PARALLEL WINDOW PROPAGATION ON MESH SURFACES

Le Tien Hung*, Do Quoc Trinh, Nguyen Huu Tho

Military Technical Academy, Hanoi, Vietnam

*Corresponding author: letienhung.lqd@gmail.com

(Received: September 12, 2022; Revised: October 06, 2023; Accepted: November 27, 2023)

Abstract - The task of calculating the shortest distance with minimal curvature between two points on a mesh surface is a well-known problem in computational geometry. This paper aims to develop a effective algorithm for computing geodesics on large, complex models with fast processing times to facilitate interactive applications. Our contribution is developing of the parallel window propagation (PWP) algorithm, which divides the sequential MMP algorithm into four phases: front node selection, window list propagation, window list merging, and vertex update. We use two separate buffers for incoming windows on each edge to avoid data dependency and conflicts in each phase, allowing for parallel processing on a CPU. As a result, our PWP algorithm can propagate a large number of windows simultaneously and independently, leading to significant improvements in performance for real-world models.

Key words - Geodesics, MMP algorithm; windows propogation; parallel computing.

1. Introduction

Geodesic is a fundamental concept in differential geometry and has been extensively researched in recent years. Simply speaking, a geodesic is a generalization of a straight line from a two-dimensional or three-dimensional space to a curved space. The correlation between these spaces is quite easy to understand: whereas a straight line minimizes the distance between two points on a flat surface, a geodesic minimizes the distance between two points on any surface with a Riemannian metric. Essentially, a geodesic path is the shortest path between two points within a particular region, and its length is referred to as the geodesic distance. In different fields, there are also other ways to define the concept of "geodesic".

The discrete geodesic problem divides itself into some special sub-problems such as “single source, single destination” [1]-[3], “single source, multiple destinations” [4], [5], [7], [8] and “all-pair geodesic” [9]-[12]. These sub-problems are also closely related to each other.

There are numerous geodesic algorithms in the literature that can be categorized as either exact or approximate. Exact algorithms, such as MMP [4], Chen and Han (CH) [5], and its improved version [13], strictly guarantee precise geodesic distances and paths. Ying et al. [14] presented a GPU-based version of the CH algorithm for more efficient geodesic computation on real-world models. Approximate algorithms, such as FMM [15], heat method [16], and SVG [17], prioritize reducing computing speed by using local information on the input mesh, such as gradient vector field and local shortest paths, to estimate geodesic distances and paths. These algorithms perform better than exact algorithms in terms of speed but sacrifice some precision.

There is another way to classify geodesic algorithms based on how they propagate information. The “windows propagation” mechanism is usually used in exact geodesic algorithms, where geodesics are computed by propagating windows created by partitioning edges [4], [5], [13]. These windows contain encoded geodesic information and are propagated in wavefront order from selected sources over vertices and edges on input surfaces. However, these algorithms are very computationally expensive and difficult to apply to large models. Approximate geodesic algorithms use the PDE mechanism [15], [16] where local geodesic information is propagated by solving the Eikonal equation, which is non-linear and can be approximated into simpler partial differential equations. Modern numerical methods make it possible to solve these equations quickly, even in linear time, giving this mechanism a time performance advantage. However, the PDE mechanism can suffer from numerical instability if the input models have rich details, as the approximation techniques applied in these details may be inaccurate.

The MMP algorithm [4] stands as an early and practical solution for accurately calculating geodesic distances on polyhedral surfaces. It utilizes a basic data structure known as a "window" or "candidate interval", essentially a portion of an edge that carries specific information and can be forwarded in a manner akin to a wavefront. This algorithm mirrors the fundamental approach of Dijkstra’s algorithm for the shortest path: initiating from a chosen source point, the "windows" radiate outwards towards all other vertices. As vertices receive new geodesic data, it's evaluated against existing data, updated if necessary, and then further disseminated. Given that each edge can possess up to $O(n)$ windows, the aggregate count of windows may reach $O(n^2)$, with n representing the total edge count of the surface. The algorithm includes a $log\, n$ priority queue to efficiently manage the progression of these windows. In terms of computational requirements, the MMP algorithm demands $O(n^2)$ space and $O(n^2\, log\, n)$ time. The process to trace the shortest path from a source to a specific destination is computed in a time frame of $O(k + log\, n)$, where k denotes the count of faces the shortest path traverses.

