Efficient Nearest Neighbor Search Using Dynamic Programming

Pengfei Wang, Jiantao Song, Shiqing Xin (Corresponding author), Shuangmin Chen, Changhe Tu, Wenping Wang, Jiaye Wang

***Abstract*—Given a collection of points in $\mathbb{R}^3$, KD-Tree and R-Tree are well-known nearest neighbor search (NNS) algorithms that rely on space partitioning and spatial indexing techniques. However, when the query point is far from the data points or the data points inherently represent a 2-manifold surface, their query performance may degrade.**

To address this, we propose a novel dynamic programming technique that precomputes a Directed Acyclic Graph (DAG) to encode the proximity structure between data points. More specifically, the DAG captures how the proximity structure evolves during the incremental construction of the Voronoi diagram of the data points. Experimental results demonstrate that our method achieves a 1x-10x speedup.

Additionally, our algorithm offers several valuable features. For instance, it naturally supports an $O(k \log n)$ algorithm for farthest point sampling, where k is the desired number of sample points. Moreover, density peak clustering, which involves finding the nearest point among the top K points, is typically considered to have a time complexity of $O(n^2)$. With our algorithm, this can be reduced to $O(n \log n)$. We believe this work will inspire further research on the NNS problem.

***Index Terms*—Nearest neighbor search, farthest point sampling, density peak clustering, delaunay, voronoi diragram.**

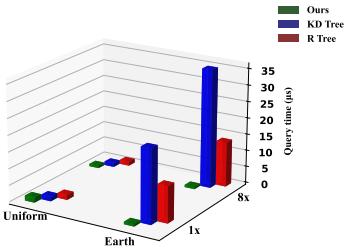
I. Introduction

GIVEN a target point set $\mathcal{P} = \{p_i\}_{i=1}^n$ and a query point q, the point p_i is considered the nearest neighbor in $\mathcal{P}$ to q if p_i satisfies

$$\forall j \in [1, n], \|q - p_i\| \leq \|q - p_j\|, \tag{1}$$

where $\|\cdot\|$ denotes the Euclidean distance. Finding the nearest neighbor is a fundamental operation in various fields, including computer graphics [1]–[3], computer vision [4], [5], and robotics [6]. Due to its critical applications across various fields, improving query speed remains an active area of research. Traditional nearest-neighbor query methods include KD Trees [7] and R Trees [8], which offer fast query speeds for general point clouds. These methods achieve efficiency by constructing hierarchical structures to partition the space or point cloud, enabling quick localization during queries. However, when handling data distributed along 2D manifold surfaces embedded in 3D space (referred to as 2D manifold data), common in computer graphics, or when the query point lies far from the dataset, the efficiency of these traditional algorithms can decline. In extreme cases, the query complexity may degrade to $O(n)$.

As shown in the inset figure, we consider two point clouds with distinct distributions: one uniformly distributed within a unit cube and the other across the surface of the Earth. Additionally, we generate two sets of query points—one within the bounding box of the input data and the other within 8 times the bounding box. It is evident that the query performance of both KD Trees and R Trees deteriorates significantly under these conditions.



In this paper, we present a simple, effective, and novel nearest point query algorithm based on Voronoi / Delaunay. For 2D manifold data in 3D, our query speed is 1x-10x faster than KD Tree and R Tree. Even for uniformly distributed random point clouds, our query algorithm achieves speeds comparable to KD Tree. Our main contributions are as follows:

- We introduced a concise nearest neighbor search algorithm for 2D manifold point cloud data that significantly improves query speed compared to traditional algorithms.
- If the point cloud possesses an order, our method can facilitate the identification of the nearest point to q among $\{p_i\}_{i=1}^k$, for any k, where $k \leq n$. This approach can be employed to reduce the time complexity of Density Peak Clustering [9], thus enhancing its applicability in large-scale scenarios.
- Based on our method, we developed a farthest point sampling algorithm with an average complexity of $O(k \log n)$ in 3D, where n represents the total number of points and k denotes the number of samples.

II. Related Work

A. Exact Nearest Neighbor Search

For spatial data, the nearest neighbor problem is typically addressed by building a spatial indexing structure [10]. These techniques can be broadly categorized into two types: those that partition space and those that partition data.

Classic structures for space partitioning include KD Trees and quadtrees. A KD Tree [7] is a spatial partitioning structure that recursively divides the dataset along one of the k coordinate system axes, typically alternating between them; for instance, with $k = 2$, splits occur alternately on the x- and y-axes. This splitting process continues until a maximum depth is reached or until fewer than a specified number of points remain in a cell. KD Trees can be extended to include objects beyond points, such as triangles or line segments.

Determining the optimal splitting plane can be challenging in these cases; however, for points (or disks) in two dimensions, one can simply sort the points along each dimension and split at the median. The construction of a KD Tree can be achieved in $O(n \log n)$ time, and it can find the nearest neighbors in $O(\log n)$ time.

An octree is another spatial partitioning structure that specifically divides 3D space into eight cubes. This is typically done recursively, with each subdivision occurring when the data within a region meets certain criteria, such as exceeding a maximum number of points or objects. Each internal node in the octree corresponds to a cube region of space, further subdivided into eight equal-sized cubes represented by its children. The construction of an octree can also be completed in $O(n \log n)$ time, and it can find the nearest neighbors in $O(\log n)$ time as well.

The R Tree [8] is a classic structure for partitioning data. It is based on B-trees and shares a similar structure: both the number of objects in a leaf node and the number of children in an internal node are bounded by specified minimum and maximum limits, and all leaf nodes are at the same level. The R Tree can be built in $O(n \log n)$ time and allows for the identification of nearest neighbors in $O(\log n)$ time.

Grids and linear search methods are also traditional approaches for nearest-neighbor searches. The grid-based approach improves efficiency by overlaying a regular grid on the entire simulation area and tracking agents within each cell. During a query, the search includes the cell containing the query points and the adjacent cells. The construction time for this method is $O(n + m)$, where m represents the number of cells in the grid. While the worst-case query time is $O(n)$, practical performance greatly depends on the distribution of agents and the size of the cells. Conversely, linear search is suitable when the number of query points is very small. This method finds the nearest points by calculating the distance from each target point to the query point.

