pseudo-source.

Given an accuracy parameter ε, the DGG has an average degree $O(1/\sqrt{\varepsilon})$ [7] and can be constructed in empirical $O\left(\frac{n}{\varepsilon^{0.75}} \log \frac{1}{\varepsilon}\right)$ time [8]. Note that the accuracy control parameter ε in DGG is not an error bound, but a rough estimation on the error produced by the algorithm. Despite of lack of theoretical guarantee, Adikusuma et al. showed that the accuracy control in DGG works practically well even for meshes with extremely high degree of anisotropy. Given meshes with more than 1 million vertices, Adikusuma et al. recommend parameter $\varepsilon \in [10^{-4}, 10^{-3}]$, for which DGG can balance graph construction, accuracy and runtime performance well [8].

4. Method

4.1. Motivation & intuition

DGG is a pre-computation method that builds *short* direct geodesic paths emanating from *every* vertex of the input mesh. Intuitively speaking, DGG approximates a long exact geodesic path using several short segments that are nearly parallel to it (see Fig. 4). After obtaining the graph, we can compute SSAD using a *modified* version of Dijkstra's algorithm. Instead of scanning all edges around a vertex, say v, as in the conventional Dijkstra's search, we only need to examine the edges inside the fan-shaped region of v that are approximately parallel to the geodesic path reaching v. Using this strategy, after doing SSAD search, we end up with paths that are approximating the exact geodesic paths

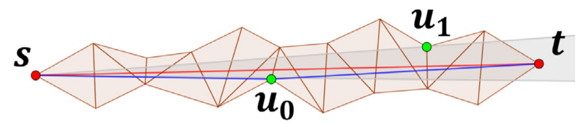


Fig. 4. When window w hits a vertex t, we can approximate a long geodesic path $\gamma(s, t)$ using its CH-relay vertices u_0 and u_1. In this example, $d_g(s, t) \approx d_g(s, u_0)+d_g(u_0, t)$. Notice that the two short geodesic paths $\gamma(s, u_0)$ and $\gamma(u_0, t)$ are nearly parallel to $\gamma(s, t)$.

with several segments of nearly parallel geodesics. It is clear that the modified Dijkstra's algorithm yields a tree which consists of only a small subset of the DGG edges.

To adapt the pre-computation-based DGG for a direct approach of computing SSAD, we construct only the required DGG edges from the given source s to all other vertices, while trying to avoid computing useless edges as much as possible. Intuitively speaking, we want to compute a "DGG" for s on the fly such that its graph edges can reach *all* vertices of the input mesh, while keeping its size as small as possible. Each time when a window w meets a vertex, we mark it as the pseudo-source of w's child windows. Therefore, **the pseudo sources in our setting include not only saddle vertices, but also Euclidean vertices and spherical vertices.** The seemingly minor change allows us to construct DGG edges from s to *all* other vertices, since every time a vertex becomes a pseudo-source, DGG edges are created from it.

To control graph size, we must avoid constructing useless DGG edges. Towards this goal, we adopt two strategies. First, since DGG edges are generated in the window propagation process, we need to monitor windows closely. Similar to [8], we assign a distance bound to each window propagated from s and update it every time when the window hits a vertex. Second, we apply window pruning rules to detect redundant windows and prevent them from propagating.

4.2. Accuracy-controllable window propagation

The key idea behind DGG-VTP is to modify the VTP algorithm by borrowing ideas from DGG. Below we first outline the major modifications to VTP before explaining them.

(1) Upon meeting any vertex in window propagation, we make it as a pseudo-source regardless of the vertex being saddle or not.
(2) From a pseudo-source v, child windows are propagated only inside a fan-shaped region, which is induced by the direction of the incident ray.
(3) When a window w meets a vertex v, we assign v as one of the CH-relay vertices of w on its left/right side. We also associate w a distance bound $d_{\max}$ to indicate how far it should travel to get the desired accuracy. This distance bound is calculated using Theorem 2 each time a window meets a vertex.
(4) Among the set of pruning rules used in VTP, we adopt only the ICH rule and the "one-angle two-sides" rule.
(5) We stop propagating a window w when its associated distance bound is reached.

Window data structure. We adopt the window data structure used in [8]. Besides the usual information stored in a window w in VTP (Fig. 2(a)), we also assign w two CH-relay vertices u_0 and u_1, as well as a distance bound $d_{\max}$.

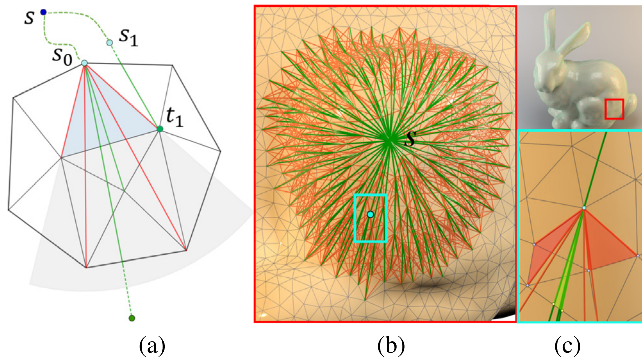


Fig. 5. Fan-shaped propagation region in DGG-VTP. (a) Consider a pseudo-vertex s_0, from which windows are propagated in a fan-shaped region (coloured in grey). We pick the blue triangle which is inside the fan-shape region. The useful and useless edges appear in green and red, respectively. The useful edges are approximately parallel to geodesic $\gamma(s, s_0)$. (b) Taking the central vertex s as the source, we compute the DGG edges (shown in green) using the fan-shaped pruning strategy. To avoid visual clutter, we use a relatively low accuracy parameter $\varepsilon = 2.5 \times 10^{-3}$ and stop the computation when 5% of total mesh vertex are reached. We also show part of the DGG constructed by Adikusuma et al.'s algorithm in red. Note that we only need the green edges in computing SSAD from source s and the red edges are completely useless. This demonstrates the effectiveness of our fan-shaped pruning strategy and the compactness of the graph generated by our method. (c) The close-up view shows the useful and useless windows in green and red respectively. In this figure, the useful window contributes to a long geodesic that is almost parallel to the incident ray. Since the useless windows are pruned, they cannot produce long geodesics that would yield redundant DGG edges. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Window propagation. Each time our algorithm generates a new window from the source s or a pseudo-source, it computes its CH-relay vertices and distance bound. Moreover, if a window meets a vertex, it also triggers the event of updating CH-relay vertices and distance bound. The distance bound is the key to our approximation quality and efficiency. We stop a window from propagating when its distance bound is reached. In this manner, we only construct DGG edges for a user-specified accuracy level. Another important difference of window propagation between our method and VTP is that we propagate windows from each pseudo-source only in a fan-shaped region whereas VTP propagates them in all directions. Due to this difference, we store in each vertex a directional information of the geodesic reaching it. When a window meets a vertex, say v_j, the direction of the incident ray is from the previous pseudo source to v_j. The fan-shaped region at v_j is defined as the region bounded by two directions from v_j, each of which forms an incident angle $\pi - \sqrt{\varepsilon}$ with the incoming direction. The geometric requirement of fan- shaped regions is to ensure that the constructed DGG edges from a pseudo-source is approximately parallel to the geodesic reaching it. To ease implementation, we simply construct the fan-shaped region as the whole triangle(s) that is/are overlapping with the region (see Fig. 5(b)).