The CH algorithm [5] employs an alternative method for organizing windows to accurately calculate geodesic distances. Unlike the MMP algorithm, which uses a priority queue, the CH algorithm utilizes a binary tree structure to manage the sequence of window propagation. Each propagating window can have a varying number of children, depending on its specific type. The relationship between parent and child windows enables the construction

of a sequence tree with up to n levels, with *n* being the total count of faces on the polyhedral surface. This sequence tree is efficient in terms of space, as it retains only the leaf and branch nodes while discarding all single-child interval windows. This approach results in a linear space complexity of $O(n)$, streamlining the process significantly.

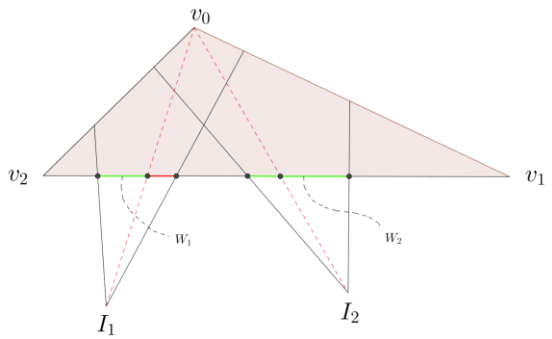

Figure 1. *The rule "one angle, one split": the red window is eliminated, and the green ones will propagate forward [12]*

To avoid exponential explosion of the sequence tree, the children windows are selected/abandoned by principle "one angle, one split". In the Figure 1, there are two windows w_1, w_2 entering into triangle $\triangle v_0v_1v_2$ from the edge (v_1, v_2). Both of w_1, w_2 cover the vertex v_0, hence four interval windows are created on the edge (v_1, v_2) as children of w_1 and w_2. The rule "one angle, one split" implies that only three of the children windows can provide shortest paths. Subsequently, only three selected interval windows will be propagated forward through triangle $\triangle v_0v_1v_2$. The CH algorithm determines the shortest paths in "single source, all vertices" problem with time complexity $O(n^2)$ and it works on polyhedral surfaces which not need to be convex.

Differing from the CH and MMP algorithms, Qin et al. [18] introduced the **VTP** algorithm, which utilizes triangles instead of windows as the primary element for propagating windows. Each iteration of this algorithm extends a set of windows from one edge of a triangle to the other two edges. The VTP employs a comprehensive set of rules to effectively eliminate unnecessary windows, thus notably reducing both computational efforts and memory requirements in later stages. Each propagating iteration involves the merging of old and new windows on each triangle edge, guided by two rules that do not depend on the order of merging. The VTP algorithm, with a time complexity of $O(n^2)$, surpasses other exact geodesic algorithms in terms of performance, particularly in CPU environment.

The original MMP algorithm does not support parallel processing due to its priority-ordered window management, handling only one window at a time. This paper introduces the PWP algorithm as a parallel adaptation of the traditional window propagation technique used in the aforementioned algorithms. This modification aims to reduce computational demands and enhance time efficiency. The PWP algorithm breaks down window propagation into four stages: front node selection, window list propagation, merging of window lists, and vertex updating, employing two distinct buffers for incoming windows on each list to prevent data dependency or conflict in each phase. This approach enables the simultaneous propagation of numerous windows in each wavefront, with all phases being parallelizable on a multi-threaded CPU.