B. Approximate Nearest Neighbor Search

Graph-based algorithms are highly effective for approximate nearest neighbor (ANN) search due to their balance of accuracy and computational efficiency [11]–[14]. Typically, the process begins with constructing connections within a given point cloud, followed by iterative traversal of the graph starting from an entry point, which may be selected randomly or determined by a separate algorithm. At each step, the algorithm selects the closest neighboring node as the next base node, iteratively refining the current solution. The search terminates once a predefined stopping condition, such as a set number of distance evaluations, is met.

One well-known graph structure used for search is the Navigable Small World (NSW, also known as Metricized Small World, MSW) [12], [14], [15]. NSW constructs the graph by inserting elements in random order. Each element is then bidirectionally connected to its M nearest neighbors among the previously inserted nodes. To improve search efficiency, many algorithms adopt a hierarchical strategy by creating multiple layers of the graph, which helps narrow the search space and accelerate the process.

Our method adopts a similar graph-based strategy, emphasizing efficient connectivity and traversal. Unlike traditional graph-based approximate nearest neighbor search, our method precisely identifies the nearest points with strict guarantees of correctness. Furthermore, although we do not explicitly use hierarchical structures, our integrated construction and traversal strategies achieve comparable benefits, such as efficient narrowing of the search space and improved accuracy.

C. Voronoi Diagram & Delaunay

Suppose we have a set of generators $\mathcal{S} = \{s_i\}_{i=1}^n$ in a given domain Ω equipped with a metric function $\mathcal{D}$. The Voronoi diagram partitions Ω into regions such that the generator s_i dominates a region defined as

$$\text{Cell}(s_i) = \{x \in \Omega \mid \mathcal{D}(s_i, x) \leq \mathcal{D}(s_j, x),\, \forall j \neq i\}.$$

The definition of Voronoi diagrams may vary based on domains, metrics, and types of generators. The most prevalent version assumes that Ω represents Euclidean spaces and that the generators are exclusively points. As a fundamental tool, Voronoi diagrams have found widespread applications in computer graphics [1], [3], [16].

The Delaunay triangulation is the dual structure of the Voronoi diagram, representing the adjacency relationships between the Voronoi cells. Currently, Voronoi diagrams are typically computed by first calculating the Delaunay triangulation and then deriving the dual structure. The Watson algorithm [17] is a classic method for computing the Delaunay triangulation. It is based on the empty circle property and works by incrementally inserting points and flipping triangles that do not satisfy this property. Common tools for computing Delaunay triangulations include TetGen [18] and CGAL [19], which can achieve an average computational complexity of $O(n \log n)$ with a worst-case complexity of $O(n^2)$.

Voronoi diagrams can also be applied to nearest neighbor searches. By definition, if the closest point to a query point q in $\mathbf{P}$ is p_k, then q lies within the Voronoi cell of p_k. This property makes Voronoi diagrams highly relevant for solving nearest neighbor problems. However, there is currently no efficient algorithm for identifying the specific Voronoi cell containing q. This restricts the practical use of Voronoi diagrams in nearest neighbor searches.

D. Farthest Point Sampling

Farthest point sampling (FPS) is a technique widely employed in computational geometry and computer graphics applications. It is designed to iteratively select points from a dataset such that each newly selected point is maximally distant from the previously chosen points. This method can be formally expressed as follows:

$$p_{i+1} = \arg\max_{p \in P} \left(\min_{p_j \in S} d(p, p_j) \right), \tag{2}$$

where:

- P represents the set of points in the point cloud,
- S denotes the subset of points that have already been selected,

- $d(p, p_j)$ is the distance between points p and p_j,
- p_{i+1} is the next point selected such that its minimum distance to any point in S is maximized.

This sampling strategy ensures that the selected points are well-distributed, making it effective for approximating shapes, surfaces, or distributions. FPS is particularly useful in applications requiring a representative subset of data points, such as mesh generation, surface simplification, or cluster initialization.

The time complexity of a typical FPS method [20], [21] is $O(kn)$, where n is the total number of points in the dataset, and k denotes the number of points to be sampled. The latest FPS method QuickFPS [22] organizes large-scale point clouds into multiple bins. By employing two mechanisms—merge computation and implicit computation on these bins—it significantly reduces external memory access and computation costs, thereby improving the speed of farthest point sampling.

III. Methodology

In this section, we introduce the key concept underlying the algorithms, beginning with a solution of linear search.

We assume that the given points are numbered from 1 to n. Given a query point q, we use $\Phi_m(q)$ to denote the closest point in the subset $\{p_i\}_{i=1}^m$, where $m \leq n$. This leads to the following state transfer equation:

$$\Phi_m(q) = \begin{cases} p_1 & \text{if } m = 1, \\ \Phi_{m-1}(q) & \text{if } m \neq 1 \wedge \|q - \Phi_{m-1}(q)\| \leq \|q - p_m\|, \\ p_m & \text{if } m \neq 1 \wedge \|q - \Phi_{m-1}(q)\| > \|q - p_m\|, \end{cases} \tag{3}$$

which can be understood as a dynamic programming problem, where $\Phi_n(q)$ is the actual nearest point to be extracted. Obviously, a naïve implementation of the query operation costs $O(n)$ time. In the following, we consider how to boost this implementation.

Recall that Voronoi diagrams [23] precisely characterize how the space of interest is divided by the given data points. A point q is nearest to p_i if and only if q is located in p_i's cell. However, Voronoi diagrams are not well-suited for NNS due to their lack of hierarchical representation. Despite this, during the incremental construction process of Voronoi Diagram, the area dominated by a site either remains unchanged or shrinks, encoding the history of how the proximity structure changes when a new point is added. Let the input point cloud be $\mathcal{P} = \{p_i\}_{i=1}^n$. We illustrate our construction process by considering any moment of constructing the Voronoi diagram through incremental point insertion, such as $\mathcal{V}_m$, which is constructed from the set $\{p_i\}_{i=1}^m$, where $m \leq n$.