Window pruning. Besides the above-mentioned fan-shaped pruning for avoid computing useless edges, we also borrow VTP's window pruning rules to reduce the total number of windows generated. We observe that not all VTP rules play equal role in window pruning. Thus, for simplicity, we decide to use the two most effective rules, namely, the ICH pruning rule and the "one-angle two-sides" rule.

4.3. Sequential algorithm

We present the pseudo code of sequential DGG-VTP in Algorithm 1. Initially, we set the vertex-oriented priority queue $\mathcal{Q}$, the FIFO queue $\mathcal{F}$ for window lists, and the propagation region R empty set using the source vertex s and its 1-ring triangles. We also set the initial CH-relay vertices for each window and compute the initial distance bounds and the directional information. Then we insert the vertices to the priority queue $\mathcal{Q}$ and update the enclosed propagation area R. See lines 1–8.

We follow the VTP framework to propagate window lists. If the priority queue $\mathcal{Q}$ is not empty, we pop the top vertex v (which has the shortest distance to s) from queue $\mathcal{Q}$ (line 10). Then, we compute the fan-shaped region around v and propagate windows in this region (lines 11–12). After that, we update the enclosed propagation region R and push any windows list surrounding vertex v to $\mathcal{F}$ (lines 13–15).

We propagate the windows lists until all of them leave the enclosed propagation area R (lines 16–28). Specifically, we prune windows using the ICH rule and the "one-angle, two-sides" rule (line 18). We update a window's CH-relay vertices and distance

Algorithm 1 Sequential DGG-VTP

Require: A manifold triangle mesh $M = (V, E, F)$, source vertex s, and the desired accuracy $\varepsilon > 0$.
Ensure: The approximate geodesic distances from s to all vertices.

```
1:  Initialize the vertex-oriented priority queue Q ← Ø, the FIFO
    queue F ← Ø and the propagation region R ← Ø
2:  for each edge e_ij = (v_i, v_j) opposite s do
3:      Create a window w with CH-relay vertices v_i and v_j
4:      Compute the distance bound for w
5:      Insert w to the window list of e_ij
6:      Insert v_i to Q sorted by distances
7:      R ← R ⋃ △s v_i v_j
8:  end for
9:  while Q ≠ Ø do
10:     v = Q.pop()
11:     Find the set of triangles T which are incident to v and are
    overlapping with v's fan-shaped region
12:     Create a window for each triangle in T and initialize its
    initial CH-relay vertices and distance bound
13:     Store the windows in the corresponding windows lists,
    update the distance for the neighbouring vertices of v, and
    push them into Q
14:     Update R using the triangles incident to v
15:     Push the newly created window lists to F if they are inside
    R and adjacent or opposite v
16:     while F ≠ Ø do
17:         wl = F.pop()
18:         Prune wl using each window's distance bound, the ICH
    rule and the "one-angle, two-sides" rule
19:         for each window w ∈ wl do
20:             Propagate w
21:             if w meets a vertex v then
22:                 Update w's CH-relay vertices and distance
    bounds
23:                 Update the distance and directional information
    for v
24:                 Push v into Q
25:             end if
26:         end for
27:         Merge the newly generated window lists and push
    them to F if they are inside propagation area R
28:     end while
29: end while
```

bound if it hits a vertex (lines 22–24). When a window has travelled a distance longer than its associated distance bound, we stop propagating it immediately. As a result, no child windows will be created.

When all windows have finally left the enclosed propagation area, we move on to pop out another vertex v from $\mathcal{Q}$ and repeat the process above until all vertices have been visited, thereby becoming pseudo-sources.

4.4. Parallel algorithm

We parallelize DGG-VTP using the PVTP framework [11]. The main technical challenge in parallelization is to ensure that there are sufficient operations to be carried out in each iteration so that the parallelization overhead does not become a significant part. We observe that the speedup factor of PAVTP with 4 threads [11] compared to AVTP often drops to below 2× when the user specifies a relatively low accuracy parameter using small value of λ. Notice that PAVTP parallelizes only the operations related to window list propagation. The other operations such as those related bucket queue, propagating windows from pseudo-sources and merging window lists are all sequential. With a lower value of λ, AVTP propagates much fewer windows. As a result, the benefit of parallelization is less significant and the speedup factor becomes smaller.

We adopt the following strategy to overcome the challenge in PVTP and PAVTP. We observe that propagating windows from pseudo-sources consists of highly independent operations. This is because each vertex is surrounded by different window lists and the resulting windows created from pseudo-sources are inserted to independent sets of window lists. In DGG-VTP, creating windows from pseudo-source is expensive because we need to update the CH-relay vertices and distance bound for each of the new windows, not to mention creating the fan-shaped region. To improve the performance of the parallel algorithm, we decide to parallelize this operation.

We present the pseudo-code of parallel DGG-VTP in Algorithm 2. As priority queue is intrinsically sequential, we replace $\mathcal{Q}$ by a bucket queue $\mathcal{B}$. At each iteration, we select k_v vertices from the bucket queue (line 6), where k_v is a self-adjustable parameter. We use the same heuristic for updating k_v in our algorithm as in PVTP [11]. We update the enclosed propagation region R (line 7), and then push K window lists surrounding all the k_v vertices to a temporary array $\mathcal{A}_{\text{from}}$ (line 8). Since we take all vertices as pseudo-sources in our algorithm, we create windows from these k_v pseudo-sources in parallel (lines 9–11). We also update the distances to the vertices surrounding the previously selected k_v vertices in a sequential manner (line 12).

After that, we propagate all the selected K window lists in parallel until they left the enclosed area R (lines 13–20). Each window list wl generates two child lists, which are on the opposite side of the edge of the triangle where wl is located. We merge the resulting window lists (line 17) and place the merged lists in a temporary array $\mathcal{A}_{\text{to}}$ if they are located inside R (line 18). At the end of the iteration, we swap the arrays $\mathcal{A}_{\text{to}}$ and $\mathcal{A}_{\text{from}}$ (line 19).

Same as the sequential counterpart, we only use the ICH rule and the "one-angle, two-sides" rule to prune windows (lines 14–15). If a window has reached its distance bound, we stop its propagation. We also update the CH-relay vertices and distance bound of a window when it hits a vertex. We also need to update the distance of vertices and push the vertex to $\mathcal{B}$ if any window can give a shorter distances to these vertices (line 16).

When all windows have finally left the enclosed propagation area, we continue to select k_v vertices from the bucket queue $\mathcal{B}$ and repeat the above process. The algorithm terminates when the bucket queue $\mathcal{B}$ is empty, i.e., all vertices have been processed.

Remark. Parallel DGG-VTP has more operations carried out in parallel than PVTP and PAVTP [11], thereby it has better performance gain. However, our algorithm is still not fully parallel since the operations for merging window lists, updating vertex distances and placing the resulting windows list to array $\mathcal{A}_{\text{to}}$ are sequential. We leave further improvement as future work.

