Efficient and practical algorithms for discrete geodesics

Xiang, Ying

2012

Xiang, Y. (2013). Efficient and practical algorithms for discrete geodesics. Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/54871>

<https://doi.org/10.32657/10356/54871>

NANYANG TECHNOLOGICAL UNIVERSITY

Efficient and Practical Algorithms for Discrete Geodesics

A Thesis
Submitted to the School of Computer Engineering
Nanyang Technological University

by

Xiang Ying

in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

Nov, 2012

Abstract

Computing exact geodesic distance plays an important role in many graphics applications. Many research studies have been undertaken in this area since the 1980s. However, the existing geodesic algorithms are not practical for large-scale models or time-critical applications. First, the existing algorithms compute exact geodesic in a serial manner due to the lack of a parallel structure. The computation of geodesic on a large-scale model could be very time-consuming. Second, the widely studied single-source all-destination geodesic algorithms are not elegant for all-pair geodesic queries. Thus, the efficiency and practicality of geodesic computation remain a challenge.

To tackle these issues, we systematically studied the discrete geodesic problem. My study focuses on all-pairs geodesic distance, geodesic offsets, and parallel geodesic algorithm on modern GPUs. Furthermore, we propose the world’s first intrinsic and parallel Poisson-disk sampling method on surfaces, which is a time-critical application of discrete geodesics. The contributions of this work are listed as follow:

First, we study the discrete geodesics on polygonal meshes with a focus on geodesic offsets and all-pairs geodesics. Geodesic offsets on discrete representation is a geodesic distance field in which the input is typically a polyline curve restricted on the piecewise linear mesh. We propose an efficient and exact algorithm to compute the geodesic offsets on triangle meshes by extending the Xin-Wang algorithm of discrete geodesics. We define a new data structure for parallel-source windows, and extend both the "one angle one split" and the filtering theorem to maintain the windows tree. We also propose an efficient algorithm to approximate the geodesic distance between an arbitrary point pair on the given surface. We compute m sample points on the given mesh using farthest point sampling algorithm, and construct a geodesic triangulation

on those sample points. Such triangulation is used to approximate the geodesic distance between a point pair in constant time $O(1)$. The pre-processing step takes $O(mn^2 \log n)$ time for an input mesh with n vertices and m samples, where $m (\ll n)$ is specified by the user, usually between a few hundreds and several thousands. Both theoretical analysis and experimental results on real-world models demonstrate that our algorithm is efficient and accurate.

Second, we propose the parallel Chen-Han (or PCH) algorithm, which extends the classic Chen-Han (CH)s discrete geodesic algorithm to the parallel setting. The original CH algorithm and its variant lack the parallel computation. The proposed PCH algorithm is able to propagate a large number of windows simultaneously and independently. Extensive experiments on real-world models demonstrate an order of magnitude improvement in execution time compared to the state-of-the-art approaches.

And finally, we present an accurate and parallel algorithm for Poisson disk sampling on arbitrary surfaces. Sampling plays an important role in a variety of graphics applications. Very few research studies have been reported with regard to the problem of generating Poisson disks on surfaces due to the complicated nature of the surface. Our approach assigns each sample candidate a random and unique priority that is unbiased with regard to the distribution. Hence, multiple threads can process the candidates simultaneously and resolve conflicts by checking the given priority values. To our knowledge, this is the first accurate and parallel algorithm for generating Poisson disk patterns on surfaces based on the exact geodesic distance.

It is highly desirable to develop a fast algorithm to compute the exact geodesic on triangle meshes. However, existing exact geodesic algorithms have very strong data dependence and do not support the parallel computation. We have presented the constant time all-pair algorithm for the approximate geodesic, the extended Xin-Wang algorithm for geodesic offsets, and the PCH algorithm, parallelizing the discrete geodesic algorithm. We also demonstrate an application of discrete geodesic on GPUs.

Acknowledgments

I am grateful to all those who have helped and supported me throughout my PHD life.

First and foremost I offer my sincerest gratitude to my supervisor, Dr. He Ying. Without his support, patience, and guidance, this study would not have been completed. His knowledge and enthusiasm on research inspired and motivated me. I have been extremely lucky to have a supervisor who cared so much about my work, and who responded to my questions and queries so promptly. One simply could not wish for a better or friendlier supervisor.

I would grateful acknowledge my undergraduate program contest coach, Mr. Yu Ruiguo, who leads me to the beautiful world of computer science.

I also wish to thank all my friends of the Geometry Modeling Group at gameLab, who supported me a lot by providing valuable feedback in many fruitful discussions. This report would not be possible in this form without the support and collaboration of several friends, in particular Ms. Chen Shuangmin, Dr. Chen Xiaoming, Ms. Dao Thi Phuong Quynh, Mr. Han Shuchu, Dr. Jin Jianqiu, Mr. Lai Chi-Fu William, Ms. Li Yusha, Dr. Lin Juncong, Ms. Liu Yaqiong, Ms. Meng Min, Mr. Wang Dayong, Mr. Wang Xiaoning, Ms. Sun Qian, Mr. Tan Fuwen, Mr. Xia Jiazhi, Dr. Xin Shiqing, Ms. Yan Xiaoqi, and Mr Zhang Minqi.

I would like to thank my parents for their support during my study. And many thanks to my friends who helped me in many ways, Mr. Li Ge, Mr. Zhou Peng, Mr. Wu Jiajun, Mr. Li Siyi. Last but not least, words alone can not express the thanks I owe to Cao Yingxiu, my girlfriend, for her trust and encouragement.

Publications

Journal Publications

- [1] **Xiang Ying**, Xiaoning Wang, and Ying He. Saddle Vertex Graph(SVG): A Novel Solution to the Discrete Geodesic Problem. *ACM Transactions on Graphics (TOG) (Special issue of SIGGRAPH Asia '13)*, conditionally accepted, 2013.
- [2] **Xiang Ying**, Shi-Qing Xin, and Ying He. Parallel Chen-Han (PCH) Algorithm for Discrete Geodesics. *ACM Transactions on Graphics (TOG)*, accepted, 2013.
- [3] **Xiang Ying**, Zhenhua Li, and Ying He. A Parallel Algorithm for Improving the Maximal Property of Poisson Disk Sampling. *Computer-Aided Design (Special issue of SIAM/ACM GD/SPM '13)*, Vol. 45, No. 11, 2013.
- [4] **Xiang Ying**, Shi-Qing Xin, Qian Sun, and Ying He. An Intrinsic Algorithm for Parallel Poisson Disk Sampling on Arbitrary Surfaces. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, Vol. 19, No. 9, pp. 1425-1437, 2013.
- [5] Long Zhang, Jiazhi Xia, **Xiang Ying**, Ying He, Wolfgang Mueller-Wittig, and Hock Soon Seah. Efficient and robust 3D line drawings using difference-of-Gaussian. *Graphical Models (Proceedings of Geometric Modeling and Processing GMP'12)*, Vol. 74, No. 4, pp. 87-98, 2012.
- [6] Shi-Qing Xin, **Xiang Ying**, and Ying He. Efficiently Computing Geodesic Offsets on Triangle Meshes by Extended Xin-Wang Algorithm. *Computer-Aided Design (Special issue of SIAM/ACM GD/SPM '11)*, Vol. 43, No. 11, pp. 1468-1476, 2011.

Conference Publications

- [1] Qian Sun, Long Zhang, Minqi Zhang, **Xiang Ying**, Shi-Qing Xin, Jiazhi Xia, and Ying He. Texture Brush: An Interactive Surface Texturing Interface. *Proceedings of ACM Symposium on Interactive 3D Graphics and Games(I3D)*, pp. 153-160, 2013.
- [2] **Xiang Ying**, Zhenhua Li, and Ying He. A parallel algorithm for improving the maximal property of Poisson disk sampling in R^2 and R^3 . *I3D '13 Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (Posters)*, pp. 179-179, 2013.
- [3] Shi-Qing Xin, Dao T.P. Quynh, **Xiang Ying**, and Ying He. Computing Defect-Insensitive Geodesic Distance on Broken Meshes. *ACM SIGGRAPH Asia 2012 Technical Briefs*. 2012.
- [4] Shi-Qing Xin, **Xiang Ying**, and Ying He. Constant time all-pairs geodesic distance query on triangle meshes. *Proceedings of ACM Symposium on Interactive 3D Graphics and Games (I3D '12)*, pp. 31-38, 2012.
- [5] **Xiang Ying**, Shi-Qing Xin, Qian Sun, and Ying He. Parallel and accurate Poisson disk sampling on arbitrary surfaces. *ACM SIGGRAPH ASIA 2011 Technical Sketches*, Article No. 18, 2011.
- [6] Shi-Qing Xin, **Xiang Ying**, and Ying He. Constant time $O(1)$ all-pairs geodesic distance query on triangle meshes. *ACM SIGGRAPH ASIA 2011 Technical Sketches*, Article No. 23, 2011.

Contents

Abstract	i
Acknowledgments	iii
Publications	iv
List of Figures	ix
List of Tables	xix
1 Introduction	1
1.1 Motivation	1
1.1.1 Discrete Geodesics	1
1.1.2 Applications	3
1.2 Objectives and Contributions	8
1.3 Thesis Organization	11
2 Related Work	12
2.1 Geodesic distance on manifolds	12
2.2 Point-source geodesic distance field	13
2.3 Geodesic path	14
2.4 Geodesic offsets	15
2.5 All-pairs geodesic	16
2.6 Poisson disk sampling	17
3 Geodesic Offset Algorithm	22
3.1 Introduction	22
3.2 Xin-Wang Algorithm of Discrete Geodesics	24
3.2.1 Edge Sequences and Windows	24

3.2.2	Window Derivation	25
3.2.3	Two Key Observations	27
3.3	Extending Xin-Wang Algorithm To The General Geodesic Offset Problem . . .	28
3.3.1	Parallel-Source Window	28
3.3.2	Extending Chen and Han’s “One-Angle-One-Split”	30
3.3.3	Extending Xin and Wang’s Filtering Theorem	31
3.3.4	Complexity Analysis	34
3.4	Experimental Results	34
4	All Pairs Geodesic Algorithm	40
4.1	Overview	42
4.2	Pre-Processing	44
4.2.1	Constructing Geodesic Triangulation	44
4.2.2	Generating Geodesic Distance File	44
4.3	Pairwise Geodesic Distance Query	45
4.3.1	Unfolding Geodesic Triangles	45
4.3.2	Algorithm	45
4.3.3	Error Analysis	47
4.4	Results & Discussions	47
4.4.1	Experimental Results	47
4.4.2	Geodesic Path & Geodesic Distance Field	48
4.4.3	Discussions	50
4.4.4	Limitations	52
4.5	Error Analysis	53
5	Parallel Algorithm for Computing Single-Source-All-Destination Geodesic	62
5.1	Introduction	63
5.2	Preliminary	65
5.3	Computing Geodesics in Parallel	69
5.3.1	Data Structure	69
5.3.2	PCH Algorithm	71
5.3.3	Correctness	77

5.4	Implementation & Experimental Results	78
5.4.1	Implementation Details	78
5.4.2	Parameter Setting	82
5.4.3	Performance	84
5.5	Comparison & Discussion	86
5.6	Conclusion	88
6	Application: Poisson Disk Sampling	90
6.1	Introduction	91
6.2	Intrinsic and Parallel Poisson Disk Sampling On Arbitrary Surfaces	93
6.2.1	Algorithmic Overview	93
6.2.2	Parallel Dart Throwing on Surfaces (PDTS)	96
6.2.3	Parallel Sampling from Dense Points (PSDP)	99
6.2.4	Correctness	100
6.3	Improving the Maximal Property of Poisson Disk Sampling in $\mathbb{R}^2$ and $\mathbb{R}^3$. . .	104
6.3.1	A Geometric Problem	104
6.3.2	Algorithm	108
6.4	Experimental Results	111
6.5	Comparison & Discussions	127
7	Conclusions & Future Work	132
7.1	Conclusions	132
7.2	Future Work	134
	References	137

List of Figures

1.1	Geodesics on sphere.	1
1.2	Geodesic is a fundamental concept in geometry. Left: a geodesic path between two points. Right: a geodesic distance field on surface.	2
1.3	Exponential map.	4
1.4	Shape distribution descriptor: an elephant model is described by a histogram of geodesic distances.	5
1.5	Example of shape retrieval using geodesic distance distribution descriptor. The leftmost shape is the query shape. The left ones are query results.	5
1.6	Poisson disk sampling is widely used in computer graphics, such as texture synthesis (left) and image stippling (right).	6
1.7	The human eye’s retinal cells constitute a distribution of Poisson disk sampling, as observed by Yellott [40]. Left: retinal cells. Right: spectrum of retinal cells.	6
1.8	Typical Poisson disk sampling and its spectrum.	7
2.1	Geodesic curve on Manifolds.	13
2.2	The improved CH algorithm [93] can compute the exact geodesic distance field with linear memory cost.	14
2.3	Geodesic path: shortest path on surface.	15
2.4	Distance offsets.	15
2.5	Holla <i>et al.</i> [38] first computed geodesic offsets on surfaces. Their method only worked well on convex surfaces.	16
2.6	Typical Poisson disk distribution in $\mathbb{R}^2$ and its power spectrum, radial mean and anisotropy.	17
2.7	Poisson disk sampling is the key technology in halftoning and image stippling.	18

2.8	Poisson disk sampling on surfaces	19
2.9	Direct Poisson disk sampling on surfaces could guide hair placement.	20
2.10	The current parallel Poisson disk sampling on surfaces[9] accept each sample from the discrete random points assigned to each cell (a). The 3D object is divided to cells, and the cells are visited in a pre-defined order (b). This cell-based sampling is not accurate, proved by [34].	20
3.1	User specifies 4 source points and 3 generator curves (drawn in yellow) on the Happy Buddha with $100K$ vertices. Our algorithm computes the exact geodesic offsets in 3.2 seconds with 3.06MB memory usage. The timing is measured on a DELL Workstation with an Xeon 2.66GHz CPU and 4GB RAM.	23
3.2	All the shortest paths from s to any point $p \in [a, b]$ pass through the same edge sequence $(e_1, e_2, \dots, e_k)$. Therefore we can use a funnel-like interval window to encode all such shortest paths.	24
3.3	Computing the children of a window: (a) A pseudo-source window at a saddle vertex v may have children both on opposite edges and at adjacent vertices; (b-d) An interval window w on edge $\overline{v_1 v_2}$ may have one or two interval-window children, depending on how the line segment $\overline{Iv}$ intersects w 's interval.	26
3.4	Two key observations in Xin-Wang algorithm [93]: (a) Chen and Han's "one angle one split" [10] states that at most one of the windows covering the same angle can have two children; and (b) Xin and Wang's filtering theorem [93] asserts that an interval window is no use if it cannot provide a shorter geodesic distance than one of the neighboring vertices.	27
3.5	Three types of windows, i.e., pseudo-source windows, interval windows and parallel-source windows are required to compute the general offset problem. (a) Taking the line segment AB as the source, we need to constrcut three parallel-source windows initially. (b) We also need to construct 6 interval windows for the endpoints A and B	29
3.6	Extended "one angle one split": If the parallel-source window w can provide a shorter distance to v_3 than the existing winning window, then we allow w to have two children. If w can have only one child, then we remove either w_1 (see (a)) or w_2 (see (b)).	30

3.7	The parallel-source window cannot contribute to determining a shortest path if v_1 gives a shorter distance than $\ BD\ $, or v_2 gives a shorter distance than $\ AC\ $, or v_3 gives a shorter distance than $\ AC\ $	31
3.8	Case 3.	33
3.9	Experimental results. The generator curves are drawn in yellow. Cold (resp. warm) colors mean small (resp. large) geodesic distances.	35
3.10	Taking salient features (e.g., eyes, forehead, nose and mouth) as generators, we compute the geodesic offsets, which can be used for face segmentation.	36
3.11	Given the generator curve on the waist, we compute the geodesic distance field of the Lucy model. Then we show the set of offset curves with spacing of 0.06 and 0.025 respectively. The model is normalized to a unit cube.	37
3.12	Our offset algorithm is intrinsic to the geometry and insensitive to the resolution/triangulation. Left: the Moai model with 11K vertices; Right: the Moai model with 3K vertices.	37
3.13	Comparison to Dijkstra’s algorithm [23] and the fast marching method [43]. Note that our algorithm is able to compute the <i>exact</i> geodesic offsets while the others compute only the <i>approximate</i> geodesic offsets (see the artifacts in (a) and (b)).	38
3.14	Geodesics (drawn in blue) starting from the generator curves (drawn in yellow) are orthogonal to the offset curves (drawn in gray) everywhere.	39
4.1	Our algorithm can approximate the geodesic distance between arbitrary pair of points in constant time . The user can control the approximation error by specifying the number of sample points on the input model. Furthermore, both the geodesic path and geodesic distance field can be approximated in linear time . With 1000 sample points on the BUNNY model ($n = 72K$), the relative root-mean-square error is less than 0.5%. (a) shows an example of geodesic distance query and the corresponding exact (in green) and approximate (in red) geodesic paths. (b) and (d) show the front and back views of the exact geodesic distance field by [Xin and Wang 2009]. (c) and (e) show the approximate result by our method. Cold (resp. warm) colors represent short (resp. long) distances.	41

4.2	Algorithm pipeline. Given the input mesh M , we first sample m points distributed uniformly on M (see (a)); Then taking the sample points as sources, we compute the geodesic distance field on M (see (b)) and construct an approximate Delaunay triangulation (see (c)), which induces a geodesic triangulation by replacing each Delaunay edge with a geodesic (see (d)). Next, the exact geodesic distances between any pair of sample points are saved in a file. Finally, with the pre-computed geodesic distance file, we can approximate the geodesic distance between any pair of query points in constant time $O(1)$ and find the corresponding geodesic path in linear time $O(k)$, where k is the number of edges crossed by the path (see (e)).	43
4.3	(a) The unfolding operation flattens curved geodesic triangles to $\mathbb{R}^2$ such that the corresponding edge lengths are equal, i.e., $d(r, s) = \ r's'\ $, $d(p, r) = \ p'r'\ $, $d(p, s) = \ p's'\ $, $d(q, r) = \ q'r'\ $, $d(q, s) = \ q's'\ $. (b) and (c) show two important cases of applying unfolding operation in our algorithm.	46
4.4	Error distribution of ARMADILLO model. 40.27%, 58.45% and 68.40% relative errors are in the range $[-0.2\%, +0.2\%]$ for $m = 1000, 2000$ and 3000 , respectively.	49
4.5	Numerical results. Top: Relative root-mean-square error VS number of sample points; Bottom: Preprocessing time (seconds) VS number of sample points. . .	50
4.6	Computing geodesic distance field on BUDDHA ($n = 50K$). The user can specify source points arbitrarily on the 3D model, then our program is able to compute the approximate geodesic distance field in linear time. The first three columns show the approximate results by our algorithm with $m = 1000, 2000$, and 3000 respectively. The last column shows the exact result by [Xin and Wang 2009].	51
4.7	Comparison to the linear interpolation method. (a) Geodesic triangulation. (b) Exact result. (c) The approximate result by unfolding technique. (d) The approximate result by linear interpolation.	52
4.8	(a) Query points located in the same geodesic triangle may lead to large approximation error. (b) Improved result can be obtained by constructing a finer geodesic triangulation that separates highly curved features.	53

4.9	Computing geodesic distance field on the GARGOYLE model ($n = 350K$). The user can specify source points arbitrarily on the 3D model, then our program is able to compute the approximate geodesic distance field in linear time, which allows the user to view the geodesic distance field at interactive speed.	53
4.10	Computing geodesic distance field on the ARMADILLO model ($n = 173K$).	54
4.11	Computing geodesic distance field on the FERTILITY model ($n = 30K$).	55
4.12	Computing geodesic distance field on the LUCY model ($n = 263K$).	56
4.13	Lemma 1	57
4.14	The upper bound in Lemma 1 is tight.	58
4.15	Lemma 2.	59
5.1	Computing the exact geodesic distance on triangle meshes by using the proposed PCH algorithm. Our implementation on Nvidia GTX 580 is one magnitude of order faster than [Xin and Wang 2009].	66
5.2	(a) A window represents an interval $[A, B]$ on an oriented edge e , which is “visible” by the pseudo source I after unfolding the corresponding face sequence. (b) The position of the pseudo source on the unfolded plane can be determined by d_0 and d_1 .	67
5.3	Window propagation results in new windows. (a) A window creates one child on the opposite edge. (b) A window creates two children on opposite edges. (c) For a saddle point v , the injection ray (the dashed red line) is split into two directions (dashed grey lines), each is an extended straight line of the injection ray after unfolding. Then the new windows are generated on the opposite edges in the fanned area formed by the two directions.	68
5.4	Window filter. (a) The “one angle one split” filter in the CH algorithm. Two windows w_1 and w_2 share the same edge (v_0, v_1) and occupy the vertex v_2 . However, the pseudo source I_1 can not provide a shorter distance to v_2 . Thus, only window w_2 can generate two children. (b) The filter used in the ICH algorithm. The window w on edge (v_0, v_1) is generated by some window on edge (v_1, v_2) . Let g_i be the <i>current</i> shortest distance for vertex v_i , $i = 0, 1, 2$. The window w is useless, if one of the following inequalities holds: $d + \ IB\ > g_0 + \ v_0B\ $ or $d + \ IA\ > g_1 + \ v_1A\ $ or $d + \ IB\ > g_2 + \ v_2B\ $.	69

5.5	The ICH algorithm maintains the windows in a priority queue. Only the nearest window is processed. Such priority queue structure is very difficult to maintain in parallel. However, the PCH algorithm put all windows together. In each iteration, we select k nearest windows and process them simultaneously.	72
5.6	During each iteration, our algorithm selects k windows that are nearest to the source points and assigns these windows to T GPU threads. Each window is taken by one GPU thread and then propagates independently, which will result in one or more children windows. After all the selected windows have been processed, our algorithm collects the newly-generated windows and organize all windows such that there are no memory gaps between adjacent windows in the memory pool. The empty slots in the memory pool are drawn in light gray.	73
5.7	The window selection and event sorting on the GPU take 3 to 6 seconds on the 144K-face Bunny model. We implement an approximate selection on the GPU and process the events on the CPU, which can significantly reduce the overall execution time. The vertical axes are the execution time in seconds and the total number of windows. The horizontal axes are the selection parameter k . Although the window selection is approximate, the computed geodesic distance is exact.	79
5.8	Testing the selection parameter k on an Nvidia GTX 580 with 512 cores. The total number of windows (in red) is approximately linear proportional to the selection parameter k (the horizontal axis). Choosing a small k does not utilize the GPU's parallel structure at all. On the other hand, choosing a very large k would produce too many useless windows, resulting in slow performance (see the blue timing curves). We found the optimal range for k is $[16 \times 2^{10}, 24 \times 2^{10}]$, which leads to the least execution time.	81
5.9	Given the Bunny model of various resolutions, we found the optimal range of k is insensitive to the mesh resolution.	83
5.10	More results.	83
5.11	The performance improvement T_{ICH}/T_{PCH} is consistent with the GPU double precision performance (GFLOPS).	84

6.1	Our parallel algorithm is able to randomly and uniformly distribute samples on arbitrary surfaces. The generated distribution faithfully exhibits the blue noise pattern.	93
6.2	Illustration of PDTs with five available threads. s_1 is an accepted Poisson disk sample. Each thread handles one active sample p_i , which has a unique and random priority value. PDTs accepts p_2, p_4 and p_5 , and rejects p_1 and p_3	98
6.3	Comparison between PDTs (left) and PSDP (right). Assume s_1 is an accepted sample. In PDTs, each thread generates one active sample at a random location. Thus, it is likely that the active samples collide with the accepted samples. For example, p_1 and p_2 are rejected due to the collision with s_1 , and p_3 is accepted since it does not conflict with any accepted and active samples. Although being parallel, this strategy is not efficient, since two geodesic disks must be computed for p_1 and p_2 respectively. However, p_1 and p_2 do not contribute to the results. By contrast, PSDP randomly picks samples from the predefined dense points $\mathcal{P}$. When s_1 is accepted, PSDP rejects all the idle samples (in yellow) inside $D(s_1, 2r)$. Therefore, in the future iterations, PSDP will pick only the samples (in green) that are outside $D(s_1, 2r)$. In other words, no active samples collide with the accepted samples in PSDP.	102
6.4	Data structure of PSDP. The array $\mathcal{P}$ contains the dense point clouds, each point has a random location on M . To allow the ICH algorithm efficiently access the samples during the wavefront propagation, we also insert each sample p_i into the triangle containing p_i . The array PointIndex is a random permutation of $\{1, \dots, \mathcal{P} \}$. We partition PointIndex into T equally sized groups, $group_i$, $i = 1, \dots, T$	102
6.5	An “forbidden” arc $\widehat{p_1 p_2}$ (in red) with two ending angles $[\alpha, \beta]$	105
6.6	When multiple circles C'_i intersect at one point p , such point p is a degenerate but “permitted” arc, which allows insertion a circle with radius r	106
6.7	(a) Several spheres intersect. (b) Sphere S' intersects another two spheres, resulting two intersecting c_1 and c_2 , respectively.	108

6.8	Maximal Poisson disk sampling in $\mathbb{R}^2$. (a) Our algorithm takes a non-maximal Poisson disk sampling as input. (b) For each sample, we draw a circle with radius $2r$ and then compute the permitted arcs in parallel. (c) The permitted arcs enclose the regions which can insert additional samples. (d) We randomly create a Poisson disk candidate for <i>each</i> permitted arc. Each candidate is assigned a random and unique priority that is unbiased with regard to the distribution. Hence, multiple threads can process the candidates simultaneously and resolve conflicts by checking the given priority values. The candidates with the higher priority win and are accepted as Poisson disks.	110
6.9	Experimental results. For each model we generate three density levels, at approximately 1K, 5K and 10K sample points. The images are generated at high resolution to allow for zoom-in examination.	111
6.10	We applied the manifold harmonics transformation [9] for spectral analysis on surfaces. As suggested in [9], we generated 300-500 samples on the models and used 10 runs for each test. The horizontal axis is the basis frequency, the vertical axes are the radial mean and the anisotropy. We tested both phase group method and our priority based parallelization with approximate and exact geodesic respectively. We also implemented the traditional dart throwing as ground truth (in green). The spectra of our method are consistent with the blue noise patterns in all examples, while the anisotropy plots of phase group methods (with the exact geodesic [93] and the approximate geodesic [9]) deviate from the ground truth. This implies that such bias of the phase group method does <i>not</i> come from the distance metric used.	113
6.11	For the samples near the self-intersection, the extrinsic approach [9] may report “false” conflicts between valid samples since their spatial distance is small. Our method can accurately detect the conflicts using the intrinsic geodesic distance, thus, resulting in the distribution with more valid samples.	114
6.12	Performance of our algorithm.	115
6.13	Our method is robust and insensitive to the mesh resolution and tessellation. From left to right: distributing 700 Poisson disks on Bunny models with 1.4K, 14K and 144K triangles respectively. Our algorithm works well for all cases. . .	115

6.14	Sampling quality. We generate samples by using the state-of-the-art technique [9] and our method. To visualize the results, we compute the geodesic disks centered at each sample with radius r . Due to the approximate geodesic distance in [9], some samples violate the minimal distance constraints, as highlighted in the black boxes. Our method measures the exact geodesic distance, thus, can guarantee that the generated distribution is free of conflicts.	117
6.15	Our algorithm is independent of the surface embedding. Row 1: two examples of Poisson disk sampling on 4D surfaces. For rendering purpose, the surfaces are projected into $\mathbb{R}^3$; Row 2: we distribute Poisson disks on one pose and the shape correspondence induces the sampling on the other pose. The intrinsic property of our algorithm guarantees the induced Poisson disk distribution is uniformly distributed and free of conflicts.	118
6.16	Adaptive sampling on Caesar model. Left: the spatially varying function specified by the user, where red means dense sampling (small radius) and green means sparse sampling (large radius); Right: the adaptive sampling result. . . .	119
6.17	Our algorithm also works for Euclidean space of arbitrary dimensions. (a) 2D case, $r = 0.0147$, # samples = 787. (b) 3D case, $r = 0.023$, # samples = 6064. (c) Cut view of (b). (d)-(f) The average power spectrum for a 2D case with $r = 0.00268$ and 21886 samples, the radial power spectrum and the anisotropy spectrum. The spectra are consistent with the blue noise patterns, since our algorithm is accurate. Our implementation on NVIDIA GTX 580 generates 13 million 2D samples/second and 1.5 million 3D samples/second.	120
6.18	Maximal Poisson disk sampling results. (a) 3691 samples in $\mathbb{R}^2$, $r = 0.00682$. (b) 3434 samples in $\mathbb{R}^3$, $r = 0.03$	121
6.19	The distribution generated by PSDP method is not maximal. For the Poisson disk sampling on a unit square with $r = 0.005$, the radius statics ρ of the maximal sampling algorithm [29] is very stable, all values fall in the range $[0.778, 0.780]$ with an average 0.7791. Our ρ of PSDP depends on the dense-points-to-Poisson-disks ratio $ \mathcal{P} / \mathcal{S} $	122
6.20	Spectrum analysis. Radius=0.00268, about 22K samples. This is average spectrum over 10 runs.	122

6.21	Performance: our maximal sampling algorithm processes about 8 million 2D samples per second, and 0.55 million 3D samples per second. The horizontal axis shows the number of samples and the vertical axis shows the sampling speed. Tested on a Dell Workstation with Xeon 2.66GHz Quad Core CPU, 12GB memory and an Nvidia GTX580.	123
6.22	Our method is effective to deal with 2D irregular domain. For arbitrary 2D polygon, we can clip the circles C'_i and put the arcs outside of the domain in the <i>forbiddenArcList</i> . This can guarantee that the permitted arcs are always inside the domain.	124
6.23	Maximal Poisson disk sampling in 2D polygons.	124
6.24	Vector field visualization.	125
6.25	Stippling by adaptive blue noise sampling.	125
6.26	Mosaic tile placement.	126
6.27	To make a fair comparison of the performance to [9], we adopt the approximate geodesic distance into our priority based parallelization framework. (a) Take the performance of priority based parallelization with exact geodesic distance as one. Adopting the approximate geodesic can greatly boost the performance of our sampling framework, when the number of samples is small. However, such speedup is not significant when the sample number is very high and each geodesic disk becomes very small. (b) shows the relative performance of our priority based parallelization to the phase group method [9]. Our method is about 5% to 15% slower than their method due to the fact that some computing samples may be rejected in our approach.	130

List of Tables

3.1	Timing statistics (seconds) measured on a workstation with an Xeon 2.66GHz CPU and 4GB RAM.	36
4.1	Query speed (# queries/second).	48
5.1	Global variables.	71
5.2	Statistics of “multiple-source-all-destination” distance. Execution time was measured in seconds.	85
5.3	Model complexity and performance. Tests were repeated 100 times with random source point and the mean values are reported in this table. The last column shows both the worst and average speedup.	85
6.1	Statistics of the number of calls for CheckStatus() on Gargoyle model. Columns show the percentage of the number of function calls.	114
6.2	Comparison to other surface sampling algorithms.	127

Chapter 1

Introduction

1.1 Motivation

1.1.1 Discrete Geodesics

The term “geodesic” is derived from *geodesy*, which is the science of measuring the size and shape of the Earth [90]. In its original definition, a geodesic is the shortest route between two points on the Earth’s surface, namely, a segment of a great circle(Figure. 1.1).