2. PWP algorithm improvements

2.1. Preliminary

In the context of a triangle mesh $M = (V, E, F)$ representing an orientable 2-manifold, with V, E, and F being sets of vertices, edges, and faces respectively, Surazhsky et al. [8] utilized a structural concept named "window" for encoding points along the shortest paths that follow identical vertex-edge sequences. These windows are formed by dividing each edge of the mesh into multiple intervals. The edges v_1v_2 and v_2v_0 may already have windows from previous propagations, as multiple wavefronts could intersect these edges. To ensure that the stored windows on these edges represent the minimum geodesic distance without overlapping, it's necessary to "intersect" existing windows with new potential ones to consolidate their minimum distance fields. The portion of a new potential window that is retained is the one offering a shorter geodesic distance than the existing data points on v_0, v_2 or v_1, v_2. The shortest paths within a window are defined locally by a 6-tuple $(x_{start}, x_{end}, x_s, y_s, d_s, \tau)$, where x_{start}, x_{end} denote the window's endpoints, (x_s, y_s) indicates the position of the pseudo source vertex s relative to the window in a planar unfolding, d_s is the shortest distance to s, and τ is the binary direction indicating the side of the edge where the source s is located (as illustrated in Figure 2). The propagation of these windows then proceeds across the mesh's faces in a manner similar to Dijkstra's algorithm.

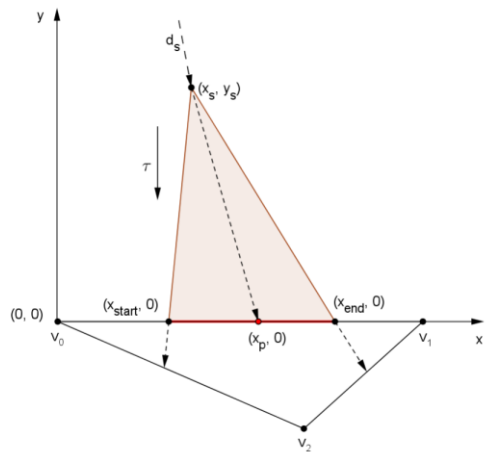

Figure 2. *In window w, the geodesic distance at $p(x_p, 0)$ is evaluated as the sum of d_s and the distance $d(s, p)$ from s to p*

When a window w propagates along the edge v_0v_1, it has the potential to create new windows on the opposite edges, specifically v_1v_2 and v_2v_0. To establish these potential windows, the pseudo-source point s is projected onto the plane (v_0, v_1, v_2). The process then involves determining the intersection points where the side rays of the "unfolded" window w intersect with edges v_0, v_2 and v_1, v_2 (see Figure 3). Let's consider that there is an existing window w_1 on edge v_0v_1 and a new window w_2 formed,

which intersects with w_1 over a specific interval, referred to as ω, where ω represents the intersection of w_1 and w_2. To ascertain which of these windows, w_1 or w_2, provides the minimum distance function for each point within the interval ω, it's necessary to identify all the points p within ω where the distance functions defined by both w_1 and w_2 are equal.

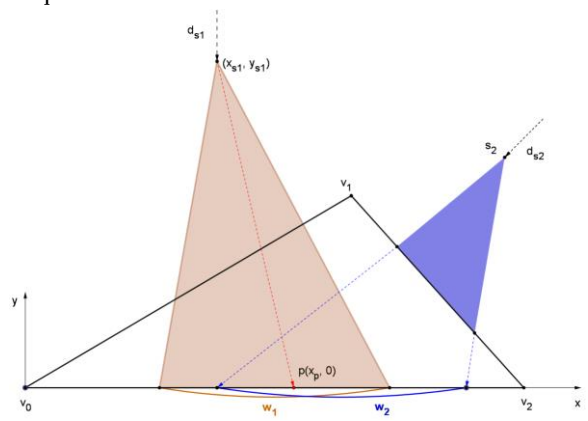

Figure 3. *Window clipping: w_1 is the old window on v_0v_1 and w_2 is the new potential window delivered onto v_0v_1*

2.2. Effective window pruning

During the propagation of windows in our algorithm, numerous windows are generated but not all contribute to the calculation of the geodesic distance at vertices. To address this, we integrate a window intersecting approach with the rule "one angle, one split" [5], utilizing it as a merging guideline to remove unnecessary windows or their redundant segments from propagation.