Figure 1(a) shows the Voronoi diagram $\mathcal{V}_{20}$ determined by 20 sites, with one of the cells highlighted in yellow. By adding three additional sites (whose cells are highlighted in green), we obtain a new Voronoi diagram $\mathcal{V}_{23}$; see Figure 1(b). It can be seen that the addition of the three green sites does not change the yellow cell. Therefore, if q is located in the yellow cell, then $\Phi_{23}(q) = \Phi_{20}(q)$, implying that the nearest site to q remains unchanged even when the three green sites are added. Only when a newly added site changes the yellow cell may the nearest site differ. Specifically, $\Phi_{24}(q) = \Phi_{23}(q)$ if q is nearer to the old site (see Figure 1(c)), and $\Phi_{24}(q) = p_{24}$ if q is nearer to the new site p_{24} (see Figure 1(d)). To summarize, by using $p_{m+1} \nvdash \text{Cell}(\Phi_m(q); \mathcal{V}_m)$ to represent that the addition of p_{m+1} does not diminish the cell of $\Phi_m(q)$ in $\mathcal{V}_m$, we have:

$$p_{m+1} \nvdash \text{Cell}(\Phi_m(q); \mathcal{V}_m) \quad \mapsto \quad \Phi_{m+1}(q) = \Phi_m(q), \tag{4}$$

or

$$\begin{gathered} p_{m+j} \nvdash \text{Cell}(\Phi_m(q); \mathcal{V}_{m+j-1}),\ \forall 1 \leq j \leq k \\ \downarrow \\ \mapsto \quad \Phi_{m+k}(q) = \Phi_{m+k-1}(q) = \cdots = \Phi_m(q). \end{gathered} \tag{5}$$

IV. Algorithm

A. Query Table Construction

Based on Section III, we can design a data structure called the Query Table, denoted as $\mathcal{L} = \{\mathcal{L}_i\}_{i=1}^n$, to store the necessary information for the query phase. Specifically, we assign each point p_i a corresponding vector $\mathcal{L}_i$, called Query List. During the incremental construction of the Voronoi diagram, when p_j is inserted, if the condition $p_j \vdash \text{Cell}(p_i; \mathcal{V}_{j-1})$ is satisfied, we push p_j to the end of $\mathcal{L}_i$.

$\mathcal{L}$ is constructed simutaneously as the Voronoi diagram is incrementally built. Initially, we have only one site p_1, leaving its Query List empty. Suppose that the addition of p_{m+1} changes p_i's cell in $\mathcal{V}_m$. We append p_{m+1} to p_i's Query List; see Figure 2. The Query Table $\mathcal{L}$ is completed once all the sites are inserted. It is important to note that the order of elements stored in each $\mathcal{L}_i$ is consistent with the insertion order.

Since the Delaunay is the dual of the Voronoi diagram and encodes the adjacency relationships between Voronoi cells, to check whether $p_m \vdash \text{Cell}(p_i; \mathcal{V}_{m-1})$, this can be further converted into checking whether p_m and p_i are adjacent in the Delaunay constructed from $\{p_i\}_{i=1}^m$, which is dual to $\mathcal{V}_m$. Algorithm 1 provides the pseudocode for the corresponding construction process.

Algorithm 1: Query Table Construction

```
Input: Point cloud P = {p_i}_{i=1}^n.
Output: Query Table {L_i}_{i=1}^n.
Initialize the Query Table L = {L_i}_{i=1}^n.
Insert p_1, initialize the Delaunay structure D.
foreach i in [2, n] do
    Initialize the set ω as empty.
    Insert p_i and update D.
    Extract all points adjacent to p_i, store them in ω.
    foreach p_x in ω do
        Push p_i to the back of L_x.
    end
end
```

B. Nearest Neighbor Query

In the query phase, we initialize the nearest site as p_1, which implies $\Phi_1(q) = p_1$. Let p_2 be the first site in p_1's Query List $\mathcal{L}_1$, and we visit $\mathcal{L}_1$ in order. If p_2 provides a smaller

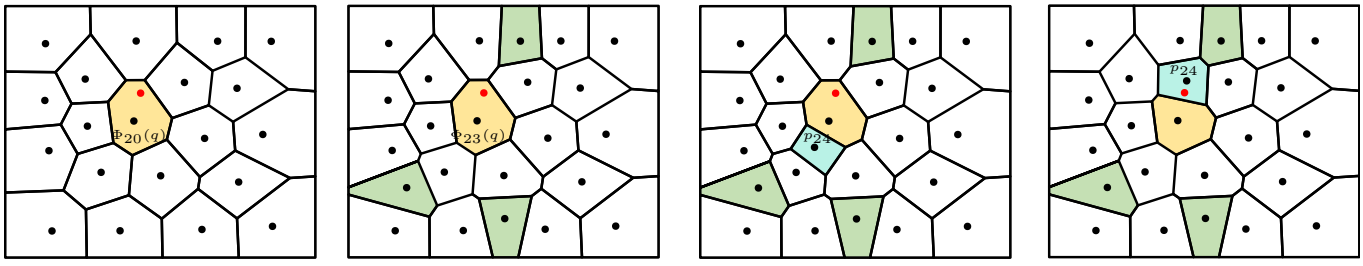


Fig. 1. (a) The Voronoi diagram of $\{p_i\}_{i=1}^{20}$, where the query point q (colored in red) belongs to the Voronoi cell colored in yellow. (b) If the addition of the three sites (whose cells are colored in green) does not change the yellow cell, $\Phi_m(q)$ remains unchanged when m is increased from 20 to 23. (b) When the 24-th site p_{24} is added (see the cyan cell), the yellow cell diminishes; yet $\Phi_{24}(q) = \Phi_{23}(q)$ if q is nearer to the old site. (d) $\Phi_{24}(q) = p_{24}$ if q is nearer to the newly added site p_{24}.