In Euclidean space, the shortest distance is the straight line between two points. In differential geometry, the geodesic is the extension of straight line on curved surface. A geodesic line is a locally length-minimizing curve. The geodesic distance is the shortest distance between two points on a surface.

Figure 1.1: Geodesics on sphere.

Figure 1.2: Geodesic is a fundamental concept in geometry. Left: a geodesic path between two points. Right: a geodesic distance field on surface.

Geodesic is useful due to its several good properties. First, it is a distance metric, meaning that the geodesic from point x to point y is non-negative, symmetric and satisfies the triangle inequality. Second, geodesic is locally isotropic, i.e., it gradually increases in the neighborhood of the source vertex. Third, geodesic is intrinsic, thus isometry invariant; it does not change with isometric transformation of the surface.

As a fundamental problem in geometric modeling, computing geodesics on triangle meshes has been widely studied since mid 1980. Many elegant algorithms have been proposed to compute the “single-source-all-destination” geodesic distance, e.g., the MMP algorithm [58], the CH algorithm [10], the improved CH (or ICH) algorithm [93], the fast marching method [43] and the approximate MMP algorithm [81]. While the state-of-the-art approaches [81] [93] work quite well for models with moderate size, they are not practical for large-scale models or time-critical applications due to their high computational costs.

The discrete geodesic problem can be classified into three categories, namely single-source-all-destination geodesic, single-source-single-destination geodesic and all-pairs geodesic. Although the three sub-problems are closely related to one another, the existing approaches usually deal with them separately. Most of the existing methods focus on single-source-all-destination geodesic, which also solve the second problem, the single-source-single-destination geodesic. However, very few work has been undertaken on the all-pairs problem due to its high complexity.

In this thesis, we systematically study the geodesic problems, including the all-pairs geodesic and geodesic offsets. For the single-source problem, even though it’s widely studied, the

geodesic computation is still very time-consuming. The past decade has witnessed an increasing trend that performs the traditionally-CPU-handled computation on the graphics processing unit (GPU), which uses a large number of graphics computational units to parallelize the computation. However, it is technically challenging to develop parallel algorithms for discrete geodesics due to the lack of parallel data structure, i.e., the geodesic distance is propagated from the source to all destinations in a sequential order. So far, the only parallel geodesic algorithm is developed by Weber et al. [85], who proposed a raster scan-based version of the fast marching algorithm. Although their method is highly efficient, it computes only the first-order approximation of geodesic and also requires parameterizing the surface into a regular domain, which is usually difficult for surfaces with complicated geometry and/or topology. To our knowledge, there are no parallel algorithms for computing exact geodesics on triangle meshes. In this thesis, we propose a parallel algorithm to significantly speed up the computation of geodesics.

1.1.2 Applications

Geodesic distance was used in robot motion and terrain navigation and has been one major topic in computational geometry in the 1980s [58]. Since geodesic measures the intrinsic, locally shortest distance between two points on the surface, it also plays a very important role in geometric computation and analysis.

There are many applications, such as surface parametrization [48], exponential map (Figure. 1.3), remeshing [70], non-rigid registration [39], shape analysis and shape segmentation [76, 96], etc, which require the query of geodesic distances between any pair of points on a given polygonal mesh.

Exponential Map. Exponential map is a local parameterization method for mapping a 2D image onto a surface. Schmidt et al. [77] introduced discrete exponential maps to interactive decal compositing.

Taking the geodesic distances and relative angles(which is the bi-product of computing geodesic), we can build a polar coordinate on surfaces (Figures 1.3(a) and (b)). Such an exponential map is an efficient local parametrization, which could be used for texture mapping (Figure 1.3(c)).

Figure 1.3: Exponential map.

Shape Descriptor. Content-based 3D surface retrieval is an important topic in computer graphics and computer vision. In 2002, Osada *et al.* [62] proposed a fast approach for shape comparison using a low-dimension descriptor, called shape distribution.

The Euclidean shape distribution is a set of distances:

$$\forall 0 \leq i, j < N, d_{i,j} = \|x_i - x_j\|,$$

where $\{x_i\}$ is a uniform sampling of manifold.

Osada *et al.* [62] also extend this shape distribution descriptor to manifold surfaces by replacing the distance $\|\cdot\|$ by geodesic distance. The distance histogram is a good low-dimensional descriptor for shape comparison. Figure 1.4 shows the geodesic distance distribution descriptor of an elephant model.

The geodesic distance histogram is a simple but practical signature of shape, which could be used for supervised or unsupervised recognition (Figure 1.5). A detailed survey of geodesic distance in shape distribution could be referred to in [70].

Poisson disk sampling on surfaces. Geodesic distances also play a significant role in the problem of Poisson disk sampling. Poisson disk sampling [19] is a widely used sampling method in computer graphics. It distributes samples that are uniformly and randomly located in a domain by requiring that the samples are as dense as possible but are at a minimum distance apart from one another. A typical example of Poisson disk sampling is the distribution of natural human eye's retinal cells, as observed by Yellott [40] (Figure 1.7).

Figure 1.4: Shape distribution descriptor: an elephant model is described by a histogram of geodesic distances.

Figure 1.6: Poisson disk sampling is widely used in computer graphics, such as texture synthesis (left) and image stippling (right).

Figure 1.7: The human eye's retinal cells constitute a distribution of Poisson disk sampling, as observed by Yellott [40]. Left: retinal cells. Right: spectrum of retinal cells.

Figure 1.8: Typical Poisson disk sampling and its spectrum.

Theoretical evidence in favor of the Poisson-disk sampling pattern was presented by Yellott [40]. He proposed that the least-conspicuous form of aliasing would be produced if the spectrum of the sampling pattern has two properties. First, the spectrum should be noisy and not have any concentrated spikes of energy. If present, such spikes could yield coherent aliasing in the same way that uniform sampling can. Second, the spectrum should have a deficiency of low-frequency energy. This causes aliasing noise to be concentrated in higher, less conspicuous frequencies. These conditions will be referred to as the *blue-noise criteria*.

Due to its excellent spatial and spectral properties, Poisson disk sampling is widely used in computer graphics, such as anti-aliasing, global illumination, non-photorealistic rendering, remeshing, texture synthesis, etc.

Dart throwing, introduced to the graphics community by [24], is the first accurate approach used to generate Poisson disk distributions in $\mathbb{R}^n$. However, this brute-force approach is impractical and inefficient, since a large number of samples are involved in the attempt but only a small percentage of them are eventually employed in the distribution. Since then, many approaches have been proposed to improve the performance, such as jittered sampling [19], spatial data structures [27, 34], procedure tiling [45, 44, 63], hierarchical sampling [57, 56], and parallel sampling [86], etc.

While tremendous efforts have been focused on multidimensional sampling in Euclidean space, very few studies have been reported on generating Poisson disk distributions on curved

surfaces. This remains a challenge due to the following reasons: first, a surface is a two-dimensional manifold which has arbitrary topology and complicated geometry, and is embedded in $\mathbb{R}^3$ or even higher dimensional space. Second, the exact geodesic distance should be used to enforce the minimum distance constraint between any pair of samples. Third, the algorithm should be parallelized such that it can make full use of all available threads. Fourth, the algorithm should be robust and insensitive to the mesh resolution and tessellation. Last but not least, the generated samples should be randomly and uniformly distributed on surfaces, exhibiting the blue noise criteria.

1.2 Objectives and Contributions

This research aims to find geodesic algorithms that are efficient. There are several parts of our objectives, including efficient algorithms for geodesic offsets, all-pair geodesic query, and parallel exact geodesic on modern GPUs. We also demonstrate an application of geodesic distance, which is the parallel and intrinsic Poisson-disk sampling on surfaces.

Geodesic Offsets. Geodesic offsets or parallels are a set of points on the surface, which are at a constant geodesic distance from the generator curves. Such offsets are useful for generating an anisotropy disk on surfaces.

The computation of geodesic offsets also plays an important role in many applications of computer-aided design, such as solid modeling [75, 32], robot path planning [53], generation of tool paths for NC machining [55, 14, 42], etc.

Although it is well studied in classical differential geometry and computer-aided design, where the underlying surface is sufficiently smooth (e.g., [68, 36, 72]), very few algorithms are available for computation of discrete geodesic offsets: Given a polygonal mesh $\mathcal{S}$ and a set of points and polylines, denoted by $\mathcal{U}$, that are restricted on $\mathcal{S}$, the distance- d geodesic offset consists of the points on $\mathcal{S}$ that are at a geodesic distance d from $\mathcal{U}$.

Holla *et al.* [38] proposed an incremental approach to approximate polyline-source offsets based on an assumption that the offset to a polyline on the surface is still a polyline. Chen *et al.* [13] computed approximate offsets by taking the samples on the source curve as source

points. Lensch *et al.* [49] extended the classic MMP algorithm [58] to compute the polyline-source geodesic distance field. However, these algorithms are either approximate or computationally expensive or memory inefficient, making it hard to apply them to large-scale real-world models.

In this thesis, we propose an efficient and exact algorithm to compute the geodesic offsets on triangle meshes by extending the Xin-Wang algorithm of discrete geodesics. Geodesic offsets on discrete representation is a geodesic distance field in which the input is typically a polyline curve restricted on a piecewise linear mesh.

Approximate Geodesic Distance. As shown before, Poisson disk sampling on surfaces rely heavily on the geodesic distance. Computing the approximate geodesic distance between an arbitrary pair of points with small error can significantly improve the performance of Poisson disk sampling.

Geodesic plays an important role in geometric computation and analysis. There are many applications, such as surface parametrization [48], remeshing [70], non-rigid registration [39] and shape segmentation [76, 96], requiring the query of geodesic distance between any pair of points on the given polygonal meshes. Rather than the widely studied “single source, all destination” discrete geodesic problem, very little work has been reported on the “all-pairs” geodesic distance query. So far, the best known result has been reported by Cook IV and Wenk [20], who pre-computed the pairwise geodesic between any two mesh vertices in $O(n^5 2^{\alpha(n)} \log n)$ time complexity and $O(n^4)$ space complexity, where n is the number of mesh vertices and $\alpha(n)$ the inverse Ackermann function. Then the geodesic distance between any pair of points on the mesh edges can be computed in $O(m + \log n)$ time, where m is the number of edges crossed by the geodesic path. Although the algorithm by Cook IV and Wenk is able to compute the *exact* pairwise geodesic, its high computational cost limits its applications to real-world models, which usually contain thousands of vertices.

In this thesis, we propose an efficient algorithm to approximate the geodesic distance between arbitrary two points on the surface. The pre-processing step takes $O(mn^2 \log n)$ time for an input mesh with n vertices and m samples, where $m (\ll n)$ is specified by the user, usually between a few hundred and several thousand. Both theoretical analysis and experimental results on real-world models demonstrate that our algorithm is efficient and accurate.

Parallel Geodesic Distance. The existing exact geodesic algorithms (e.g., the MMP, CH and ICH algorithms) do not support parallel computing due to the lack of a parallel data structure. These algorithms partition each mesh edge into a set of intervals, called windows, which are maintained in a queue and then propagated across the mesh faces. The propagation step pops a window from the queue and then computes its children windows and performs clipping or merging if necessary, which can add, modify, or remove existing windows, and the queue is then updated accordingly. Although these algorithms generate the correct solution regardless of the order in which windows are removed from the queue, selecting windows in arbitrary order leads to an extremely slow performance.

The MMP algorithm and the ICH algorithm propagate the windows as a wavefront by ordering them in a priority queue according to the distance from the source. However, maintaining a priority queue is a sequential and global process. It is highly non-trivial to implement a parallel priority queue on GPU. Furthermore, it is worth noting that the performance of the MMP and ICH algorithm drops significantly if the priority queue is replaced by a first-in-first-out queue.

The CH algorithm organizes the windows in a tree structure and propagates windows in a breadth-first-search sweep from the root to the leaves. Due to the data dependency between the parents and children, only windows on the same level can be processed in parallel. To maintain this global tree structure, a CPU/GPU synchronization has to be carried out for each layer. As the depth of the tree equals the number of mesh faces, the $O(n)$ times synchronization is too expensive for large models, hence making the parallel implementation impractical.

In this thesis, we extend the classic CH algorithm for parallel computing of the exact geodesic distance on triangle meshes. Our algorithm, called the parallel Chen-Han or PCH algorithm, follows the windows propagation structure as all the exact algorithms do. Our idea is to divide CH's sequential algorithm into a number of phases such that there is no data dependence in each phase and the operations within each phase can be done in parallel.

Parallel Algorithms for Sampling on Surfaces. The existing algorithms for Poisson disk sampling on surfaces can be distinguished into two categories: direct and indirect approaches. The indirect approaches (e.g., [51]) parameterize the input 3D surface to a regular 2D domain, followed by applying the existing 2D Poisson disk sampling algorithms, and finally mapping

the generated 2D samples back to the surface. Although the parameterization-based approaches allow one to directly borrow the well-studied 2D sampling techniques to 3D surface sampling, it is technically challenging to parameterize surfaces of complicated geometry and arbitrary topology. Furthermore, the shape distortion induced by the parameterization may cause numerical problems in 2D sampling. The direct approaches, as the name suggests, distribute samples on the surfaces without parameterization. Thus, they can be applied to a wide range of models. Cline et al. [17] presented optimal dart throwing to generate maximal Poisson disk distribution on surfaces. However, their approach is not suitable for real-time applications due to its sequential nature and high computational cost. Bowers et al. [9] extended the parallel Poisson disk sampling [86] to arbitrary surfaces. Their method is highly efficient, allowing sampling on large-scale models at interactive speed. However, the generated distribution is not fully random as the sequence of processing the phase groups follows a predefined order. Moreover, the approximate geodesic computation in their approach results in large errors in models with rich features and thus compromises the sampling quality. In a nutshell, while each existing approach has its own merits and limitations, none of them have all the features of being accurate, direct, and parallel.

In this thesis, we have proposed an accurate and parallel algorithm for direct Poisson disk sampling on arbitrary surfaces.

1.3 Thesis Organization

The remainder of this report is organized as follow: Chapter 2 reviews the related works of discrete geodesic and Poisson disk sampling. Then we introduce three algorithms for geodesic: efficient geodesic offset algorithm in Chapter 3, efficient approximate geodesic algorithm in Chapter 4 and a parallel geodesic algorithm on GPU in Chapter 5. Chapter 6 presents a parallel sampling algorithm on arbitrary surfaces. The last chapter concludes our geodesic-based Poisson disk sampling and discusses some future works.

Chapter 2

Related Work

This chapter surveys some related work in geodesic distance.

Computing discrete geodesics on polygonal meshes is a classical problem in computational geometry and computer graphics. Loosely speaking, it has three variants, i.e., “single source, any destination”, “geodesic path” and “all-pairs geodesic”, which are also closely related to one another. The readers are suggested to refer [59] for a comprehensive survey.

2.1 Geodesic distance on manifolds

Computer graphics applications often face problems that require the extraction of meaningful curves on surfaces. A detail definition of geodesic curve is given in [69]. In this section, we consider a smooth surface $\mathcal{S}$ embedded into the 3D Euclidean space, $\mathcal{S} \subset \mathbb{R}^3$.

A curve $\tilde{\gamma} : [0, 1] \rightarrow \mathcal{S}$ traced on the surface has a length computed as

$$L(\tilde{\gamma}) = \int_0^1 \|\tilde{\gamma}'(t)\| dt,$$

where $\tilde{\gamma}'(t) \in \mathcal{T}_{\tilde{\gamma}(t)} \subset \mathbb{R}^3$ is the derivative vector, that lies in the embedding space $\mathbb{R}^3$, and is in fact a vector belonging to the 2D tangent plane $\mathcal{T}_{\tilde{\gamma}(t)}$ to the surface at point $\tilde{\gamma}(t)$.

The geodesic curve is defined as the shortest curve joining two points $\tilde{x}_s, \tilde{x}_e \in \mathcal{S}$

$$\tilde{\gamma}^* = \arg \min_{\tilde{\gamma} \in \mathcal{P}(\tilde{x}_s, \tilde{x}_e)} L(\tilde{\gamma}),$$

The geodesic distance between $\tilde{x}_s$ and $\tilde{x}_e$ is defined as the length of $\tilde{\gamma}^*$.

Figure 2.1: Geodesic curve on Manifolds.

2.2 Point-source geodesic distance field

In 1986, Sharir and Schorr [79] pioneered an algorithm to compute the “single-source-all-destination” discrete geodesic on convex polyhedral with complexity $O(n^3 \log n)$. And in 1987, Mitchell *et al.* [58] (MMP) improved the time complexity bound to $O(n^2 \log n)$ by using the “continuous Dijkstra” technique. Later, Chen and Han [10] (CH) suggested building a binary tree to encode all the edge sequences that can possibly contain a shortest path, thereby improving the time complexity to $O(n^2)$. Both MMP and CH algorithm have $O(n^2)$ memory complexity, which may not work well on memory limited devices, such as GPUs.

Interestingly, in 2005, Surazhsky *et al.* [81] provided experimental evidence demonstrating that the MMP algorithm runs many times faster, in practice, than the CH algorithm. Surazhsky also extended the MMP algorithm to compute approximate geodesics with bounded error.

In 2009, Xin and Wang [93] improved the CH algorithm by exploiting a filtering theorem. Their improved algorithm outperforms both the MMP and CH algorithms. Besides the exact algorithms, there are also many algorithms [37, 84, 3, 47, 6, 43, 35, 2, 80, 4] to approximate the discrete geodesic distance field. Among them, the Fast Marching Method [43], with an $O(n \log n)$ time complexity, is a numerical method for solving boundary value problems of the Eikonal equation and has been widely used in the research community.

Figure 2.2: The improved CH algorithm [93] can compute the exact geodesic distance field with linear memory cost.

So far, the improved CH algorithm (Figure 2.2) is the most efficient algorithm to compute geodesic distance field on surfaces. This algorithm maintains the geodesic windows from near to far in a priority queue. It can be easily adapted to compute a geodesic disk with a user-specified radius. Such geodesic disk is very useful in Poisson disk sampling on surfaces.

Improved CH algorithm also significantly reduced the memory cost of computing a geodesic field. It is linear to the number of mesh vertices. A lot of memory limited devices, such as GPUs, can benefit from such memory complexity. We will later revise this improved CH algorithm for our parallel Poisson disk sampling.

2.3 Geodesic path

Geodesic path is the shortest curve between two points on surface. It is widely used in many computer graphics applications.

The ϵ -approximation algorithms [67, 16, 15] are well studied to compute an approximate path at most $(1 + \epsilon)d(s, t)$ in length, where $d(s, t)$ denotes the exact geodesic distance between two points s and t on the input polyhedral surface. These algorithms make a trade-off between computation cost and accuracy. Polthier and Schmieß [73] introduced discrete geodesic curvature on polyhedral surfaces and defined discrete straightest geodesics that allow a unique solution of the initial value problem for geodesics. Later, they developed a method to compute the evolution of distance circles on polyhedral surfaces using geodesic flow [74]. Xin and Wang [92]

Figure 2.3: Geodesic path: shortest path on surface.

Figure 2.4: Distance offsets.

presented an algorithm to compute a locally exact geodesic between two distinct vertices on a polyhedral surface. Recently, Xin *et al.* [91] proposed an efficient and exact method for computing closed geodesics within finite number of iterations.

2.4 Geodesic offsets

Geodesic offset curves are important for many industrial applications, such as solid modeling, robot-path planning, the generation of tool paths for NC machine, etc.

Distance offset is well studied in classical differential geometry and computer-aided design, however, few algorithms are available for computing geodesic offsets on discrete representation.

There is a rich body of literatures of computing geodesic offsets on parametric surfaces. We

Figure 2.5: Holla *et al.* [38] first computed geodesic offsets on surfaces. Their method only worked well on convex surfaces.

refer the readers to the complete survey [71, 30, 54]. However, very few algorithms are available for computing discrete geodesic offsets on polygonal meshes. Holla *et al.* [38] computed offsets at distances $d_1, d_2, \dots, d_k$ from the source curve, where $d_i < d_{i+1}, i = 1, 2, \dots, k$. For any input d , the geodesic offset can be reported by interpolating the offsets at d_j and d_{j+1} , where $d_j \leq d \leq d_{j+1}$. However, their algorithm heavily depends on the assumption that the offset to a polyline is still a polyline. In fact, the assumption only holds on convex polyhedral surfaces (Figure 2.5).

Chen *et al.* [13] took a set of sample points on the source curve as source points and then computed the geodesic offset at a given distance. Both [38] and [13] are approximate algorithms that require interactive detection and correction of global self-intersections. Lensch *et al.* [49] extended the MMP algorithm [58] onto the polyline-source geodesic offset problem. However, due to the large memory cost $O(n^2)$, their method is not practical for large-scale models.

2.5 All-pairs geodesic

Agarwal *et al.* [1] proved that the $\Theta(n^4)$ shortest path edge sequences can be computed in $O(n^6 2^{\alpha(n)} \log n)$ -time and $O(n^5)$ -space complexity, where $\alpha(n)$ is the inverse Ackermann function. Recently, Cook IV and Wenk [20] improved this bound to $O(n^5 2^{\alpha(n)} \log n)$ -time and $O(n^4)$ -space. These algorithms are computationally expensive, thus, can hardly be applied to real-world applications.

Figure 2.6: Typical Poisson disk distribution in $\mathbb{R}^2$ and its power spectrum, radial mean and anisotropy.

Besides all pairs geodesic, there are also some related work on computing all pairs shortest paths on weighted or unweighted graphs, such as [25], [97], [7], [98]. Using these graph-based approaches, the resultant shortest paths always follow the mesh edges, thereby resulting in a “zigzag” structure on the mesh. As a result, shortest paths highly depend on the mesh resolution and tessellation. Therefore, shortest paths are usually not robust and can lead to inaccurate results on polygonal meshes, especially with poor triangulation.

2.6 Poisson disk sampling

Poisson disk sampling in $\mathbb{R}^n$. Poisson disk sampling distribution was first observed by Cook [19] that this distribution usually yields more desirable quality than alternative distributions with the same number of samples. Such a Poisson disk distribution requires that the samples are randomly located but remain at least a minimum distance d apart from one another (figure 2.6). The Fourier spectrum of a set of Poisson disk samples exhibits blue noise property with low anisotropy and small amount of low frequency energy. Basically, visually pleasing results could be obtained by a blue noise sampling. Low frequency aliasing are replaced by high frequency noise via such distance constraint.

Poisson disk sampling in $\mathbb{R}^n$ is used extensively in computer graphics for various applications such as avoiding aliasing artifacts in ray tracing, halftoning, stippling, shown in Figure 2.7.

Dart throwing [19] is a classical algorithm that generates isotropic blue noise samples sequentially one by one. Dunbar and Humphreys [27] presented an effective spatial data structure in

Figure 2.7: Poisson disk sampling is the key technology in halftoning and image stippling.

$\mathbb{R}^2$ to encode the regions of space where the insertion of samples is allowed. However, it is unknown whether their data structure can be extended to three or higher dimensions. Gamito and Maddock [34] presented an efficient and accurate Poisson disk sampling algorithm that generates distributions with the same statistical properties of those generated with the brute-force dart-throwing algorithm. This method has $O(N \log N)$ space and time complexity, where N is the number of samples, and the resultant distribution is maximal in the sense that no further samples can be inserted at the completion of the algorithm.

As the above accurate algorithms require sequential computation, many approximate algorithms are proposed to improve the speed by compromising the sampling quality. The hierarchical approaches [56, 57, 89, 64] allow several Poisson-disk distributions with different radii to be generated from a single run of the algorithm. The tile-based approaches [18, 45, 44, 63, 50] generate infinite non-periodic tilings on the plane or surfaces at runtime by carefully designed rules. Dunbar and Humphreys [27] presented a fast approximation algorithm with linear time complexity by sacrificing the uniform distribution property. Jones [41] proposed a sampling algorithm based on Voronoi diagram. From quite a different perspective, Balzer et al. [5], Fattal [31], Chen et al. [12], Xu et al. [95] and de Goes et al. [22] formulated the Poisson disk sampling on surface into an optimization problem. Recently, Wei [86] pioneered a parallel Poisson disk sampling algorithm by subdividing the sample domain into grid cells and drawing samples concurrently from multiple cells that are sufficiently far apart to avoid

Figure 2.8: Poisson disk sampling on surfaces

conflicts. Wei [87] further extended parallel blue noise sampling to multiple classes where each individual class as well as their unions exhibit blue noise characteristics.

The Fourier spectrum is a common method to evaluate the quality of blue noise sampling [9]. Recently, some analyzing methods are proposed to evaluate the quality of non-uniform sampling, such as [66] and [88].

Poisson disk sampling on surfaces. Li et al. [51] generalized the traditional isotropic dart throwing and relaxation for the anisotropic setting, and introduced warping and sphere sampling based approaches to extend Fourier spectrum analysis for adaptive and/or anisotropic samples. By using parameterization, their technique can generate high quality anisotropic sampling on surfaces.

Fu and Zhou [33] extended Dunbar and Humphreys [27]’s fast approximate algorithm for direct sampling on surfaces. Similar to its planar counterpart, this extended approach cannot guarantee the uniform sampling property since samples are not free to be placed anywhere on the surface with equal probability.

By using the fast marching method [78] with spherical wavefront to approximate the geodesic distance, Cline et al. [17] proposed the optimized dart throwing algorithm to generate maximal Poisson disk point sets on 3D surface and made some applications based on sampling (Figure 2.9). Their approach works directly on surfaces of various types, e.g., mesh, subdivision, parametric and implicit surfaces. However, their method requires sequential computation and thus is not suitable for real-time applications involving large-scale objects.

Recently, Bowers et al.[9] presented a parallel algorithm for direct sampling on arbitrary surfaces. They used grid cell structure in $\mathbb{R}^3$ [86] to efficiently manage the samples (Figure 2.10)

Figure 2.9: Direct Poisson disk sampling on surfaces could guide hair placement.

Figure 2.10: The current parallel Poisson disk sampling on surfaces[9] accept each sample from the discrete random points assigned to each cell (a). The 3D object is divided to cells, and the cells are visited in a pre-defined order (b). This cell-based sampling is not accurate, proved by [34].

and approximated the geodesic distance by finding the length of a curve smoothly interpolating the normals of the two end points. Bowers et al.’s approach is fully parallel and highly efficient, however, it heavily depends on the embedding space and their geodesic approximation algorithm does not work well for models with high detailed features.

Our algorithm, which belongs to the category of *accurate* algorithms, can generate Poisson disk patterns on arbitrary surfaces as well as in $\mathbb{R}^n$. Furthermore, our algorithm is parallel, while all the existing accurate algorithms are sequential.

Other surface sampling techniques. There is a rich body of literatures in surface sampling and its applications. Some representative work includes point repulsion [82], centroid Voronoi tessellation (CVT) [26], stratified sampling [61], spectral sampling [65], and polyominoes [63], just name a few. These techniques however do not guarantee Poisson disk distributions.

Chapter 3

Geodesic Offset Algorithm

Geodesic offset curves are important for many industrial applications, such as solid modeling, robot-path planning, the generation of tool paths for numerical control machine (NC machine), etc. Although it is well studied in classical differential geometry and computer-aided design, where the underlying surface is sufficiently smooth, very few algorithms are available for computing geodesic offsets on discrete representations, in which the input is typically a polyline curve restricted on the piecewise linear mesh. In this chapter, we propose an efficient and exact algorithm to compute the geodesic offsets on triangle meshes by extending the Xin-Wang algorithm of discrete geodesics. We define a new data structure for parallel-source windows, and extend both the “one angle one split” and the filtering theorem to maintain the windows tree. Similar to the original Xin-Wang algorithm, our extended algorithm has an $O(n)$ space complexity and an $O(n^2 \log n)$ asymptotic time complexity, where n is the number of vertices on the input mesh. We tested our algorithm on numerous real-world models and showed that our algorithm is efficient and insensitive to the mesh resolution/tessellation.

3.1 Introduction

In this chapter, we propose an *efficient* and *exact* algorithm to compute the geodesic offsets on triangle meshes. Our algorithm extends Xin-Wang algorithm of “single-source-all-destination” discrete geodesic [93] to the general geodesic offset problem where the source is a set of points

and polylines restricted on the input triangle mesh. Our algorithm has an $O(n)$ space complexity and an $O(n^2 \log n)$ asymptotic time complexity, where n is the number of vertices on the input mesh. Numerous experimental results show that our offset algorithm runs very fast in practice. As shown in Figure. 6.1, it takes 3.2 seconds and 3.06 MB memory to compute the geodesic offset on the Happy Buddha with $100K$ vertices.

Figure 3.1: User specifies 4 source points and 3 generator curves (drawn in yellow) on the Happy Buddha with $100K$ vertices. Our algorithm computes the exact geodesic offsets in 3.2 seconds with 3.06MB memory usage. The timing is measured on a DELL Workstation with an Xeon 2.66GHz CPU and 4GB RAM.

The rest of the chapter is structured as follows. Section 3.2 briefly describes Xin-Wang algorithm of “single-source-all-destination” discrete geodesic. Section 3.3 documents the extension of Xin-Wang algorithm for computing geodesic offsets on triangle meshes, followed by the experimental results and discussions in Section 3.4.

3.2 Xin-Wang Algorithm of Discrete Geodesics

This section briefly describes the key concepts and techniques in Xin-Wang algorithm of “single-source-all-destination” discrete geodesics [93], which will be extended to solve the discrete geodesic offsets in the next section. We begin with several definitions for *edge sequences* and *windows* followed by the outline of the Xin-Wang algorithm.

3.2.1 Edge Sequences and Windows

Face sequences, edge sequences and windows are fundamental concepts in discrete geodesic algorithms [58, 93, 79, 10, 81].

Figure 3.2: All the shortest paths from s to any point $p \in [a, b]$ pass through the same edge sequence $(e_1, e_2, \dots, e_k)$. Therefore we can use a funnel-like interval window to encode all such shortest paths.

A *face sequence* $\mathcal{F}$ is an ordered list of faces, say $f_1, f_2, \dots, f_{k+1}$, such that each pair of consecutive faces, i.e., f_i and f_{i+1} , share a common edge, say e_i . The associated list of shared edges, say $\mathcal{E} = (e_1, e_2, \dots, e_k)$, is called an *edge sequence*. We say that $\mathcal{E}$ is *simple* if it contains no repeated edges. Figure 3.2 gives an example of edge sequence.