Algorithm 2 Parallel DGG-VTP

Require: A manifold triangle mesh $M = (V, E, F)$, source s, the accuracy parameter $\varepsilon > 0$, and the number of threads n_{thd}
Ensure: The approximate distances from s to all vertices

1: Create windows to all surrounding triangles from source s and insert them to the corresponding windows list
2: Update distances to triangle vertices surrounding s and store the geodesic directions when it hits these vertices inside each vertex data structure
3: Insert these vertices data structure to bucket queue $\mathcal{B}$ sorted by distances
4: Update propagation region R using the one ring triangles area around s
5: **while** $\mathcal{B}$ is not empty **do**
6: Pop k_v vertices from $\mathcal{B}$
7: Update the enclosed propagation region R using the one ring triangles surrounding the k_v vertices.
8: Push all K windows lists surrounding vertex k_v to array $\mathcal{A}_{from}$
9: parallel Compute set of triangles $\mathcal{T}$ around v where v's fan-shaped region has some overlaps
10: parallel Propagate windows from v to the triangles in $\mathcal{T}$ and store their initial CH-relay vertex
11: parallel Store these windows to its corresponding windows lists
12: Update the vertex distance surrounding all k_v vertices, while inserting them to $\mathcal{B}$
13: **while** $\mathcal{A}_{from}$ is not empty **do**
14: parallel Prune the windows in the window lists using the ICH rule and the distance bound of each window
15: parallel Propagate and prune all window lists using the "one-angle, two-sides" rule while updating their CH-relay vertices and distance bound; Store the resulting window lists and updated vertices in their own independent buffers
16: Push the updated vertices into $\mathcal{B}$
17: Merge the newly created window lists
18: Push the merged lists to $\mathcal{A}_{to}$ if they are inside propagation area R
19: Swap $\mathcal{A}_{from}$ and $\mathcal{A}_{to}$
20: **end while**
21: **end while**

5. Results & comparisons

We implemented our algorithm in C++ and made the code available at https://github.com/GeodesicGraph/DGG-VTP. Since our focus is on high accuracy geodesic computation, we evaluated it on large-scale meshes with up to 10 million faces and compared it to the existing exact algorithms. We randomly chose 95 models from the Thingi10K dataset and subdivided them to reach the desired combinatorial complexity. Most of the models have irregular tessellations.

We conducted 2 separate tests for DGG-VTP. The first one is to compare it with AVTP in the way that both algorithms produce comparable accuracy. This test was done on Intel Xeon E5-2643

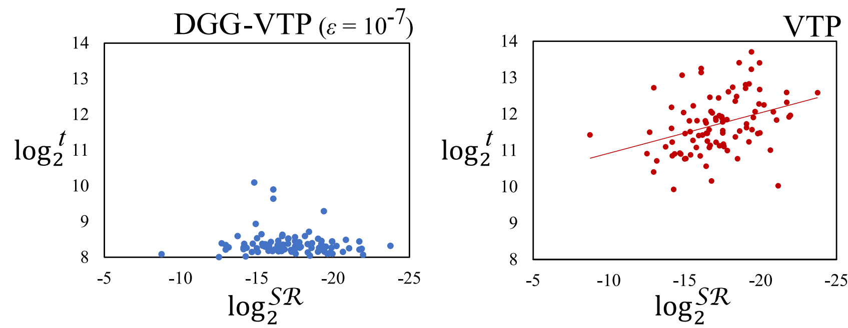


Fig. 6. Surface roughness influence to runtime. In the log–log chart above, we present the correlation between the surface roughness parameter $\mathcal{SR}$ and runtime t measured in seconds. DGG-VTP's times not closely related to surface roughness, while VTP's runtime shows a slightly moderate inverse correlation to $\mathcal{SR}$.

with 48 GB RAM using a smaller testing dataset that has varying meshes size from 5–15M vertices and the degree of anisotropy up to 150.

The second test is to compare DGG-VTP with VTP [5] and PVTP [11] on the 95 randomly chosen models from Thingi10K dataset described above. This test was carried out on a workstation with an Intel Core i5-7500 CPU and 16 GB RAM. Setting the accuracy parameter $\varepsilon = 10^{-5}$, DGG-VTP runs up to 75× faster than VTP on the large-scale meshes. With $\varepsilon = 10^{-7}$, DGG-VTP produces results with accuracy comparable to exact VTP using single precision floating point setting with 23 bits of precision. Given this setting, we see that our algorithm can improve the speed of VTP by a factor of 35× in single threaded setting. In multi-threaded setting, e.g., 4 threads, we can achieve up to 60× speedup compared to PVTP [11]. Moreover, we show that our algorithm is capable of giving more than 99% of distances computed to have error less than the desired accuracy control parameter in high accuracy settings of $\varepsilon \in \{10^{-4}, 10^{-5}, 10^{-6}, 10^{-7}\}$. This experiment demonstrates that our algorithm is an alternative for highly accurate geodesic computation other than the VTP algorithm on large scale meshes.

Surface roughness influence. One important observation regarding VTP's performance stems from our comparison with DGG-VTP. We observed that VTP tends to run slower on smooth meshes in which there are less roughness and jaggedness, whereas DGG-VTP's performance is very subtle. We used the variance of Gaussian curvature to measure surface roughness. Specifically, we randomly chose 1% vertices of the vertex set V and let V' be the sample set. Denote by $G(v)$ the discrete Gaussian curvature for vertex v, and $N(v)$ the set of 500 nearest neighbours to v. Then, we computed the measure of surface roughness for mesh M as

$$\mathcal{SR} = \frac{1}{|V'|}\sum_{v\in V'} \mathrm{Var}(G(p)), \qquad p \in N(v).$$

As Fig. 6 shows, surface roughness affects VTP's runtime performance moderately. Taking the logarithm with base 2 for the quantity $\mathcal{SR}$ and the VTP runtime, we can see a slightly moderate negative correlation between $\mathcal{SR}$ and VTP's runtime. The Pearson correlation between the two quantities is −0.35, indicating VTP takes longer time on smooth models than on rough models. In contrast, we see that DGG-VTP runtime performance is insensitive to surface roughness. Due to space limit, we only reported DGG-VTP's runtime correlation with surface roughness for $\varepsilon = 10^{-7}$. We observed similar trend with other values of ε.

The reason for the influence of surface roughness to VTP's performance can be explained as follows. Intuitively, any window produced from the source vertex becomes narrower as it travels farther. Narrow windows mean more windows are required to cover the perimeter of wavefront. However, as the surface becomes rougher and more jagged, these narrow windows are more likely to collide with each other, resulting in fewer windows being created. Besides, there are more wide windows created from saddle vertices that can cover more wavefront perimeter.

DGG-VTP differs from VTP in the way it explicitly prunes narrow windows from propagating too far using Adikusuma et al.'s distance bound, which limits windows to be created on smooth surfaces. As a result, the runtime is not affected by surface roughness. Indeed, we see negligible correlation between DGG-VTP's runtime and $\mathcal{SR}$ with Pearson Correlation of 0.11.

Memory efficiency. Due to the better error control, our algorithm needs to propagate much fewer windows than AVTP to reach the same accuracy level. As a result, we can see significant memory saving in our algorithm as it scales with higher accuracy. In Table 1, we can see 2-16× improvement for $\varepsilon \in \{10^{-4}, 1.4 \times 10^{-5}\}$ in memory usage as compared to AVTP. For $\varepsilon = 1.8 \times 10^{-6}$, we can see memory saving of up to 20×.