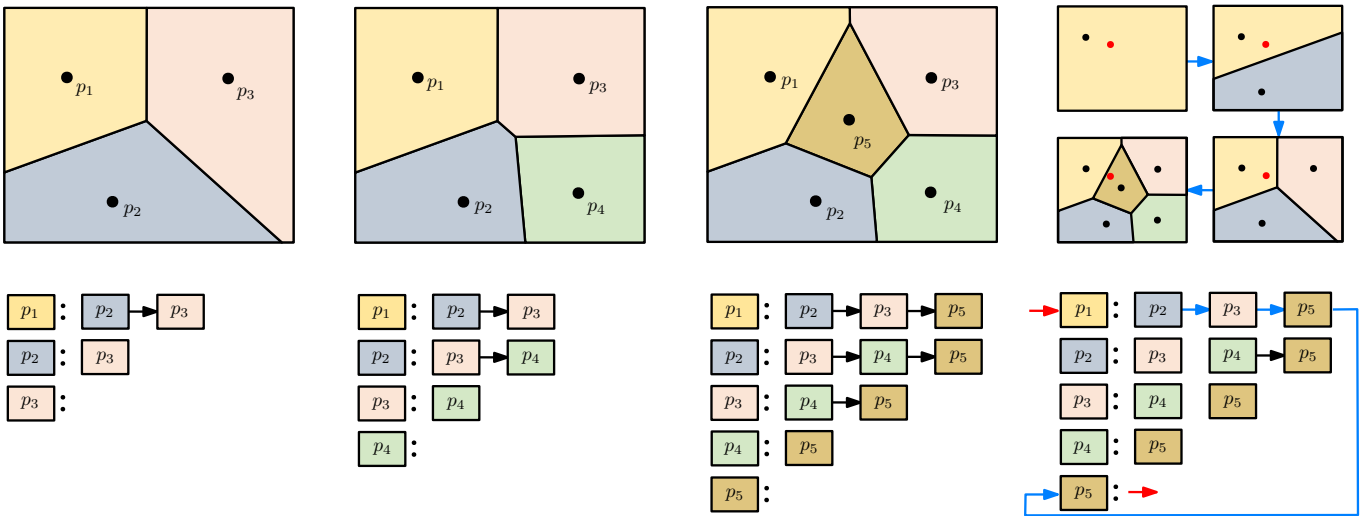


Fig. 2. We consider an incremental construction process of the Voronoi diagram of $\{p_i\}_{i=1}^{n}$. During the process, we append p_j to p_i's Query List if the addition of p_j changes p_i's cell. The rightmost figure shows the query process.

distance than p_1, we switch to p_2's Query List $\mathcal{L}_2$. Otherwise, we move to the next site in $\mathcal{L}_1$ to see if it can provide a smaller distance. See the rightmost figure of Figure 2 for how the query algorithm operates. See Algorithm 2 for the pseudo-code of the query algorithm.

Algorithm 2: Nearest Neighbor Search

Input: Query Table $\{\mathcal{L}_i\}_{i=1}^{n}$ and query point q.
Output: The closest point to q.
`getNearestNeighbor(`$p_I = p_1$`)`
foreach p *in* $\mathcal{L}_I$ **do**
  if $\|q - p\| < \|q - p_I\|$ **then**
    return `getNearestNeighbor(`p`)`
  end
end
return p_I

C. Farthest Point Sampling

Sections IV-A and IV-B describe the algorithm for constructing the Query Table and performing nearest neighbor searches. The algorithm assumes a specific order of the point cloud and incrementally constructs the Voronoi / Delaunay based on this order. When the point cloud is unordered or there is no need to compute the nearest point to q among the first n points, we can sort the points in a specific manner to enhance the efficiency of the query phase.

Intuitively, the more uniformly the inserted point cloud is distributed in space, the faster the query speed. This is because it facilitates a transition from large to small spatial jumps during the query process. Farthest point sampling, a commonly used downsampling method, aims to distribute the sampled points as uniformly as possible across the point cloud. Drawing inspiration from this, we set the number of sampled points equal to the total number of points in the cloud and use the sampling order to incrementally construct the Voronoi diagram. Notably, by integrating this approach into our construction process, we developed a farthest point sampling strategy with reduced complexity, leading to faster sampling times.

An important step in the farthest point sampling process involves calculating the distance between the newly sampled point and all remaining points, followed by updating the distance information. However, in most cases, the distances for the majority of the remaining points do not require updating, leading to a significant waste of computational resources. By integrating this process with our construction method, we can optimize the calculations by updating the distances for only a small subset of remaining points that are likely to change. This approach improves the algorithm's efficiency.

As an example, consider a state from the incremental

construction of the Voronoi diagram, as shown in Figure 3(a). The black points represent the points that have already been inserted, while the gray points denote the remaining points waiting to be inserted. When a new point is inserted based on the farthest point sampling strategy, as illustrated in Figure 3(b), the red point represents the newly sampled point from the gray points, and the green area corresponds to its Voronoi cell. A key observation is that only the distances of the remaining points located within the cells adjacent to the green cell (specifically, those within these cells prior to the insertion of the red point) need to be updated. Since the white Voronoi cell does not change during the insertion of new points, this implies that the nearest neighbors of the uninserted points within it remain unchanged. This significantly reduces the computational cost of sampling. The remaining task involves two parts: dynamically tracking the remaining points contained within the cells of the already inserted points during the incremental construction of the Voronoi diagram, and dynamically obtaining the next sampling point. The first part can be accomplished efficiently due to the local properties of Voronoi cells, while for the second part, we utilize a segment tree structure in our implementation.

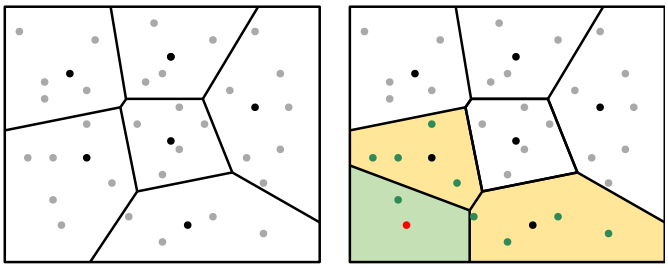


Fig. 3. Coupling of Farthest Point sorting with incremental Voronoi Construction. Left: Uninserted points (gray) are distributed within the Voronoi cells of the inserted points (black). Right: When a new point is inserted, only the distance of the uninserted points within the green and yellow cells may need to be updated.