Planar unfolding is a widely used technique for computing shortest paths on polygonal meshes. A face sequence $\mathcal{F}$ can be unfolded in this way: rotate f_2 around e_1 to make f_1 and f_2 coplanar; rotate f_3 around e_2 to make f_2 and f_3 coplanar; and repeat this until all faces are coplanar. Obviously, this process takes $O(k)$ time.

Mitchell *et al.* [58] observed that any shortest path passes through an alternating list of vertices and edge sequences. Surazhsky *et al.* [81] defined a *window* to be a continuous subset of an

edge, say, $[a, b]$, such that the shortest path from the source point s to any point $p \in [a, b]$ passes through the same vertex-edge sequence, as shown in Figure 3.2. The last vertex v in this vertex-edge sequence is called *root* or *pseudo-source*, which is crucial to define a window since $d(s, p)$ equals the sum of $d(s, v)$ and the straight-line distance between the unfolded image point of v and p , where $d(\cdot, \cdot)$ returns the geodesic distance. We can further classify *window* into *pseudo-source window* that ends at a vertex and *interval window* that ends at a non-empty interval of an edge, which can be implemented as follows:

```

Struct PseudoSourceWindow
{
    int      vertexID;
    int      root;
    double   distanceFromSourceToRoot;
    Point2D  unfoldedImageOfRoot;
};

Struct IntervalWindow
{
    int      edgeID;
    Interval interval;
    int      root;
    double   distanceFromSourceToRoot;
    Point2D  unfoldedImageOfRoot;
};

```

3.2.2 Window Derivation

Unlike the graph-based shortest path problem, the discrete geodesic problem on a polyhedral surface is continuous in essence. The key to discretize the geodesic problem is to encode the shortest paths that share the common edge sequence into a window and then take care of the

Figure 3.3: Computing the children of a window: (a) A pseudo-source window at a saddle vertex v may have children both on opposite edges and at adjacent vertices; (b-d) An interval window w on edge $\overline{v_1v_2}$ may have one or two interval-window children, depending on how the line segment $\overline{Iv}$ intersects w 's interval.

derivation of windows. Most of the existing exact geodesic algorithms [58, 93, 10] compute the children of an existing window w in a similar fashion (see Figure 3.3):

If w is a pseudo-source window at vertex v with geodesic distance d

If v is a saddle vertex (see Figure 3.3(a))

For each adjacent vertex, compute a pseudo-source window;

For each opposite edge, compute an interval window;

Else /* v is a convex vertex*/

No children need to be computed;

Else /* w is an interval window. Suppose w covers an interval $[a, b]$ of the edge $\overline{v_1v_2}$. Let v be the vertex opposite to $\overline{v_1v_2}$ and I be the unfolded image of the last vertex passed through by w .*/

If the line segment $\overline{Iv}$ is right to the interval $[a, b]$ (see Figure 3.3(b))

Compute the only interval-window child on edge $\overline{v_1v}$;

Elseif the line segment $\overline{Iv}$ is left to the interval $[a, b]$ (see Figure 3.3(c))

Compute the only interval-window child on edge vv_2 ;

Else /* $\overline{Iv}$ intersects the interval $[a, b]$ at some point (see Figure 3.3(d)).*/

Compute two interval windows as the children, one on edge v_1v and the other on edge vv_2 ;

Compute a pseudo-source window at vertex v .

3.2.3 Two Key Observations

Figure 3.4: Two key observations in Xin-Wang algorithm [93]: (a) Chen and Han’s “one angle one split” [10] states that at most one of the windows covering the same angle can have two children; and (b) Xin and Wang’s filtering theorem [93] asserts that an interval window is no use if it cannot provide a shorter geodesic distance than one of the neighboring vertices.

The existing geodesic algorithms [58, 10, 93] use different techniques to avoid the combinatorial explosion of the number of windows. The MMP algorithm [58] transforms each edge into a covering of a set of ordered windows. Chen and Han [10] observed an important fact called “one angle one split”, with which they built an $O(n^2)$ tree to encode the windows that possibly determines a shortest path. Xin and Wang [93] observed that the majority of windows created by the CH algorithm are useless and presented a filtering theorem that significantly reduces the number of windows.

One angle one split. As shown in Figure 3.4(a), $\triangle v_1 v v_2$ is a mesh triangle. w_1 and w_2 are two interval windows that cover the same angle $\angle v_1 v v_2$, with pseudo sources I_1 and I_2 , respectively. According to the window derivation algorithm in Section 3.2.2, both w_1 and w_2 may have two children, which lead to four new windows $w_1^1, w_1^2, w_2^1, w_2^2$. However, Chen and Han’s “one angle one split” theorem [10] states that at least one of the four children does not help when determining shortest paths. That is to say, among all the windows covering the same angle, at most one of them can have two children that possibly determine a shortest path. Thus, one associates each angle with a “winning” window, by which one can check the validity of new windows. The CH algorithm guarantees that the window tree is of $O(n^2)$ in size and $O(n)$ in depth.

Filtering theorem. Xin and Wang [93] observed that the majority of windows created by the CH algorithm [10] are useless. Therefore they proposed a filtering theorem to remove those useless windows. As shown in Figure 3.4(b), w is an interval window on the edge v_1v_2 and the unfolded image of w 's root is located on the plane of $\triangle v_1v_2v_3$. The filtering theorem asserts that w is useless if one of the following conditions satisfies:

- The shortest path from s to v_1 and then to B is shorter than what w can give;
- The shortest path from s to v_2 and then to A is shorter than what w can give;
- The shortest path from s to v_3 and then to A is shorter than what w can give.

In addition, Xin and Wang [93] suggested maintaining a priority queue such that the “nearest” window can be handled first. Numerous experimental results show that Xin-Wang algorithm [93] outperforms the MMP algorithm [58] and the CH algorithm [10] in both time and space. This motivates us to extend Xin-Wang algorithm to the general geodesic offset problem.

3.3 Extending Xin-Wang Algorithm To The General Geodesic Offset Problem

This section details our algorithm to compute the general geodesic offset on triangle meshes. The key components of our algorithm are to define a new type of window called *parallel-source window* and then extend Chen and Han’s “one angle one split” and Xin and Wang’s filtering theorem (see Figure 3.4).

3.3.1 Parallel-Source Window

Consider the source set $\mathcal{U}$ including source points and polyline curves. After computing the geodesic distance field, we can backtrack the discrete geodesic, for any destination point p , which is actually the shortest path from p to $\mathcal{U}$. We use $\mathcal{G}$ to represent the family of backtraced discrete geodesics and $\mathcal{O}$ the family of offset curves. In the continuous setting, the offset curves are iso-distance curves and the backtraced geodesics follow the gradient direction of

Figure 3.5: Three types of windows, i.e., pseudo-source windows, interval windows and parallel-source windows are required to compute the general offset problem. (a) Taking the line segment AB as the source, we need to construct three parallel-source windows initially. (b) We also need to construct 6 interval windows for the endpoints A and B .

the geodesic distance field, thus, each curve in $\mathcal{G}$ is orthogonal to the curves in $\mathcal{O}$, i.e., $\forall \mathcal{C}_1 \in \mathcal{G}, \mathcal{C}_2 \in \mathcal{O}$, if $\mathcal{C}_1$ and $\mathcal{C}_2$ intersect, then $\mathcal{C}_1 \perp \mathcal{C}_2$.

On triangle meshes, the above condition implies that the four angles $\mathcal{C}_1$ intersecting $\mathcal{C}_2$ are greater than or equal to $\pi/2$. Specially, for the case of Figure 3.5 (a) where the source is the line segment AB , we only need to consider the geodesics orthogonal to AB . Therefore, we replace the vertex-type root with a straight-line segment and define the *parallel-source* window as follows:

```

Struct ParallelSourceWindow
{
    int      edgeID;
    Interval interval;
    Interval rootSegment;
    Point2D unfoldedImageOfLeftEndpoint;
    Point2D unfoldedImageOfRightEndpoint;
};
    
```

Taking Figure 3.5 as an example, we need to push 3 parallel-source windows (see (a)) and 6 interval windows rooted at A and B (see (b)) into a priority queue when initializing the Xin-Wang algorithm [93].

3.3.2 Extending Chen and Han’s “One-Angle-One-Split”

Figure 3.6: Extended “one angle one split”: If the parallel-source window w can provide a shorter distance to v_3 than the existing winning window, then we allow w to have two children. If w can have only one child, then we remove either w_1 (see (a)) or w_2 (see (b)).

Chen and Han’s “one angle one split” [10] states that at most one of the windows covering the same angle can have two children (see Figure 3.4 (a)). For the implementation point of view, we need to keep a “winning” window that can provide the up-to-date shortest geodesic distance for the opposite vertex. In this subsection, we discuss how to extend “one angle one split” onto the general offset problem.

As shown in Figure 3.6, assume v is the *last* vertex that passed through by the existing shortest geodesic path between s and v_3 . The unfolded image of v is R . Since any window encodes a set of shortest paths that go through the same edge sequence, there are no vertices enclosed in the yellow area in (a). Therefore, R must be outside of the the yellow area. Then we consider the following cases in the extended “one angle one split”:

Extended “one angle one split”:

Case 1. If $p \in AB$ can give a shorter distance than the existing winning window and R is left to AB , then the parallel-source window w is allowed to have two children; and the winning window of v_3 is set to be w .

Case 2. If $p \in AB$ cannot give a shorter distance,

Case 2.1. If R is left to AB , then we remove the left child w_1 ; see Figure 3.6 (a).

Case 2.1. If R is right to AB , then we remove the right child w_2 ; see Figure 3.6 (b).

Note that if Rv_3 intersects AB at point q , then q can give a shortest distance than the existing winning window, which belongs to Case 1.

3.3.3 Extending Xin and Wang’s Filtering Theorem

Figure 3.7: The parallel-source window cannot contribute to determining a shortest path if v_1 gives a shorter distance than $\|BD\|$, or v_2 gives a shorter distance than $\|AC\|$, or v_3 gives a shorter distance than $\|AC\|$.

As shown in Figure 3.7, the side boundaries AC and BD of a parallel-source window are still orthogonal to the source segment AB even if upon unfolded. Therefore, A (resp. B) is nearer to C (resp. D) than any other point in AB . This motivates us to extend Xin and Wang’s filtering theorem [93]:

Extended filtering theorem. *Suppose the parallel-source window w arrives at the face $\triangle v_1 v_2 v_3$, entering from $v_1 v_2$ and exiting from $v_1 v_3$ (see Figure 3.7). Let AB be the unfolded image of the source segment and CD be the interval in $v_1 v_3$ covered by w . Then the parallel-source window w can be filtered out (i.e., it does not help to determine a shortest path) if*

$$d(s, v_1) + \|v_1 D\| \leq \|BD\|, \quad (\text{Eq. 3.1})$$

or

$$d(s, v_3) + \|v_3 C\| \leq \|AC\|, \quad (\text{Eq. 3.2})$$

or

$$d(s, v_2) + \|v_2 C\| \leq \|AC\|, \quad (\text{Eq. 3.3})$$

where $d(\cdot, \cdot)$ is the geodesic distance notation.

Proof. We consider three cases:

Case 1. If v_1 provides a shorter distance for the point D , i.e.,

$$d(s, v_1) + \|v_1 D\| \leq \|BD\|,$$

then for any point $p \in CD$, the geodesic distance given by v_1 is

$$d(s, v_1) + \|v_1 p\|,$$

while the geodesic distance given by the parallel window w is $\|pp'\|$, where p' is the projection point on the segment AB ; see Figure 3.8. Considering

$$\|BD\| - \|pp'\| < \|pD\|,$$

we have

$$\begin{aligned} d(s, v_1) + \|v_1 p\| &= d(s, v_1) + \|v_1 D\| - \|pD\| \\ &< (\|BD\|) - (\|BD\| - \|pp'\|) \\ &= \|pp'\|. \end{aligned} \quad (\text{Eq. 3.4})$$

Observing that Eq. 3.4 holds for any $p \in CD$, we can show that the parallel-source window w can be filtered out.

Case 2. If v_3 provides a shorter distance for C , i.e.,

$$d(s, v_3) + \|v_3 C\| \leq \|AC\|,$$

we can prove that w is filtered out in a similar fashion as case 1.

Case 3. If v_2 provides a shorter distance for C than that of the parallel-source window w , we can easily show that v_2 and C are on the same side of the straight line AB , as shown in Figure 3.8. We construct a parabola $\mathcal{P} \in \mathbb{R}^2$ such that

$$d(s, v_2) + \|x - v_2\| = \|xx'\|,$$

where x is a moving point on the regarding plane and x' is the projection point of x on the straight line AB . Obviously, C is located in the interior of $\mathcal{P}$ due to the assumption

$$d(s, v_2) + \|v_2 C\| \leq \|AC\|.$$

Figure 3.8: Case 3.

Then we show that D must be in the interior of $\mathcal{P}$ by contradiction. Assume that D is outside of $\mathcal{P}$. D must be on the righthand side of the axis of $\mathcal{P}$ because

- (i) D is right to C along the direction AB , i.e., $\overrightarrow{AB} \cdot \overrightarrow{CD} > 0$;
- (ii) D is above the straight line $v_1 v_2$ (BD intersects $v_1 v_2$ at one point), where v_2 is the focus of $\mathcal{P}$.

But on the other hand, if D is on the righthand side of the axis of $\mathcal{P}$, then v_2 must be inside the quadrangle $ABDC$. This contradicts to the observation that the quadrangle $ABDC$ completely lies in an edge sequence and cannot enclose any vertex inside.

With the fact that D is in the interior of $\mathcal{P}$, the entire segment CD is in the interior of $\mathcal{P}$ due to the convexity. This implies

$$d(s, v_2) + \|v_2 p\| \leq \|pp'\|,$$

for any $p \in CD$, where p' is the projection point of p onto AB . Therefore, the parallel-source window w can be removed.

Putting the above three cases together, we complete the proof.

3.3.4 Complexity Analysis

Our extended algorithm applies to the general geodesic offset problem, where the source is a set of points and polylines. We assume that the source consists of a subset of vertices and each face contains at most one line segment of the generator curves (polylines), whose endpoints serve as the entry point and the exit point respectively. It is clear that we have $O(m)$ parallel-source windows initially, where m is the number of faces ($m \approx 2n$ for triangle meshes). The property “one angle one split” guarantees that the number of windows on each level can be bounded by $O(m)$ when building the window tree [10]. Since the window tree cannot exceed m levels (a shortest path intersects every face at most once), the total number of windows is $O(m^2)$. Note that we only need to maintain the bottom level (i.e., leaves) of the windows tree and all branch nodes, therefore, the extended algorithm still has an $O(n)$ space complexity and an $O(n^2 \log n)$ asymptotic time complexity as in the original Xin-Wang algorithm [93].

3.4 Experimental Results

This section documents the experimental results and demonstrates that our algorithm is efficient, exact, intrinsic and robust. We implemented the extended Xin-Wang algorithm in C++ on Windows 7. Our experiments were carried out on a DELL Workstation with an Xeon 2.66GHz CPU and 4GB RAM.

Figure 3.9: Experimental results. The generator curves are drawn in yellow. Cold (resp. warm) colors mean small (resp. large) geodesic distances.

Performance. We tested our algorithm on numerous commonly-used models in graphics and geometric modeling community. Due to the $O(n)$ space complexity and $O(n^2 \log n)$ time complexity, our algorithm works efficiently and effectively for large scale models (see Table 1 for the statistics). Taking the Dragon model (with $750K$ vertices) for an example, our algorithm computes the geodesic offsets for the “SPM11” logo within only 77 seconds and costs 23MB memory. Figure 4.12 shows the geodesic parallels on the Lucy model with various spacings. Figure 3.10 demonstrates the geodesic offset on the 3D scanned human face model, which usually contains artifacts (such as disconnected components, holes, etc) near the face boundary. Computing the geodesic offset based on the generator curves of some facial features (e.g., eyes, nose and mouth) leads to a good segmentation.

Robustness. Our geodesic computation is intrinsic to the geometry and insensitive to the mesh resolution and tessellation. As shown in Figure 3.12, we experiment our method with the same 3D model of different resolutions and triangulations. The same generator curves are specified by the same cutting plane. Our algorithm is stable and robust, and produces consistent resultant geodesic offsets.

Exactness. The proposed geodesic offset algorithm is exact in the sense that the geodesic distance from each vertex to the generator curve(s) is accurate. As a result, the integral curves

Table 3.1: Timing statistics (seconds) measured on a workstation with an Xeon 2.66GHz CPU and 4GB RAM.

Models	#vertices	Timing (second)	Memory (Mb)
Armadillo	173K	13.385	5.29
Buddha	100K	3.224	3.06
Bunny	75K	8.537	2.23
Dragon	750K	77.173	22.90
Gargoyle	352K	31.840	10.76
Laurana	10K	0.733	0.33
Lucy	263K	26.114	8.04
Moai	11K	0.655	0.34
Moai	3K	0.06	0.08
Two Kids	42K	3.931	1.29

Figure 3.10: Taking salient features (e.g., eyes, forehead, nose and mouth) as generators, we compute the geodesic offsets, which can be used for face segmentation.

Figure 3.11: Given the generator curve on the waist, we compute the geodesic distance field of the Lucy model. Then we show the set of offset curves with spacing of 0.06 and 0.025 respectively. The model is normalized to a unit cube.

Figure 3.12: Our offset algorithm is intrinsic to the geometry and insensitive to the resolution/triangulation. Left: the Moai model with 11K vertices; Right: the Moai model with 3K vertices.

of the gradient of the geodesic distance field are exact geodesics. As shown in Figure 3.14, the (blue) geodesics are orthogonal to the (gray) offset curves everywhere.

Figure 3.13: Comparison to Dijkstra’s algorithm [23] and the fast marching method [43]. Note that our algorithm is able to compute the *exact* geodesic offsets while the others compute only the *approximate* geodesic offsets (see the artifacts in (a) and (b)).

To demonstrate the exactness of the proposed algorithm, we also compared our algorithm with Dijkstra’s algorithm [23] and the fast marching method [43], which are *approximation* algorithms for geodesic offsets. As shown in Figure 4.7, the approximation algorithms usually result in poor quality geodesic offsets on models with complicated geometry and topology. We must point out the overhead for the exactness is the computational cost. Both Dijkstra’s algorithm [23] and fast marching method [43] are of time complexity $O(n \log n)$, while ours is $O(n^2 \log n)$. This is because the geodesic offsets are G^2 continuous except at conjugate points.

Discussions. Holla *et al.* [38] and Chen *et al.* [13] used a traditional approach to compute geodesic offsets, where the offsets are computed segment by segment followed by local and

Figure 3.14: Geodesics (drawn in blue) starting from the generator curves (drawn in yellow) are orthogonal to the offset curves (drawn in gray) everywhere.

global trimmings. It is known that this kind of approach is computational expensive, and can hardly be used for meshes with more than a few thousand vertices. Furthermore, it is technically challenging to fix global intersections. Usually, manual inspection and correction of self-intersections are required.

Lensch *et al.* [49] extended the MMP algorithm [58] onto the polyline-source geodesic offset problem. Similar to our algorithm, the extended MMP algorithm can compute the *exact* geodesic offset on polygonal meshes. Although working well for small and medium sized meshes, their approach is impractical for large scale models due to its high space complexity $O(n^2)$. For example, their algorithm failed on a mesh with $200K$ vertices by running out of memory. In contrast, our algorithm successfully computed the exact geodesic offsets by using only 6 MB memory. To our knowledge, our algorithm is the first one that is able to compute *exact* geodesic offsets on large scale polygonal meshes.

Chapter 4

All Pairs Geodesic Algorithm

Computing discrete geodesics on polyhedral surfaces plays an important role in computer graphics. In contrast to the well-studied “single-source, all-destination” discrete geodesic problem, little progress has been reported to the pairwise geodesic, i.e., computing the geodesic distance between two arbitrary points on the surface. To our knowledge, the existing pairwise geodesic algorithms have very high computational cost, thus, can not be applied to real-world models, which usually contain thousands of vertices. In this chapter, we propose an efficient algorithm to approximate the pairwise geodesic on triangular meshes. The pre-processing step takes $O(mn^2 \log n)$ time for the input mesh with n vertices and m samples, where $m (\ll n)$ is specified by the user, usually between a few hundred and several thousand. Then in the query step, our algorithm can compute the *approximate* geodesic distance between arbitrary pair of points (not necessarily mesh vertices) in constant time $O(1)$. Furthermore, the geodesic path and the geodesic distance field can be approximated in linear time. Both theoretical analysis and experimental results on real-world models demonstrate that our algorithm is efficient and accurate.

In this chapter, we present an efficient algorithm for approximating all-pairs geodesic distances on triangle meshes. Our algorithm consists of two steps. In the preprocessing step, we construct a geodesic triangulation on m sample points, where m is specified by the user, usually, between a few hundred and several thousand. The preprocessing step takes $O(mn^2 \log n)$ time and outputs an $O(m^2 + n)$ -sized file that encodes 1) the exact geodesic distances between every pair of sample points and 2) the exact geodesic distances between a mesh vertex and the

Figure 4.1: Our algorithm can approximate the geodesic distance between arbitrary pair of points in **constant time**. The user can control the approximation error by specifying the number of sample points on the input model. Furthermore, both the geodesic path and geodesic distance field can be approximated in **linear time**. With 1000 sample points on the BUNNY model ($n = 72K$), the relative root-mean-square error is less than 0.5%. (a) shows an example of geodesic distance query and the corresponding exact (in green) and approximate (in red) geodesic paths. (b) and (d) show the front and back views of the exact geodesic distance field by [Xin and Wang 2009]. (c) and (e) show the approximate result by our method. Cold (resp. warm) colors represent short (resp. long) distances.

three vertices of the enclosing geodesic triangle. Then in the query step, we can approximate the geodesic distance between any two points¹ (not necessarily mesh vertices) on the surface in constant time $O(1)$. We show that the upper bound of the approximation error is related to the geodesic triangulation, which in turns can be controlled by the user-specified number of samples m . Experiments on real-world models demonstrate that our method can generate accurate results with a reasonable number of sample points and preprocessing time. For example, the relative root-mean-square error is less than 0.5%, for the BUNNY model with $m = 1000$ samples, see Figure. 4.1. Furthermore, our method has two by-products. First, the approximate geodesic path can be computed in $O(k)$ time, where k is the number of edges crossed by the path. Second, the approximate multi-source geodesic distance field can be computed in $O(ln)$ time, where l is the number of source points.

The remaining of this chapter is organized as follows: Section 4.1 presents the notations and overview of our algorithm, followed by the details of pre-processing in Section 4.2 and run-time query in Section 4.3. Section 4.4 shows the experimental results and discusses the strengths and limitations of the proposed method.

¹We assume each query point q is given in the form of the mesh triangle containing q and a barycentric coordinate to determine the location of q . Thus, point location also takes $O(1)$ time.

Our work focuses on approximating all pairs geodesics on triangle meshes in constant time. However, our method can be easily extended to compute the approximate geodesic paths and geodesic distance fields in linear time.

4.1 Overview

Notations. The input triangle mesh M is represented by $M = (V, E, F)$, where V , E , and F are the sets of vertices, edges and faces respectively. Let $n = |V|$ denote the number of vertices.

Throughout this chapter, we distinguish *vertex*, *sample point*, and *query point* as follows:

- a *vertex* v is an element of V ;
- a *sample point* s is a point on M , which is computed using farthest point sampling algorithm.
- a *query point* q is an arbitrary point on M , i.e., it can be a vertex, or on an edge or inside a triangle.

Furthermore, we use the following notations to denote the path and various distances between two query points p and q :

- $\|pq\|$ the Euclidean distance;
- $d(p, q)$ the exact geodesic distance;
- $\tilde{d}(p, q)$ the approximate geodesic distance computed by our algorithm.

Finally, let $[i] = i \bmod 3$ be the modulo 3 operator.

Algorithmic pipeline. Our algorithm contains two steps, the pre-processing step and the query step, as shown in Figure 4.2.

The pre-processing step first samples the given surface with m points, where m is specified by the user (see (a)) and then computes the geodesic distance fields with the samples as sources (see (b)), which induces a Delaunay triangulation (see (c)). Next, the Delaunay triangulation edges are replaced by geodesics (see (d)). Finally, we save two kinds of exact geodesic

Figure 4.2: Algorithm pipeline. Given the input mesh M , we first sample m points distributed uniformly on M (see (a)); Then taking the sample points as sources, we compute the geodesic distance field on M (see (b)) and construct an approximate Delaunay triangulation (see (c)), which induces a geodesic triangulation by replacing each Delaunay edge with a geodesic (see (d)). Next, the exact geodesic distances between any pair of sample points are saved in a file. Finally, with the pre-computed geodesic distance file, we can approximate the geodesic distance between any pair of query points in constant time $O(1)$ and find the corresponding geodesic path in linear time $O(k)$, where k is the number of edges crossed by the path (see (e)).

distances into a file, i.e., the distances between any pair of sample points, and the distances between each mesh vertex and the three vertices of the geodesic triangle containing it. The pre-processing step takes $O(mn^2 \log n)$ time and outputs a file of size $O(m^2 + n)$.

In the query step, given the two query points q_1 and q_2 on the surface, we first find the geodesic triangles containing them, say T_1 and T_2 (see (e)). Then, we *unfold* the geodesic triangles T_1 and T_2 to $\mathbb{R}^2$, such that the Euclidean side lengths coincide with the geodesic distances of the curved geodesic triangle. The query points q_1 and q_2 are also mapped to $\mathbb{R}^2$ using unfolding operations. Their 2D images are denoted by q'_1 and q'_2 , respectively. Finally, the geodesic distance between q_1 and q_2 is approximated by the Euclidean distance between q'_1 and q'_2 . Note that this approximation can be done in constant time $O(1)$ as unfolding is a simple and local operation.

4.2 Pre-Processing

4.2.1 Constructing Geodesic Triangulation

As shown in Section. 4.3.3, the approximation error is closely related to the maximal side length of geodesic triangles. Thus, an isotropic sampling, which induces geodesic triangulation of similar sizes, is desired. In this chapter, we adopt a variant of the farthest point sampling method [70], which iteratively updates the sampling set with the farthest point to the existing sample points. The difference is that we use the improved CH algorithm [93] rather than the fast marching method [43] for geodesic distance computation. The time complexity for sampling m points is $\sum_{i=1}^m f(\frac{n}{i}) = O(n^2 \log n)$, where $f(x) = O(x^2 \log x)$.

Taking m sample points as source points, we compute the multi-source geodesic distance field using [93], then construct an approximate Delaunay triangulation using [94]. It is shown that the resultant Delaunay triangulation is well defined in the sense that all the Delaunay edges do not intersect [94] under moderate condition. However, the edges are not “straight” in general. Finally, we need to replace each curved Delaunay edge with a geodesic path.

Note that it takes $O(mn^2 \log n)$ time to run the exact geodesic algorithm [93] with each sample point as a source point. Also, it takes $O(n)$ time to replace each curved Delaunay edge with a geodesic. Considering the number of edges in the Delaunay triangulation is $O(m)$, the time complexity for constructing geodesic triangulation is $O(mn^2 \log n)$.

4.2.2 Generating Geodesic Distance File

After constructing the geodesic triangulation, we are ready to generate a file containing the following information, which will be used our all-pairs geodesic distance query algorithm.

- The exact geodesic distances between any pair of sample points.
- For each vertex v , output the geodesic triangle $\triangle s_1 s_2 s_3$ that contains v and the exact geodesic distances $d(v, s_i)$, $i = 1, 2, 3$.

Note that all the required geodesic distances are readily available when we constructed the geodesic triangulation in the previous step.

Putting them all together, the entire pre-processing takes $O(mn^2 \log n)$ time and outputs a file of size $O(m^2 + n)$.

4.3 Pairwise Geodesic Distance Query

4.3.1 Unfolding Geodesic Triangles

We first present two simple *unfolding* operations that play key roles in our constant-time query algorithm. Observing that geodesic distances follow triangle inequality, thus, we can flatten the curved geodesic triangle to a unique triangle (up to rigid motion) in $\mathbb{R}^2$.

Given two curved geodesic triangles $\triangle psr$ and $\triangle qrs$, we can unfold them to 2D triangles $\triangle p's'r'$ and $\triangle q'r's'$, where the corresponding Euclidean edge lengths are equal to those of the geodesic edges. Let $u(p, q|rs)$ denote such unfolding operation with respect to edge rs . We call it one-side unfolding if p and q are on the same side of rs and two-side unfolding, otherwise (see Figure. 4.3(a)).

4.3.2 Algorithm

We assume that the query point q is given in the form of a triangle $t = (t_1, t_2, t_3)$ and a Barycentric coordinate (α, β, γ) such that $q \in t$ and $q = \alpha t_1 + \beta t_2 + \gamma t_3$. Thus, the point location takes constant time. This assumption is reasonable, since many sampling algorithms (e.g., [62]) do output the samples in this format.

Given two arbitrary query points $p, q \in M$, we consider the following cases to compute the approximate geodesic distance $\tilde{d}(p, q)$:

Case 1: both p and q are sample points. Their exact geodesic distance can be reported immediately by retrieving the information from the pre-computed geodesic file.

Case 2: one of p and q is a sample point, while the other is a vertex. Without loss of generality, say p is a sample point, and q is a vertex.

Figure 4.3: (a) The unfolding operation flattens curved geodesic triangles to $\mathbb{R}^2$ such that the corresponding edge lengths are equal, i.e., $d(r, s) = \|r's'\|$, $d(p, r) = \|p'r'\|$, $d(p, s) = \|p's'\|$, $d(q, r) = \|q'r'\|$, $d(q, s) = \|q's'\|$. (b) and (c) show two important cases of applying unfolding operation in our algorithm.

- **Case 2.1:** p and q are in the same geodesic triangle. Then p is the vertex of the geodesic triangle containing q . So, the exact geodesic distance $d(p, q)$ can be obtained directly from the geodesic file.
- **Case 2.2:** p and q are in different geodesic triangle. Let $\triangle s_1 s_2 s_3$ be the geodesic triangle containing q , then the exact geodesic distances $d(p, s_i)$, $i = 1, 2, 3$, are known. We apply the two-side unfolding operation with respect to each edge of geodesic triangle $\triangle s_1 s_2 s_3$, i.e., $u(p, q|s_i s_{[i]+1})$, $i = 1, 2, 3$. Let p_i and q_i denote the images of p and q , respectively. Then the approximate geodesic distance is $\tilde{d}(p, q) = \min_{1 \leq i \leq 3} \|p_i q_i\|$ (see Figure. 4.3(b)).

Case 3: both p and q are vertices, rather than sample points.