Figure 4. *After intersecting of w_{income} with w_i, the red part of w_{income} is used as the new window for intersecting with w_j*

In our algorithm, windows on the same edge are sorted into two lists: w_{lf} and w_{lb}. The list w_{lf} contains windows aligned with the forward halfedge, while w_{lb} encompasses those aligned with the backward halfedge. All windows within a single list are stored sequentially without any overlap. On an edge e within a list w_l, if a window w_j follows another window w_i, the endpoint of w_i is always positioned before the starting point of w_j. Suppose a new window w_{income} propagates onto w_l and intersects with w_i. Our intersecting scheme allows for determining the subintervals within the areas covered by both w_i and w_{income}, and identifying which of the two windows more effectively covers these subintervals. If w_{income} better covers the final subinterval, we retain this portion of w_{income} for further intersecting steps with w_j, as depicted in Figure 4. In such cases, we set the flag of w_{income}'s survival as TRUE. Post-merging, all newly formed windows derived from w_{income} in w_l undergo reassessment through the rule "one angle one split". Newly created windows from w_i are re-evaluated only when the geodesic distances of two start/end vertices of w_i's edge are updated. Experimental results demonstrate that our algorithm is more efficient in pruning redundant windows and requires less computation of Euclidean distances compared to the VTP schemes.

2.3. Parallelized window propagation

A common limitation of the exact geodesic algorithms, such as those cited in [5] and [8], is their inability to operate in parallel due to their structural design. These algorithms use a "window propagation" strategy to spread geodesic data from sources to vertices. All generated windows are placed in an associative container and processed one at a time, in a global sequence. Absent a well-defined queuing order, the efficiency of these wavefronts diminishes, leading to a marked decrease in performance.

Since there are GPU-based parallel CH approaches [14], it raises an obvious question about the existing of a parallel version of MMP algorithm on CPU and how much the performance could be improved from other algorithms. However, in MMP algorithm, generated windows are managed in a priority queue and they are pop out one by one following a very strict order. It means the MMP algorithm is hard to be parallelized and its performance strongly depends on the windows management.

The existence of GPU-based parallel CH approaches [14] poses a critical question about the feasibility of a parallel version of the MMP algorithm on CPU, and how much the performance could be improved. The challenge with the MMP algorithm lies in its use of a priority queue for managing windows, where they are handled individually in a very specific sequence. This characteristic makes parallelization of the MMP algorithm difficult and its performance strongly depends on on the management of windows.

To overcome these challenges, our study introduces a novel parallel window propagation approach that allows for simultaneous processing of window lists associated with multiple vertices. The window propagation is divided into four distinct phases: selection of the front node, propagation of window lists, merging of window lists, and updating of vertices. This division is designed to ensure there are no issues with data dependence or conflicts (see Algorithm 1).

Algorithm 1: PWP Algorithm

Input: The triangle mesh M, the source p, the size factor f.

Output: The geodesic distance vector D at vertices of M.

initiate propagation

$D \leftarrow \vec{0}$

$v\_queue \leftarrow p$

While $v\_queue \neq \emptyset$

  $T \leftarrow [f * v\_queue.size]$

  $V' \leftarrow$ select maximally T vertices from v_queue

  parallel using omp **For** each $v_i \in V'$

    FIFO queue $wl\_queue_i \leftarrow$ non-empty lists on adjacent edges of v_i in counterclockwise order

  End for

  parallel using omp **For** each $v_i \in V'$

    While $wl\_queue_i \neq \emptyset$

      $wl \leftarrow wl\_queue_i$.front()

check wl using "one angle one split" rule

If $wl \neq \emptyset$

$wl_{out\ left}$, $wl_{out\ right} \leftarrow$ propagate wl on edges e_{left}, e_{right} in counterclockwise order

If e_{left} is an adjacent edge of v_i

merge $wl_{out\ left}$ into wl_{left} on e_{left}

Else

merge $wl_{out\ right}$ into wl_{right} on e_{right}

End if

If the first/last window in wl_{left} changed

generate update event at start/end vertex ($v_{start/end}$, $d_{start/end}$)

End if

End if

$wl \leftarrow \emptyset$

End while

End for

parallel using omp **For** each $v_i \in V'$

For each adjacent list wl around v_i in counterclockwise order

If $wl_{start/end} \neq \emptyset$

merge $wl_{start/end}$ into wl

End if

If the first/last window in wl changed

generate update event at start/end vertex ($v_{start/end}$, $d_{start/end}$)