D. Incremental Delaunay Construction

Currently, numerous incremental methods for Delaunay construction, such as parallel acceleration [24], [25] and presorting points [26], are commonly used to speed up the process. However, due to the specific requirements of farthest point sorting and the need to generate a query table during the construction phase in our approach, traditional acceleration techniques for Delaunay construction are not directly applicable.

During the incremental Delaunay construction process, one of the most time-consuming operations is locating the tetrahedron that contains a newly inserted point. According to Section IV-A and IV-B of our algorithm, when inserting point p_k, we have already constructed a query table for finding the nearest point within the set $\{p_i\}_{i=1}^{k-1}$. This query table stores information that allows for rapid identification of nearest points among the already inserted points. Leveraging this structure, we can quickly determine the nearest point to p_k. This nearest point then serves as the starting point for efficiently locating the tetrahedron containing p_k, significantly accelerating the Delaunay construction process.

E. Complexity Analysis

In this section, we will analyze the complexity of the farthest point sampling as a standalone component, rather than as part of the construction phase.

Time Complexity of Query Table Construction. The complexity of the Query Table Construction is the same as that of the incremental Delaunay construction. On average, the time complexity for constructing a Delaunay in 3D is $O(n \log n)$, while the worst-case time complexity is $O(n^2)$. However, this worst-case scenario is seldom encountered in typical construction processes.

Space Complexity of the Query Table. On average, the number of neighbors for a Voronoi cell in 3D is a constant, denoted as c. This implies that when a new point is inserted, it is adjacent to c cells on average, requiring the point to be appended to the end of c Query Lists. Therefore, the average space complexity of the Query Table is

$$O(\sum_{i=2}^{n} c) = c \times O(n) = O(n), \tag{6}$$

where n is the number of points.

Time Complexity of Nearest Neighbor Query. When querying the nearest neighbor for the point q, the complexity of the query phase depends on the number of times the cell containing q (which may change during the query process) shrinks during the incremental construction, as a comparison is needed each time the Voronoi cell containing q shrinks. Simply put, assuming $\Phi_m(q)$ is known and a new point p_{m+1} is inserted, the cell of p_{m+1} typically adjacent to c cells on average. Therefore, the probability that $\text{Cell}(\Phi_m(q); \mathcal{V}_m)$ shrinks is approximate c/m, indicating the possibility of needing a comparison. Thus, the average time complexity of the nearest neighbor query phase is

$$O(\sum_{i=1}^{n-1} \frac{c}{i}) = c \times O(\log n) = O(\log n). \tag{7}$$

Time Complexity of Farthest Point Sampling. When we have already inserted k points to construct the Voronoi diagram, each cell, on average, contains $(n-k)/k$ remaining points. Given the distances from all remaining points to the nearest sampled points are known, selecting the next point for sampling has a complexity of $O(\log(n-k))$. Knowing that the newly inserted point is adjacent to c cells, the number of remaining points that potentially need their distances updated is $c \times (n-k)/k$.

Thus, the additional complexity caused by the farthest point sampling process is:

$$\begin{aligned} &O(\sum_{k=1}^{n-1} \left[\log(n-k) + \left(c \times \frac{n-k}{k} \right) \right]) \\ &= O(\sum_{k=1}^{n-1} \log(k) + cn \times \sum_{k=1}^{n-1} \frac{1}{k} - (n-1) \times c) \\ &= O(n \log n) \end{aligned} \tag{8}$$

V. Results

Hardware environment. Our algorithm was implemented in C++ on a platform featuring a 3.4 GHz AMD Ryzen 9 5950X 16-Core CPU, 64GB of RAM, and running the Windows 11 operating system.

Experimental parameters. We utilized CGAL [19] for Delaunay computations. In our experiments, we used random sorting and farthest point sorting as insertion order. We measured and recorded both the construction performance and query performance for each method. The space for randomly generating query points was confined within the bounding box with 8x the volume (with edge lengths twice that of the original bounding box) of the input point cloud. The number of query points generated was set at 1000K.

Relevant approaches for comparison. Currently, there are numerous algorithms [12], [27]–[29] available for nearest neighbor search. In this paper, we selected two of the fastest and most classic methods for comparison.

1) KD Tree: We utilized the implementation from the nanoflann library [30], setting the maximum number of leaf nodes to 10. According to tests referenced in [31], this implementation is currently the fastest for nearest neighbor searches.
2) R Tree: We used the implementation from Boost Library [32].

In all the experiments below, Ours[1] refers to our method using a random order for incremental Delaunay construction, while Ours[2] indicates that the farthest point sorting strategy was applied.

A. Query Table

a) Length of the query list for a vertex: Recall that the Query Table maintains a list of points encroached upon during the incremental construction of the Delaunay for each vertex in V. It is evident that the Voronoi cells of points inserted earlier often shrink more times, resulting in longer Query Lists. Therefore, to investigate how the length of the Query List varies with different insertion indexes, we conducted a study using the 100K-vertex dragon point cloud as an example. As shown in Figure 4, we have plotted the length of the query list for each point with respect to its insertion order. Although the impact of random sorting and farthest point sorting strategies on the length of the Query List is not readily apparent from the figure, we calculated the average lengths of the Query Lists to be 15.684 and 14.165, with standard deviations of 17.705 and 16.491, respectively. The farthest point sorting resulted in a shorter and more uniform distribution of the Query List. Similar results were obtained across multiple data. This characteristic enables the query table generated through farthest point sorting to facilitate faster nearest-neighbor searches.

b) Number of tested vertices: As shown in Algorithm 2, the nearest neighbor search problem is transformed into the task of calculating distances between points.