- **Case 3.1:** p and q are in the same geodesic triangle $\triangle s_1 s_2 s_3$. We apply one-side unfolding $u(p, q|s_i s_{[i]+1})$, $i = 1, 2, 3$, with respect to each geodesic edge. Let p_i and q_i denote the 2D images of p and q . Then the approximate geodesic distance is $\tilde{d}(p, q) = \min_{1 \leq i \leq 3} \|p_i q_i\|$ (see Figure. 4.3 (c)).
- **Case 3.2:** p and q are in different geodesic triangles, say $p \in \triangle s_1 s_2 s_3$ and $q \in \triangle s_4 s_5 s_6$. Note that the sample point s_4 and query point p are in different geodesic triangles, we can compute $\tilde{d}(p, s_4)$ according to Case 2.2. Similarly, we compute $\tilde{d}(p, s_5)$ and $\tilde{d}(p, s_6)$. Then with the approximate geodesic distances $\tilde{d}(p, s_i)$, $4 \leq i \leq 6$, we can apply Case 2.2 again to compute $\tilde{d}(p, q)$.

Case 4: one of p and q is a vertex, while the other is an arbitrary point. Without loss of generality, assume $q \in V$ is a vertex, but p is not. Let $\triangle v_1 v_2 v_3$ be the mesh triangle that contains p . By using Case 3, we can compute $\tilde{d}(v_i, q), i = 1, 2, 3$. Then, the approximate geodesic distance $\tilde{d}(p, q)$ is given by applying Case 3 again, i.e., two-side unfolding with respect to each edge $v_i v_{[i]+1}, i = 1, 2, 3$.

Case 5: neither of p and q is a vertex. Assume p is in a mesh triangle $\triangle v_1 v_2 v_3$. Following Case 4, we can compute $\tilde{d}(v_i, q), i = 1, 2, 3$, immediately. Note that $d(p, v_i), i = 1, 2, 3$, can be computed using Euclidean distance. Again, we can apply two-side unfolding operation in Case 3 to each edge of $\triangle v_1 v_2 v_3$, i.e., $u(p, q|v_i v_{[i]+1}), i = 1, 2, 3$, to obtain the final estimate $\tilde{d}(p, q)$.

The above all-pairs geodesic distance query algorithm contains only three kinds of operations, retrieving geodesic distance from pre-computed geodesic file, unfolding operations and computing the Euclidean distance between two 2D points, which are all constant-time operations. Thus, each query can be answered in constant time $O(1)$.

4.3.3 Error Analysis

If two query points are in the *different* geodesic triangles, the upper bound of approximation error is given by the following Theorem, where the proof is provided in the Appendix:

Theorem 1. *Given two query points q_1 and q_2 in different geodesic triangles, the approximation error $|d(q_1, q_2) - \tilde{d}(q_1, q_2)| \leq 2L + 2l$, where L (resp. l) is the maximum edge length of geodesic (resp. mesh) triangles containing q_1 and q_2 .*

4.4 Results & Discussions

4.4.1 Experimental Results

We implemented our algorithm in C++ on a 64-bit Workstation with Xeon 2.66GHz CPU and 8GB memory. Table 4.1 shows the query speed of the BUDDHA and the LUCY models.

Table 4.1: Query speed (# queries/second).

BUDDHA ($n = 50K$)			LUCY ($n = 263K$)		
$m = 1000$	$m = 2000$	$m = 3000$	$m = 1000$	$m = 2000$	$m = 3000$
27.3K	36.7K	45.4K	8.2K	15.5K	21.0K

To evaluate the accuracy of our algorithm, we compared the approximate geodesic distance with the exact geodesic distance. Specifically, we randomly sampled k query points² using [62] for each test model. Then for any pair of query points p and q , we computed the *exact* geodesic distance $d(p, q)$ using [93] and the *approximate* geodesic distance $\tilde{d}(p, q)$ using our algorithm. Next, we computed the relative error $\varepsilon(p, q) = \frac{|d(p, q) - \tilde{d}(p, q)|}{d(p, q)}$. Finally, we computed the relative root-mean-square error

$$\epsilon = \frac{1}{\binom{k}{2}} \sqrt{\sum_{p, q} \varepsilon^2(p, q)}.$$

As the upper bound of approximation error is closely related to the longest edge of the geodesic triangles that containing the query points (see Theorem 1), which in turn depends on the number of sample points m . Clearly, the more number of sample points, the more accurate geodesic distances one obtains, but the more pre-processing time one has to take. As shown in Figure 4.5(top), for smooth model such as FERTILITY (see Figure 4.11) and BUNNY (see Figure 4.1), we can obtain query results with $\epsilon \leq 0.5\%$ with just a few hundred samples. For large models with detailed features, such as ARMADILLO (see Figure 4.10) and LUCY (see Figure 4.12), we observed that a few thousand samples are usually sufficient to produce good results. As shown in Figure 4.5(bottom), the pre-processing time is roughly a linear function of m . The error distribution histogram in Figure 4.4 also demonstrates that increasing m is helpful to reduce the approximation error. Thus, this is a tradeoff between the accuracy and pre-processing time, which is usually application dependent.

4.4.2 Geodesic Path & Geodesic Distance Field

Our algorithm has two byproducts that 1) the approximate geodesic path can be computed in $O(k)$ time, where k is the number of edges crossed by the path; and 2) the approximate geodesic distance field with l source points can be computed in $O(ln)$ time.

²We experienced that $k = 10000$ leads to very stable statistical result.

Figure 4.4: Error distribution of ARMADILLO model. 40.27%, 58.45% and 68.40% relative errors are in the range $[-0.2\%, +0.2\%]$ for $m = 1000$, 2000 and 3000, respectively.

Geodesic distance field. Computing geodesic distance field with l source points is straight forward, i.e., for each source point, simply computing the geodesic distances with all mesh vertices, and then choosing the smallest distance. Thus, the time complexity is $O(ln)$. Figure 4.6 shows the approximate multi-source geodesic distance fields with various number of samples on BUDDHA model.

Geodesic path. Given two query points s and t on the surface, let $\gamma : [0, 1] \rightarrow M$ be the geodesic path between s and t , where $\gamma(0) = s$ and $\gamma(1) = t$. Let $D(\cdot)$ denote the geodesic distance field with source point s . The geodesic path satisfies the following Eikonal equation [43]:

$$\frac{d\gamma}{dt} = -\nabla D,$$

where ∇D is the gradient of D . This ODE can be solved efficiently by tracing the gradient vector field ∇D on triangle mesh.

Let $Nb(t)$ be the set of one-ring neighbors of t , if t is a vertex; otherwise, $Nb(t)$ includes the three vertices of the mesh triangle that contains t . We compute $\tilde{d}(s, t)$ and the geodesic distances between each element of $Nb(t)$ and s . Then we compute the gradient ∇D for t 's neighboring triangles and find the one with the largest magnitude, which is used as the initial tracing direction.

Starting from t and following the initial tracing direction, we can reach one mesh edge, which induces a neighboring triangle f . Then we can compute the gradient ∇D in f , which further induces the next tracing direction. Repeat the tracing procedure until we reach the source point s .

Note that computing the gradient ∇D inside a mesh triangle f takes constant time, so is the tracing in f . Thus, the time complexity for tracing the geodesic path is $O(k)$, where k is the number of edges crossed by the path.

Figure 4.5: Numerical results. Top: Relative root-mean-square error VS number of sample points; Bottom: Preprocessing time (seconds) VS number of sample points.

4.4.3 Discussions

We compared our algorithm with a simple linear interpolation method: Considering two mesh vertices v_1 and v_2 , which are located in geodesic triangles $\triangle_{s_1 s_2 s_3}$ and $\triangle_{s_4 s_5 s_6}$ respectively. As the exact geodesic distances $d(v_1, s_i)$, $i = 1, 2, 3$, are known (from the pre-computed geodesic file), we can compute the Barycentric-like coordinate $(\lambda_1, \lambda_2, \lambda_3)$ of v_1 with respect to geodesic triangle $\triangle_{s_1 s_2 s_3}$. Similarly, compute $(\lambda_4, \lambda_5, \lambda_6)$ for v_2 with respect to $\triangle_{s_4 s_5 s_6}$. Let $d_{ij} = d(s_i, s_j)$ denote the exact geodesic distance between s_i and s_j , $1 \leq i \leq 3, 4 \leq j \leq 6$, which can also be obtained from the pre-computed geodesic file. Then the linear interpolation method computes the geodesic between v_1 and v_2 by

$$\tilde{d}(v_1, v_2) = \begin{pmatrix} \lambda_1 & \lambda_2 & \lambda_3 \end{pmatrix} \begin{pmatrix} d_{14} & d_{15} & d_{16} \\ d_{24} & d_{25} & d_{26} \\ d_{34} & d_{35} & d_{36} \end{pmatrix} \begin{pmatrix} \lambda_4 \\ \lambda_5 \\ \lambda_6 \end{pmatrix}.$$

Although conceptually simple and easy to implement, this linear interpolation method performs poorly as shown in Figure. 4.7. This is mainly because the linear interpolation acts like a low-pass filter that averages all geodesic distances d_{ij} . In sharp contrast, our algorithm can generate much more accurate results by unfolding with respect to each side of the geodesic triangle, and

Figure 4.6: Computing geodesic distance field on BUDDHA ($n = 50K$). The user can specify source points arbitrarily on the 3D model, then our program is able to compute the approximate geodesic distance field in linear time. The first three columns show the approximate results by our algorithm with $m = 1000$, 2000 , and 3000 respectively. The last column shows the exact result by [Xin and Wang 2009].

then choose the shortest distance. Loosely and intuitively speaking, the linear interpolation is “isotropic”, but ours is “anisotropic”.

Figure 4.7: Comparison to the linear interpolation method. (a) Geodesic triangulation. (b) Exact result. (c) The approximate result by unfolding technique. (d) The approximate result by linear interpolation.

4.4.4 Limitations

The proposed framework has limitations.

First, Theorem 1 presents the upper bound of the approximation error for two query points located in *different* geodesic triangles. In fact, if two query points are close enough that they are in the *same* geodesic triangle, the approximation error could be much higher. Figure 4.8 (a) illustrates an example, where a single geodesic triangle contains two highly curved features. Using the one-side unfolding, the two images q'_1 and q'_2 are close to each other, which leads to inaccurate result. In practice, such case can be improved by constructing a finer geodesic triangulation that separates the highly curved features, e.g., splitting $\triangle_{s_1 s_2 s_3}$ such that the two features are separated in two geodesic triangles as shown in Figure. 4.8(b). Then our two-side unfolding operation can generate much more accurate result.

Second, the pre-processing step has $O(m^2 + n)$ space complexity. Although $m \ll n$, the number of sampling points m clearly depends on the mesh geometry and complexity. For a large-scale mesh with $n = 1000K$ vertices and $m = 20K$ sampling points, the memory cost is 3.2GB memory. This limits our method for no more than $20K$ sampling points on a 32-bit PC.

Figure 4.8: (a) Query points located in the same geodesic triangle may lead to large approximation error. (b) Improved result can be obtained by constructing a finer geodesic triangulation that separates highly curved features.

Figure 4.9: Computing geodesic distance field on the GARGOYLE model ($n = 350K$). The user can specify source points arbitrarily on the 3D model, then our program is able to compute the approximate geodesic distance field in linear time, which allows the user to view the geodesic distance field at interactive speed.

4.5 Error Analysis

We analyze the error bound of the proposed algorithm in this section.

Lemma 1. *Given a triangle mesh M , let $\triangle_{s_1 s_2 s_3} \in M$ be a geodesic triangle and $p \in M$ is*

Figure 4.10: Computing geodesic distance field on the ARMADILLO model ($n = 173K$).

Figure 4.11: Computing geodesic distance field on the FERTILITY model ($n = 30K$).

an arbitrary point such that $p \notin \triangle_{s_1 s_2 s_3}$. Assume that the exact geodesic distances $d(p, s_2)$ and $d(p, s_3)$ are known, one can map p (resp. s_i) to $p' \in \mathbb{R}^2$ (resp. s'_i) by two-side unfolding $u(p, s_1 | s_2 s_3)$. Let $q \in s_2 s_3$ be the intersection point of geodesics $s_2 s_3$ and $p s_1$. Let $q' \in \mathbb{R}^2$ be the intersection point between $p' s'_1$ and $s'_2 s'_3$.

The approximate geodesic distance $\tilde{d}(p, s_1)$ is given by

$$\begin{cases} \|p' s'_1\|, & \text{if } q' \in s'_2 s'_3, \\ \|p' s'_2\| + \|s'_2 s'_1\|, & \text{if } q' \text{ is on the left side of } s'_2 s'_3, \\ \|p' s'_3\| + \|s'_3 s'_1\|, & \text{otherwise.} \end{cases}$$

Then the approximation error is bounded by $d(s_2, s_3)$, i.e., $|d(p, s_1) - \tilde{d}(p, s_1)| \leq d(s_2, s_3)$.

Proof. By using triangle inequality, one obtains

$$d(p, s_1) \leq \min_{2 \leq i \leq 3} \{d(s_1, s_i) + d(p, s_i)\}. \quad (\text{Eq. 4.2})$$

The following two inequalities hold

$$d(s_1, s_2) - d(s_2, q) \leq d(s_1, q) \quad (\text{Eq. 4.3})$$

and

$$d(p, s_2) - d(s_2, q) \leq d(p, q). \quad (\text{Eq. 4.4})$$

Adding (Eq. 4.3) and (Eq. 4.4) together, we have

$$d(s_1, s_2) + d(p, s_2) - 2d(s_2, q) \leq d(s_1, q) + d(p, q) = d(s_1, p). \quad (\text{Eq. 4.5})$$

Figure 4.12: Computing geodesic distance field on the LUCY model ($n = 263K$).

Figure 4.13: Lemma 1

Similarly,

$$d(s_1, s_3) + d(p, s_3) - 2d(s_3, q) \leq d(s_1, p). \quad (\text{Eq. 4.6})$$

Adding (Eq. 4.5) and (Eq. 4.6) together, one obtains

$$d(s_1, s_2) + d(p, s_2) + d(s_1, s_3) + d(p, s_3) - 2d(s_2, s_3) \leq 2d(s_1, p), \quad (\text{Eq. 4.7})$$

since $d(s_2, s_3) = d(s_2, q) + d(q, s_3)$.

Inequality (Eq. 4.7) implies the following inequality

$$\min_{2 \leq i \leq 3} \{d(s_1, s_i) + d(p, s_i)\} - d(s_2, s_3) \leq d(s_1, p). \quad (\text{Eq. 4.8})$$

Applying similar idea to 2D case (Figure. 4.13 right), we have

$$\min_{2 \leq i \leq 3} \{d(s_1, s_i) + d(p, s_i)\} - d(s_2, s_3) \leq \|p's'_1\| \quad (\text{Eq. 4.9})$$

and

$$\|p's'_1\| \leq \min_{2 \leq i \leq 3} \{d(s_1, s_i) + d(p, s_i)\}. \quad (\text{Eq. 4.10})$$

for the case $q' \in s'_2 s'_3$, which implies

$$|d(p, s_1) - \tilde{d}(p, s_1)| \leq d(s_2, s_3).$$

One can easily verify the conclusion for the other two cases, i.e., q' is on the left or right side of $s'_2 s'_3$, in a similar way. $\square$

Remark. Here we show that the upper bound in Lemma 1 is tight. As shown in Figure. 4.14, Let $A = (1, 0, 0)$, $B = (0, 0, 1)$, $C = (-1, 0, 0)$, $D = (0, 0, -1)$ be four points on the unit sphere $\mathbb{S}^2$. Let $a = \frac{\pi}{2}$ be $\frac{1}{4}$ of the length of a great circle.

Figure 4.14: The upper bound in Lemma 1 is tight.

Then we have

$$d(A, B) = d(B, C) = d(C, D) = d(D, A) = a,$$

and

$$d(A, C) = 2a$$

.

Using two-side unfolding $u(B, D|AC)$, the resultant 2D triangles $\triangle A'B'C'$ and $\triangle A'C'D'$ are degenerate, such that $\tilde{d}(B, D) = \|B'D'\| = 0$. However, the exact geodesic distance $d(B, D) = 2a$. Thus, the approximation error is $2a$, which reaches the upper bound in Lemma 1.

Lemma 2. Given two vertices v_1 and v_2 that are in different geodesic triangles, say $v_1 \in \triangle s_1 s_2 s_3$ and $v_2 \in \triangle s_4 s_5 s_6$, the approximation error $|d(v_1, v_2) - \tilde{d}(v_1, v_2)| \leq 2L$, where L is the maximum edge length of geodesic triangles $\triangle s_1 s_2 s_3$ and $\triangle s_4 s_5 s_6$.

Figure 4.15: Lemma 2.

Proof. Let $L_1 \triangleq \max_{1 \leq i \leq 3} d(s_i, s_{[i]+1})$ and $L_2 \triangleq \max_{4 \leq i \leq 6} d(s_i, s_{[i]+1})$ be the maximal side lengths of geodesic triangles $\triangle s_1 s_2 s_3$ and $\triangle s_4 s_5 s_6$ respectively. Naturally, $L = \max\{L_1, L_2\}$.

Following Equation (Eq. 4.8), the exact geodesic distance $d(v_1, v_2)$ satisfies

$$\min_{4 \leq i \leq 6} \{d(v_1, s_i) + d(s_i, v_2)\} - L_2 \leq d(v_1, v_2), \quad (\text{Eq. 4.11})$$

and

$$d(v_1, v_2) \leq \min_{4 \leq i \leq 6} \{d(v_1, s_i) + d(s_i, v_2)\}. \quad (\text{Eq. 4.12})$$

By extending this property onto 2D case, we have $\tilde{d}(v_1, v_2)$ satisfies (note the correspondence in length during unfolding)

$$\min_{4 \leq i \leq 6} \{\tilde{d}(v_1, s_i) + d(s_i, v_2)\} - L_2 \leq \tilde{d}(v_1, v_2), \quad (\text{Eq. 4.13})$$

and

$$\tilde{d}(v_1, v_2) \leq \min_{4 \leq i \leq 6} \{\tilde{d}(v_1, s_i) + d(s_i, v_2)\}. \quad (\text{Eq. 4.14})$$

Combining (Eq. 4.11)-(Eq. 4.14), we have

$$|d(v_1, v_2) - \tilde{d}(v_1, v_2)| \leq L_2 + \max_{4 \leq i \leq 6} |d(v_1, s_i) - \tilde{d}(v_1, s_i)|. \quad (\text{Eq. 4.15})$$

Note that $v_1 \in \triangle_{s_1 s_2 s_3}$ while $s_i \notin \triangle_{s_1 s_2 s_3}$ for any $4 \leq i \leq 6$. From Lemma 1, we have

$$\|d(v_1, s_i) - \tilde{d}(v_1, s_i)\| \leq L_1. \quad (\text{Eq. 4.16})$$

Putting Equation (Eq. 4.15) and (Eq. 4.16) together, we complete the proof. $\square$

Theorem 1. *If the pair of query points $q_1 \in \triangle_{v_1 v_2 v_3}$ and $q_2 \in \triangle_{v_4 v_5 v_6}$ (may be surface points) are far away enough from each other, i.e., any pair of v_i and v_j ($i=1,2,3, j=4,5,6$) are not the same geodesic triangle, then we have the approximation error $|d(q_1, q_2) - \tilde{d}(q_1, q_2)| \leq 2L + 2l$, where L is the maximum edge length of geodesic triangles that contains q_1 and q_2 , and l is the maximum edge length of triangles $\triangle_{v_1 v_2 v_3}$ and $\triangle_{v_4 v_5 v_6}$.*

Proof sketch. According to our unfolding technique, we estimate the geodesic distance between q_1 and q_2 by the following three steps:

- (i) Compute the approximate geodesic distances between v_i and v_j ($i = 1, 2, 3, j = 4, 5, 6$) based on exact geodesic distances between related sample points;
- (ii) Compute the approximate geodesic distances between q_1 and $v_j, j = 4, 5, 6$, based on $\tilde{d}(v_i, v_j)$ that are approximated in the last step;

(iii) Note that $q_2 \in \triangle v_4 v_5 v_6$ while $q_1 \notin \triangle v_4 v_5 v_6$, we can estimate $\tilde{d}(q_1, q_2)$ based on $\tilde{d}(q_1, v_j), j = 4, 5, 6$, in the last step.

According to the conditions that q_1 and q_2 are far away enough such that v_i and v_j are in different geodesic triangles, all unfolding processes are performed in a two-side style. Therefore we can complete the proof similarly to Lemma 1 and Lemma 2.

Chapter 5

Parallel Algorithm for Computing Single-Source-All-Destination Geodesic

In many graphics applications, the computation of exact geodesic distance is very important. However, the high computational cost of the existing geodesic algorithms means that they are not practical for large-scale models or time-critical applications. To tackle this challenge, we propose the parallel Chen-Han (or PCH) algorithm, which extends the classic Chen-Han (CH) discrete geodesic algorithm to the parallel setting. The original CH algorithm and its variant both lack a parallel solution because the windows (a key data structure that carries the shortest distance in the wavefront propagation) are maintained in a strict order or a tightly coupled manner, which means that only one window is processed at a time. We propose dividing the CH's sequential algorithm into four phases, window selection, window propagation, data organization, and events processing so that there is no data dependence or conflicts in each phase and the operations within each phase can be carried out in parallel. The proposed PCH algorithm is able to propagate a large number of windows simultaneously and independently. We also adopt a simple yet effective strategy to control the total number of windows. We implement the PCH algorithm on modern GPUs (such as Nvidia GTX 580) and analyze the performance in detail. The performance improvement (compared to the sequential algorithms) is highly consistent with GPU double-precision performance (GFLOPS). Extensive experiments on real-world models demonstrate an order of magnitude improvement in execution time compared to the state-of-the-art.

5.1 Introduction

Computing geodesics on triangle meshes, as a fundamental problem in geometric modeling, has been widely studied since the mid-1980s. Over the years, several algorithms have been proposed to compute the “single-source-all-destination” geodesic distance, such as, the MMP algorithm [58], the CH algorithm [10] and the improved CH (or ICH) algorithm [93], the fast marching method [43] and the approximate MMP algorithm [81]. While the state-of-the-art approaches [81] [93] work quite well for models of moderate size, their high computational cost means that they are not practical for large-scale models or time-critical applications.

In the past decade, there has been an increasing trend of performing the traditionally-CPU-handled computation on a graphics processing unit (GPU), which uses large numbers of graphics chips to parallelize the computation. However, developing parallel algorithms for discrete geodesics is technically challenging due to the lack of parallel structure; specifically, the geodesic distance is propagated from the source to all destinations in a sequential order. To date, the only parallel geodesic algorithm is that of Weber et al. [85], who developed a raster scan-based version of the fast marching algorithm. Although Weber et al.’s method is highly efficient, it only computes the first-order approximation of geodesic and requires parameterization of the surface into a regular domain, which is usually difficult for surfaces with complicated geometry and/or topology. To our knowledge, there is no parallel algorithm with which to compute an exact geodesic on triangle meshes.

Technical challenges The existing exact geodesic algorithms (for example, the MMP, CH and ICH algorithms) do not support parallel computing due to the lack of parallel structure. These algorithms partition each edge into a set of intervals, called windows, which are maintained in a queue and then propagated across the mesh faces. The propagation step pops a window from the queue and then computes its children windows and performs clipping or merging if necessary, which can add, modify, or remove existing windows, and the queue is updated accordingly. These algorithms generate the correct solution regardless of the order in which windows are removed from the queue; however, selecting windows in arbitrary order causes extremely slow performance.

The MMP algorithm and the ICH algorithm propagate the windows as a wavefront by ordering them in a priority queue according to their distance from the source. However, maintaining a

priority queue is a sequential and global process. It is highly non-trivial to implement a parallel priority queue on GPUs. The performance of the MMP and ICH algorithms drops significantly without the priority queue.

The CH algorithm organizes the windows in a tree data structure and propagates the windows in a breadth-first-search sweep from the root to the leaves. Due to the data dependency between parents and their children nodes, only windows on the same level can be processed in parallel. To maintain this global tree data structure, a CPU/GPU synchronization must be conducted for each layer. As the depth of the tree equals the number of mesh faces, the $O(n)$ times synchronization is too expensive for large models, making parallel implementation unpractical.

Our contributions This chapter extends the classic CH algorithm for parallel computing the exact geodesic distance on triangle meshes. Our algorithm, which is called the Parallel Chen-Han (or PCH) algorithm, follows the window propagation framework, as all the exact algorithms do. Our idea is to divide CH’s sequential algorithm into several phases in such a way that there is no data dependence in each phase and the operations within each phase can be done in parallel. Our parallel algorithm design considers three key factors: (1) propagating the windows in as parallel a manner as possible; (2) controlling the total number of windows effectively; and (3) avoiding data conflicts during window propagation. The following concepts made it possible for us to address these issues.

- First, in contrast to the existing approaches, which maintain the windows in a “tightly-coupled” structure (such as the tree data structure in the CH algorithm) or a “strictly-ordered” structure (such as the priority queue in MMP/ICH algorithms), the windows in the proposed PCH algorithm are loosely organized. Consequently, our method neither sorts the windows nor traces the ancestor windows. Therefore, the windows can be propagated in a fully parallel and independent manner.
- Second, to reduce the total number of windows, the PCH algorithm adopts a k -selection to determine the k windows nearest to the source, where the parameter k is specified by the user. These windows will then be processed independently by multiple GPU threads. Together with a parallel k -selection and the window filtering techniques in the CH and ICH algorithms, our parallel algorithm is very effective for window propagation,

as well as for controlling the total number of windows. Extensive evaluation of real-world models shows that our algorithm only has slightly more windows than the ICH algorithm, which uses a priority queue to maintain the wavefront.

- Third, during window propagation, if a window brings the vertex v a shorter distance than its current distance, our method does not immediately update the geodesic distance at v immediately, since other threads may also access v at the same time and such an update would cause data conflicts. Instead, our method triggers an update event. Once all the selected windows have been propagated, our method processes these events and updates the corresponding data. This updating can also be done in a parallel manner.

The proposed PCH algorithm is the first parallel technique for calculating exact geodesic distance on triangle meshes. We have implemented our algorithm on modern GPUs and obtained promising results on a wide range of models. We have observed that the performance improvement is highly consistent with the GPUs' double-precision performance (GFLOPS). For example, our method can compute the exact geodesic distance field on the 1.8-million-face Blade model (see Figure 5.1) in 11 seconds on an Nvidia GTX580 graphics card, which is an order of magnitude faster than the state of the art [93]. This indicates that our algorithm is suitable for applications involving intensive geodesic distance computations.

5.2 Preliminary

Let $M = (V, E, F)$ be a triangle mesh representing an orientable 2-manifold where V , E and F are the vertex, edge and face sets, respectively. For a vertex $v \in V$, the total angle is the sum of interior angles formed between each pair of neighboring edges incident at v . A vertex v is called *spherical* if its total angle is less than 2π , *Euclidean* if the total angle equals 2π , and *saddle* if the total angle is greater than 2π . Mitchell et al. [58] proved that there exists a geodesic path from the source s , typically a mesh vertex, to any other point on the surface. They also showed that a geodesic path cannot pass through a spherical vertex except that it is an end point.

Theorem [58] The general form of a geodesic path is a path that goes through an alternating sequence of vertices and edges such that the unfolded image of the path along any edge se-

Figure 5.1: Computing the exact geodesic distance on triangle meshes by using the proposed PCH algorithm. Our implementation on Nvidia GTX 580 is one magnitude of order faster than [Xin and Wang 2009].

Figure 5.2: (a) A window represents an interval $[A, B]$ on an oriented edge e , which is “visible” by the pseudo source I after unfolding the corresponding face sequence. (b) The position of the pseudo source on the unfolded plane can be determined by d_0 and d_1 .

quence is a straight line segment and the angle of the path passing through a vertex is greater than or equal to π .

If a geodesic path from the source s to destination t passes through one or more saddle vertices, we call the vertex, which is nearest to t , a *pseudo* source. Clearly, a geodesic path must be a straight line inside a triangle. When crossing over an edge, the geodesic path must also correspond to a straight line if the two adjacent faces are unfolded into a common plane. The exact geodesic algorithms partition each mesh edge into a set of intervals, called *windows* [81] [93], each of which encodes the geodesic paths passing through the *same* face sequence.

As shown in Figure 5.2, a window data structure associated to an edge e is a 6-tuple (d, A, B, d_0, d_1, e) where

- d is the distance from the pseudo source to the source s ;
- A and B are the left and right endpoints of the interval;
- d_0 and d_1 are the distances from the edge endpoints to the (pseudo) source.

An important step in the exact geodesic algorithms is to propagate the windows across the adjacent triangle, which yields new windows on the opposite edge(s). In general, a window

at edge e can have up to two children at the edges opposite to e . See Figure 5.3(a)-(b). For a special case where the geodesic path passes through a saddle vertex v , one or more windows are then created for the edges in a fanned area. See Figure 5.3(c).

Figure 5.3: Window propagation results in new windows. (a) A window creates one child on the opposite edge. (b) A window creates two children on opposite edges. (c) For a saddle point v , the injection ray (the dashed red line) is split into two directions (dashed grey lines), each is an extended straight line of the injection ray after unfolding. Then the new windows are generated on the opposite edges in the fanned area formed by the two directions.

Chen and Han [10] proposed using a tree data structure to keep record of the parent-child relationship. The depth of the tree is equal to the number of faces. To avoid exponential explosion in the tree, Chen and Han adopted a simple “one angle one split” scheme: if two windows occupy a vertex, at most one of them can have two children. See Figure 5.4(a). Xin and Wang [93] further reduced the number of windows by using a strict window filtering technique. See Figure 5.4(b). They also suggested using the priority queue to organize the windows according to their distance back to the source. Adopting the priority queue together with the two window filtering techniques, the ICH algorithm outperforms both the MMP and CH algorithms.

Mitchell et al. [58] proved that each edge may have $O(|E|)$ windows and therefore the total number of windows is $O(|E|^2)$, which is the theoretical upper bound. In practice, Surazhsky et al. [81] observed that for typical meshes an edge has an average of $O(\sqrt{|E|})$ windows. Thus, the key for developing an efficient geodesic algorithm is to organize the windows effectively.

Figure 5.4: Window filter. (a) The “one angle one split” filter in the CH algorithm. Two windows w_1 and w_2 share the same edge (v_0, v_1) and occupy the vertex v_2 . However, the pseudo source I_1 can not provide a shorter distance to v_2 . Thus, only window w_2 can generate two children. (b) The filter used in the ICH algorithm. The window w on edge (v_0, v_1) is generated by some window on edge (v_1, v_2) . Let g_i be the *current* shortest distance for vertex v_i , $i = 0, 1, 2$. The window w is useless, if one of the following inequalities holds: $d + \|IB\| > g_0 + \|v_0B\|$ or $d + \|IA\| > g_1 + \|v_1A\|$ or $d + \|IB\| > g_2 + \|v_2B\|$.