We compared the memory usage of DGG-VTP to VTP in Fig. 7. We observed that given a high accuracy parameter $\varepsilon = 10^{-5}$, DGG-VTP can save memory by up to 2 orders of magnitude, with geometric mean of 30×. Using single precision floating point accuracy of $\varepsilon = 10^{-7}$, we have memory saving of up to 30× with geometric mean of 9×. This confirms that DGG-VTP with $\varepsilon = 10^{-7}$ is an efficient alternative to exact methods.

Runtime performance. Since DGG-VTP propagates fewer number of windows than AVTP, we can also see speedup in runtime performance here. In Table 1, we see 2-22× speedup as compared with AVTP for $\varepsilon = 10^{-4}, 1.4 \times 10^{-5}$, and 1.8×10^{-6} respectively. In Fig. 7, we compare DGG-VTP to VTP algorithm using our diverse testing dataset of 94 remeshed random models from Thingi10K, we see up to 75× speedup given $\varepsilon = 10^{-5}$. Meanwhile, given single precision floating point accuracy of $\varepsilon = 10^{-7}$, we see performance improvement of up to 35×. The geometric mean of speedup for $\varepsilon = 10^{-5}$ and 10^{-7} is 20× and 10× respectively. Moreover, in parallel setting using 4 threads, DGG-VTP is able to outperform VTP by up to 60× when computing using single floating point precision accuracy of 10^{-7}. This effectively suggests DGG-VTP can be a great alternative to VTP when exactness is only required up to the limit of single floating point accuracy. These results justify DGG-VTP significant advantage with respect to both VTP and AVTP.

Time and space complexities. Our algorithm scales well with higher accuracy and larger scale meshes. Experiments show that on DGG-VTP empirically generates $O\left(\frac{n}{\varepsilon^{0.23}}\right)$ windows (see Fig. 8(a)). The runtime for window propagation, excluding overheads for priority queue and window list operations, also scales with $O\left(\frac{n}{\varepsilon^{0.23}}\right)$ as shown in Fig. 8(b). As we can see, given this

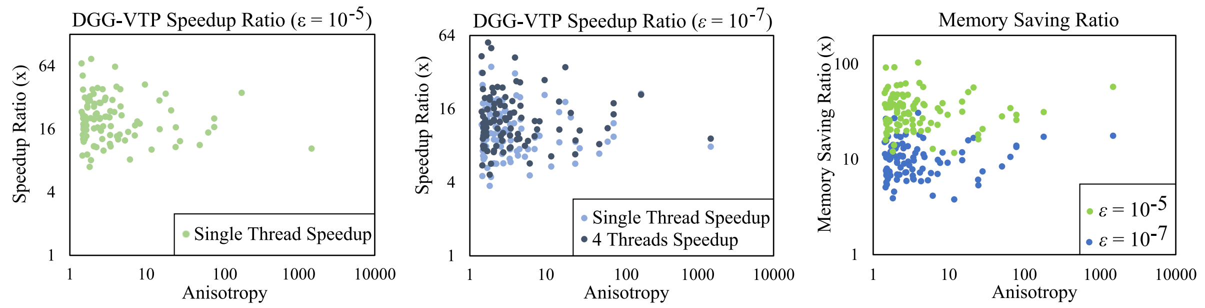


Fig. 7. DGG-VTP performance on 95 large-scale meshes with around 10M vertices and varying anisotropy. We use VTP as the baseline and test DGG-VTP with $\varepsilon = 10^{-5}$ and 10^{-7} respectively.

Table 1
Comparison with A-VTP and VTP on models with varying degree of anisotropy. We manually tune A-VTP's parameter λ so that A-VTP produces results with similar accuracy as DGG-VTP. T: running time (seconds); RAM: peak memory consumption (MB); s.u.: speed-up factor; m.r.: memory ratio (i.e., the peak memory of A-VTP or VTP to that of our method).

Model	DGG-VTP				A-VTP				vs. A-VTP		vs. VTP		w/ all pruning rules		
$(\tau, \|V\|)$	ε	T	$\bar{\varepsilon}/\varepsilon$	RAM	λ	T	$\bar{\varepsilon}/\varepsilon$	RAM	m.r.	s.u.	s.u.	m.r.	T	$\bar{\varepsilon}/\varepsilon$	RAM
100388	10^{-4}	90	0.21	5.3	56	189.6	0.21	28.1	5.30	2.11	33.3	60.79	101.7	0.21	5.09
Cat	1.4×10^{-5}	131	0.15	8.9	292	707	0.15	114	12.81	5.40	22.9	36.20	135.7	0.15	8.4
(1.62, 8M)	1.8×10^{-6}	168	0.18	14.6	1200	1733	0.18	315	21.58	10.32	17.8	22.07	183	0.18	13.8
101634	10^{-4}	190.5	0.25	7.48	44	397.5	0.28	31	4.14	2.09	15.7	34.97	216	0.25	7.2
Sphere	1.4×10^{-5}	246	0.15	12.5	340	1374	0.15	195.4	15.63	5.59	12.2	20.93	305	0.15	11.9
(1.51, 15M)	1.8×10^{-6}	381	0.2	20.5	1100	2865	0.19	252.3	12.31	7.52	7.8	12.76	382	0.2	19.5
106619	10^{-4}	94.1	0.31	7.2	56	175	0.3	34	4.72	1.86	18.6	46.17	105	0.31	6.9
Terrain	1.4×10^{-5}	114.8	0.22	11.8	388	579	0.23	137	11.61	5.04	15.2	28.17	129	0.22	11.2
(1.46, 7M)	1.8×10^{-6}	169	0.25	18.8	1300	1421	0.15	369	19.63	8.41	10.3	17.68	193	0.25	17.6
264708	10^{-4}	98	0.18	5.99	44	201.5	0.21	21.4	3.57	2.06	54.9	100.5	118	0.18	5.83
Heart	1.4×10^{-5}	127.8	0.15	8.87	188	517.6	0.16	84.2	9.49	4.05	42.1	67.86	154	0.15	8.46
(1.74, 7M)	1.8×10^{-6}	183	0.16	14.5	700	1609	0.18	311	21.45	8.79	29.4	41.51	202	0.16	13.8
Golfball	10^{-4}	171.3	0.19	7.04	44	275.5	0.23	29	4.12	1.61	20.5	35.92	194	0.19	6.65
(1.7, 13M)	1.4×10^{-5}	233	0.15	11.9	220	835.4	0.17	125.3	10.53	3.59	15.1	21.25	231	0.15	11.2
	1.8×10^{-6}	315	0.17	19.8	900	2202.3	0.18	246	12.42	6.99	11.1	12.77	336	0.17	18.5
Lucy	10^{-4}	213	0.36	12.3	44	368.5	0.29	42.5	3.46	1.73	9.5	13.50	262	0.36	11.8
(1.01, 16M)	1.4×10^{-5}	327	0.17	20.2	244	891	0.18	143	7.08	2.72	6.2	8.22	339	0.17	19.2
	1.8×10^{-6}	370	0.2	32	1200	1496	0.21	184	5.75	4.04	5.5	5.19	445	0.2	29.9
1036429	10^{-4}	184	0.23	9.2	88	1224	0.25	85	6.7	9.2	11.5	14.6	214	0.23	8.9
(145, 8M)	1.4 x 10^{-5}	212	0.19	12.2	335	1883	0.21	138	8.9	11.3	10.0	11.0	259	0.19	11.6
	1.8 x 10^{-6}	272	0.22	18	672	2085	0.13	148	7.7	8.2	7.8	7.5	321	0.22	16.9
1036473	10^{-4}	366	0.2	14.6	52	1028	0.19	74	2.8	5.1	18.2	38.6	367	0.2	13.9
(34, 13M)	1.4 x 10^{-5}	431	0.19	20	245	2855	0.15	259	6.6	13.0	23.8	28.2	484	0.19	18.3
	1.8 x 10^{-6}	550	0.19	28.1	1016	7702	0.17	626	14.0	22.3	18.6	20.0	577	0.19	25.4
1158269	10^{-4}	580	0.31	40	136	4022	0.3	252	6.9	6.3	12.9	9.8	652	0.31	39.9
(74, 12M)	1.4 x 10^{-5}	686	0.22	44.2	470	5359	0.19	360	7.8	8.1	10.9	8.8	814	0.22	43.4
	1.8 x 10^{-6}	945	0.25	51	1016	7555	0.24	434	8.0	8.5	7.9	7.6	961	0.25	48.5
1336194	10^{-4}	656	0.23	17.9	52	2297	0.26	107	3.5	6.0	26.7	35.8	697	0.23	16.5
(72, 17M)	1.4 x 10^{-5}	790	0.2	23	245	5890	0.21	333	7.5	14.5	21.7	27.9	873	0.2	19.95
	1.8 x 10^{-6}	1023	0.34	30.9	672	9308	0.36	613	9.1	19.8	16.8	20.7	1095	0.34	25.8
58874	10^{-4}	118	0.26	5.5	28	200	0.25	24	1.7	4.4	38.3	118.4	115	0.2	5.3
(3.8, 10M)	1.4 x 10^{-5}	144	0.2	9.1	136	593	0.19	74	4.1	8.1	31.4	71.5	156	0.2	8.8
	1.8 x 10^{-6}	195	0.2	15.1	470	1751	0.17	243	9.0	16.1	23.2	43.1	210	0.2	14.4
63785	10^{-4}	294	0.31	16.2	88	669	0.3	88.1	2.3	5.4	11.3	6.5	303	0.31	15.3
(2.0, 19M)	1.4 x 10^{-5}	345	0.26	25.4	672	1110	0.28	113	3.2	4.4	3.4	4.1	396	0.26	23.6