The number of distance comparisons between points during the query process partially reflects the efficiency of the nearest neighbor search. We tested several classical models and recorded the number of comparisons, along with the mean and variance of the Query Lists, as shown in Table I. It can be observed that, after utilizing the farthest point sorting method, both the average length (Avg) and variance (Var) of the Query Lists decreased, accompanied by a reduction in the average number of comparisons during the nearest neighbor search process.

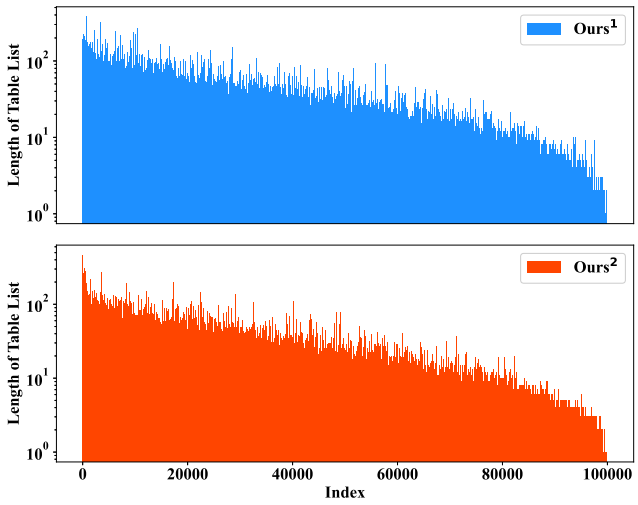


Fig. 4. The length of the Query List for different index points. Top: random sorting. Bottom: farthest point sorting.

TABLE I
THE AVERAGE (AVG) AND VARIANCE (VAR) OF THE LENGTH OF THE QUERY LIST AND THE NUMBER OF COMPARISONS REQUIRED PER QUERY ON CLASSICAL MODELS.

		Camel	Bunny	Dragon	Kitten	Armadillo	Lucy	Sponza
Vertices		28934	72911	100313	291023	726367	1018219	1313504
Ours[1]	Avg	15.913	15.805	15.684	15.397	15.832	16.034	14.631
	Var	17.911	17.264	17.705	18.834	17.631	17.968	20.431
	Tested	171.783	166.726	209.516	182.719	234.742	276.549	418.285
Ours[2]	Avg	14.414	14.345	14.165	13.964	14.370	14.478	13.067
	Var	16.783	15.906	16.491	17.126	16.172	16.584	19.681
	Tested	144.754	135.831	161.822	155.806	197.042	235.085	358.413

B. Preprocessing Cost

The preprocessing time of our algorithm is primarily concentrated in the Delaunay construction phase, as it involves significant computational overhead in building and managing the complex graph structure. In this section, we present the preprocessing times of different algorithms across several classic models in Table II. Additionally, Figure 5 illustrates the preprocessing time relative to the resolution of the Dragon point cloud, providing a comparative analysis between our method and others. Furthermore, Figures 9 and Figure 10 show the preprocessing time of different methods on subsets of the ABC dataset [33] and the Thingi10k dataset [34]. It can be observed that at this stage, our method currently exhibits longer preprocessing times compared to existing nearest neighbor search algorithms, particularly at higher resolutions, indicating areas that require further optimization.

TABLE II
PREPROCESSING TIME (s) ON CLASSICAL MODELS.

	Camel	Bunny	Dragon	Kitten	Armadillo	Lucy	Sponza
Vertices	28934	72911	100313	291023	726367	1018219	1313504
KD-tree	0.004	0.011	0.016	0.051	0.149	0.215	0.281
R-tree	0.005	0.014	0.022	0.062	0.170	0.243	0.311
Ours[1]	0.188	0.703	0.984	3.531	10.359	15.219	18.985
Ours[2]	0.265	0.765	1.406	4.782	15.157	22.953	26.938

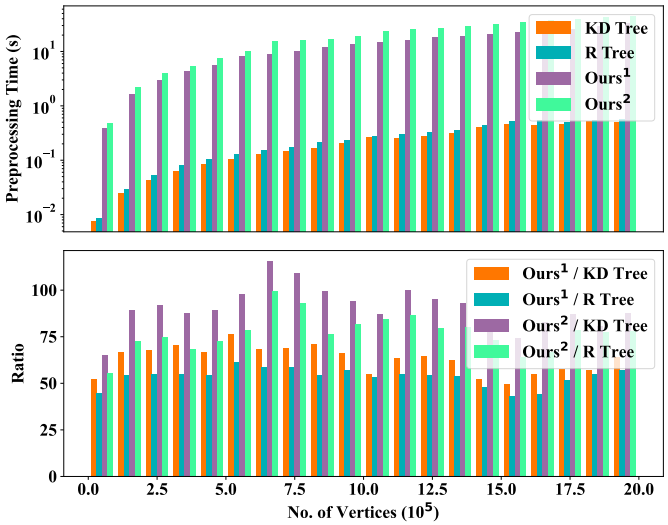


Fig. 5. The preprocessing cost of KD Tree, R Tree and Ours on the Dragon data with varying resolutions. Top: preprocessing time. Bottom: the comparison about the preprocessing cost.

C. Query Performance

Using the Dragon dataset as an example, we present the query cost of our method compared to other approaches with respect to resolution in Figure 8, demonstrating the significant advantage of our method in this scenario. In Table III, we provide more detailed examples for several classical models.

TABLE III
AVERAGE QUERY TIME (μs) OF KD TREE, R TREE, AND OURS ON CLASSICAL MODELS. THE **HIGHEST** AND SECOND-FASTEST QUERY TIMES ARE HIGHLIGHTED IN BOLD AND UNDERLINED, RESPECTIVELY.

	Camel	Bunny	Dragon	Kitten	Armadillo	Lucy	Sponza
Vertices	28934	72911	100313	291023	726367	1018219	1313504
KD Tree	1.639	3.795	2.475	8.475	12.81	9.796	4.407
R Tree	1.623	2.739	2.034	5.359	5.241	4.518	1.851
Ours[1]	0.423	0.559	0.540	0.957	1.132	1.293	2.146
Ours[2]	**0.344**	**0.482**	**0.388**	**0.752**	**0.834**	**1.070**	**1.694**

D. Tests on ABC & Thingi10K

Thingi10K [34] and ABC [33] are two large-scale 3D datasets that contain diverse models. To conduct a comprehensive comparison among KD Tree, R Tree, and Ours, we ran them on the dataset to compare both the preprocessing cost and the query cost. The query time are shown in Figures 9 and Figure 10. And the preprocessing time can be found in Figure 6 and Figure 7.