5.3 Computing Geodesics in Parallel

The proposed parallel Chen-Han algorithm is based on the shared memory model so that all of the data (such as mesh data structure, geodesic distance, etc.) are assessable to all processors. This section presents our algorithm on modern GPUs, as these are readily available to the graphics community.

5.3.1 Data Structure

Our algorithm maintains several global variables, as shown in Table 5.1. Let $M = (V, E, F)$ denote the input triangle mesh, where V , E and F are the set of vertices, edges and faces. We use the half-edge data structure to encode M 's incidence information. Considering the limited GPU memory, we adopt a minimal half-edge data structure. Each edge is decomposed into two half-edges with opposite orientations. Each half-edge stores the index of the starting vertex, the index of the opposite half-edge and its length. Each vertex references one outgoing half-edge.

```

struct half_edge {
    int starting_vertex_id;
    int opposite_half_edge_id;
    double length;
};

half_edge he[]; // size 3|F|
half_edge outgoing_he[]; // size |V|

```

Compared to the standard half-edge data structure, our struct *half_edge* does not have the face that it borders and the next half-edge around the face. The complete incidence information is encoded into two arrays, *he* of dimension $3|F|$, which stores all half-edges of M , and *outgoing_he*, of dimension $|V|$, which stores one of the half-edges emanating from each vertex. The three half-edges of the i -th triangle are stored at $he[3i]$, $he[3i + 1]$ and $he[3i + 2]$, respectively. Given the *half_edge* $[j]$, the triangle that it borders is the $\lfloor j/3 \rfloor$ -th triangle and the next-half-edge around that face is *half_edge* $[3\lfloor j/3 \rfloor + (j + 1)\%3]$, where $\lfloor \cdot \rfloor$ is the floor function.

To access the one-ring neighbors of the i -th vertex v_i , one can simply start with the outgoing half-edge *outgoing_he* $[i]$ and iteratively switch to the opposite half-edge and find the next half-edge that points to neighboring vertex. The half-edge data structure, *he* and *outgoing_he*, are located in the GPU memory and accessible to the GPU threads in the real-only manner. The PCH algorithm does not require the vertex coordinates, instead, it needs only the metric (the edge length).

```

double geod_dist[]; // size |V|
window angle_split[]; // size 3|F|

```

The array *geod_dist* $[0..|V|-1]$ stores the distance value at each vertex. The initial value for each non-source vertex is ∞ . When the algorithm terminates, the value *geod_dist* $[i]$ gives the globally shortest geodesic distance at the i -th vertex.

The array *angle_split* $[0..3|F|-1]$ contains the window that can provide the shortest distance to each angle. Such information is used by the Chen-Han's "one angle one split" window filter.

Data	Location	CPU access	GPU access
half_edge data structure	GPU RAM	-	read only
<i>angle_split</i> [] <i>geod_dist</i> []	CPU RAM (mapped to GPU RAM)	read & write	read only
memory pool	GPU RAM	-	read & write

Table 5.1: Global variables.

During the wavefront propagation, a large number of windows will be created, processed, and then discarded. To allow efficient access of the windows, our program creates a memory pool in the GPU memory. The memory pool contains all the active windows, which are stored consecutively so that there are no memory gaps between adjacent windows. This condition allows us to distribute the active windows *evenly* to all GPU threads. The memory pool also provides each GPU thread with a buffer for the temporary storage of the new windows. The global array *buffer_address*[0..*T*-1] stores the beginning address and size of each buffer. Unlike the continuous storage of the active windows, these thread buffers are not physically continuous in the memory pool.

5.3.2 PCH Algorithm

Let $S = \{s_i\}_{i=1}^m$ be the set of source points specified by the user. For each source point s_i , our algorithm creates a window for every edge facing s_i , and then iteratively processes the windows.

Each iteration contains the following four steps: First, the algorithm selects k windows that are nearest to the source points s_i , $i = 1, \dots, m$. Second, the selected k windows are processed by T threads in a completely independent manner. The window propagation results in new windows, which are stored in each thread's own buffer. If a window w can provide a shorter distance for a vertex v , our algorithm does not immediately update the corresponding entries of *geod_dist* or *angle_split*, as doing so would cause conflicts. Instead, the algorithm creates an update event, which will be processed later. Third, the algorithm collects the new windows and organizes them with the existing windows in the memory pool, so that there are no memory gaps between the adjacent windows. Finally, the algorithm processes these update events and updates the arrays *geod_dist* and *angle_split* for the corresponding vertices and corner angles.

Figure 5.5: The ICH algorithm maintains the windows in a priority queue. Only the nearest window is processed. Such priority queue structure is very difficult to maintain in parallel. However, the PCH algorithm put all windows together. In each iteration, we select k nearest windows and process them simultaneously.

Figure 5.6: During each iteration, our algorithm selects k windows that are nearest to the source points and assigns these windows to T GPU threads. Each window is taken by one GPU thread and then propagates independently, which will result in one or more children windows. After all the selected windows have been processed, our algorithm collects the newly-generated windows and organize all windows such that there are no memory gaps between adjacent windows in the memory pool. The empty slots in the memory pool are drawn in light gray.

These four steps are repeated until all windows in the memory pool have been processed. The data flow for each iteration is shown in Figure 5.6 and the pseudo code is shown in Algorithm 1. And Figure 5.5 shows the key idea of our PCH algorithm, which parallelizes the complicate computation. Next, we explain the algorithm in details.

Initialization (lines 1, 2) In the initialization stage, our algorithm creates a memory pool, and two global arrays: the first array $geod_dist[|V|]$ contains the geodesic distance at each vertex, where the initial value is 0 for the source points and ∞ for all other vertices; the second array

Algorithm 1: Parallel Chen-Han Algorithm

Input: A triangle mesh $M = (V, E, F)$, the set of source points $S = \{s_i | s_i \in V, 1 \leq i \leq m\}$, the selection parameter k , and the number of GPU threads T

Output: The geodesic distance for each vertex

- 1: initialize the global variables
 - 2: **parallel** create a window for every edge opposite to the source points
 - 3: **Repeat**
 - 4: **parallel** select k nearest windows
 - 5: **parallel** propagate the selected windows
 - 6: organize the newly-generated windows and update events
 - 7: **parallel** process the update events
 - 8: **Until** all windows are processed
-

$angle_split[3|F|]$ contains the windows for each angle, which is used by the “one angle one split” window filter. For every edge opposite to the source point s_i , a window is created and put in the memory pool.

Step 1. Parallel selection (line 3) Rather than sorting all windows, our algorithm selects k windows, which are closest to the source points. The k -selection is a fundamental problem in computer science, which has been widely studied. The state-of-the art parallel algorithm is due to Monroe et al. [60]. Their algorithm proceeds via an iterative probabilistic guess-and-check process on pivots for a three-way partition. When the guess is correct, the problem is reduced to selection on a much smaller set. They proved that their probabilistic algorithm always gives a correct result and always terminates. We adopt Monroe et al.’s parallel algorithm in our GPU implementation. We also develop an approximate k -selection to improve the speed. We choose $k \gg T$ to fully utilize the pipeline of the GPU cores. The details of optimal parameter selection and the approximated version of k -selection will be presented in Section. 5.4.2.

Step 2. Parallel window propagation (line 4) All of the selected windows are evenly distributed to the T GPU threads. Some windows may generate one or more children, while others may not. Statistics on a wide range of real-world models show that each window generates an *average* maximum of 3.8 children¹. Therefore, we allocate each GPU thread a $4\lceil \frac{k}{T} \rceil$ -sized

¹In general, each window has at most two children during the propagation. See Figure 5.3. However, a saddle vertex can produce more than 2 windows. The worse case is that one window is created for each edge opposite to the saddle vertex. Our statistics show that when processing $L(\geq 8)$ windows in parallel, the number of new windows is less than $3.8L$.

buffer in the memory pool to contain the new windows. This fixed-size buffer works well in practice and we do not observe any buffer overflow in our experiments. In case of an overflow in the buffer of the i -th thread, for example, the thread i creates a new buffer of doubled size, copies the contents to the new buffer, and then updates the entry $buffer_address[i]$. These buffers are not required to be consecutive in the memory, since they store the new windows *temporarily*.

Algorithm 2: PropagatingWindow(w)

```
// assume  $w = (d, A, B, d_0, d_1, e)$  is on the oriented
// edge  $e = (v_0, v_1)$  and  $v_2$  is the opposite vertex of  $e$ ;
if  $A = v_0$  &&  $d_0 + d < geod\_dist[v_0]$  then
    //  $w$  provides a short distance to  $v_0$ ;
    create a distance update event  $(v_0, d_0 + d)$ ;
    if  $v_0$  is a saddle vertex then
        | create windows in the fanned area (see Figure. 5.3(c));
if  $B = v_1$  &&  $d_1 + d < geod\_dist[v_1]$  then
    //  $w$  provides a short distance to  $v_1$ ;
    create a distance update event  $(v_1, d_1 + d)$ ;
    if  $v_1$  is a saddle vertex then
        | create windows in the fanned area (see Figure. 5.3(c));
if  $w$  occupies  $v_2$  then
    // apply the one-angle-one-split filter;
    if  $w$  provides  $v_2$  a shorter distance than  $angle\_split[v_2]$  then
        | create an angle update event  $(e, w)$ ;
        | create  $w$ 's two children (see Figure. 5.3(b));
    else
        | //  $w$  has only one child on edge  $(v_0, v_2)$  or  $(v_1, v_2)$ ;
        | create  $w$ 's child (see Figure. 5.4(a));
    if  $\|Iv_2\| + d < geod\_dist[v_2]$  then
        | create a distance update event  $(v_2, \|Iv_2\| + d)$ ;
        | if  $v_2$  is a saddle vertex then
        | | create windows in the fanned area (see Figure. 5.3(c));
else
    | //  $w$  has only one child on edge  $(v_0, v_2)$  or  $(v_1, v_2)$ ;
    | create  $w$ 's child (see Figure. 5.3(a));
    apply the ICH filter for the new windows and store the useful
    windows in the buffer (see Figure. 5.4(b));
```

Note that during the window propagation, if a new window can provide the vertex v a shorter distance than its current one $geod_dist[v]$, we should update $geod_dist[v]$ accordingly. However, this would cause data conflicts, as other GPU threads may also access v at the same time. To avoid conflicts, we create a *distance* update event for v . Similarly, if a window w occupies the vertex v and provides v a shorter distance than that of the window in the corresponding entry of *angle_split*, we also create an *angle* update event. All these events will be processed later. The pseudo code for window propagation is shown in function `PropagatingWindow()`.

Step 3. Data organization (line 5) In the above window propagation step, each GPU thread T_i generates a certain number of windows, which are stored in T_i 's own buffer. The purpose of this step is to organize these newly-generated windows in the memory pool such that there are no memory gaps between adjacent windows, which is a critical condition for the parallel selection and parallel updating.

Since each GPU thread counts the number of new windows independently, we use $O(T)$ space to store these numbers, where T is the number of GPU threads. We then compute the accumulated number of new windows for each thread. Finally, each GPU thread copies its new windows from its buffer to the memory pool. The pseudo code for data organization is shown in function `OrganizingData()`. We use the same procedure to organize the generated distance and angle update events.

Algorithm 3: `ReorganizeData({dataCount}, {data}, addr)`

```
// datai: the buffer storing the output of the i-th thread;
// dataCounti: the size of datai;
// addr: the address of the first free slot in the memory pool;
cum0 ← 0;
for  $i = 1$  to  $T - 1$  do
  |  $cum_i \leftarrow cum_{i-1} + dataCount_{i-1}$ 
parallel for each thread  $i$  do
  | copy datai to  $addr + cum_i$ 
```

Step 4. Processing the update events (line 6) The window propagating step produces distance and angle update events, which are organized in step 3 such that they are stored in the memory pool in a seamless manner. This allows us to parallel process these events.

An event is a 2-tuple $(key, value)$. For the distance update event, key is the vertex id, and $value$ is the geodesic distance. For the angle update event, key is id of the oriented edge, which determines the opposite angle/vertex, say v , and $value$ is a window occupies v .

To avoid the conflicts in events updating, we must sort all events by their keys in increasing or decreasing order. If two keys tie, we compare the two events by their values. For example, for two distance events (v_i, d') and (v_i, d'') occurring at the same vertex v_i , the one with smaller distance wins. To break a tie between two windows, we compare the shortest distance from the source to the corresponding interval. This tie-breaking enables us to process only the first event at a vertex or angle, since all the subsequent events occurring at the same vertex or angle do not carry useful information. We then assign the ordered events evenly to T GPU threads, which update the corresponding vertices or angles. The pseudo code is shown in function `ProcessingEvents()`.

Algorithm 4: `ProcessingEvents({event}, {Data})`

```

// eventi: a 2-tuple {key, value};
// {Data}: the to-be-updated array geod_dist or angle_split;
thrust sort {event} by key and use value for tie breaking;
S ← size of {event};
parallel for each thread i do
    for  $j = i \times \lceil \frac{S}{T} \rceil$  to  $(i + 1) \times \lceil \frac{S}{T} \rceil - 1$  do
        if  $event_j.key \neq event_{j-1}.key$  then
             $Data[event_j.key] \leftarrow event_j.value;$ 

```

5.3.3 Correctness

The correctness of our parallel algorithm is based on two observations. First, during the window propagation, we adopt Chen and Han’s “one angle one split” and Xin and Wang’s filtering techniques to reduce the total number of windows. It is shown [10][93] that both techniques remove only the *useless* windows, which do not carry the shortest distance. Second, Mitchell et al. [58] proved that there exists a geodesic path from the source s to any other vertex. Our window propagation step keeps the *useful* windows, which carry the shortest distance. Thus, for a vertex v , we can always find a sequence of windows $w_1, w_2, \dots, w_l$, to encode the geodesic

path, where w_l is the last window on the edge e opposite to v such that w_l occupies the vertex v . So w_l finally assigns the geodesic distance at v . Regardless of the order in which the windows are processed, such window sequence always exists. Thus, the resulting distance field is correct.

5.4 Implementation & Experimental Results

5.4.1 Implementation Details

We implemented our parallel geodesic algorithm on a 64-bit PC with an Intel Xeon 2.66GHz CPU and 12GB memory. The graphics card is an Nvidia GTX 580 with 512 cores and 1.5GB memory. Our program is compiled using CUDA 4.2.

The input mesh is maintained by using the half-edge data structure. Only one copy of the mesh structure is transferred to GPU memory. More specifically, we need only an array of half-edges, each of which has the length information. We do not need to store the vertex in the GPU memory; the face connectivity is also encoded in the half-edge structure.

Our algorithm requires a parallel selection to choose the k nearest windows and a parallel sorting to order all update events by their keys and values. Selection and sorting can be fully implemented in parallel. We adopted the probabilistic parallel selection [60] and CUDA's thrust sorting [8] in our implementation, but, surprisingly, observed poor performance. As shown in Figure 5.7 (a), the GPU selection and sorting are very slow for the 144K-face Bunny, taking more than 80% of the execution time. This slowness is mainly because the GPU selection algorithms require a large amount of CPU/GPU synchronization. As a result, these algorithms obtain good performance only for a very large number of elements (for example, more than 1 million). To improve the performance, we adopted the following simple yet effective strategies in our implementation.

First, we implement an approximate k -selection on the GPU. Assume that there are $N(> k)$ windows, $w_1, \dots, w_N$, in the memory pool and that there are T threads available. We allocate the following windows $w_i, w_{i+T}, w_{i+2T}, \dots$, to the i -th thread, which enables each thread to obtain approximately $\lceil N/T \rceil$ windows, among which the $\lceil k/T \rceil$ nearest windows are selected.

Figure 5.7: The window selection and event sorting on the GPU take 3 to 6 seconds on the 144K-face Bunny model. We implement an approximate selection on the GPU and process the events on the CPU, which can significantly reduce the overall execution time. The vertical axes are the execution time in seconds and the total number of windows. The horizontal axes are the selection parameter k . Although the window selection is approximate, the computed geodesic distance is exact.

Thus, T threads select k windows in total. This approximate selection uses only a single GPU launch and can significantly reduce the execution time.

Second, since GPU sorting is not efficient, we copy the update events from GPU to CPU. To minimize the CPU/GPU data transfer, we use `cudaHostAlloc()` to allocate the memory for global variables `geod_dist[]` and `angle_split[]`. The memory is mapped to CUDA address space using `cudaHostAllocMapped`, so both CPU and GPU can access it. All writing operations on such global variables are performed by CPU and the GPU procedure `PropagatingWindow()` accesses these data in a read-only manner. Although this CPU operation makes the event updating step sequential, we find it is much more efficient than the GPU sorting and updating and improves the overall performance significantly. See Figure 5.7(b).

Remark 1. Our PCH algorithm requires the windows to be stored consecutively in the physical memory. This condition plays an important role in the parallel selection step, where the i -th thread checks the $i, i + T, i + 2T, \dots$, positions in the memory pool, and then selects the windows those are the nearest to the source. The consecutively stored windows guarantee that each thread requires approximately the same number of windows, which means that the workload is balanced. Also, because different threads may generate different numbers of windows during the window propagation step, the procedure `ReorganizeData()` is required in order to reorganize the new windows and ensure the condition for the next-round parallel selection.

Remark 2. Although our implementation adopts the *approximation* k -selection to choose the windows, the correctness of the computed geodesic distance is guaranteed, since our algorithm does not delete any useful windows and the correctness of the geodesic distance is independent of the order in which the windows are processed. Therefore, the resulting geodesic distance is exact.

Remark 3. One of the key factors in designing a parallel algorithm is to effectively avoid data conflicts. Because our PCH algorithm propagates the windows and then updates the arrays *geod_dist* and *angle_split* in parallel, it is critical to maintain the data consistency in these arrays. Mutex and atomic operations are two commonly used techniques to access the shared memory in parallel computing. However, these techniques are not suitable to the proposed PCH algorithm, which requires frequent access of the global arrays *geod_dist* and *angle_split*: each call of *PropagatingWindow* reads *geod_dist* three times and *angle_dist* once. It may also write *geod_dist* up to three times and *angle_dist* once.

The large size of both arrays makes it expensive to set a mutex lock for each element. More importantly, by using mutex lock, each read/write operation locks the corresponding resources to avoid the access by other threads, which will significantly compromise the performance. On the other hand, using the atomic operations in PCH is not effective either, since doing so when reading/writing the two global arrays will slow down the procedure *PropagatingWindow* - the most expensive of the four steps - thereby compromising the performance of the entire program.

Therefore, our implementation does not contain the mutex or atomic operations. Instead, we adopt the delayed update strategy to avoid data conflicts. Note that the procedure *PropagatingWindow* does not update *geod_dist* and *angle_dist*. Thanks to the read-only operations in *PropagatingWindow*, all the selected k windows can be propagated without any data dependence. Then, in the *ProcessingEvents* procedure, we sort the distance and angle updating events to detect and delete the conflicted data, after which the arrays can be updated in parallel. This delayed update strategy works quite well in practice, which is justified by the promising experimental results.

Figure 5.8: Testing the selection parameter k on an Nvidia GTX 580 with 512 cores. The total number of windows (in red) is approximately linear proportional to the selection parameter k (the horizontal axis). Choosing a small k does not utilize the GPU's parallel structure at all. On the other hand, choosing a very large k would produce too many useless windows, resulting in slow performance (see the blue timing curves). We found the optimal range for k is $[16 \times 2^{10}, 24 \times 2^{10}]$, which leads to the least execution time.

5.4.2 Parameter Setting

The performance of our PCH algorithm depends on the mesh complexity $n = |V|$, the GPU's computational power (such as the number of cores and frequency), and the selection parameter k , which specifies the number of nearest windows to be processed by GPU threads in parallel.

The value of k is closely related to the *total* number of windows produced in the wavefront sweep. Both the MMP algorithm and the ICH algorithm use a priority queue to determine which window is nearest to the source. Xin and Wang [93] showed that the priority queue is very effective in terms of controlling the total number of windows. If $k = 1$, the k -selection becomes a priority queue, leading to a small number of windows. For such a case, however, our program becomes a sequential program, as only one window is processed at a time. In order to take full advantage of the parallel nature of our algorithm, a large k is usually preferred, so that multiple windows can be processed independently. On the other hand, for a sufficiently large k , the total number of windows increases dramatically. For example, if k is larger than the total number of windows, then all windows are selected and processed simultaneously, which means that the from-near-to-far wavefront is not maintained at all. As observed in both the MMP and CH algorithms, such cases will result in an extremely large number of windows and, therefore, very poor performance. Consequently, there is a trade-off between controlling the total number of windows and utilizing the GPU's parallel structure. We tested various k on a large number of models and observed that the total number of windows is almost linear to the parameter k ; the smaller the k , the fewer windows, and vice versa. Figure 5.8 shows that the performance curves for Lucy, Bunny and Golf Ball, and the other models have very similar patterns. We found the optimal range for k is $[16 \times 2^{10}, 24 \times 2^{10}]$, which leads to the least execution time.

We also investigated the relationship between the optimal range of k and the mesh resolution. As Figure 5.9 shows, all the performance curves have the similar pattern, which reveals the trade-off between utilizing the GPU's parallel structure and obtaining the optimal performance on the Bunny model of various resolutions. We found the optimal range of k is insensitive to the mesh resolution.

Figure 5.9: Given the Bunny model of various resolutions, we found the optimal range of k is insensitive to the mesh resolution.

Figure 5.10: More results.

Figure 5.11: The performance improvement T_{ICH}/T_{PCH} is consistent with the GPU double precision performance (GFLOPS).

5.4.3 Performance

We tested our program on a wide range of 3D models, from the small Bunny (144K faces) to the large-scale Dragon (4 million faces). We observed that the execution time of the MMP/ICH algorithms and ours depends on the location of the source point, and that different source points may result in up to 10% – 15% difference in computation time because the total number of windows is dependent of the source point. To obtain a consistent evaluation, we repeated our test 100 times, where a random source point is generated each time. Table 5.3 shows the mesh complexity and the *average* execution time, memory cost, the number of windows, etc. The consistent experimental results show that our method obtains an order-of-magnitude improvement for models with more than 500K faces. Our algorithm is also flexible in terms of computing the “multiple-source-all-destination” geodesic distance. Table 5.2 shows the performance of our algorithm and the ICH algorithm on the Golf Ball and Pegasus.

Figure 5.7(b) shows that GPU processing takes more than 75% of the total execution time, while the corresponding ratio for large-scale models may exceed 90%. As all windows are processed completely in GPU, the performance of our program depends heavily on the GPU’s parallel computing capacity. We verified this by testing our program on three Nvidia graphics card: GTX 285, GTX 470 and GTX 580. As Figure 5.11 shows, we observed that the perfor-

# source points	Golf ball			Pegasus		
	T_{ICH}	T_{PCH}	T_{ICH}/T_{PCH}	T_{ICH}	T_{PCH}	T_{ICH}/T_{PCH}
1	23.71	2.51	9.45	32.61	3.33	9.79
2	22.02	2.30	9.59	33.10	3.04	10.80
4	19.36	2.01	9.65	29.94	2.83	10.56
8	14.61	1.46	10.04	34.55	3.16	10.94
16	11.17	1.02	10.90	31.67	3.40	9.33
32	8.49	0.69	12.32	26.02	2.83	9.21
64	5.99	0.50	11.97	24.33	2.34	10.41
128	4.60	0.37	12.53	20.08	1.99	10.10
256	3.62	0.29	12.65	15.39	1.57	9.81
512	2.85	0.23	12.46	12.68	1.25	10.17
1,024	2.36	0.20	12.06	9.95	0.98	10.15

Table 5.2: Statistics of “multiple-source-all-destination” distance. Execution time was measured in seconds.

Models	# faces	MMP		ICH			PCH				Speedup	
		Time (s)	Peak mem. (MB)	Time (s)	Peak mem. (MB)	Total #windows	Time (s)	Peak mem. (MB)	Total # windows	Peak # windows	T_{ICH}/T_{PCH}	
Bunny	144,036	8.19	250.49	6.24	7.04	6,910,123	0.95	15.10	8,912,758	47,729	5.28	6.57
Golf ball	245,760	26.86	676.32	26.10	12.61	27,770,244	2.49	22.50	28,541,081	59,218	9.78	10.48
Lion	305,608	17.18	476.36	12.60	14.43	13,986,984	1.51	26.80	16,315,130	54,916	7.45	8.34
Sphere	327,680	86.94	1,894.02	70.39	21.55	81,022,375	8.11	37.08	82,412,805	201,499	8.51	8.68
Armadillo	345,944	13.25	412.36	10.34	15.91	11,169,475	1.40	28.51	13,159,141	35,599	6.04	7.39
Lucy	525,814	22.27	627.03	18.20	22.17	21,590,268	1.71	42.93	22,996,519	57,691	7.43	10.64
Pegasus	667,474	49.80	1,224.35	33.69	27.47	28,471,897	3.09	53.15	31,632,899	56,460	8.55	10.90
Buddha	1,439,116	179.67	3,698.71	160.70	67.42	113,292,750	15.20	114.77	122,402,062	139,761	7.95	10.57
Ramesses	1,652,528	76.43	1,734.26	52.09	57.90	47,374,020	5.79	124.72	53,404,855	55,083	7.25	8.99
Gargoyle	1,726,398	211.25	4,096.23	179.90	71.30	119,568,420	17.50	139.77	137,773,318	208,614	8.47	10.28
Blade	1,765,388	205.91	3,725.16	122.80	72.12	90,044,841	11.00	137.67	97,250,102	134,014	8.82	11.16
Isidore horse	2,208,936	80.85	2,026.17	67.77	69.04	56,403,922	6.31	165.04	68,469,783	54,294	8.95	10.74
Dragon	4,000,000	534.53	8,311.39	421.20	174.37	265,327,798	41.32	308.40	325,421,513	240,455	8.63	10.19

Table 5.3: Model complexity and performance. Tests were repeated 100 times with random source point and the mean values are reported in this table. The last column shows both the worst and average speedup.

mance improvement T_{ICH}/T_{PCH} is highly consistent with GPU double-precision performance (GFLOPS). Accordingly, we believe that our algorithm can achieve better performance on the next-generation graphics card.

In addition to performance, we also measured the peak memory² of the MMP, ICH, and PCH algorithms. In order to avoid tiny windows, we adopt the same tolerance 10^{-6} for all the three algorithms. As Table 5.3 shows, the memory cost of our algorithm is much smaller than the MMP algorithm, which has $O(n^2)$ space complexity. Our memory cost is 1.5 – 2 times higher than the ICH algorithm due to the greater number of windows generated, as well as the fact that some extra space is required to store the distance and angle update events.

5.5 Comparison & Discussion

This section compares our method to the other exact geodesic algorithms, including the MMP algorithm, the CH algorithm, and the ICH algorithm. As mentioned before, all these algorithms do not have parallel structure.

Comparison to the MMP algorithm The MMP algorithm takes a continuous Dijkstra sweep and maintains the windows on the wavefront using a priority queue. It is highly non-trivial to maintain the priority queue in parallel. The MMP algorithm requires windows clipping during the window propagation. The strong dependency among the windows makes it difficult to process the windows in parallel. The MMP algorithm has $O(n^2)$ space complexity, which diminishes its application on the GPU because the GPU usually has much less RAM than CPU. Unlike the MMP algorithm, our method does not require the priority queue and window clipping operation. Our algorithm also has linear memory complexity.

Comparison to the CH algorithm Instead of using a priority queue, the CH algorithm organizes the windows in a tree structure and propagates windows from the root to the leaves. Because only the branch nodes are retained, the CH algorithm has linear memory complexity $O(n)$. Chen and Han also proposed a “one angle one split” scheme to avoid exponential explosion. Whenever a new window w occupies the vertex v and provides v a shorter distance,

²The peak memory of PCH is the actual GPU memory taken by our algorithm, rather than the size of the memory pool.

the original sub-tree at v is deleted. Due to the parent-child data dependency, only windows on the same level can be processed in parallel. As a result, a CPU/GPU synchronization must be conducted for each layer. Note that the depth of the tree is equal to the number of mesh faces. Such $O(n)$ synchronization is very time consuming, which means it is not practical to parallelize the CH algorithm.

Our algorithm also adopts the “one angle one split” scheme to identify the useless windows, which do not carry the shortest distance. However, our algorithm neither has the tree structure nor maintains the parent/child relation. This allows the windows to be propagated in a fully independent manner.

Comparison to the ICH algorithm The ICH algorithm is a variant of the CH algorithm, which adopts an effective window filtering technique to reduce the number of windows. Like the original CH algorithm, the ICH algorithm maintains the parent-child relation and the tree structure. Unlike the original algorithm, however, the window in our algorithm does not contain its parent information and no windows are sorted. This feature allows us to propagate a large number of windows in a completely independent manner.

The ICH algorithm also uses the priority queue to maintain the wavefront, which is able to minimize the total number of windows. Instead of the priority queue, our algorithm uses a simple k -selection to determine the windows, which are close to the source points. This strategy is very effective in terms of controlling the total number of windows. Our experimental results show that the total number of windows in our parallel algorithm is only slightly higher than that of the ICH algorithm.

The PCH algorithm on the CPU We also implement our PCH algorithm on the CPU in such a way that all the parallel procedures are replaced by their sequential counterparts. We adopt quickselection for the sequential k -selection, which selects the top k windows without ordering them, and then compare its performance to the ICH algorithm on the same hardware setting. We observe that the CPU PCH algorithm is 5-30% slower than the ICH algorithm for all test models.

The priority queue is known to play a critical role in the ICH algorithm, since it guarantees that the to-be-processed window is the closest to the source among all active windows. Strictly

ordering the windows by their distances to the source is a very effective way to control the total number of windows. The CPU PCH algorithm selects the top k windows and then propagate them one by one. Because the selected k windows are not ordered, the children windows of a far-but-early-processed window may be over-ridden by a close-but-late-processed window. This means that the CPU PCH algorithm must process process 10-40% more windows than the ICH algorithm, resulting in slower performance.

The GPU PCH algorithm processes the same number of windows as the CPU PCH algorithm. However, the overhead cost of processing the extra windows is very small compared to the performance speedup by the high throughput of the GPU. Therefore, the GPU PCH algorithm significantly outperforms its CPU counterpart and the ICH algorithm.

Window deletion & complexity analysis The CH algorithm organizes the windows in a tree data structure. Chen and Han [10] suggested that once a window becomes useless, the useless window and its sub-tree should be deleted (or tagged) immediately, which can guarantee the total number of windows is $O(n^2)$, leading to an $O(n^2)$ time complexity. Our PCH algorithm does not require the tree data structure to maintain the parent-children relation among windows. Also, it does not delete the useless windows until the update process. These features significantly reduce the data dependence, thereby providing great flexibility in the parallel algorithm design. However, the price to of such flexibility is the lack of rigorous complexity analysis.