End if

End for

End for

parallel using omp **For** each $v_i \in V'$

For each opposite list wl of v_i

If $wl_{start/end} \neq \emptyset$

merge $wl_{start/end}$ into wl

End if

If the first/last window in wl changed

generate updating event at start/end vertex ($v_{start/end}$, $d_{start/end}$)

End if

If the start/end vertex of wl is covered by income windows

generate new front node ($v_{start/end}.id$, $d_{start/end}$)

End if

End for

End for

update distances $d_i \in D$ from the generated updating events

update front queue from the generated new front nodes

End while

2.3.1. Selection of Front Node

During each iteration of propagation in our algorithm, we disseminate window lists around T vertices that are in closest proximity to the source points. Specifically, in each ith thread, the algorithm handles only those window lists that are adjacent to the selected vertex from the i-th vertex queue. This procedure is executed sequentially to ensure that the selected vertices are consistently retrieved from the top of the vertex queue, denoted as v queue. This sequential approach is essential to maintain the integrity and order of the vertex selection process.

2.3.2. Propagation of Window Lists

In our algorithm, to handle the possibility of two separate threads directing window flows into the same window list, we assign two temporary buffers, w_{lstart} and w_{lend}, for each window list w_l. These buffers are used to store incoming windows, as depicted in Figure 5. This approach differs from the MMP algorithm in that if a vertex v_i is chosen during the front node selection phase, then only the adjacent lists of v_i, following a counterclockwise sequence, are added to the FIFO queue w_l. As shown in Figure 5(a), we first merge the incoming window in w_{lstart} with the forward list w_{l31} on the edge (v_3, v_1), and then we add w_{l31} to the FIFO queue of v_1. Similarly, as illustrated in Figure 5(b), the incoming window in w_{lend} is merged with the forward list w_{l31} on the edge (v_3, v_1) before adding w_{l31} to the FIFO queue of v_1. This strategy is crucial for managing the flow of windows efficiently and avoiding conflicts between threads.

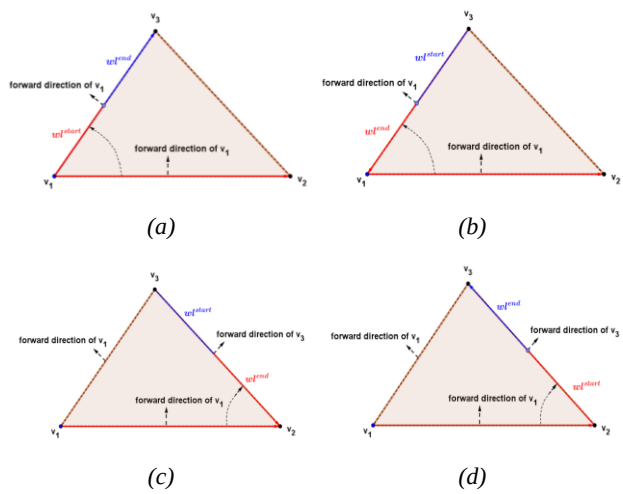

Figure 5. *During the propagation at v_1, window list w_{l12} on the edge (v_1, v_2) moves to the edges (v_3, v_1) in (a), (b) and (v_2, v_3) in (c), (d)*

2.3.3. Merging of Window Lists

For every chosen vertex v, we merge the forward adjacent lists surrounding v with their respective non-empty buffers. As an illustration, in Figure 5(a), we merge the newly introduced window in w_{lend} with the forward list w_{l31} located on the edge (v_3, v_1). Similarly, in Figure 5(b), we merge the new window found in w_{lstart} with the forward list w_{l31} on the same edge (v_3, v_1), prior to placing w_{l31} into the FIFO (First-In, First-Out) queue of vertex v_1.