window complexity, 2.5× increase in the number of windows to be propagated will create 100× increase in geodesic distance accuracy. Adding the cost of priority queue and window list operations, the total time complexity is $O(n \log n + \frac{n}{\varepsilon^{0.23}})$. In conclusion, this makes our algorithm suitable to compute high accuracy geodesic distance in large scale meshes (e.g., with more than 5M vertices) as alternative to exact algorithm that often takes much longer time.

Direct accuracy control. The existing approximate window propagation methods, such as AMMP [9], AICH [10] and AVTP [11], control the error via a *local* parameter that is not directly related to the expected global accuracy. For example, AMMP merges two windows by bounding a local error. The reason that it cannot support a global accuracy parameter is that many merges will only occur near the source vertex. Our method, in contrast, controls the accuracy via an explicit formula (Eq. (1)) linking the window's travel distance and the global error. As a result, our method allows the user to directly specify the expected accuracy. In Table 2 we can see that DGG-VTP is able to give accurate distances with error below the user specified error control for more than 99% of the distances it computed in average. Moreover,

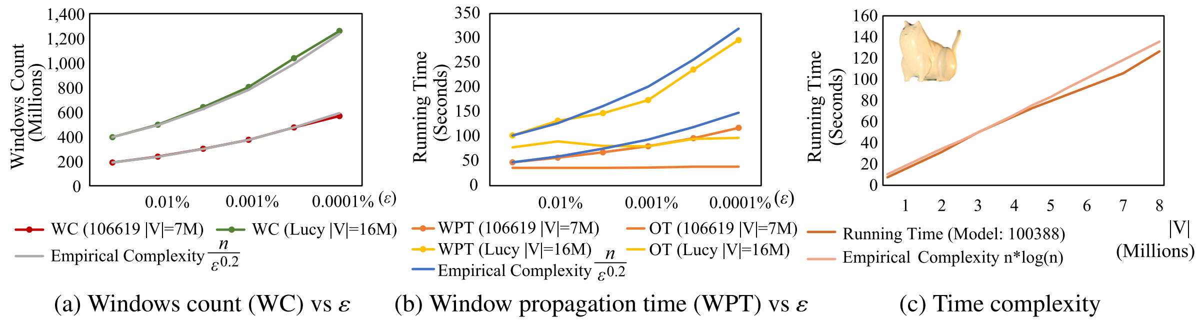


Fig. 8. Complexities. (a) DGG-VTP empirically produces $O(\frac{n}{\varepsilon^{0.23}})$ windows. (b) We show the window propagation runtime as accuracy increases. Here we exclude constant time overheads (OT) to compute fan-shape region, priority queue and related window list operations such as merging and selections from the window propagation runtime. (c) Runtime complexity with increasing number of vertices. It can be seen that our runtime complexity is increasing at $O(n \log n)$ rate with the increasing number of vertices.

Table 2
Accuracy control of DGG-VTP. We set the target accuracy parameter $\varepsilon = 10^{-4}, 10^{-5}, 10^{-6}$, and 10^{-7}, respectively. We randomly choose a source vertex and compute the SSAD distances for each mesh. Here, we report the average of the percentage of paths with error below the user-specified accuracy ε and the average of the percentage of results with mean error below ε.

ε	% accurate paths	% mean error $\leq \varepsilon$
10^{-4}	99.3	100
10^{-5}	99.4	100
10^{-6}	99.5	98.9
10^{-7}	99.2	91.5

speaking of the mean error of the resulting geodesics distances, DGG-VTP is able to give more than 90% of results with mean error less than the desired accuracy control parameter in a wide ranging error settings from 10^{-4} to 10^{-7}. These results indicate the effectiveness of our strategy to scale the accuracy control parameter given by the user before computation to create good accuracy control for DGG-VTP.

Window pruning rules. VTP has a long list of prune rules. As mentioned above, our method only keeps two of them, i.e., the ICH pruning rule and the "one-angle, two-sides" rule. In Table 1, we show the comparison of our algorithm using all VTP pruning rules against our simplified setting. We see similar runtime and memory usage in both cases. In fact, the runtime of our algorithm is up to 20% better than using all VTP pruning rules. Meanwhile, our memory usage is around 5%–10% worse than using all VTP pruning rules. This shows that for the sake of simplicity, it is recommended not to use VTP pruning rules in our algorithm. Furthermore, since we do not need to use most of VTP pruning rules, we are able to use bucket sorting for propagating windows, same as the FWP algorithm. This would introduce easier parallelization strategy with potential for future GPU parallelization.

Comparison with DGG. Although our method is an adaption of the precomputation-based DGG for computing SSAD directly, it has a unique feature that is not available to DDG. Our method can achieve much higher accuracy than the usual DGG method [8]. Note that DGG has space complexity $O\left(\frac{n}{\sqrt{\varepsilon}}\right)$. In order to obtain high accuracy such as $\varepsilon = 10^{-6}$, there are on average 1000 DGG edges incident to each vertex. Such a big graph not only takes lots of memory but also slows down distance query. As a result, the recommended accuracy parameter for DGG is $\varepsilon \in [10^{-4}, 10^{-3}]$. In contrast, our method can produce geodesics with much higher accuracy than DGG at very little cost.