Because time complexity is linearly proportional to the total number of windows, we empirically evaluated the time complexity by counting the total number of windows. As Table 5.3 shows, we observed that the total number of windows of the PCH algorithm is only 5%-20% more than that of the ICH algorithm. This implies that such a delay in windows deletion only has a very minor effect on the time complexity. As Table 5.3 shows, our PCH algorithm can achieve 8x-10x speedup for most of the models, which justifies the good performance of our algorithm.

5.6 Conclusion

It is highly desirable to develop a parallel algorithm to compute exact geodesic on triangle meshes. However, the existing exact geodesic algorithms have very strong data dependence

and lack the parallel solution. This chapter presents the PCH algorithm, which extends the classic Chen-Han algorithm to parallel setting. Our idea is to divide the sequential CH algorithm into four phases, namely window selection, window propagation, data re-organization and events processing, such that the operations in each phase have no dependence or conflicts and can be done in parallel. In contrast to the original CH algorithm, the proposed PCH algorithm neither maintains the windows in a tightly coupled manner nor sorts the windows according to their distance to the source. As a result, it can process a large number of windows simultaneously and independently. We also adopt a simple selection strategy which is effective to control the total number of windows. We implemented our parallel geodesic algorithm on modern GPUs (e.g., Nvidia GTX 580) and observed that the performance improvement (compared to the conventional sequential algorithms) is highly consistent with GPU double precision performance (GFLOPS), which justifies the parallel nature of our algorithm. Extensive experiments on real-world models demonstrate an order of magnitude improvement in execution time compared to the state-of-the-art.

Chapter 6

Application: Poisson Disk Sampling

Poisson disk sampling has excellent spatial and spectral properties, and plays an important role in a variety of visual computing areas. Although many promising algorithms have been proposed for multi-dimensional sampling in Euclidean space, very few research studies have been reported with regard to the problem of generating Poisson disks on surfaces due to the complicated nature of the surface. This chapter presents an intrinsic algorithm for parallel Poisson disk sampling on arbitrary surfaces. In sharp contrast to conventional parallel approaches, our method neither partitions the given surface into small patches nor uses any spatial data structure to maintain the voids in the sampling domain. Instead, our approach assigns each sample candidate a random and unique priority that is unbiased with regard to the distribution. Hence, multiple threads can process the candidates simultaneously and resolve conflicts by checking the given priority values. Our algorithm guarantees that the generated Poisson disks are uniformly and randomly distributed without bias. It is worth noting that our method is intrinsic and independent of the embedding space. This intrinsic feature allows us to generate Poisson disk patterns on arbitrary surfaces in $\mathbb{R}^n$ ($n \geq 3$). To our knowledge, this is the first *intrinsic, parallel* and *accurate* algorithm for surface Poisson disk sampling. Furthermore, by manipulating the spatially varying density function, we can obtain adaptive sampling easily. We implement our Poisson disk sampling algorithm on GPU and demonstrate practical applications including vector field visualization, mosaic tiling, and stippling.

6.1 Introduction

Poisson disk sampling distributes all samples that are uniformly and randomly located. It requires that the samples are as dense as possible but are at a minimum distance apart from one another. Due to its excellent spatial and spectral properties, Poisson disk sampling is widely used in computer graphics and visualization, such as anti-aliasing, global illumination, non-photorealistic rendering, remeshing, texture synthesis, vector field visualization, etc. Dart throwing, proposed by Dippé and Wold [24], is the first accurate approach to generate Poisson disk patterns in $\mathbb{R}^n$. However, this brute-force approach is impractical and inefficient, since a large number of samples are involved in the attempt but only a small percentage of them are eventually inserted into the distribution. Since then, many approaches have been proposed to improve the performance, e.g., jittered sampling [19], spatial data structures [27, 34], procedure tiling [45, 44, 63], and hierarchical sampling [57, 56].

While tremendous efforts have been focused on the multidimensional sampling in Euclidean space, very few studies have been reported on generating Poisson disk patterns on curved surfaces. A very natural research direction is to extend the key ideas and data structures of the existing 2D sampling algorithms to curved surfaces. However, this extension is usually technically challenging due to the following reasons: first, a surface is a two-dimensional manifold that has arbitrary topology and complicated geometry, and is embedded in $\mathbb{R}^3$ or even higher dimensional space. Second, the generated samples should be randomly and uniformly distributed on surfaces, and exhibit the blue noise property. Third, the algorithm can be parallelized and implemented on modern graphics hardware easily. Fourth, the algorithm should be intrinsic to the geometry and insensitive to the mesh resolution and tessellation. Last but not the least, the exact geodesic distance should be used to enforce the minimum distance constraint between any pair of samples.

Recently, Wei pioneered the phase group technique [86], which subdivides the sample domain into grid cells and then draws samples concurrently from multiple cells that are sufficiently far apart so that their samples cannot conflict one another. The phase group technique is very intuitive, inherently parallel and highly efficient for Poisson disk sampling in Euclidean space of arbitrary dimension. Wei's algorithm can be extended to generate Poisson disks on 3D surface

by creating the phase groups in its embedding space [9]. However, due to its dependency of the ambient space, it is difficult to apply the phase graph method [9] to surfaces with complicated geometry/topology or embedded in high dimensional spaces. Furthermore, as pointed out in [34], the phase group algorithm is approximate in that the generated distribution is not fully random.

In this chapter, we present a new method for parallel Poisson disk sampling. Rather than the phase group method that explicitly partitions the sampling domain to generate the samples in parallel, our approach assigns each sample candidate a random and unique priority that is unbiased with regard to the distribution, and organizes the samples by their indices. Hence, multiple threads can process the candidates simultaneously and resolve conflicts by checking the given priority values. Our algorithm is accurate as the Poisson disks are uniformly and randomly distributed without bias. Our algorithm also distinguishes itself from the existing approaches by its intrinsic feature, since all the computations are purely based on the intrinsic metric. This embedding-space-independent feature allows us to generate Poisson disk distribution on arbitrary surfaces in $\mathbb{R}^n$ ($n \geq 3$). To our knowledge, this is the first *intrinsic*, *accurate* and *parallel* algorithm for generating Poisson disk samples on arbitrary surfaces. Furthermore, by manipulating the spatially varying radius function, we can obtain adaptive sampling easily. The spectrum analysis shows that the proposed algorithm is accurate and the generated Poisson disk distributions faithfully exhibit blue noise property.

In addition, we present a simple and yet effective technique to compute maximal Poisson disk sampling in $\mathbb{R}^2$ and $\mathbb{R}^3$. Taking a non-maximal Poisson disk sampling as input, our algorithm efficiently detects the regions that allow additional samples and then generates Poisson disks in these regions. The key idea is to convert the complicated plane or space searching problem into a simple searching on circles or spheres, which is one dimensional lower than the original sampling domain. Compared to the existing maximal Poisson disk sampling algorithm, our method is memory efficient, highly fast and fully parallel, which can generate more than 8 million samples per second in $\mathbb{R}^2$ and 0.55 million samples per second in $\mathbb{R}^3$ on Nvidia GTX580. Furthermore, our algorithm is flexible that can generate maximal Poisson disk sampling on arbitrary polygonal meshes. The spectrum analysis shows the high quality of the generated maximal Poisson disk sampling.

Figure 6.1: Our parallel algorithm is able to randomly and uniformly distribute samples on arbitrary surfaces. The generated distribution faithfully exhibits the blue noise pattern.

6.2 Intrinsic and Parallel Poisson Disk Sampling On Arbitrary Surfaces

6.2.1 Algorithmic Overview

Notations. The notations used throughout this chapter include:

- $M = (V, E, F)$: the input triangle mesh, where V , E and F are the set of vertices, edges and faces;
- A : the area of M ; $\mathcal{S}$: the set of Poisson disk samples;
- N : the number of Poisson disk samples, i.e., $N = |\mathcal{S}|$; $\mathcal{P}$: the set of dense points representing M ;
- T : the total number of threads;
- $d(p, q)$: the geodesic distance between points p and q ;

- r : the radius of the Poisson disk, so $d(p, q) \geq 2r$ for arbitrary two Poisson disk samples p and q ;
- $D(p, r)$: the geodesic disk of radius r centered at point $p \in M$;
- ρ : radius statistics measuring the packing density of Poisson disks. The ideal range is between 0.65 and 0.85, as suggested in [46].

Motivation. The dart throwing algorithm generates the Poisson disk distribution by repeatedly drawing a random point on the surface, checking the point's distance with existing samples, and accepting it if no violation is found. Due to its sequential nature and high computational cost, the brute force algorithm is not practical for interactive applications involving large-scale models. Wei [86] pioneered a parallel dart throwing algorithm that starts by partitioning the domain into grid cells such that the diagonal of each cell is $2r$. For each cell, Wei's algorithm makes up to k trials to draw a random point that satisfies the minimum distance requirement with existing samples in the neighboring cells. To achieve parallelism, the cells are organized into subsets such that all cells in a subset are separated by at least a distance of $2r$ from each other, and thus can be processed in parallel without causing conflicts. Although the sampling inside each group is random, the sequence of groups visited for every resolution level follows a predetermined order, which violates the uniform sampling condition.

Our algorithm is based on a different strategy. It is observed that samples have a uniform random position and a uniform random birth time in the sequential dart throwing algorithm. As time progresses, samples are accepted or rejected based on their distance from the samples that have already been born. We parallelize the classic dart throwing algorithm by assigning a *unique random* value to each sample, which represents the priority or order of the sample. The candidate samples can be processed by multiple threads simultaneously. Each thread checks the collision of the processed sample against its neighbors. When conflicts occur, the samples with the higher priority win and are accepted as Poisson disks.

Data structure. Our data structure is as follows:

```
enum SampleStatus{
    IDLE,          //not processed
    ACTIVE,        //being processed by a thread
    ACCEPTED,      //accepted as a Poisson disk
    REJECTED,      //rejected
};
```



```

struct Sample {
    SampleStatus status;

    //random number between 0 and 1
    float priority;

    //point coordinates
    vector3d pos;

    //the corresponding mesh triangle
    int face_id;
};

```

Each sample has a time-dependent status and a unique random number to represent its priority. During the run time, each thread takes one active sample. Let p_i be the active sample that is being processed by the i -th thread. To ensure that no two active samples have the same priority, the priority of p_i is given by

$$priority(p_i) = \frac{rand() * T + i}{RAND_MAX * T},$$

where T is the total number of threads, and the function `rand()` returns a pseudo-random integer number in the range 0 to `RAND_MAX-1`. Noted there is a summation of `rand()` and i in the denominator. Such i is used to avoid the same priority value for different p_i . The value of i could not determine the comparison result between two priority values, unless the function `rand()` generates the same pseudo-random number for two samples. However, such probability is extremely low ($1/RAND_MAX$) and the bias could be ignored. Given a high quality pseudo-random generator, $RAND_MAX \rightarrow \infty$, thus, such probability is zero, and $priority(p_i) = rand()/RAND_MAX$.

We followed Osada et al. [62]’s algorithm to generate random points with respect to the surface area, which are unbiased and insensitive to mesh tessellation. For each triangle, we computed its area and stored it in an array along with the cumulative areas of triangles visited so far. Then, we selected a triangle with probability proportional to its area by generating a random number between 0 and the total cumulative area and performing a binary search on the array of cumulative areas. For each selected triangle with vertices (v_1, v_2, v_3) , we constructed a random point $(1 - \sqrt{\alpha})v_1 + \sqrt{\alpha}(1 - \beta)v_2 + \sqrt{\alpha}\beta v_3$, where $\alpha, \beta \in [0, 1]$ are two random numbers.

Geodesic distance. To enforce the minimal distance constraints, we implemented the Improved Chen-Han (ICH) algorithm [93] on CUDA to compute the exact geodesic disks. The original ICH algorithm computes the “single-source all-destination” discrete geodesic on triangle mesh by maintaining the geodesic windows from near to far in a priority queue. It can be easily adapted to compute a geodesic disk with a user-specified radius. In our framework, each GPU thread initializes the CUDA-based ICH algorithm, which starts from the corresponding active sample, propagates the wavefronts and terminates when the geodesic distance exceeds $2r$. The memory cost of each ICH thread is linear to the number of mesh vertices in the geodesic disk of radius $2r$.

Resolving conflicts. For each active sample, the corresponding thread computes a local geodesic disk of radius $2r$. If there is an accepted Poisson disk inside the geodesic disk, the sample is rejected immediately. Otherwise, the algorithm recursively checks whether conflicts have occurred among the nearby active samples. If so, the one with the highest priority wins and will be marked as accepted and added in $\mathcal{S}$.

To facilitate our explanation, we first propose a simple algorithm of Parallel Dart Throwing on Surfaces, or PDTS, in Section 6.2.2. PDTS is an accurate algorithm in the sense that it generates randomly and uniformly distributed samples on surfaces at run time. However, PDTS is not efficient, since it throws a large number of darts, but only accepts a small percentage of them. Then we present an improved algorithm, Parallel Sampling from Dense Points, or PSDP, in Section 6.2.3. PSDP first converts the input mesh into a set of dense points $\mathcal{P}$, and then randomly selects the samples from $\mathcal{P}$ and processes them in a similar fashion as PDTS. Once a sample is accepted, all the samples within the geodesic disk of radius $2r$ are rejected. The algorithm terminates when all predefined points have been processed. PSDP is an accurate and efficient algorithm that works for arbitrary surfaces. We present PDTS only for explanation purposes, since the strategy of conflict resolving in PDTS is similar to that of PSDP. Note that all the experimental results reported in Section 6.4 are actually based on PSDP.

6.2.2 Parallel Dart Throwing on Surfaces (PDTS)

Starting from an empty set $\mathcal{S}$, our Parallel Dart Throwing on Surfaces (PDTS) algorithm iteratively updates the Poisson disk samples. For each iteration, PDTS randomly generates T

Algorithm Parallel Dart Throwing on Surfaces (PDTs)

```

global Mesh  $M$  //Triangle mesh
global list<Sample>  $\mathcal{S}[[F]]$  //  $\mathcal{S}[i]$  is the list of accepted samples at  $i$ -th triangle
global list<Sample>  $\mathcal{A}[T]$  //  $\mathcal{A}[i]$  is the list of active samples colliding with  $p_i$ 
global list<Sample>  $\mathcal{D}[T]$  //  $\mathcal{D}[i]$  is the list of accepted samples colliding with  $p_i$ 

function ParallelDartThrowing( $r, N$ )
    //  $r$ : Poisson disk radius
    //  $N$ : number of Poisson disk samples

     $\mathcal{S} \leftarrow \emptyset$ 
    while  $|\mathcal{S}| < N$  do
        parallel for each thread  $i$  from 1 to  $T$ 
             $p_i.pos \leftarrow$  random point
             $p_i.status \leftarrow ACTIVE$ 
             $p_i.priority \leftarrow \frac{rand()*T+i}{RAND.MAX*T}$ 
        parallel end for
        parallel for each thread  $i$  from 1 to  $T$ 
            // find the accepted and active samples in geodesic disk  $D(p_i, 2r)$ 
             $\{\mathcal{D}[i], \mathcal{A}[i]\} \leftarrow DetectCollision(p_i, 2r)$ 
            if  $\mathcal{D}[i] \neq \emptyset$ 
                 $p_i.status \leftarrow REJECTED$ 
                 $\mathcal{A}[i] \leftarrow \emptyset$ 
            end if
        parallel end for
        parallel for each thread  $i$  from 1 to  $T$ 
            CheckStatus( $p_i$ )
            if  $p_i.status == ACCEPTED$  then
                atomic  $\mathcal{S}[p_i.face\_id].samples.insert(p_i)$ 
            end if
        parallel end for
    end while
    return  $\mathcal{S}$ 

function CheckStatus( $p_i$ )
    if atomic  $p_i.status \neq ACTIVE$  then
        return
    end if
    for each  $q \in \mathcal{A}[i]$ 
        if  $q.priority > p_i.priority$  then
            CheckStatus( $q$ )
            if  $q.status == ACCEPTED$  then
                atomic  $p_i.status \leftarrow REJECTED$ 
            return
        end if
    end if
    atomic  $p_i.status \leftarrow ACCEPTED$ 
    return

function DetectCollision( $p_i, 2r$ )
    //Compute the geodesic disk centered at  $p_i$  using the ICH algorithm;
    //Collect the ACTIVE and ACCEPTED samples when wavefronts propagate;
    //Terminate the ICH algorithm when the geodesic disk radius exceeds  $2r$ ;
    //Return the collected ACTIVE and ACCEPTED samples.
    
```

samples on surfaces, where T is the number of threads. Each sample is marked as *active* and is given a unique random number for the priority. Let p_i denote the i -th active sample that is being processed by the i -th thread. For each p_i , the corresponding thread computes the geodesic disk $D(p_i, 2r)$ by the CUDA-based ICH algorithm. When the wavefronts propagate, the thread collects the active and accepted samples. The ICH algorithm terminates when the wavefront touches a point with geodesic distance of more than $2r$. If an accepted sample has been found inside this geodesic disk, the current active sample p_i is rejected immediately, since it conflicts with an existing Poisson disk. Otherwise, the thread recursively checks the conflicts among p_i and its nearby active samples. The one with the highest priority wins and is added to the Poisson disk sample set $\mathcal{S}$. It should be noted that all the threads proceed in parallel, and the accepted samples are independent of the order of the threads. Figure 6.2

Figure 6.2: Illustration of PDTS with five available threads. s_1 is an accepted Poisson disk sample. Each thread handles one active sample p_i , which has a unique and random priority value. PDTS accepts p_2 , p_4 and p_5 , and rejects p_1 and p_3 .

shows an example of PDTS. Assume that the current Poisson disk sample set $\mathcal{S}$ contains one accepted sample s_1 and there are 5 threads available. PDTS generates five active samples, p_i , $i = 1, \dots, 5$, at random locations, each of which has a unique and random priority value. All circles in Figure 6.2 are of the radius r . The i -th thread computes a geodesic disk $D(p_i, 2r)$ centered at p_i with radius $2r$ and collects the active samples inside $D(p_i, 2r)$. Let $\mathcal{A}[i]$ (resp. $\mathcal{D}[i]$) denote the list of active (resp. accepted) samples colliding with p_i . Then, $\mathcal{D}[1] = \{s_1\}$, $\mathcal{D}[2] = \dots = \mathcal{D}[5] = \mathcal{A}[1] = \emptyset$, $\mathcal{A}[2] = \{p_3\}$, $\mathcal{A}[3] = \{p_2, p_4, p_5\}$, $\mathcal{A}[4] = \{p_3\}$, $\mathcal{A}[5] = \{p_3\}$. The sample p_1 is immediately rejected since it collides with an accepted sample s_1 , i.e., $\mathcal{D}[1] \neq \emptyset$. Both p_2 and p_5 are accepted, because they don't have conflicted neighbors with higher prior-

ity. To determine p_3 's status, the third thread invokes $\text{CheckStatus}(p_3)$ and checks all elements in $\mathcal{A}[3]$ which have higher priority than p_3 . Thus, it will recursively call $\text{CheckStatus}(p_2)$ or $\text{CheckStatus}(p_5)$, and accepts p_2 or p_5 , depending on which element appears earlier in $\mathcal{A}[3]$. For either case, p_3 is rejected. To determine p_4 's status, the fourth thread invokes $\text{CheckStatus}(p_4)$, which will recursively call $\text{CheckStatus}(p_3)$, since p_3 has higher priority than p_4 . However, since p_3 is rejected, the fourth thread accepts p_4 .

Here we should mention that the final accepted samples are independent of the order of the threads. To illustrate this, let's consider the third and fourth threads. Depending on their order, there are three cases:

- Case 1: thread 3 is executed before thread 4. Thread 3 sets p_3 's status to *REJECTED*. Then thread 4 invokes $\text{CheckStatus}(p_4)$, which will call $\text{CheckStatus}(p_3)$, since $p_3 \in \mathcal{A}[4]$ has higher priority than p_4 . As p_3 has already been rejected, $\text{CheckStatus}(p_4)$ set $p_4.\text{status}$ to *ACCEPTED*.
- Case 2: thread 4 is executed before thread 3. Thread 4 invokes $\text{CheckStatus}(p_4)$, which will call $\text{CheckStatus}(p_3)$, since p_3 has higher priority than p_4 . The function $\text{CheckStatus}(p_3)$ will reject p_3 , since its priority is lower than that of p_2 and p_5 . Thus, $\text{CheckStatus}(p_4)$ sets $p_4.\text{status}$ to *ACCEPTED*. Later, when the third thread calls $\text{CheckStatus}(p_3)$ again, the function simply returns, since p_3 is not active any more.
- Case 3: thread 3 and thread 4 are executed simultaneously. Thread 3 invokes $\text{CheckStatus}(p_3)$ directly and thread 4 invokes $\text{CheckStatus}(p_4)$ which in turns calls $\text{CheckStatus}(p_3)$. Both threads reject p_3 , i.e., $p_3.\text{status}$ is set to *REJECTED* twice¹. Then p_4 is accepted.

Thus, even though each thread may be executed at different time frames, the accepted samples are totally determined by their priority values and independent of the order of the threads.

6.2.3 Parallel Sampling from Dense Points (PSDP)

PDTS randomly throws darts on surface in parallel, and abandons the darts that are within $2r$ distance of existing darts. This is not efficient for two reasons: first, PDTS has to compute

¹The read/write operation of a sample's status is atomic.

the geodesic disk for *every* sample; second, a large number of trials does not contribute to the results.

To improve the performance of PDTs, we adopt a different strategy for parallelizing the sampling process in PSDP: given a mesh model M , PSDP converts it to dense point clouds by randomly and uniformly throwing a large number of darts on the surface. Each generated point is initialized as an idle sample. For each iteration in the run time, PSDP randomly picks T idle samples and updates their status to ACTIVE. Similar to PDTs, each thread takes one active sample, say p_i , computes a geodesic disk $D(p_i, 2r)$ and finds the active and idle samples inside it. The status of p_i is determined by `CheckStatus()`, which recursively checks the conflicts among the *active* samples inside $D(p_i, 2r)$. If p_i is accepted, then all the idle samples inside $D(p_i, 2r)$ are rejected. PSDP repeats the iterations until all idle points have been processed. PSDP is more efficient than PDTs, since each accepted sample will result in the removal of all idle samples within geodesic distance $2r$. Therefore, no active sample conflicts with the accepted samples in PSDP (see Figure 6.3).

As shown in Figure 6.4, the entry in the array `PointIndex` is a pointer to the dense point clouds $\mathcal{P}$. To ensure that the accepted samples are randomly and uniformly distributed on M , we randomly shuffle the array `PointIndex`, and then partition `PointIndex` into T equally sized groups, which allows parallel processing by T threads. It should be noted that that our partition is *not* spatial dependent in the sense that the points in each group are not spatially related, i.e., they are located randomly on M . This distinguishes our algorithm from the existing parallel sampling algorithms [86, 9], which partition the domain according to the spatial information.

6.2.4 Correctness

We show that our algorithms, Parallel Dart Throwing on Surfaces (PDTs) and Parallel Sampling from Dense Points (PSDP), are accurate, i.e., the generated samples are randomly and uniformly distributed in the domain.

Given a sample domain $\mathcal{D}$, Poisson Disk sampling is a set $X = \{x_i \in \mathcal{D}; i = 1, 2, \dots, N\}$ of N samples having the properties:

$$\forall x_i \in X, \forall M \subseteq \mathcal{D} : P(x_i \in M) = \int_M dx, \quad (\text{Eq. 6.1})$$

Algorithm Parallel Sampling from Dense Points (PSDP)

```

global Mesh  $M$  //Triangle mesh
global list<Sample>  $\mathcal{S}[[F]]$  //  $\mathcal{S}[i]$  is the list of accepted samples at  $i$ -th triangle
global vector<Sample>  $\mathcal{P}$  //Dense point clouds of  $M$ 
global vector<int> PointIndex //The point indices
global list<Sample>  $\mathcal{A}[T]$  //  $\mathcal{A}[i]$  is the list of active samples colliding with  $p_i$ 
global list<Sample>  $\mathcal{I}[T]$  //  $\mathcal{I}[i]$  is the list of idle samples colliding with  $p_i$ 

function ParallelSamplingFromDensePoints( $r, N, L$ )
    //  $r$ : Poisson disk radius
    //  $N$ : number of Poisson disks
    //  $L$ : number of points in  $|\mathcal{P}|$ 

     $\mathcal{P} \leftarrow \text{GenerateDensePoints}(L)$ 
    parallel PointIndex  $\leftarrow 1 \dots L$ 
    parallel Randomly shuffle PointIndex
    // partition the PointIndex into  $T$  groups, i.e.,  $group_i, i = 1, \dots, T$ 
    for each  $i$  from 1 to  $T$ 
         $group_i.range \leftarrow (\frac{i-1}{T}L, \frac{i}{T}L]$ 
        // the index of the next IDLE sample in  $group_i$ 
         $group_i.curr\_index \leftarrow \frac{i-1}{T}L + 1$ 
    end for
     $S \leftarrow \emptyset$ 

    repeat
        parallel for each thread  $i$  from 1 to  $T$ 
            while  $\mathcal{P}[\text{PointIndex}[group_i.curr\_index]].status \neq IDLE$  do
                 $group_i.curr\_index \leftarrow group_i.curr\_index + 1$ 
                Terminate this thread if  $group_i.curr\_index$  exceeds its range
            end while
             $p_i \leftarrow \mathcal{P}[\text{PointIndex}[group_i.curr\_index]]$ 
             $p_i.status \leftarrow ACTIVE$ 
             $p_i.priority \leftarrow \frac{rand()*T+i}{RAND\_MAX*T}$ 
        parallel end for
        parallel for each thread  $i$  from 1 to  $T$ 
            //Find the active and idle samples colliding with  $p_i$ 
             $\{\mathcal{A}[i], \mathcal{I}[i]\} \leftarrow \text{DetectCollision}(p_i, 2r)$ 
        parallel end for
        parallel for each thread  $i$  from 1 to  $T$ 
            //Check conflicts among the active samples colliding with  $p_i$ 
            CheckStatus( $p_i$ )
            if  $p_i.status == ACCEPTED$ 
                atomic  $\mathcal{S}[p_i.face\_id].insert(p_i)$ 
                for each  $q \in \mathcal{I}[i]$ 
                    atomic  $q.status \leftarrow REJECTED$ 
                end for
            end if
        parallel end for
    until no more IDLE points
    return  $S$ 

function GenerateDensePoints( $L$ )
    parallel for each  $i$  from 1 to  $L$ 
         $p_i.pos \leftarrow \text{random point on } M$ 
         $p_i.status \leftarrow IDLE$ 
         $\mathcal{P}[i] \leftarrow p_i$ 
        Insert  $p_i$  into the sample list of the corresponding triangle
    parallel end for
    return
    
```

Figure 6.3: Comparison between PDTs (left) and PSDP (right). Assume s_1 is an accepted sample. In PDTs, each thread generates one active sample at a random location. Thus, it is likely that the active samples collide with the accepted samples. For example, p_1 and p_2 are rejected due to the collision with s_1 , and p_3 is accepted since it does not conflict with any accepted and active samples. Although being parallel, this strategy is not efficient, since two geodesic disks must be computed for p_1 and p_2 respectively. However, p_1 and p_2 do not contribute to the results. By contrast, PSDP randomly picks samples from the predefined dense points $\mathcal{P}$. When s_1 is accepted, PSDP rejects all the idle samples (in yellow) inside $D(s_1, 2r)$. Therefore, in the future iterations, PSDP will pick only the samples (in green) that are outside $D(s_1, 2r)$. In other words, no active samples collide with the accepted samples in PSDP.

Figure 6.4: Data structure of PSDP. The array $\mathcal{P}$ contains the dense point clouds, each point has a random location on M . To allow the ICH algorithm efficiently access the samples during the wavefront propagation, we also insert each sample p_i into the triangle containing p_i . The array PointIndex is a random permutation of $\{1, \dots, |\mathcal{P}|\}$. We partition PointIndex into T equally sized groups, $group_i, i = 1, \dots, T$.

$$\forall x_i, x_j \in X : \|x_i - x_j\| \geq 2r, \quad (\text{Eq. 6.2})$$

where $P(\cdot)$ is a conditional probability.

The first condition states that a uniformly distributed random sample x_i of X has a probability of falling inside a subset M of $\mathcal{D}$ that is equal to the hypervolume of M . The second condition requires that any two samples are $2r$ apart from each other.

An accurate Poisson disk sampling algorithm must satisfy both conditions. Clearly, our approaches satisfy the second condition, since exact geodesic distances are used. Next, we will show the first condition also holds in PDTs and PSDP.

Let $\mathcal{S} = \{s_1, \dots, s_k\}$ be a set containing k Poisson disk samples generated by the brute-force dart throwing algorithm. Let $p_i, i = 1, \dots, T$, be points that are uniformly and randomly distributed on D . If we randomly shuffle the indices $\{1, \dots, T\}$, and then add the shuffled T points *sequentially* into $\mathcal{S}$ by checking the minimal distance constraint, the resultant set is obviously a Poisson disk set, since the process is the same as a sequential dart throwing. Observe that each trial has a unique order, which is the index of the point.

In PDTs, we assign each sample p_i a *unique* and *random* priority value, which is equivalent to randomly shuffle the indices $\{1, \dots, T\}$. Therefore, PDTs can make T trials *simultaneously*, and the resultant set $\mathcal{S}$ contains uniformly and randomly distributed Poisson disks.

In PSDP, we construct a dense point clouds $\mathcal{P}$, such that the points are randomly and uniformly distributed on $\mathcal{D}$. If we randomly choose T points $p_i \in \mathcal{P}, i = 1, \dots, T$, and make T trials simultaneously, this is equivalent to PDTs algorithm.

Note that PSDP randomly shuffles the point indices $\{1, \dots, |\mathcal{P}|\}$ and then partitions them into T equally-sized groups. During the run time, each thread picks one index from each group. Thus, we only need to show this selection is unbiased.

Let g_i be the number of *available* idle samples in i -th group, and $I = \sum_{i=1}^T g_i$ be the total number of *available* idle samples in all groups.

Without the group partition, the probability for any idle sample being selected is $1/I$. With the group partition, the probability for an arbitrary idle sample q being located in the i -th group is g_i/I . Also, the probability for q being selected by the i -th thread is $1/g_i$. Therefore, the

probability for q being selected is also equal to $1/I$, which implies that the partition based selection is unbiased.

Thus, PSDP is also an accurate algorithm that distributes Poisson disks randomly and uniformly on $\mathcal{D}$.