Subsequently, it's necessary to merge the lists on the opposite side of vertex *v* with their non-empty buffers. Demonstrated in Figures 5(c) and 5(d), the newly arriving windows stored in w_{lstart} and w_{lend} are merged into the opposing list on the edge (v_2, v_3). This step ensures that the windows are consistently updated and integrated across both sides of each vertex.

2.3.4. Updating of Vertecies

In our algorithm, the geodesic distances at the start and end vertices of the lists, which have been merged during the previously mentioned two phases, are updated. Specifically, if a vertex *v* is extracted from the *w* queue during the "front node selection" phase, it is re-entered into the *v* queue, but only under the condition that at least one of its forward adjacent lists is not empty. This approach ensures that vertices are revisited and updated as necessary, based on the propagation and merging of window lists in the surrounding areas.

3. Experimental Results

Our geodesic algorithm was executed on a 64-bit PC equipped with an Intel Core i7-6700k CPU 4.0GHz (supporting up to 8 threads) and 32GB memory. The parallel components of the algorithm were developed to run on the CPU, utilizing the multi-threading capabilities of the OpenMP interface.

As Chen and Han [5] suggested, our algorithm generates a number of windows that also scale to $O(n^2)$. in its most demanding scenarios. However, a key feature of our algorithm is the immediate deletion of windows post-propagation, thereby ensuring the space complexity remains at $O(n^2)$. While this space complexity is on par with other existing exact algorithms, Table 1 in our paper demonstrates that the actual space required by our algorithm is considerably lower.

To experimentally estimate the optimal value of *f* for the PWP algorithm, we run it on a variety of models with different resolution. Our results shown that the PWP algorithm archives its best time and space performances with *f* is around 0.5. For the values less than 0.5, our algorithm still performs well because we set the minimum number of available threads on the system to take the advantage of multi-threading CPUs. Our PWP algorithm runs slower and requires much more memory when *f* increases closer to 1. It can be explained as if we select too many nodes in vertices queue to propagate their window lists around then the number of unnecessary “one angle, one split” checking/merging is called too often and of running threads equal to the maximum number the selected front vertices have very high change to be inserted back into the queue. Thus, the size of the queue stays long in almost all propagating iterations, which leads to very high memory usage.

In order to determine the most effective value of the factor *f*, we conducted tests across a range of models with varying resolutions. Our findings indicate that the PWP algorithm achieves peak performance in both time and space efficiency when *f* is approximately 0.5. When *f* is set below 0.5, the algorithm still delivers robust results due to the implementation of a minimum thread utilization strategy, capitalizing on the benefits of multi-threaded CPUs. However, as *f* approaches 1, the PWP algorithm's performance declines, requiring substantially more memory. This decrease in efficiency can be attributed to the excessive selection of nodes in the vertex queue for window list propagation, leading to frequent and unnecessary "one angle, one split" checks and merges. As a result, the vertex queue often remains extensive throughout the propagation phases, causing significant memory consumption.

This phenomenon is illustrated in Figures 6 and 7, where we present the outcomes of running our algorithm on two models: Bunny (160k faces) and Kitten (1.1M faces), using varying *f* values. Opting for a smaller *f* value still maximizes the potential of parallel propagation, as the minimum number of active threads is controlled. Conversely, a larger *f* value results in numerous redundant window list checks and merges, prolonging the length of the vertex queue, thus slowing down the algorithm and increasing memory requirements. We have identified the optimal *f* value to be around 0.5, at which point the PWP algorithm demonstrates its best execution time and memory efficiency. Selecting the appropriate *f* value is crucial for optimizing performance, underscoring the importance of a theoretical estimation of this factor.