Comparison with VTP. Given varying anisotropy meshes as inputs, on average, more than 99% of the geodesic distances computed are always less than the user specified error control. Comparing performance with VTP, we have up to 35× speedup and more than 40× memory saving with high accuracy setting of $\varepsilon = 10^{-7}$ in randomly sampled Thingi10K models that are remeshed to large scale models with 10M vertex with varying degree of anisotropy. It is worth noting that this high accuracy setting has already reached beyond the limit of single floating point precision that offers 23 bits of accuracy. Moreover, the speed of our method is much less sensitive to the surface roughness than VTP does, meaning that **DGG-VTP is an alternative to exact methods in computing accurate geodesic distances on large-scale mesh models, especially when input meshes are smooth.**

Parallelization performance. We developed parallel DGG-VTP by improving the PVTP framework. The notable change is we propagate the newly created windows from pseudo-sources in parallel, after observing that pseudo-sources are surrounded by strictly different half-edges. Table 3 reports the results of our parallel algorithm in 4-threads setting. Fig. 9(a) shows the parallelization speed-up with respect to the number of threads. Fig. 9(b) shows the results of parallelization with 4 threads using our diverse testing dataset consisting of 95 large scale meshes. As we can see in the table, the parallelization speedup of our algorithm is similar to that of PAVTP despite of the fact that it produces much fewer windows than PAVTP. Taking geometric mean of the speedup factor, on average our parallel algorithm confers 2.33× speedup for isotropic models, as comparing to PAVTP that gives 2.5× speedup. For anisotropic models, our parallel algorithm gives 2.12× speedup on average, while PAVTP's average speed-up factor is 2.23×. The speedup difference is less than 7%. This shows the effectiveness of our strategy of parallelization as compared to PAVTP.

We also compared parallel DGG-VTp with PVTP, the parallel version of the exact VTP algorithm. Our parallelization speedup factor is greater than PVTP on a high accuracy setting of $\varepsilon = 10^{-7}$ with 4 threads as illustrated in Fig. 9. With $\varepsilon = 10^{-5}$, the speedup rate of parallel DGG-VTP to DGG-VTP is lower than that of PVTP speedup rate compared to VTP. This is because DGG-VTP propagates far fewer windows. However, by setting $\varepsilon = 10^{-7}$ for DGG-VTP in single floating point precision, it shows better speedup ratio than PVTP. This is likely due to our optimization strategy of creating windows from new pseudo-sources in parallel. Apparently, this operation takes significant amount of time as DGG-VTP creates only small number of windows. As a result, the speedup from this strategy can outperform PVTP speedup. We believe it is unlikely that this optimization strategy can speedup PVTP by the same amount as parallel DGG-VTP since PVTP creates far more windows.

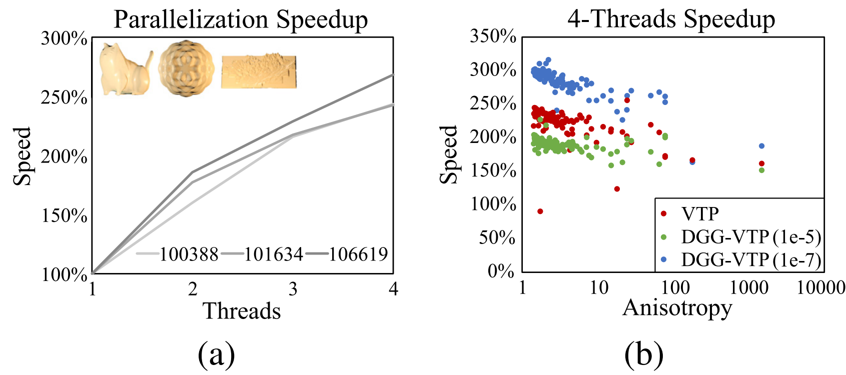


Fig. 9. Parallelization speedup. (a) We present the parallelization speedup with respect to the number of threads. (b) We show the parallelization speedup of our method with high accuracy settings of 10^{-5} and 10^{-7} together with VTP parallelization speedup. Our method with high accuracy setting of 10^{-7} can have better parallelization speedup than VTP even when it propagates significantly fewer windows. We tested the methods under 4-threads setting.

Table 3
Parallelization on isotropic and anisotropic models with 4 CPU threads.

Model	ε	DGG-VTP			A-VTP			$\frac{t_{4T-A}}{t_{4T-DGG}}$
$(\tau, \lvert V\rvert)$		t_{1T-DGG}	t_{4T-DGG}	$\frac{t_{1T-DGG}}{t_{4T-DGG}}$	t_{1T-A}	t_{4T-A}	$\frac{t_{1T-A}}{t_{4T-A}}$	
100388	10^{-4}	90	44	2.05	189.6	93	2.04	2.11
(τ = 1.62, 8M)	1.4 x 10^{-5}	131	52	2.52	707	234	3.02	4.50
	1.8 x 10^{-6}	168	65	2.58	1733	574	3.02	8.83
101634	10^{-4}	190.5	91	2.09	397.5	178	2.23	1.96
(τ = 1.51, 15M)	1.4 x 10^{-5}	246	117	2.10	1374	540	2.54	4.62
	1.8 x 10^{-6}	381	146	2.61	2865	904	3.17	6.19
106619	10^{-4}	94.1	43	2.19	175	80	2.19	1.86
(τ = 1.46, 7M)	1.4 x 10^{-5}	114.8	50	2.30	579	245	2.36	4.90
	1.8 x 10^{-6}	169	63	2.68	1421	493	2.88	7.83
264708	10^{-4}	98	47	2.09	201.5	94	2.14	2.00
(τ = 1.74, 7.5M)	1.4 x 10^{-5}	127.8	56	2.28	517.6	199	2.60	3.55
	1.8 x 10^{-6}	183	72	2.54	1609	511	3.15	7.10
Golfball	10^{-4}	171.3	80	2.14	275.5	140	1.97	1.75
(τ = 1.7, 13M)	1.4 x 10^{-5}	233	96	2.43	835.4	347	2.41	3.61
	1.8 x 10^{-6}	315	116	2.72	2202.3	732	3.01	6.31
Lucy	10^{-4}	213	102	2.09	368.5	170	2.17	1.67
(τ = 1.01, 16M)	1.4 x 10^{-5}	327	129	2.53	891	325	2.74	2.52
	1.8 x 10^{-6}	370	165	2.24	1496	637	2.35	3.86
		Average:		2.33	Average:		2.53	
1036429	10^{-4}	184	107	1.72	1224	671	1.82	6.27
(τ = 145, 8M)	1.4 × 10^{-5}	212	115	1.84	1883	941	2.00	8.18
	1.8 × 10^{-6}	272	142	1.92	2085	1037	2.01	7.30
1036473	10^{-4}	366	176	2.08	1028	570	1.80	3.24
(τ = 34, 13M)	1.4 × 10^{-5}	431	200	2.16	2855	1211	2.36	6.06
	1.8 × 10^{-6}	550	252	2.18	7702	3090	2.49	12.26
1158269	10^{-4}	580	287	2.02	4022	1765	2.28	6.15
(τ = 74, 12M)	1.4 × 10^{-5}	686	328	2.09	5359	2461	2.18	7.50
	1.8 × 10^{-6}	945	392	2.41	7555	3177	2.38	8.10
1336194	10^{-4}	656	324	2.02	2297	1096	2.10	3.38
(τ = 72, 17M)	1.4 × 10^{-5}	790	377	2.10	5890	2682	2.20	7.11
	1.8 × 10^{-6}	1023	450	2.27	9308	4041	2.30	8.98
58874	10^{-4}	118	56	2.11	200	102	1.96	1.82
(τ = 3.8, 10M)	1.4 × 10^{-5}	144	65	2.22	593	226	2.62	3.48
	1.8 × 10^{-6}	195	80	2.44	1751	625	2.80	7.81
63785	10^{-4}	294	128	2.30	669	282	2.37	2.20
(τ = 2, 19M)	1.4 × 10^{-5}	345	147	2.35	1110	430	2.58	2.93
		Average:		2.12	Average:		2.23	