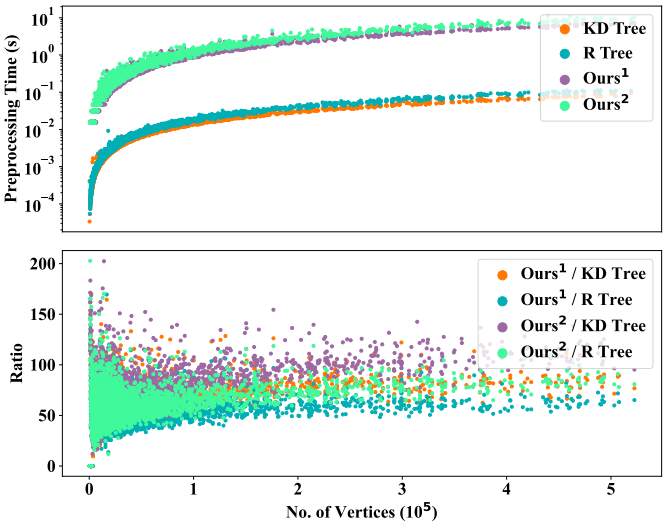


Fig. 6. Comparison about preprocessing cost on ABC dataset. Top: the time cost of preprocessing for KD Tree, R Tree and ours. Bottom: the comparison about the preprocessing cost among KD Tree, R Tree and ours.

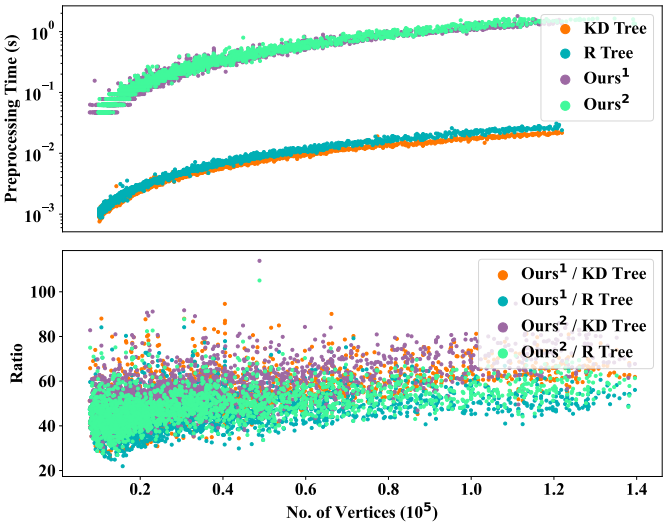


Fig. 7. Comparison about preprocessing cost on Thingi10K dataset. Top: the time cost of preprocessing for KD Tree, R Tree and ours. Bottom: the comparison about the preprocessing cost among KD Tree, R Tree and ours.

E. Other Types of Data

a) Uniform Point Cloud.: To fully demonstrate the efficiency of the algorithm, we compared KD Tree, R Tree, and Ours on uniform point clouds. As shown in Figure 11, it can be observed that even on completely uniform point clouds, our method remains comparable in query efficiency to the fastest current implementation of KD Tree and R Tree.

b) Earth City.: Additionally, we tested the efficiency of the KD Tree, R Tree, and our method on an Earth data, distributed over a sphere, with the results shown in Figure 12. At this distribution, both KD Tree and R Tree exhibit a significant drop in efficiency. However, our method remains extremely fast, achieving up to 40x the query speed of KD Tree.

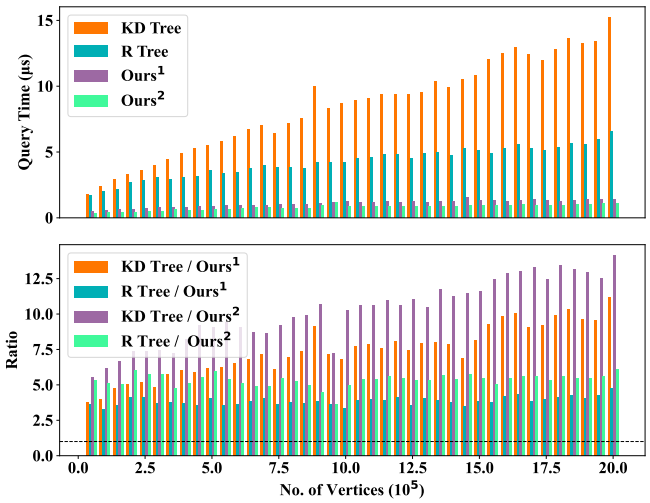


Fig. 8. Average query time on the Dragon model with varying resolutions. Top: Average query time (μs) among KD tree, R tree and ours. Bottom: The comparison about the average time different methods. Dashed line indicates where the ratio equals 1.

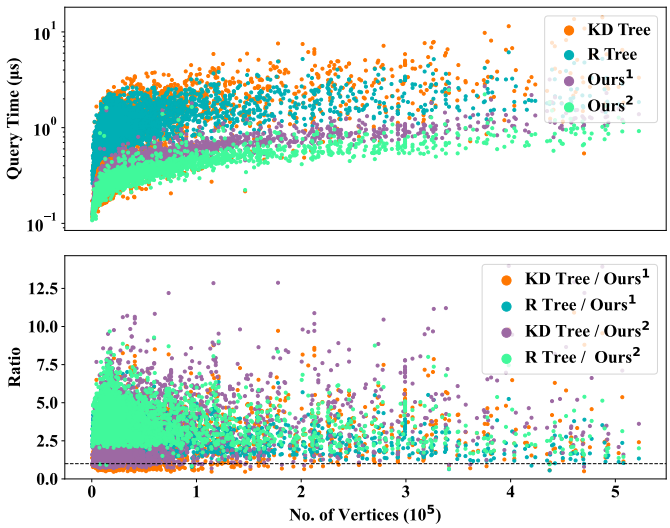


Fig. 9. Comparison about query performance on ABC dataset. Top: the average timing cost per query (μs) for KD Tree, R Tree and Ours. The y-axis is scaled logarithmically. Bottom: the comparison about the query cost among KD Tree, R Tree and Ours. Dashed line indicates where the ratio equals 1.