6.3 Improving the Maximal Property of Poisson Disk Sampling in $\mathbb{R}^2$ and $\mathbb{R}^3$

Our PSDP algorithm parallelizes the dart throwing by assigning each sample candidate a random and unique priority that is unbiased with regard to the distribution. Hence, multiple threads can process the candidates simultaneously and resolve conflicts by checking the given priority values. However, this algorithm is not maximal, i.e., when the algorithm terminate, there is still room for additional samples. Recently, Ebeida et al. [29] presented a parallel algorithm to generate maximal Poisson disk sampling in $\mathbb{R}^d$. Their idea is to maintain a flat quadtree structure to keep track of the uncovered region. Their algorithm is theoretical sound and also practical for dimension up to 3 in GPU. However, compared to the other parallel algorithms, Ebeida et al's approach requires large memory due to maintaining all the active cells. This diminishes their application to large scale application due to the limited memory on modern graphics card.

This section presents a simple and yet effective technique to compute maximal Poisson disk sampling in $\mathbb{R}^2$ and $\mathbb{R}^3$. Taking a non-maximal Poisson disk sampling as input, our algorithm efficiently detects the regions that allow additional samples and then generates Poisson disks in these regions. The key idea is to convert the complicated plane or space searching problem into a simple searching on circles or spheres, which is one dimensional lower than the original sampling domain.

6.3.1 A Geometric Problem

Before introducing our algorithm, we discuss an interesting geometric problem, which is closely related to maximal Poisson disk sampling.

Figure 6.5: An “forbidden” arc $\widehat{p_1 p_2}$ (in red) with two ending angles $[\alpha, \beta]$.

Problem statement. Consider a set of circles $C_i = (c_i, r_i)$ in $\mathbb{R}^2$ (resp. spheres S_i in $\mathbb{R}^3$) covering a finite domain $\mathcal{D} \subset \mathbb{R}^2$ (resp. $\mathbb{R}^3$), where c_i and r_i are the center and radius respectively. Any two circles (resp. spheres) do not intersect. Can we insert a circle (resp. sphere) of radius r into $\mathcal{D}$ without intersecting the existing circles (resp. spheres)

Solution in 2D. For each circle $C_i = (c_i, r_i)$, we draw a new circle $C'_i = (c_i, r_i + r)$ with radius $r_i + r$. Note that these new circles C'_i may intersect due to the increased radius.

Now consider two circles C'_1 and C'_2 intersecting at p_1 and p_2 (see Figure 6.5). Clearly, we cannot insert additional circle of radius r in the overlapped region between C'_1 and C'_2 , since such circle will intersect C_1 or C_2 . Observe that the red arc $\widehat{p_1 p_2}$ of circle C'_1 is completely inside circle C'_2 . We call this arc “forbidden” since no circles of radius r can be inserted between C_1 and C_2 . Other circles C'_i can also generate forbidden arcs on C'_1 if they intersect C'_1 . Then we define the “permitted” arc(s) on C'_1 as the complement of all forbidden arcs.

Thus, in order to find the room to insert additional circle near C_1 , it is equivalent to test whether the circle C'_1 has at least one permitted arc.

Take an arbitrary unit vector $\vec{u}$ as the reference axis for circle C'_1 . Then we compute the angles $\alpha = \text{angle between } \vec{u} \text{ and } \overrightarrow{c_1 p_1}$ and $\beta = \text{angle between } \vec{u} \text{ and } \overrightarrow{c_1 p_2}$. The forbidden arc $\widehat{p_1 p_2}$ is given by (α, β) if the reference axis $\vec{u}$ does not intersect the arc. Otherwise, we split it into two small arcs, i.e., $(-\epsilon, \beta)$ and $(\alpha, 2\pi + \epsilon)$, where $\epsilon = 1 \times 10^{-8}$ is a small value. Figure 6.6 shows a special case where multiple circles intersect at one point p . In such situation, a circle of radius r can be inserted at p , thus, the point p is a degenerate but permitted arc. The detailed algorithm to find all permitted arcs is documented in Algorithm 1.

Figure 6.6: When multiple circles C'_i intersect at one point p , such point p is a degenerate but “permitted” arc, which allows insertion a circle with radius r .

Solution in 3D. The 3D problem can be solved by using the above 2D results. For each sphere $S_i = (c_i, r_i)$, we make a new sphere $S'_i = (c_i, r_i + r)$. Due to the increased radius, the spheres may intersect each other. Similar to 2D case, if two spheres S'_i and S'_j intersect, we cannot insert a sphere with radius r in between S_i and S_j .

Observe that the intersection of two spheres is a circle, which bounds a disc on each sphere. Similar to 2D case, we call such disc forbidden, since it does not allow inserting additional sphere. Also, the complement of all forbidden discs is called “permitted”. Note that the boundary of the permitted region is a set of permitted arcs. Thus, in order to find the room to insert additional sphere near S_i , it is equivalent to test whether the permitted region on S'_i is non-empty.

This permitted region searching problem, on the other hand, can be formulated as finding permitted arcs. As shown in Figure 6.7, let $C_1 = S'_i \cap S'_1$ and $C_2 = S'_i \cap S'_2$ be two intersecting circles on S'_i . And C_1 and C_2 also intersect at points p_1, p_2 . Thus, we can use the permitted arcs technique to find the permitted region, i.e., a region that is not enclosed by any circles. Therefore, by searching all permitted regions on all spheres, we can find the available spaces for inserting new spheres.

Algorithm 5: FindPermittedArcsForCircle(Circle C'_i)

```
//Step 1. Compute forbidden arcs on  $C'_i$ 
forbiddenArcListi  $\leftarrow \emptyset$ 
 $\{\hat{C}'\} \leftarrow$  all circles intersecting with  $C'_i$ 
foreach  $\hat{C}'_j$  in  $\{\hat{C}'\}$  do
    //compute intersection between  $C'_i$  and  $\hat{C}'_j$ 
    forbiddenArcj  $\leftarrow$  intersecting arc on  $C'_i$ 
    Insert forbiddenArcj to forbiddenArcListi
end for
//Step 2. Compute permitted arcs.
//In 2D plane, there are at most 18 circles intersecting with  $C'_i$ . So this sort runs very fast.
Sort forbiddenArcListi by  $\alpha$  value with increasing order
 $\sigma \leftarrow 0$ 
permittedArcListi  $\leftarrow \emptyset$ 
foreach arc in forbiddenArcListi do
    // $\sigma = \text{arc}.\alpha$  occurs while multiple circles intersect at one point.
    if  $\sigma \leq \text{arc}.\alpha$  then
        newArc  $\leftarrow (\sigma, \text{arc}.\alpha)$ 
        Insert newArc to permittedArcListi
    end if
     $\sigma = \max(\sigma, \text{arc}.\beta)$ 
end for
if  $\sigma \leq 2\pi$  then
    newArc  $\leftarrow (\sigma, 2\pi)$ 
    Insert newArc to permittedArcListi
end if
Return permittedArcListi
```

Figure 6.7: (a) Several spheres intersect. (b) Sphere S' intersects another two spheres, resulting two intersecting c_1 and c_2 , respectively.

6.3.2 Algorithm

Our algorithm takes a non-maximal Poisson disk sampling as input, which is generated by PSDP. Given the non-maximal Poisson disks, our goal is to search the domain $\mathcal{D}$ and then find the regions which are big enough to contain additional samples. This is exactly the problem we mentioned in the previous subsection, which can be solved by searching on circles or spheres, which are one dimensional less than the sampling domain $\mathcal{D}$. It is also worth noting that such searching operation on each circle or sphere does not have data or control dependency on others, since all circles or spheres are already given. As a result, finding permitted arcs or regions can be implemented in a fully parallel manner.

Once the permitted arcs or regions are found, we randomly create a Poisson disk candidate for *each* permitted arc. Similar to our PDTs or PSDP, each candidate is assigned a random and unique priority that is unbiased with regard to the distribution. Hence, multiple threads can process the candidates simultaneously and resolve conflicts by checking the given priority values. The candidates with the higher priority win and are accepted as Poisson disks.

After inserting new Poisson disks, we update the permitted arcs or regions and then repeat this procedure until the permitted arcs or regions are empty. Figure 6.8 shows the pipeline of our algorithm in $\mathbb{R}^2$.

Algorithm 6: FindPermittedArcsForSphere(Sphere S'_i)

```

//Step 1. Compute forbidden discs on  $S'_i$ 
forbiddenDisci  $\leftarrow \emptyset$ 
 $\{\hat{C}'\} \leftarrow$  all spheres intersecting with  $S'_i$ 
foreach  $\hat{S}'_j$  in  $\{\hat{S}'\}$  do
    //Compute intersecting disc between  $S'_i$  and  $\hat{S}'_j$ 
    forbiddenDisci  $\leftarrow$  intersecting disc on  $S'_i$ 
end for
//Step 2. Compute permitted arcs on discs via FindPermittedArcsForCircle()
permittedArcListi  $\leftarrow \emptyset$ 
foreach disc in forbiddenDisci do
    //Compute intersection between disc and other discs on  $S'_i$ 
    //And find all permitted arcs on disc.
    arcList  $\leftarrow$  FindPermittedArcsForCircle(disc);
    Merge arcList to permittedArcListi
end for
Return permittedArcListi

```

Algorithm 7: Parallel and Maximal Poisson Disk Sampling

```

compute a non-maximal Poisson disk sampling by using the parallel algorithm PSDP
parallel: foreach  $x_i$  do
    generate the circle  $C'_i = (x_i, 2r)$ 
    //resp. sphere  $S'_i = (x_i, 2r)$ 
    permittedArcListi  $\leftarrow$ 
        FindPermittedArcsForCircle( $C'_i$ )
    //resp., FindPermittedArcsForSphere( $S'_i$ )
end for
repeat //Generate new points
    parallel: foreach arcj in permittedArcListi do
         $p_j \leftarrow$  a new sample point on arcj
         $p_j.\text{priority} \leftarrow$  a random and unique value
    end for
    parallel: for all  $p_j$  do
        Resolving conflicts among  $\{p_j\}$ 
        Output all accepted points as  $\{q\}$ 
    end for
    parallel: foreach accepted point  $q_i$  in  $\{q\}$  do
        Update the permitted arcs for all neighbor points.
    end for
until all permittedArcListi are empty

```

Figure 6.8: Maximal Poisson disk sampling in $\mathbb{R}^2$. (a) Our algorithm takes a non-maximal Poisson disk sampling as input. (b) For each sample, we draw a circle with radius $2r$ and then compute the permitted arcs in parallel. (c) The permitted arcs enclose the regions which can insert additional samples. (d) We randomly create a Poisson disk candidate for *each* permitted arc. Each candidate is assigned a random and unique priority that is unbiased with regard to the distribution. Hence, multiple threads can process the candidates simultaneously and resolve conflicts by checking the given priority values. The candidates with the higher priority win and are accepted as Poisson disks.

6.4 Experimental Results

Quality. Following [9], we calculated the radius statistics $\rho = r\sqrt{2\sqrt{3}N/A}$ to measure the packing density [46]. In all cases, our ρ values fall between $0.70 - 0.76$, which is within the ideal range of $0.65 - 0.85$ recommended by [46]. We observed that the radius statistics ρ is closely related to the number of points in $|\mathcal{P}|$. Intuitively speaking, $|\mathcal{P}|$ can be considered as the total number of thrown darts. The more darts that are thrown, the larger the area covered by the accepted darts. However, ρ remains almost constant when the ratio $|\mathcal{P}|/|\mathcal{S}|$ is above 50. As the sampling speed is inversely proportional to the ratio $|\mathcal{P}|/|\mathcal{S}|$, we usually set $|\mathcal{P}|/|\mathcal{S}| = 20$ in our experiments for the tradeoff between quality and performance.

Figure 6.9: Experimental results. For each model we generate three density levels, at approximately 1K, 5K and 10K sample points. The images are generated at high resolution to allow for zoom-in examination.

Using our geodesic program, we can easily visualize the Poisson disks on surfaces (see Figure 6.14 and 6.9), which allows us to visually check the sampling quality. We compared our method with the state-of-the-art technique [9]. Our algorithm measures the exact distance between two samples, while Bowers et al. [9] approximates the geodesic distance by a smooth curve interpolating the normals of two endpoints. They derived a closed form formula that produces exact geodesic distances for the plane and sphere, which works well for smooth surfaces, such as Bunny. However, for surfaces with rich details, such as Gargoyle and Lion, their

approximation has larger errors, such that some samples violate the minimal distance constraints in the generated distribution (see the highlighted boxes in Figure 6.14). Our method, in contrast, is guaranteed to generate conflict-free Poisson disk distributions.

Spectrum analysis. We applied the spectrum analysis method in [9] to verify our method. As required in [9], we tessellated each model to $|V| = 100,000$ vertices and then computed the first 10,000 eigenvectors of the Laplacian matrix by using the spectral shift method [83]. For comparison, we generated 300 – 500 samples on surfaces by using our method, [9] and the brute-force dart throwing, which is considered as the ground truth. We then evaluated the radial means and anisotropy² of the samples power spectrum averaged over 10 runs. Our algorithm is accurate in that it distributes Poisson disks randomly and uniformly on surfaces. As expected, the plots of both the radial mean and anisotropy of our method are highly consistent with the brute-force dart throwing in all examples (see Figure 6.4). Bowers et al. [9] embed the surface into $\mathbb{R}^3$ and partitions the domain into grid cells of size $\frac{2r}{\sqrt{3}}$, which are organized into phase groups such that all cells in a phase group are separated by at least a distance of $2r$ from each other. This allows sampling in parallel without causing conflicts. Although the sampling inside each group is random, the sequence of groups visited follows a predetermined order. Therefore, the generated distribution violates the uniform sampling condition because samples within a group cannot be placed until all previous groups have been sampled. We tested our method and the phase group method [9] by using the exact geodesic [93] and the approximate geodesic used in [9]. As shown in Figure 6.4, the spectra of our method are consistent with the blue noise patterns in all examples. In contrast, the anisotropy plots of phase group methods (with and without the exact geodesics), deviate from the ground truth, which implies that such deviation in the anisotropy of the phase group method does *not* come from the distance metric used.

Performance. We implemented and tested our PSDP algorithm it on a PC with an Intel Xeon 2.66GHz CPU and 12GB memory. The graphics card is an NVIDIA GTX 580 with 512 cores and 1.5GB memory. Our program is compiled using CUDA 4.0 RC2, which supports recursion for device function.

²As in [9], the generated anisotropy plots are centered around the -7.0dB line instead of the -10dB line, since the basis functions in manifold harmonics transformation are the real forms of the analytic Fourier basis, which causes a factor 2 of the variance of the power spectrum.

Figure 6.10: We applied the manifold harmonics transformation [9] for spectral analysis on surfaces. As suggested in [9], we generated 300-500 samples on the models and used 10 runs for each test. The horizontal axis is the basis frequency, the vertical axes are the radial mean and the anisotropy. We tested both phase group method and our priority based parallelization with approximate and exact geodesic respectively. We also implemented the traditional dart throwing as ground truth (in green). The spectra of our method are consistent with the blue noise patterns in all examples, while the anisotropy plots of phase group methods (with the exact geodesic [93] and the approximate geodesic [9]) deviate from the ground truth. This implies that such bias of the phase group method does *not* come from the distance metric used.

Figure 6.11: For the samples near the self-intersection, the extrinsic approach [9] may report “false” conflicts between valid samples since their spatial distance is small. Our method can accurately detect the conflicts using the intrinsic geodesic distance, thus, resulting in the distribution with more valid samples.

	# of calls for CheckStatus()									
N/T	1	2	3	4	5	6	7	8	9	≥ 10
5	75.94%	21.45%	2.16%	0.15%	0.13%	0.09%	0.03%	0.02%	0.02%	0.01%
10	84.27%	14.25%	0.81%	0.52%	0.07%	0.04%	0.02%	0.01%	0.01%	0%
20	91.39%	8.30%	0.15%	0.14%	0.02%	0.01%	0%	0%	0%	0%
40	95.32%	4.58%	0.06%	0.04%	0%	0%	0%	0%	0%	0%

Table 6.1: Statistics of the number of calls for CheckStatus() on Gargoyle model. Columns show the percentage of the number of function calls.

 (a) Sampling speed vs number of samples N ($|\mathcal{P}|/|\mathcal{S}| = 20$)

 (b) Sampling speed vs $|\mathcal{P}|/|\mathcal{S}|$ ($N = 50K$)

 (c) Radius statistics vs $|\mathcal{P}|/|\mathcal{S}|$ ($N = 50K$)

Figure 6.12: Performance of our algorithm.

Figure 6.13: Our method is robust and insensitive to the mesh resolution and tessellation. From left to right: distributing 700 Poisson disks on Bunny models with 1.4K, 14K and 144K triangles respectively. Our algorithm works well for all cases.

We observed that the sampling speed depends on the number of Poisson disk samples N (see Figure 6.12 (a)). Among all the functions in PSDP, `DetectCollision()` is computationally expensive, since it computes a geodesic disk of radius $2r$ using the ICH algorithm [93] with time complexity $O(n^2 \log n)$, where n is the number of mesh vertices in the geodesic disk. When N is small, the Poisson disk radius r is big, thus, it takes a longer time for `DetectCollision()` to compute the geodesic disk. Conversely, it takes less time to compute the geodesic disk when N is big and r is small. Thus, the sampling speed increases when N is increasing. However, when N is comparable or exceeds the number of mesh vertices $|V|$, the radius of the geodesic disk is similar to or less than the average edge length. As a result, the time for computing the geodesic disk is insignificant and remains roughly constant. Hence the sampling speed does not depend on N anymore.

We also observed that the sampling speed is closely related to the ratio $|\mathcal{P}|/|\mathcal{S}|$ (see Figure 6.12 (b)), since the greater number of predefined points in $\mathcal{P}$, the greater number of idle samples collided; thus, the more time taken for function `DetectCollision()` to collect the idle samples.

Efficiency of our conflicts resolving strategy. Function `CheckStatus()` plays a key role in our parallel Poisson disk sampling algorithm, since it converts the sequential dart throwing process into a parallel collision detection. Each thread will invoke `CheckStatus()` for the corresponding active sample, say p_i . If there is an accepted sample inside its geodesic disk $D(p_i, 2r)$, p_i is rejected immediately. Otherwise, `CheckStatus()` recursively checks whether conflicts have occurred among its active neighbors. The recursive function stops when a winner is identified.

Consider the worst case that the priorities of all samples are in ascending order, i.e., $p_i.\text{priority} < p_{i+1}.\text{priority}$ and p_i conflicts with p_{i+1} , $i = 1, \dots, T-1$. Then the i -th thread invokes `CheckStatus( $p_i$ )`, which will recursively call `CheckStatus( $p_{i+1}$ )`, $\dots$, until `CheckStatus( $p_T$ )` to identify the winner p_T . Therefore, the number of calls of `CheckStatus()` for each thread is of complexity $O(T)$. However, the probability of such a scenario is extremely low in practice. In fact, the number of calls of `CheckStatus()` depends on the number of samples N and the number of active samples T . The bigger value of N , the smaller radius r , the smaller probability of conflicts among active samples. Let c be the average number of calls of `CheckStatus()` for each thread. Our statistical results show that $c = 1.09$ when $N/T = 20$, and more than 91% function

calls returns immediately, since there are no active samples collided (see Table 6.1). We observed the value of c remains roughly the same for all models, since PSDP converts all models to dense point clouds before sampling, thus the performance is nearly uniform regardless of the mesh complexity.

Figure 6.14: Sampling quality. We generate samples by using the state-of-the-art technique [9] and our method. To visualize the results, we compute the geodesic disks centered at each sample with radius r . Due to the approximate geodesic distance in [9], some samples violate the minimal distance constraints, as highlighted in the black boxes. Our method measures the exact geodesic distance, thus, can guarantee that the generated distribution is free of conflicts.

Intrinsic algorithm. Our algorithm is intrinsic in the sense that it depends on the metric rather than the surface embedding, since all distances are measured intrinsically by the exact geodesic algorithm [93]. In contrast, [9] is an extrinsic approach, since it partitions the ambient space $\mathbb{R}^3$ to form the phase groups and approximates the geodesic distance by computing a curve that smoothly interpolates the normals of two end points. Furthermore, our partition method is purely based on the surface area and does not require each partitioned group to be connected, which is fundamentally different from the space or surface segmentation strategies used in other surface sampling algorithms [9][33]. Therefore, our algorithm works for surfaces in the arbitrary dimension. Figure 6.15 shows examples of intrinsic sampling on surfaces in $\mathbb{R}^4$.

Figure 6.15: Our algorithm is independent of the surface embedding. Row 1: two examples of Poisson disk sampling on 4D surfaces. For rendering purpose, the surfaces are projected into $\mathbb{R}^3$; Row 2: we distribute Poisson disks on one pose and the shape correspondence induces the sampling on the other pose. The intrinsic property of our algorithm guarantees the induced Poisson disk distribution is uniformly distributed and free of conflicts.

Figure 6.15 (row 2) also shows two poses of a lion model, which have the same mesh connectivity and are nearly isometric. As our method is intrinsic to the geometry, we can distribute Poisson disks on one pose and the shape correspondence naturally induces the sampling on the other pose, which is guaranteed to be valid, i.e., uniformly distributed and free of conflicts.

Robustness. Our method converts the input model into a dense point clouds by [62], which randomly generates points with respect to surface area, i.e., the number of points on a mesh triangle is proportional to its area. Moreover, the exact geodesic distance algorithm [93] is intrinsic to the shape geometry and insensitive to mesh resolution and tessellation, so is our algorithm. See Figure 6.13 for an example of sampling on Bunny with varying resolutions.

Figure 6.16: Adaptive sampling on Caesar model. Left: the spatially varying function specified by the user, where red means dense sampling (small radius) and green means sparse sampling (large radius); Right: the adaptive sampling result.

Adaptive sampling. Our algorithm can be easily extended for adaptive sampling, where the Poisson disk radius r is guided by a spatially varying function $g : V \rightarrow [0, 1]$ on the surface, where 0 means dense sampling (small radius) and 1 means sparse sampling (large radius). Let r_{max} be maximal radius of a Poisson disk, specified by the user. Therefore, the Poisson disk centered at p has radius $g(p)r_{max}$. Two samples $p, q \in M$ conflict if their geodesic distance $d(p, q) < (g(p) + g(q))r_{max}$. For each active sample p_i , we compute a geodesic disk $D(p_i, (g(p_i) + 1)r_{max})$ centered at p_i and with radius $(g(p_i) + 1)r_{max}$. Note that this radius is conservative to ensure all the conflicted samples are included. Then for each active/idle sample

$q \in D(p_i, (g(p_i) + 1)r_{max})$, we detect the collision by checking $d(p_i, q) < (g(p_i) + g(q))r_{max}$. Figure 6.16 shows the adaptive sampling guided by user-specified radius function.

Figure 6.17: Our algorithm also works for Euclidean space of arbitrary dimensions. (a) 2D case, $r = 0.0147$, # samples = 787. (b) 3D case, $r = 0.023$, # samples = 6064. (c) Cut view of (b). (d)-(f) The average power spectrum for a 2D case with $r = 0.00268$ and 21886 samples, the radial power spectrum and the anisotropy spectrum. The spectra are consistent with the blue noise patterns, since our algorithm is accurate. Our implementation on NVIDIA GTX 580 generates 13 million 2D samples/second and 1.5 million 3D samples/second.

Sampling in $\mathbb{R}^n$. Our algorithm is general in that it works for both arbitrary surfaces and Euclidean space of arbitrary dimensions. Given an unit hypercube H in $\mathbb{R}^n$, we maintain a grid structure of hypercells, each with side length $\frac{2r}{\sqrt{n}}$. Then we randomly generate points in each hypercell to make a dense point set $\mathcal{P}$. During the run time, for each active sample p_i , our algorithm detects the collision of p_i in its neighboring cells. Similar to [9], the predefined points $\mathcal{P}$ are stored in a hash table. The geodesic distance in function DetectCollision() is simply the Euclidean distance in $\mathbb{R}^n$. Figure 6.17 shows sampling examples in $\mathbb{R}^2$ and $\mathbb{R}^3$.

Maximal Property in $\mathbb{R}^2$ and $\mathbb{R}^3$. Our improvement algorithm is also fully parallel and highly efficient. The processing speed is up to 8 million 2D samples per second, and 0.55 million 3D samples per second (see Figure 6.12). Compared to the existing maximal Poisson disk sampling algorithms [34, 28], our algorithm is much more efficient due to its parallel nature.

Figure 6.18: Maximal Poisson disk sampling results. (a) 3691 samples in $\mathbb{R}^2$, $r = 0.00682$. (b) 3434 samples in $\mathbb{R}^3$, $r = 0.03$

Figure 6.9 shows our maximal sampling results in $\mathbb{R}^2$ and $\mathbb{R}^3$. Figure 6.20 shows the frequency spectrum, radial mean and anisotropy of a set of 22K samples across 10 trials. The spectrum is visually identical to the results of pure dart-throwing. The radial mean and anisotropy do not show visual bias either. This is because our program takes the result of PSDP as input and the same priority based sampling strategy is applied to the additional samples. Our method is also very flexible to handle the irregular 2D sampling domain (Figure 6.22). Figure 6.23 shows the maximal sampling in various 2D shapes.

We applied our PSDP algorithm to $\mathbb{R}^2$ and computed the radius statistics ρ to measure how far our distribution is from the maximal distribution. We applied our program with various number of the pre-generated dense points $|\mathcal{P}|$. For each configuration, we also run our program 10 times and obtained the average ρ . To obtain the ground truth, we applied the bias-free maximal sampling [29] to a unit square with $r = 0.005$ and run it for 10 times. We found the values of ρ fall in the range $[0.778, 0.780]$ with average 0.7791. And our improved maximal sampling has almost the same value of ρ as [29]. As shown in Figure 6.19, the value of our ρ of PSDP (the blue curve) tends to approach the ground truth (the red line) when increasing $|\mathcal{P}|$. This result is not a surprise, since the more number of pre-defined samples, the larger pool our algorithm can choose from, thus, more Poisson disks it can produce, and more closer the generated distribution to the maximal distribution.

The percentage of samples that are added on the arcs is

$$1 - \left(\frac{\rho_i}{\rho_m}\right)^2,$$

Figure 6.19: The distribution generated by PSDP method is not maximal. For the Poisson disk sampling on a unit square with $r = 0.005$, the radius statics ρ of the maximal sampling algorithm [29] is very stable, all values fall in the range $[0.778, 0.780]$ with an average 0.7791. Our ρ of PSDP depends on the dense-points-to-Poisson-disks ratio $|\mathcal{P}|/|\mathcal{S}|$.

, where ρ_i is the ρ of PSDP, and ρ_m is the ρ of maximal sampling. For $|\mathcal{P}|/|\mathcal{S}| = 20, 50, 100, 300$, such percentage is about 13.6%, 8.4%, 5.6%, 2.6%, respectively.

Applications. We demonstrate three applications of our algorithm. Figure 6.24 shows the vector field visualization, where the arrow textures are placed randomly and uniformly on surfaces with user-defined vector field. Figure 6.26 shows surfaces decorated with a mosaic of tiles, which are generated by the Voronoi diagram of the Poisson disk samples. Figure 6.25

Figure 6.20: Spectrum analysis. Radius=0.00268, about 22K samples. This is average spectrum over 10 runs.

Figure 6.21: Performance: our maximal sampling algorithm processes about 8 million 2D samples per second, and 0.55 million 3D samples per second. The horizontal axis shows the number of samples and the vertical axis shows the sampling speed. Tested on a Dell Workstation with Xeon 2.66GHz Quad Core CPU, 12GB memory and an Nvidia GTX580.

Figure 6.22: Our method is effective to deal with 2D irregular domain. For arbitrary 2D polygon, we can clip the circles C'_i and put the arcs outside of the domain in the *forbiddenArcList*. This can guarantee that the permitted arcs are always inside the domain.

Figure 6.23: Maximal Poisson disk sampling in 2D polygons.

shows stippling to represent the image with isotropic blue noise samples.

Figure 6.24: Vector field visualization.

Figure 6.25: Stippling by adaptive blue noise sampling.

Figure 6.26: Mosaic tile placement.

6.5 Comparison & Discussions

The existing algorithms are clearly divided into groups of surface and Euclidean space sampling, due to the difference between the segmentation of $\mathbb{R}^n$ and curved surfaces, which also results in different spatial data structures. Table 2 compares our method and the existing Poisson disk sampling algorithms. In the following, we elaborate the key difference between our technique and the existing approaches.

Method	Sampling Domain	Intrinsic	Parallel	Accurate	Maximal	Numerical solver	Spatial partition	Performance
Dart throwing	$\mathbb{R}^n$	-	No	Yes	No	No	No	Slow
Dunbar [27]	$\mathbb{R}^2$	-	No	Yes	Yes	No	No	Fair
Gamito [34]	$\mathbb{R}^n$	-	No	Yes	Yes	No	Yes	Fair
Wei [86]	$\mathbb{R}^n$	-	Yes	No	No	No	Yes	Very Good
Chen [11]	$\mathbb{R}^2$	-	Yes	No	Yes	No	Yes	Good
Ebeida. [28]	$\mathbb{R}^2$	-	No	Yes	Yes	No	Yes	Good
Ebeida. [29]	$\mathbb{R}^n$	-	Yes	Yes	Yes	No	Yes	Good
Cline. [17]	Surfaces in $\mathbb{R}^n$	Yes	No	Yes	Yes	No	Yes	Fair
Fu [33]	Surfaces in $\mathbb{R}^n$	Yes	No	No	Yes	No	No	Slow
Bowers [9]	3D surfaces	No	Yes	No	No	No	Yes	Very Good
Chen [12]	Surfaces in $\mathbb{R}^n$	Yes	No	No	No	Yes	No	Fair
Xu [95]	Surfaces in $\mathbb{R}^n$	Yes	No	No	No	Yes	No	Slow
Corsini [21]	3D Surfaces	No	No	Yes	No	No	Yes	Fair
Our method	Surfaces in $\mathbb{R}^n$	Yes	Yes	Yes	No	No	No	Very Good

Table 6.2: Comparison to other surface sampling algorithms.

Comparison to the phase group method [86] [9]. The phase group method, pioneered by Wei [86], is the first technique for parallel Poisson disk sampling. By subdividing the sampling domain into grid cells, darts can be thrown concurrently into multiple cells that are sufficiently far apart to avoid conflicts. The phase group method is highly efficient and can be applied to Euclidean space of arbitrary dimension, making it very attractive in practice. However, as pointed out in [34], the Poisson disk distribution generated by the phase group method is approximate since the sequence of processing the phase groups follows a pre-defined order. Although such bias can be reduced by increasing the grid resolution, it will increase the computational cost as well. Our method is proven to be equivalent to the classical dart throwing. As a result, it guarantees to generate Poisson disk distribution without bias. Furthermore, our method does not require the explicit space partitioning. This intrinsic feature makes our method flexible and can be applied to both Euclidean space $\mathbb{R}^n$ and arbitrary surfaces easily.