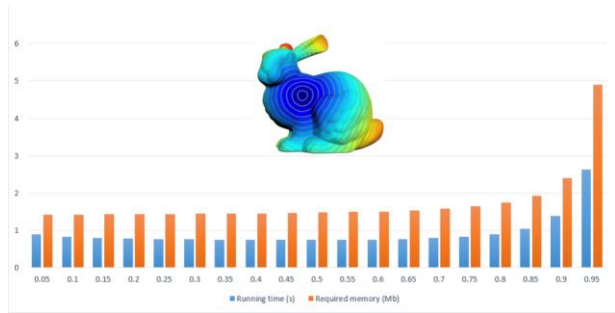

Figure 6. *The impact of f on the model Bunny (160k faces)*

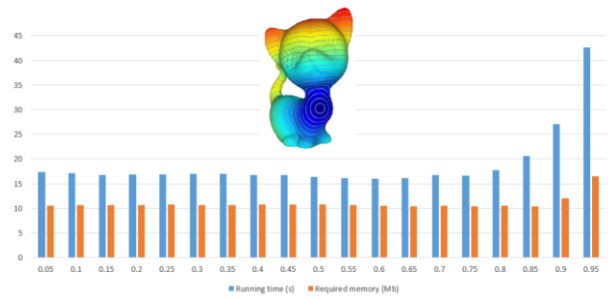

Figure 7. *The impact of f on the model Kitten (1.1M faces)*

We also tested the PWP, MMP and VTP algorithms on a diverse array of models, ranging from smaller ones like the Gargoyle (40k faces), to larger models such as the Bunny (1.6M faces). For a comprehensive evaluation, we measured various parameters including the running time, the maximum number of active windows, and the peak memory usage. The data presented in Table 1 reveals that our PWP algorithm significantly enhances time efficiency, especially in large-scale models, while only requiring a marginal increase in memory usage. Additionally, as depicted in Table 2, our algorithm demonstrates competent

performance even on anisotropic models, highlighting its versatility and effectiveness across different types of geometrical configurations.

Table 1. *Performance comparison of the PWP, MMP and VTP algorithms by three factors: The running time T_s, the maximal memory usage M and the maximal number of active windows n_w*

Model	Comparison	MMP	VTP	PWP (f = 0.5)
Gargoyle (40k faces)	T_s (s)	0.66	0.16	0.09
	n_w	668,091	6,061	6,53
	M (MB)	48.58	0.39	0.42
Armadillo (346k faces)	T_s (s)	8.93	1.80	1.22
	n_w	6,724,512	20,436	20,447
	M (MB)	488.32	1.31	1.31
Fertility (800k faces)	T_s (s)	249.38	14.12	10.62
	n_w	71,607,326	205,856	180,301
	M (MB)	5,165,33	13.17	11.54
Kitten (1.1M faces)	T_s (s)	378.01	19.57	15.74
	n_w	112,886,256	153,886	168,508
	M (MB)	8,140,97	9.85	10.78
Bunny (1.6M faces)	T_s (s)	496.06	25.77	18.83
	n_w	154,547,216	164,644	206,641
	M (MB)	11,146,6	10.54	13.23

Table 2. *Performance comparison of the PWP, MMP and VTP algorithms on anisotropic models.*

Model	Comparison	MMP	VTP	PWP (f = 0.5)
Block (800k faces)	T_s (s)	90.7	8.0	6.9
	n_w	37,940,667	97,032	92,643
	M (MB)	2741.33	6.21	5.93
Rocker arm (960k faces)	T_s (s)	225.9	13.0	12.4
	n_w	6,724,512	20,436	20,447
	M (MB)	5804.25	9.37	9.38
Impeller (1.6M faces)	T_s (s)	162.0	17.7	15.6
	n_w	72,780,334	89,928	111,279
	M (MB)	5,259,38	5.76	7.12

4. Conclusion

In this paper, we introduce PWP algorithm, a novel approach designed for discrete geodesic computation that excels in both temporal and memory efficiency. The PWP algorithm employs dual buffers for incoming windows on each list, effectively handling window data. The new structure of PWP ensures that each phase is free from data dependencies or conflicts, allowing for parallel processing on CPU. A key advantage of the PWP algorithm is its capability to process numerous windows simultaneously and autonomously, significantly enhancing its performance, particularly when applied to real-world models. This feature makes the PWP algorithm a highly efficient tool in the realm of geodesic computation.