Comparison with GSP. Geodesic source propagation [31] is a highly efficient algorithm that can be implemented on both GPUs and CPUs. It adopts virtual source propagation and visibility heuristics to improve robustness and accuracy. However, it does not allow the user to control accuracy. We compared DGG-VTP and GSP (using the authors' provided code) on the testing dataset of 95 large-scale meshes in terms of running time and accuracy. As shown in Table 4, GSP yields a fixed average error 4×10^{-3}. With $\varepsilon = 10^{-4}$, GSP is on average 13.7 times faster than DGG-VTP, but their error is also 172 times larger than ours. With $\varepsilon = 10^{-7}$, the average speed factor is 33.2, and the accuracy difference is as large as 5 orders of magnitude.

Since the two methods have different objectives, their application domains are also different. GSP is highly desired in

Table 4
Comparison with GSP [31] on 95 large-scale meshes. GSP has a fixed mean error $\bar{\varepsilon}_{GSP} = 4.1 \times 10^{-3}$. $I_{accuracy}$ is the mean value of the error ratio $\frac{\varepsilon_{GSP}}{\varepsilon_{DGG-VTP}}$ for each individual mesh.

ε	$\bar{\varepsilon}_{DGG-VTP}$	$\bar{\varepsilon}_{GSP}$	$\frac{I_{accuracy}}{T_{DGG-VTP}/T_{GSP}}$	$T_{DGG-VTP}/T_{GSP}$	$I_{accuracy}$
10^{-4}	2.5×10^{-5}	4.1×10^{-3}	10.75	13.7	172
10^{-5}	2.2×10^{-6}	4.1×10^{-3}	100.62	17.3	2011.5
10^{-6}	2.5×10^{-7}	4.1×10^{-3}	874.88	23.2	23 173
10^{-7}	9×10^{-8}	4.1×10^{-3}	5875.30	33.2	220 731

time-critical applications, whereas our method, as an alternative to exact methods, is desired in accuracy-critical applications, such as computing curve-sourced geodesic distances [32–34], geodesic Voronoi diagrams [35,36] and power diagrams [37,38].

To quantitatively evaluate the trade-off between accuracy and performance, we computed the ratio of accuracy improvement to performance slowdown for each testing mesh and reported the mean ratio. Table 4 reports the results for $\varepsilon = \{10^{-4}, 10^{-5}, 10^{-6}, 10^{-7}\}$. We observed that the ratio goes up with higher accuracy parameter and is 5875× when $\varepsilon = 10^{-7}$, which indicates that DGG-VTP scales with higher accuracy while still having good runtime performance.

6. Conclusion & future work

We developed a new approximate method for computing discrete geodesic distances and paths on triangle meshes. Our method allows the user to specify the desired accuracy using a single parameter ε. Technically, it extends the conventional window propagation framework by monitoring the accuracy of the computed distances so that propagation can terminate immediately when the desired accuracy is reached. With $\varepsilon = 10^{-7}$, our method is able to produce results with accuracy comparable to exact methods, but runs 3−40× faster. Such feature distinguishes our method from all the existing approximate methods. We evaluated the performance of our method on large-scale 3D shapes with varying complexity and degree of anisotropy. Quantitative comparisons with state-of-the-art approximate algorithms show that our method can achieve good results that balance accuracy and running time.

Our approach calls for possible further improvements that need to be addressed in the near future. First, we implemented parallel DGG-VTP only on CPUs. To fully explore the potential of DGG-VTP, GPU parallelization is highly desired. Since our method adopts only two simple window pruning rules, i.e., the ICH rule and the "one-angle two-sides" rule, our method does not rely on the order of windows in window lists, which plays an important role for window pruning in VTP. Therefore, it is possible to replace the current window propagation scheme by FWP, which is easier for GPU parallelization than VTP. Second, there are many window pruning rules proposed in the literature [2,5,10]. However, they do not play an equal role in practice. Therefore, it is interesting to quantitatively examine their performance with respect to mesh tessellation, anisotropy and smoothness. Third, although our method allows the user to directly specify the expected accuracy and works well for a wide range of triangle meshes including the ones with extremely high degree of anisotropy, it cannot guarantee the actual error is less than the accuracy parameter ε. There is a pressing need to develop an approximate method with guaranteed error bound. Last but not the least, window propagation is the only known technique for computing exact or highly accurate SSAD geodesics on triangle meshes. Window propagation methods are accurate and robust, however, they are computationally expensive. The last two decades have witnessed extensive investigation on window pruning rules and window management schemes. Although it is still possible to further improve both, we believe the room for performance gain is small. An interesting direction is to develop new techniques for computing exact geodesics. Notice that DGG-VTP can well approximate a long geodesic path through shorter geodesics that are nearly parallel to it. Applying curve straightening/shortening methods to those nearly parallel geodesics can produce exact paths faster than using other types of initial paths. It is interesting to examine whether combining accuracy-aware window propagation and curve shortening would lead to a method faster than the pure windows propagation methods.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

We thank the anonymous reviewers for their constructive comments. This project was partially supported by the Ministry of Education, Singapore, under its Academic Research Fund Tier 1 (RG20/20) and Tier 2 (MOE-T2EP20220-0005).