Bowers et al. extended the phase group method to curved surface by subdividing the embedding space [9]. Although their method works fairly well for general models in graphics community, this space depending property may reduce their performance and compromise the quality. For example, if the input model has very complicated geometry, the uniform space partition does not lead to a uniform partition of the points inside each cell, i.e., some cells may have significantly larger points than others. Such unbalance would reduce the performance. Our method does not require the space partition. Instead, it simply organizes the sample points by their indices.

Figure 6.11 shows another scenario where the input surface is self-intersected. The phase group method may organize spatially-close-but-topologically-far points (imagine the points near the intersection and are on different sheets) into the same cell. Due to the “false” conflicts, Bowers et al.’s method results in fewer samples near the intersected region than our method.

We also observed that computing the exact geodesic distance is the bottleneck of our surface sampling framework, while Bowers et al. [9] adopted a simple closed-form formula to approximate geodesics. To make a fair comparison of the performance, we tested our priority based parallelization with the approximate geodesic distance [9]. As shown in Figure 6.27(b), the performance of our priority based parallelization is about 85%-95% of the phase group method, which demonstrates that our method is fairly efficient.

Comparison to [12] [95]. From quite a different perspective, Chen et al. [12] and Xu et al. [95] formulated the Poisson disk sampling on surface into an optimization problem. Both approaches are intrinsic and independent of the ambient space. Although the spectrum analysis shows the patterns generated by [12] and [95] exhibit blue noise characteristics, these approaches are only approximate, since the samples not distributed in a fully random and uniform manner. Furthermore, both approaches are computational expensive due to solving the centroidal Voronoi tessellation [12] and computing capacity-constrained triangulation. Furthermore, it is unclear whether [12] and [95] can be implemented in parallel. Our method is simple and robust, and significantly faster than [12] [95].

Comparison to the Euclidean space $\mathbb{R}^n$ sampling algorithms. The existing Euclidean space algorithms either adopt some spatial data structure for maintaining the not-yet-sampled domain [34] [27] or explicitly partition the sampling domain [28] [29]. Like the phase group

method [9], some algorithms could be extended to curved surface by partitioning the surface's embedding space. However, as mentioned above, such extension is extrinsic and space dependent, which may fail for surfaces with complicated geometry/topology and is not suitable for surfaces in high dimensional space. Our method is fundamentally different than the aforementioned approaches in that it neither partitions the surface or its embedding space nor adopts any spatial data structure to maintain the sampling domain. As shown in PDTs and PSDP, our algorithms take only a simple mesh data structure (half-edge in our implementation) and a few lists/vectors to organize the candidates by their indices instead of their spatial locations. Furthermore, the geodesic distance between samples is measured using the metric of the given mesh, which is independent of the surface's embedding space. Thanks to its simple and intrinsic nature, our method is flexible and can be applied to arbitrary surfaces in $\mathbb{R}^n$.

Limitations. First, although it can make full use of all the available GPU threads to obtain maximal performance, our algorithm has to synchronize all threads three times for each iteration. Our experimental results show that each synchronization usually takes $0.04 - 0.05$ milliseconds. Note that this overhead is independent of the number of threads T . The number of iterations for PSDP to terminate is in the range of $[|\mathcal{S}|/T, |\mathcal{P}|/T]$. Therefore, the total number of synchronization is between $3|\mathcal{S}|/T$ and $3|\mathcal{P}|/T$. The number of synchronization required for [9] is gl , where g is the number of phase groups (e.g., $3 \times 3 \times 3$), and l is the number of trials in each phase group (e.g., 10). Thus, the overhead in [9] is independent of the sampling complexity and GPU configuration, and much less than ours. Second, our method converts the input model to dense point clouds on surface M . To obtain high quality sampling, we usually require the ratio $|\mathcal{P}|/|\mathcal{S}| \geq 20$, which results in high memory costs. Thus, our method is limited by the available physical memory of the graphics card. Third, our method does not generate maximal distribution at the completion of the algorithm and it is difficult to control the exact number of samples generated.

Our improvement of maximal property also has two limitations: First, our improvement cannot guarantee the resulting maximal sampling is free of bias. This is because our method inserts new samples only on the boundaries of the permitted regions. Ideally, these new samples should be distributed randomly in such regions. However, as the output of the PSDP algorithm is close to maximal, the permitted regions are relatively small compared to the entire sampling domain.

(a) Speed up when using the approximate geodesic distance [9].

(b) Relative performance.

Figure 6.27: To make a fair comparison of the performance to [9], we adopt the approximate geodesic distance into our priority based parallelization framework. (a) Take the performance of priority based parallelization with exact geodesic distance as one. Adopting the approximate geodesic can greatly boost the performance of our sampling framework, when the number of samples is small. However, such speedup is not significant when the sample number is very high and each geodesic disk becomes very small. (b) shows the relative performance of our priority based parallelization to the phase group method [9]. Our method is about 5% to 15% slower than their method due to the fact that some computing samples may be rejected in our approach.

Thus, such distribution bias of new samples is very tiny. Second, our current implementation is limited to 2D and 3D. Although the idea can be extended to higher dimensional space by recursively reducing the problem from $\mathbb{R}^d$ to $\mathbb{R}^{d-1}$, the implementation is not easy.

Chapter 7

Conclusions & Future Work

7.1 Conclusions

The discrete geodesic problem has attracted a great deal of attention since Mitchell, Mount and Papadimitriou published their seminal paper in 1987. However, the existing geodesic algorithms are not practical for large-scale models or time-critical applications. First, the existing algorithms compute exact geodesic in a serial manner due to the lack of a parallel structure. The computation of geodesic on a large-scale model could be very time-consuming. Second, the widely studied single-source all-destination geodesic algorithms are not elegant for all-pair geodesic queries. Thus, the efficiency and practicality of geodesic computation remain a challenge.

In this thesis, I systematically studied the discrete geodesic problem. My study focuses on all-pairs geodesic distance, geodesic offsets, and parallel geodesic algorithm on modern GPUs. Furthermore, we propose the world’s first intrinsic and parallel Poisson-disk sampling on surfaces, which is a typical time-critical application of discrete geodesics.

In chapter 3, we propose an efficient and exact algorithm to compute the geodesic offsets on triangle meshes by extending the Xin-Wang algorithm of discrete geodesics. We define a new data structure for parallel-source windows, and extend both the “one angle one split” and the filtering theorem to maintain the windows tree. Similar to the original Xin-Wang algorithm, our extended algorithm has an $O(n)$ space complexity and an $O(n^2 \log n)$ asymptotic time complexity, where n is the number of vertices on the input mesh. We tested our algorithm on

numerous real-world models and showed that our algorithm is efficient and insensitive to the mesh resolution/tessellation.

In chapter 4, we propose an efficient algorithm to approximate the pairwise geodesic on triangular meshes. The pre-processing step takes $O(mn^2 \log n)$ time for the input mesh with n vertices and m samples, where $m (\ll n)$ is specified by the user, usually between a few hundred and several thousand. Then in the query step, our algorithm can compute the *approximate* geodesic distance between arbitrary pair of points (not necessarily mesh vertices) in constant time $O(1)$. Furthermore, the geodesic path and the geodesic distance field can be approximated in linear time. Both theoretical analysis and experimental results on real-world models demonstrate that our algorithm is efficient and accurate.

In chapter 5, we propose the parallel Chen-Han (or PCH) algorithm, which extends the classic Chen-Han (CH)s discrete geodesic algorithm to the parallel setting. The original CH algorithm and its variant lack the parallel solution since the windows (a key data structure which carries the shortest distance in the wavefront propagation) are maintained in a strict order or a tightly coupled manner. As a result, only one window can be processed at a time. Our idea is to divide the CHs sequential algorithm into four phases, namely window selection, window propagation, data organization and events processing, such that there is no data dependence or conflicts in each phase and the operations within each phase can be done in parallel. The proposed PCH algorithm is able to propagate a large number of windows simultaneously and independently. In addition, we adopt a simple yet effective strategy to control the total number of windows. We implement the PCH algorithm on modern GPUs (e.g., NVIDIA GTX 580) and observe that the performance improvement (compared to the sequential algorithms) is highly consistent with GPU double precision performance (Gflops).

The proposed parallel geodesic computation technique has the following contributions: 1) It is the first parallel exact geodesic algorithm for 2-manifold meshes; 2) To make the CH algorithm parallelizable, two novel techniques, k -selection scheme and update event triggering, are proposed; and 3) Experimental results demonstrate that the proposed implementation improves running time performance by an order of magnitude when compared to the state-of-the-art.

And in chapter 6, we propose an application of discrete geodesics. We present an accurate and parallel algorithm for Poisson disk sampling on arbitrary surfaces. Sampling plays an

important role in a variety of graphics applications. Among existing sampling methods, Poisson disk sampling is popular thanks to its useful statistical property in distribution and the absence of aliasing artifacts. Although many promising algorithms have been proposed for multi-dimensional sampling in Euclidean space, very few research studies have been reported with regard to the problem of generating Poisson disks on surfaces due to the complicated nature of the surface. Rather than the conventional approaches that explicitly partition the spatial domain to generate the samples in parallel, our approach assigns each sample candidate a random and unique priority that is unbiased with regard to the distribution. Hence, multiple threads can process the candidates simultaneously and resolve conflicts by checking the given priority values. Our algorithm is accurate as it uniformly and randomly distributes the Poisson disks on surfaces. All the computations are based on the intrinsic metric and are independent of the embedding space. Therefore, it works for arbitrary surfaces in $\mathbb{R}^n$. To our knowledge, this is the first accurate and parallel algorithm for generating Poisson disk patterns on surfaces based on the exact geodesic distance.

It is highly desirable to develop a fast algorithm to compute the exact geodesic on triangle meshes. However, the existing exact geodesic algorithms have very strong data dependence and lack the parallel solution. We have presented the constant time all-pair algorithm to compute the approximate geodesic, the extended Xin-Wang algorithm for geodesic offsets, and the PCH algorithm, which move the discrete geodesic algorithm from serial to parallel. We also demonstrate an application of discrete geodesic on GPUs. Hence, we believe our algorithms could significantly contribute to the computer graphics society.

7.2 Future Work

There are a few interesting future directions based on PCH. First, we will study effective windows filtering techniques to further reduce the total number of windows, thus, improving the overall performance. Second, our PCH algorithm naturally supports the multiple-source-all-destination geodesic distance, thus, it has the potential to improve the performance of the geodesic Voronoi diagram [52]. Third, as the first parallel geodesic algorithm with good performance, the PCH algorithm is highly desired in many time-critical applications which require extensive computation of geodesics. We will investigate such applications in the near future.

There are also some problems of the discrete geodesics that confusing us. The MMP algorithm [58] computes the single-source geodesic distance in $O(n^2 \log n)$ time, where n is the number of vertices of the input mesh. In 1990, Chen and Han [10] improved the time complexity to $O(n^2)$, which remains the best-known complexity. However, extensive experiments have shown that the CH algorithm often runs 10^3 to 10^4 times slower than the MMP algorithm. Xin and Wang realized that such slowness is due to the extremely large number of operations to the useless windows (a data structure to that will be explained later), so they modified the CH algorithm by adopting window filtering and maintaining windows in a priority queue. Although the improved CH (or ICH) algorithm has $O(n^2 \log n)$ time complexity, it outperforms both the MMP and the CH algorithms.

So why can the *theoretical* time complexity not indicate the *real* performance in the discrete geodesic problem? The exact geodesic algorithms, such as MMP, CH and ICH, all adopt the same computational framework: each edge is partitioned into a set of intervals, called windows, and each window encodes the exact geodesic distance from the source to any point in the interval. The windows are maintained in either a sorted data structure (like a priority queue in the MMP and ICH algorithms) or a hierarchical data structure (like a tree in the CH algorithm). In each iteration, the algorithms propagate one window across a triangle by computing the window's children windows and performing clipping or merging if necessary, which can add, modify, or remove existing windows. The algorithms terminate when all windows have been processed. Thus, window complexity is a key factor with which to measure the real performance of the algorithm. [58] and [10] noted that the number of windows is $O(n^2)$, which means that none of the the window propagation algorithms can break the *theoretical* $O(n^2)$ time barrier. On the other hand, Surazhsky et al. [81] observed that the window complexity grows as approximately $O(n^{1.5})$ on real-world models. In practice, therefore, the MMP and the ICH algorithms in practice run in sub-quadratic time.

Surazhsky et al. [81] also observed that given a smooth surface, adding a random noise at each vertex can reduce the window complexity significantly. This interesting observation is confirmed by the experience that computing the geodesic on a bumpy real-world model usually takes less time than that for a smooth synthetic model of similar complexity. However, the existing literature has not addressed the question of the relation between the algorithm's time complexity and the surface's geometric features.

All of the existing algorithms compute the globally shortest geodesic by propagating the distance information in the wavefront order, thereby tackling the discrete geodesic problem in a global manner. I have observed that the discrete surface has some interesting and unique properties, such as the discrete geodesic's very strong local structure. The richer the geometric features of the mesh, the stronger such local structures are. This could allow us to break down the original problem into easily solvable sub-problems.

References

- [1] P. K. Agarwal, B. Aronov, J. O'Rourke, and C. A. Schevon. Star unfolding of a polytope with applications. *SIAM J. Comput.*, 26(6):1689–1713, 1997.
- [2] Pankaj K. Agarwal, Sarel Har-Peled, and Meetesh Karia. Computing approximate shortest paths on convex polytopes. In *Symposium on Computational Geometry*, pages 270–279, 2000.
- [3] Pankaj K. Agarwal, Sarel Har-Peled, Micha Sharir, and Kasturi R. Varadarajan. Approximating shortest paths on a convex polytope in three dimensions. *J. ACM*, 44(4):567–584, 1997.
- [4] Lyudmil Aleksandrov, Anil Maheshwari, and Jörg-Rüdiger Sack. An improved approximation algorithm for computing geometric shortest paths. In *Fundamentals of Computation Theory, 14th International Symposium, FCT 2003*, pages 246–257, 2003.
- [5] Michael Balzer, Thomas Schlömer, and Oliver Deussen. Capacity-constrained point distributions: a variant of lloyd's method. In *ACM SIGGRAPH 2009 papers*, SIGGRAPH '09, pages 86:1–86:8, 2009.
- [6] Michael Barbehenn. A note on the complexity of Dijkstra's algorithm for graphs with weighted vertices. *IEEE Trans. Comput.*, 47(2):263, 1998.
- [7] Surender Baswana, Ramesh Hariharan, and Sandeep Sen. Maintaining all-pairs approximate shortest paths under deletion of edges. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 394–403, 2003.

- [8] Nathan Bell and Jared Hoberock. *GPU Computing Gems*, chapter Thrust: a productivity-oriented library for CUDA, pages 359–371. Morgan Kaufmann Publishers, jade edition, 2011.
- [9] John Bowers, Rui Wang, Li-Yi Wei, and David Maletz. Parallel poisson disk sampling with spectrum analysis on surfaces. *ACM Trans. Graph.*, 29:166:1–166:10, 2010.
- [10] J. Chen and Y. Han. Shortest paths on a polyhedron. In *SCG '90: Proceedings of the Sixth Annual Symposium on Computational Geometry*, pages 360–369, 1990.
- [11] R. Chen and C. Gotsman. Parallel blue-noise sampling by constrained farthest point optimization. *Computer Graphics Forum (Proc. Symp. Geometry Proc. '12)*, 31(5), 2012.
- [12] Zhonggui Chen, Zhan Yuan, Yi-King Choi, Ligang Liu, and Wenping Wang. Variational blue noise sampling. *IEEE Transactions on Visualization and Computer Graphics*, 18(10):1784–1796, 2012.
- [13] Z.M. Chen, Y.G. Chen, and F.J. Liang. A new offset approach for curves on triangle mesh surfaces. In *Technology and Innovation Conference 2006*, pages 202–208, 2006.
- [14] B. K. Choi and R. Jerard. *Sculptured Surface Machining - Theory and Applications*. Kluwer Academic Publishers, 1998.
- [15] Joonsoo Choi, Jürgen Sellen, and Chee-Keng Yap. Approximate euclidean shortest path in 3-space. In *Proceedings of the tenth annual symposium on Computational geometry*, pages 41–48, 1994.
- [16] K. Clarkson. Approximation algorithms for shortest path motion planning. In *STOC '87*, pages 56–65, 1987.
- [17] David Cline, Stefan Jeschke, Anshuman Razdan, Kenric White, and Peter Wonka. Dart throwing on surfaces. *Computer Graphics Forum*, 28(4):1217–1226, 6 2009.
- [18] Michael F. Cohen, Jonathan Shade, Stefan Hiller, and Oliver Deussen. Wang tiles for image and texture generation. In *ACM SIGGRAPH 2003 Papers*, pages 287–294, 2003.
- [19] Robert L. Cook. Stochastic sampling in computer graphics. *ACM Trans. Graph.*, 5:51–72, 1986.

REFERENCES

- [20] Atlas F. Cook IV and Carola Wenk. Shortest path problems on a polyhedral surface. In *WADS '09*, pages 156–167, 2009.
- [21] M. Corsini, P. Cignoni, and R. Scopigno. Efficient and flexible sampling with blue noise properties of triangular meshes. *IEEE Transactions on Visualization and Computer Graphics*, 18(6):914–924, 2012.
- [22] Fernando de Goes, Katherine Breeden, Victor Ostromoukhov, and Mathieu Desbrun. Blue noise through optimal transport. *ACM Trans. Graph.*, 31(6):171:1–171:11, November 2012.
- [23] E.W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.
- [24] Mark A. Z. Dippé and Erling Henry Wold. Antialiasing through stochastic sampling. In *SIGGRAPH '85*, pages 69–78, 1985.
- [25] Dorit Dor, Shay Halperin, and Uri Zwick. All-pairs almost shortest paths. *SIAM J. Comput.*, 29:1740–1759, 2000.
- [26] Qiang Du, Vance Faber, and Max Gunzburger. Centroidal voronoi tessellations: Applications and algorithms. *SIAM Rev.*, 41:637–676, 1999.
- [27] Daniel Dunbar and Greg Humphreys. A spatial data structure for fast poisson-disk sample generation. *ACM Trans. Graph.*, 25:503–508, 2006.
- [28] Mohamed S. Ebeida, Andrew A. Davidson, Anjul Patney, Patrick M. Knupp, Scott A. Mitchell, and John D. Owens. Efficient maximal poisson-disk sampling. *ACM Trans. Graph.*, 30:49:1–49:12, 2011.
- [29] Mohamed S. Ebeida, Scott A. Mitchell, Anjul Patney, Andrew A. Davidson, and John D. Owens. A simple algorithm for maximal poisson-disk sampling in high dimensions. *Computer Graphics Forum*, 31(2):785–794, 2012.
- [30] G. Elber, I.-K. Lee, and M. S. Kim. Comparing offset curve approximation methods. *IEEE Computer Graphics and Applications*, 17(3):62–71, May/June 1997.
- [31] Raanan Fattal. Blue-noise point sampling using kernel density model. In *ACM SIGGRAPH 2011 papers*, SIGGRAPH '11, pages 48:1–48:12, 2011.

- [32] Mark Forsyth. Shelling and offsetting bodies. In *Proceedings of the third ACM symposium on Solid modeling and applications*, SMA '95, pages 373–381, 1995.
- [33] Yan Fu and Bingfeng Zhou. Direct sampling on surfaces for high quality remeshing. In *ACM SPM '08*, pages 115–124, 2008.
- [34] Manuel N. Gamito and Steve C. Maddock. Accurate multidimensional poisson-disk sampling. *ACM Trans. Graph.*, 29:8:1–8:19, 2009.
- [35] Sariel Har-Peled. Approximate shortest paths and geodesic diameter on a convex polytope in three dimensions. *Discrete & Computational Geometry*, 21(2):217–231, 1999.
- [36] A. Hatna, R. J. Grieve, and P. Broomhead. Offsetting 3d contours on parametric surfaces. *The International Journal of Advanced Manufacturing Technology*, 16(3):189–195, 2000.
- [37] John Hersberger and Subhash Suri. Practical methods for approximating shortest paths on a convex polytope in $\mathbb{R}^3$. In *SODA '95: Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 447–456, 1995.
- [38] V. Devaraja Holla, K. G. Shastry, and B. G. Prakash. Offset of curves on tessellated surfaces. *Computer-Aided Design*, 35(12):1099–1108, 2003.
- [39] Qi-Xing Huang, Bart Adams, Martin Wicke, and Leonidas J. Guibas. Non-rigid registration under isometric deformations. In *SGP '08*, pages 1449–1457, 2008.
- [40] Jr JI Yellott. Spectral consequences of photoreceptor sampling in the rhesus retina. *Science*, 221(4608):382–385, July 1983.
- [41] Thouis R. Jones. Efficient generation of poisson-disk sampling patterns. *Journal of Graphics, GPU, and Game Tools*, 11(2):27–36, 2008.
- [42] K. I. Kim and K. Kim. A new machine strategy for sculptured surfaces using offset surface. *International Journal of Production Research*, 33(6):1683–1697, 1995.
- [43] R. Kimmel and J. A. Sethian. Computing geodesic paths on manifolds. In *Proc. Natl. Acad. Sci. USA*, pages 8431–8435, 1998.
- [44] Johannes Kopf, Daniel Cohen-Or, Oliver Deussen, and Dani Lischinski. Recursive wang tiles for real-time blue noise. In *SIGGRAPH '06*, pages 509–518, 2006.

REFERENCES

- [45] Ares Lagae and Philip Dutré. A procedural object distribution function. *ACM Trans. Graph.*, 24:1442–1461, 2005.
- [46] Ares Lagae and Philip Dutré. A comparison of methods for generating poisson disk distributions. *Computer Graphics Forum*, 27:114–129, 2008.
- [47] Mark Lanthier, Anil Maheshwari, and Jörg-Rüdiger Sack. Approximating weighted shortest paths on polyhedral surfaces. In *SCG '97: Proceedings of the thirteenth annual symposium on Computational geometry*, pages 485–486, 1997.
- [48] Haeyoung Lee, Yiyong Tong, and Mathieu Desbrun. Geodesics-based one-to-one parameterization of 3d triangle meshes. *IEEE MultiMedia*, 12:27–33, January 2005.
- [49] Hendrik P. A. Lensch, Bodo Rosenhahn, Hans-Peter Seidel, Philipp Slusallek, and Joachim Weickert, editors. *Proceedings of the Vision, Modeling, and Visualization Conference 2007*, 2007.
- [50] Hongwei Li, Kui-Yip Lo, Man-Kang Leung, and Chi-Wing Fu. Dual poisson-disk tiling: An efficient method for distributing features on arbitrary surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 14(5):982–998, 2008.
- [51] Hongwei Li, Li-Yi Wei, Pedro V. Sander, and Chi-Wing Fu. Anisotropic blue noise sampling. *ACM Trans. Graph.*, 29:167:1–167:12, 2010.
- [52] Yong-Jin Liu, Zhanqing Chen, and Kai Tang. Construction of iso-contours, bisectors, and Voronoi diagrams on triangulated surfaces. *IEEE Trans. Pattern Anal. Mach. Intell.*, 33(8):1502–1517, 2011.
- [53] T Lozano-Perez and MA Wesley. An algorithm for planning collision free paths amongst polyhedral obstacles. *Communications of the ACM*, 25(9):560–570, 1979.
- [54] T. Maekawa. An overview of offset curves and surfaces. *Computer-Aided Design*, 31(3):165–173, March 1999.
- [55] K. Marciniak. *Geometric modeling for numerically controlled machining*. Oxford University Press, 1991.
- [56] Michael McCool and Eugene Fiume. Hierarchical poisson disk sampling distributions. In *GI '92*, pages 94–105, 1992.

- [57] Don P. Mitchell. Spectrally optimal sampling for distribution ray tracing. In *SIGGRAPH '91*, pages 157–164, 1991.
- [58] J. S. B. Mitchell, D. M. Mount, and C. H. Papadimitriou. The discrete geodesic problem. *SIAM J. Comput.*, 16(4):647–668, 1987.
- [59] Joseph S. B. Mitchell. New results on shortest paths in three dimensions. In *SCG '04*, pages 124–133, 2004.
- [60] Laura Monroe, Joanne Wendelberger, and Sarah Michalak. Randomized selection on the gpu. *High Performance Graphics, HPG '11*, pages 89–98, 2011.
- [61] Diego Nehab and Philip Shilane. Stratified point sampling of 3D models. In *IEEE/Eurographics Symposium on Point Based Graphics, PBG '04*, pages 49–56, 2004.
- [62] Robert Osada, Thomas Funkhouser, Bernard Chazelle, and David Dobkin. Shape distributions. *ACM Trans. Graph.*, 21:807–832, 2002.
- [63] Victor Ostromoukhov. Sampling with polyominoes. *ACM Trans. Graph.*, 26, 2007.
- [64] Victor Ostromoukhov, Charles Donohue, and Pierre-Marc Jodoin. Fast hierarchical importance sampling with blue noise properties. In *ACM SIGGRAPH 2004 Papers, SIGGRAPH '04*, pages 488–495, 2004.
- [65] A. Cengiz Öztireli, Marc Alexa, and Markus H. Gross. Spectral sampling of manifolds. *ACM Trans. Graph.*, 29(6):168, 2010.
- [66] A. Cengiz Öztireli and Markus Gross. Analysis and synthesis of point distributions based on pair correlation. *ACM Trans. Graph.*, 31(6):170:1–170:10, November 2012.
- [67] C. H. Papadimitriou. An algorithm for shortest-path motion in three dimensions. *IPL*, 20:259–263, 1985.
- [68] N. M. Patrikalakis and L. Bardis. Offsets of curves on rational b-spline surfaces. *Engineering with Computers*, 5(1):39–46, 1989.
- [69] Gabriel Peyr, Mickael Pchaud, Renaud Keriven, and Laurent D. Cohen. Geodesic methods in computer vision and graphics. *Foundations and Trends in Computer Graphics and Vision*, 5(3-4):197–397, 2010.

REFERENCES

- [70] Gabriel Peyré and Laurent D. Cohen. Geodesic remeshing using front propagation. *Int. J. Comput. Vision*, 69(1):145–156, 2006.
- [71] Binh Pham. Offset curves and surfaces: a brief survey. *Computer-Aided Design*, 24(4):223–229, 1992.
- [72] L. A. Piegl and W. Tiller. Computing offsets of NURBS curves and surfaces. *Computer-Aided Design*, 31(2):147–156, February 1999.
- [73] Konrad Polthier and Markus Schmies. *Mathematical Visualization*, chapter Straightest geodesics on polyhedral surfaces, page 391. Springer Verlag, 1998.
- [74] Konrad Polthier and Markus Schmies. Geodesic flow on polyhedral surfaces. In *Proceedings of Eurographics-IEEE Symposium on Scientific Visualization '99*, pages 179–188, 1999.
- [75] J R Rossignac and A A G Requicha. Offsetting operations in solid modelling. *Comput. Aided Geom. Des.*, 3:129–148, August 1986.
- [76] P. V. Sander, Z. J. Wood, S. J. Gortler, J. Snyder, and H. Hoppe. Multi-chart geometry images. In *SGP '03*, pages 146–155, 2003.
- [77] Ryan Schmidt, Cindy Grimm, and Brian Wyvill. Interactive decal compositing with discrete exponential maps. *Transactions on Graphics*, 25(3):605–613, July 2006.
- [78] J. A. Sethian. A fast marching level set method for monotonically advancing fronts. *Proc. Nat. Acad. Sci*, pages 1591–1595, 1996.
- [79] M. Sharir and A. Schorr. On shortest paths in polyhedral spaces. *SIAM J. Comput.*, 15(1):193–215, 1986.
- [80] Alon Spira and Ron Kimmel. Geodesic curvature flow on parametric surfaces. In *Curve and Surface Design*, pages 365–373, 2002.
- [81] Vitaly Surazhsky, Tatiana Surazhsky, Danil Kirsanov, Steven J. Gortler, and Hugues Hoppe. Fast exact and approximate geodesics on meshes. *ACM Trans. Graph.*, 24(3):553–560, 2005.
- [82] Greg Turk. Re-tiling polygonal surfaces. In *SIGGRAPH '92*, pages 55–64, 1992.

- [83] Bruno Vallet and Bruno Lévy. Spectral geometry processing with manifold harmonics. *Comput. Graph. Forum*, 27(2), 2008.
- [84] Kasturi R. Varadarajan and Pankaj K. Agarwal. Approximating shortest paths on a non-convex polyhedron. In *IEEE Symposium on Foundations of Computer Science*, pages 182–191, 1997.
- [85] Ofir Weber, Yohai S. Devir, Alexander M. Bronstein, Michael M. Bronstein, and Ron Kimmel. Parallel algorithms for approximation of distance maps on parametric surfaces. *ACM Trans. Graph.*, 27(4), 2008.
- [86] Li-Yi Wei. Parallel poisson disk sampling. In *SIGGRAPH '08*, pages 20:1–20:9, 2008.
- [87] Li-Yi Wei. Multi-class blue noise sampling. *ACM Trans. Graph.*, 29, 2010.
- [88] Li-Yi Wei and Rui Wang. Differential domain analysis for non-uniform sampling. *ACM Trans. Graph.*, 30(4):50:1–50:10, July 2011.
- [89] K.B. White, D. Cline, and P.K. Egbert. Poisson disk point sets by hierarchical dart throwing. In *IEEE Symposium on Interactive Ray Tracing. RT '07.*, pages 129–132, 2007.
- [90] Wikipedia. Geodesic.
- [91] Shi-Qing Xin, Ying He, and Chi-Wing Fu. Efficiently computing exact geodesic loops within finite steps. *IEEE Transactions on Visualization and Computer Graphics*, 18(6):879–889, 2011.
- [92] Shi-Qing Xin and Guo-Jin Wang. Efficiently determining a locally exact shortest path on polyhedral surfaces. *Computer-Aided Design*, 39(12):1081–1090, 2007.
- [93] Shi-Qing Xin and Guo-Jin Wang. Improving Chen and Han’s algorithm on the discrete geodesic problem. *ACM Trans. Graph.*, 28(4):1–8, 2009.
- [94] Shi-Qing Xin and Guo-Jin Wang. Applying the improved Chen and Han’s algorithm to different versions of shortest path problems on a polyhedral surface. *CAD*, 42:942–951, 2010.
- [95] Yin Xu, Ruizhen Hu, Craig Gotsman, and Ligang Liu. Blue noise sampling of surfaces. *Computers & Graphics*, 36(4):232–240, 2012.

REFERENCES

- [96] Kun Zhou, John Synder, Baining Guo, and Heung-Yeung Shum. Iso-charts: stretch-driven mesh parameterization using spectral analysis. In *SGP '04*, pages 45–54, 2004.
- [97] Uri Zwick. All pairs shortest paths in weighted directed graphs exact and almost exact algorithms. In *Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, FOCS '98, page 310, 1998.
- [98] Uri Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *J. ACM*, 49:289–317, 2002.