REFERENCES

[1] D. Martınez, L. Velho, and P. C. Carvalho, "Computing geodesics on triangular meshes", *Computers & Graphics*, vol. 29, no. 5, pp. 667–675, 2005. https://doi.org/10.1016/j.cag.2005.08.003

[2] K. Polthier and M. Schmies, "Straightest geodesics on polyhedral surfaces", in *Proc. ACM SIGGRAPH 2006 Courses (SIGGRAPH '06)*, pp. 30-38, 2006. DOI:10.1145/1185657.1185664

[3] S.-Q. Xin and G.-J. Wang, "Efficiently determining a locally exact shortest path on polyhedral surfaces", *Computer-Aided Design*, vol. 39, no. 12, pp. 1081–1090, 2007. DOI: 10.1016/j.cad.2007.08.001

[4] J. S. Mitchell, D. M. Mount, and C. H. Papadimitriou, "The discrete geodesic problem", *SIAM Journal on Computing*, vol. 16, no. 4, pp. 647–668, 1987.

[5] J. Chen and Y. Han, "Shortest paths on a polyhedron", in *Proceedings of the 6th Annual symposium on Computational geometry*, 1990, pp. 360–369. https://doi.org/10.1145/98524.98601

[6] M. Novotni and R. Klein, "Computing geodesic distances on triangular meshes", in *Proceedings of WSCG'2002*, vol. 11, 2002, pp. 341-347.

[7] V. Surazhsky, T. Surazhsky, D. Kirsanov, S. J. Gortler, and H. Hoppe, "Fast exact and approximate geodesics on meshes", *ACM Transactions on Graphics (TOG)*, vol. 24, no. 3, pp. 553–560, 2005. DOI: 10.1145/1073204.1073228

[8] K. R. Varadarajan and P. K. Agarwal, "Approximating shortest paths on a nonconvex polyhedron", *SIAM Journal on Computing*, vol. 30, no. 4, pp. 1321–1340, 2000.

[9] S. Har-Peled, "Approximate shortest paths and geodesic diameter on a convex polytope in three dimensions", *Discrete & Computational Geometry*, vol. 21, no. 2, pp. 217–231, 1999.

[10] S. Har-Peled, "Constructing approximate shortest path maps in three dimensions", in *Proceedings of the 44th Annual symposium on Computational geometry*, 1998, pp. 383–391. Doi: 10.1145/276884.276927

[11] L. Aleksandrov, A. Maheshwari, and J.-R. Sack, "An improved approximation algorithm for computing geometric shortest paths", *Fundamentals of Computation Theory*, *Springer*, Vol. 2751, pp. 246–257, 2003. DOI:10.1007/978-3-540-45077-1_23

[12] S.-Q. Xin and G.-J. Wang, "Improving Chen and Han's algorithm on the discrete geodesic problem", *ACM Transactions on Graphics (TOG)*, vol. 28, no. 4, pp. 1-8, 2009. DOI: 10.1145/1559755.1559761.

[13] X. Ying, S. Xin, and Y. He, "Parallel Chen-Han (PCH) algorithm for discrete geodesics", *ACM Transactions on Graphics (TOG)*, vol. 33, no. 1, pp. 1-11, 2014. DOI: 10.1145/1559755.1559761.

[14] J. A. Sethian, "Fast marching methods", *SIAM Review*, vol. 41, no. 2, pp. 199–235, 1999. DOI: 10.1137/S0036144598347059

[15] K. Crane, C. Weischedel and M. Wardetzky, "Geodesics in heat: a new approach to computing distance based on heat flow", *ACM Transactions on Graphics (TOG)*, vol. 32, no. 5, pp. 1-11, 2013. DOI: 10.1145/2516971.2516977.

[16] X. Ying, X. Wang, and Y. He, "Saddle vertex graph (SVG): A novel solution to the discrete geodesic problem", *ACM Transactions on Graphics (TOG)*, vol. 32, no. 6, pp. 1-12, 2013. DOI: 10.1145/2508363.2508379.

[17] Y. Qin, X. Han, H. Yu, Y. Yu, and J. Zhang, "Fast and exact discrete geodesic computation based on triangle-oriented wavefront propagation", *ACM Transactions on Graphics (Proc. SIGGRAPH)*, vol. 35, no. 4, pp. 1-13, 2016. DOI: 10.1145/2897824.2925930.