References

[1] Mitchell Joseph SB, Mount David M, Papadimitriou Christos H. The discrete geodesic problem. SIAM J Comput 1987;16(4):647–68.
[2] Chen Jindong, Han Yijie. Shortest paths on a polyhedron. In: Proceedings of the sixth annual symposium on computational geometry. ACM; 1990, p. 360–9.
[3] Xin Shi-Qing, Wang Guo-Jin. Improving Chen and Han's algorithm on the discrete geodesic problem. ACM Trans Graph 2009;28(4):104.
[4] Xu Chun-Xu, Wang Tuanfeng Y, Liu Yong-Jin, Liu Ligang, He Ying. Fast wavefront propagation (FWP) for computing exact geodesic distances on meshes. IEEE Trans Vis Comput Graphics 2015;21(7):822–34.
[5] Qin Yipeng, Han Xiaoguang, Yu Hongchuan, Yu Yizhou, Zhang Jianjun. Fast and exact discrete geodesic computation based on triangle-oriented wavefront propagation. ACM Trans Graph 2016;35(4).
[6] Ying Xiang, Wang Xiaoning, He Ying. Saddle vertex graph (SVG): A novel solution to the discrete geodesic problem. ACM Trans Graph 2013;32(6):170:1–2.
[7] Wang Xiaoning, Fang Zheng, Wu Jiajun, Xin Shi-Qing, He Ying. Discrete geodesic graph (DGG) for computing geodesic distances on polyhedral surfaces. Comput-Aided Geom Des 2017;52:262–84.
[8] Adikusuma Yohanes Yudhi, Fang Zheng, He Ying. Fast construction of discrete geodesic graphs. ACM Trans Graph 2020;39(2):14:1–14:14.
[9] Surazhsky Vitaly, Surazhsky Tatiana, Kirsanov Danil, Gortler Steven J, Hoppe Hugues. Fast exact and approximate geodesics on meshes. ACM Trans Graph 2005;24(3):553–60.
[10] Xin Shi-Qing, Wang Guo-Jin. Applying the improved Chen and Han's algorithm to different versions of shortest path problems on a polyhedral surface. Comput Aided Des 2010;42(10):942–51.
[11] Du Jie, He Ying, Fang Zheng, Meng Wenlong, Xin Shi-Qing. On the vertex-oriented triangle propagation (VTP) algorithm: Parallelization and approximation. Comput Aided Des 2021;130:102943.
[12] Ying Xiang, Xin Shi-Qing, He Ying. Parallel chen-han (PCH) algorithm for discrete geodesics. ACM Trans Graph 2014;33(1):9:1–9:11.
[13] Ying Xiang, Huang Caibao, Fu Xuzhou, He Ying, Yu Ruiguo, Wang Jianrong, Yu Mei. Parallelizing discrete geodesic algorithms with perfect efficiency. Comput Aided Des 2019;115:161–71.
[14] Dijkstra Edsger W. A note on two problems in connexion with graphs. Numer Math 1959;1(1):269–71.
[15] Xin Shi-Qing, Ying Xiang, He Ying. Constant-time all-pairs geodesic distance query on triangle meshes. In: Proceedings of the ACM SIGGRAPH symposium on interactive 3D graphics and games; 2012, p. 31–8.
[16] Kimmel Ron, Sethian James A. Computing geodesic paths on manifolds. Proc Natl Acad Sci 1998;95:8431–5.
[17] Crane Keenan, Weischedel Clarisse, Wardetzky Max. Geodesics in heat: A new approach to computing distance based on heat flow. ACM Trans Graph 2013;32(5):152.

[18] Melvær Eivind Lyche, Reimers Martin. Geodesic polar coordinates on polygonal meshes. Comput Graph Forum 2012;31(8):2423–35.
[19] Belyaev Alexander G, Fayolle Pierre-Alain. On variational and PDE-based distance function approximations. Comput Graph Forum 2015;34(8):104–18.
[20] Campen Marcel, Kobbelt Leif. Walking on broken mesh: Defect-tolerant geodesic distances and parameterizations. Comput Graph Forum 2011;30(2):623–32.
[21] Tao Jiong, Zhang Juyong, Deng Bailin, Fang Zheng, Peng Yue, He Ying. Parallel and scalable heat methods for geodesic distance computation. IEEE Trans Pattern Anal Mach Intell 2021;43(2):579–94.
[22] Xia Qianwei, Zhang Juyong, Fang Zheng, Li Jin, Zhang Mingyue, Deng Bailin, He Ying. GeodesicEmbedding (GE): A high-dimensional embedding approach for fast geodesic distance queries. IEEE Trans Vis Comput Graphics 2021;(01).
[23] Yuan Na, Wang Peihui, Meng Wenlong, Chen Shuang-Min, Xu Jian, Xin Shiqing, He Ying, Wang Wenping. A variational framework for curve shortening in various geometric domains. IEEE Trans Vis Comput Graphics 2021;1.
[24] Liu Bangquan, Chen Shuang-Min, Xin Shi-Qing, He Ying, Liu Zhen, Zhao Jieyu. An optimization-driven approach for computing geodesic paths on triangle meshes. Comput Aided Des 2017;90:105–12.
[25] Cao Luming, Zhao Junhao, Xu Jian, Chen Shuang-Min, Liu Guozhu, Xin Shiqing, Zhou Yuanfeng, He Ying. Computing smooth quasi-geodesic distance field (QGDF) with quadratic programming. Comput Aided Des 2020;127:102879.
[26] Meng Wenlong, Xin Shiqing, Zhao Jinhui, Chen Shuangmin, Tu Changhe, He Ying. A variational framework for computing geodesic paths on sweep surfaces. Comput Aided Des 2021;140:103077.
[27] Sharp Nicholas, Crane Keenan. You can find geodesic paths in triangle meshes by just flipping edges. ACM Trans Graph 2020;39(6):249:1–249:15.
[28] Xin Shi-Qing, He Ying, Fu Chi-Wing. Efficiently computing exact geodesic loops within finite steps. IEEE Trans Vis Comput Graph 2012;18(6):879–89.
[29] Aleksandrov Lyudmil, Lanthier Mark, Maheshwari Anil, Sack Jörg-Rüdiger. An epsilon-approximation for weighted shortest paths on polyhedral surfaces. In: Arnborg, Stefan, Ivansson, Lars (Eds.), Proceedings of SWAT '98; 1998, p. 11–22.
[30] Meng Wenlong, Xin Shiqing, Tu Changhe, Chen Shuang-Min, He Ying, Wang Wenping. Geodesic tracks: Computing discrete geodesics with track-based steiner point propagation. IEEE Trans Vis Comput Graphics 2021;1.
[31] Trettner Philip, Bommes David, Kobbelt Leif. Geodesic distance computation via virtual source propagation. Comput Graph Forum 2021;40(5):247–60.
[32] Bommes David, Kobbelt Leif. Accurate computation of geodesic distance fields for polygonal curves on triangle meshes. In: Proceedings of the vision, modeling, and visualization conference 2007; 2007, p. 151–60.
[33] Xin Shi-Qing, Ying Xiang, He Ying. Efficiently computing geodesic offsets on triangle meshes by the extended xin-wang algorithm. Comput Aided Des 2011;43(11):1468–76.
[34] Sun Qian, Zhang Long, Zhang Minqi, Ying Xiang, Xin Shi-Qing, Xia Jiazhi, He Ying. Texture brush: an interactive surface texturing interface. In: ACM symposium on interactive 3D graphics and games, I3D '13. 2013, p. 153–60.
[35] Xu Chunxu, Liu Yong-Jin, Sun Qian, Li Jinyan, He Ying. Polyline-sourced geodesic voronoi diagrams on triangle meshes. Comput Graph Forum 2014;33(7):161–70.
[36] Wang Xiaoning, Ying Xiang, Liu Yong-Jin, Xin Shi-Qing, Wang Wenping, Gu Xianfeng, Müller-Wittig Wolfgang, He Ying. Intrinsic computation of centroidal voronoi tessellation (CVT) on meshes. Comput Aided Des 2015;58:51–61.
[37] Xin Shi Qing, Wang Wenping, Chen Shuangmin, Zhao Jieyu, Shu Zhenyu. Intrinsic girth function for shape processing. ACM Trans Graph 2016;35(3):1–14.
[38] Xin Shi-Qing, Lévy Bruno, Chen Zhonggui, Chu Lei, Yu Yaohui, Tu Changhe, Wang Wenping. Centroidal power diagrams with capacity constraints: Computation, applications, and extension. ACM Trans Graph 2016;35(6):244:1–244:12.