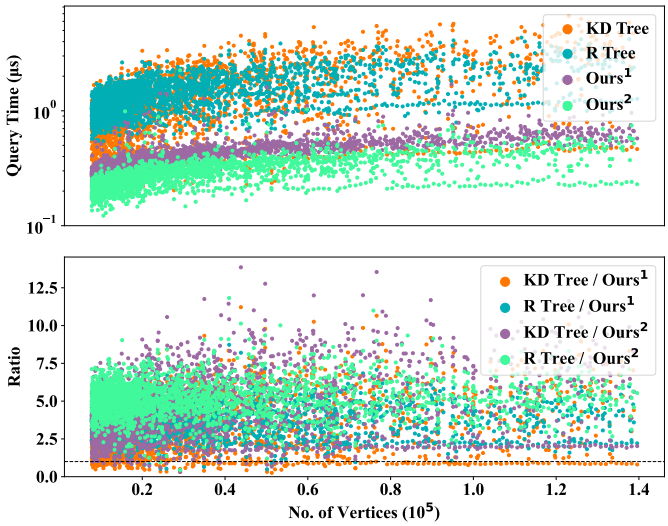


Fig. 10. Comparison about query performance on Thingi10K data. Top: the average timing cost per query (μs) for KD Tree, R Tree and Ours. The y-axis is scaled logarithmically. Bottom: the comparison about the query cost among KD Tree, R Tree and Ours. Dashed line indicates where the ratio equals 1.

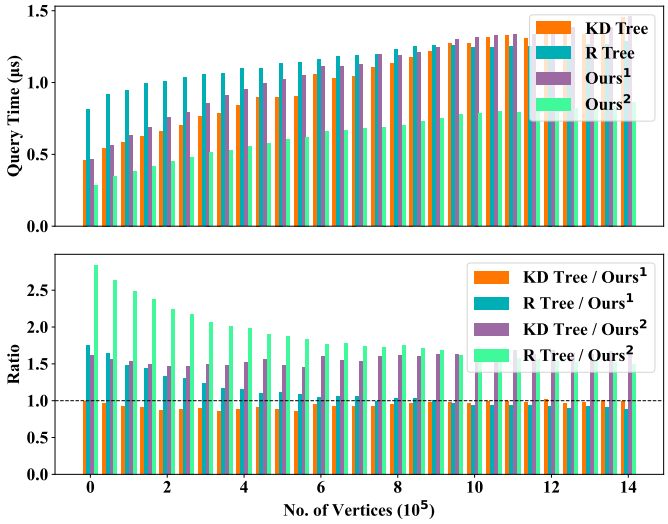


Fig. 11. Average query time on the uniform point cloud with varying resolutions. Top: Average query time per query (μs) among KD Tree, R Tree and Ours. Bottom: The comparison of the average time of different methods. Dashed line indicates where the ratio equals 1.

F. Near and far query points.

Additionally, we observed that the location of query points at different positions can impact the efficiency of the query. To further investigate this, we conducted tests on uniformly distributed point clouds, point clouds representing Earth cities, and the Dragon model, with the number of points in each point cloud controlled at 1,000K. The results are presented in Figure 13. The horizontal axis represents the ratio of the length of the generation space for the query points to the length of the bounding box of the target points along any coordinate axis. It can be observed that even in a completely uniform point cloud distribution, where the query point generation space matches the bounding box of the target points, our query speed remains comparable to that of the KD Tree. In other scenarios, our method demonstrates a significant advantage in query speed.

G. Farthest point sampling

QuickFPS [22] is currently the fastest farthest point sampling strategy, achieving performance speedups of 230.5× over traditional CPU implementations, 43.4× over traditional GPU implementations, and 12.2× over previous state-of-the-art point cloud accelerators. It divides the point cloud into multiple buckets based on spatial location, and utilizes advanced techniques such as merged computation and implicit computation to accelerate the farthest point sampling process. Although the method proposed in this paper is not specifically designed for farthest point sampling, it also demonstrates impressive speed in this task. Using several traditional models as examples, with the farthest point sampling rate set at 25%, the sampling times of QuickFPS (CPU Version) and the method proposed in this paper are shown in Table IV. Moreover, compared to the complexity of QuickFPS, our

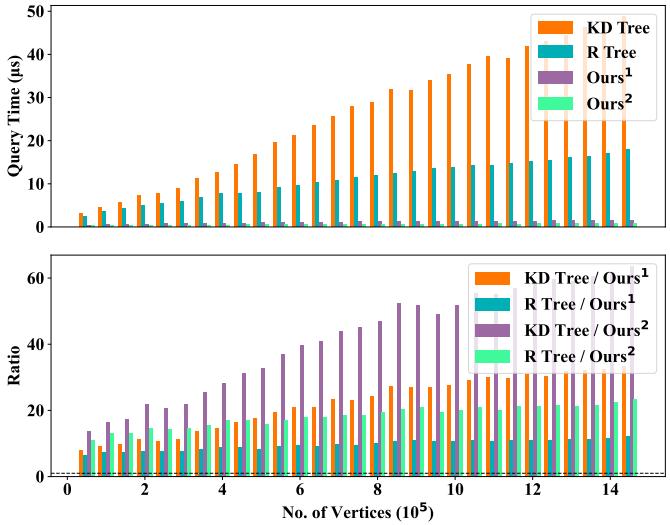


Fig. 12. Average query time on the earth data with varying resolutions. Top: Average query time among KD tree, R tree and ours. Bottom: The comparison about the average time different methods. Dashed line indicates where the ratio equals 1.

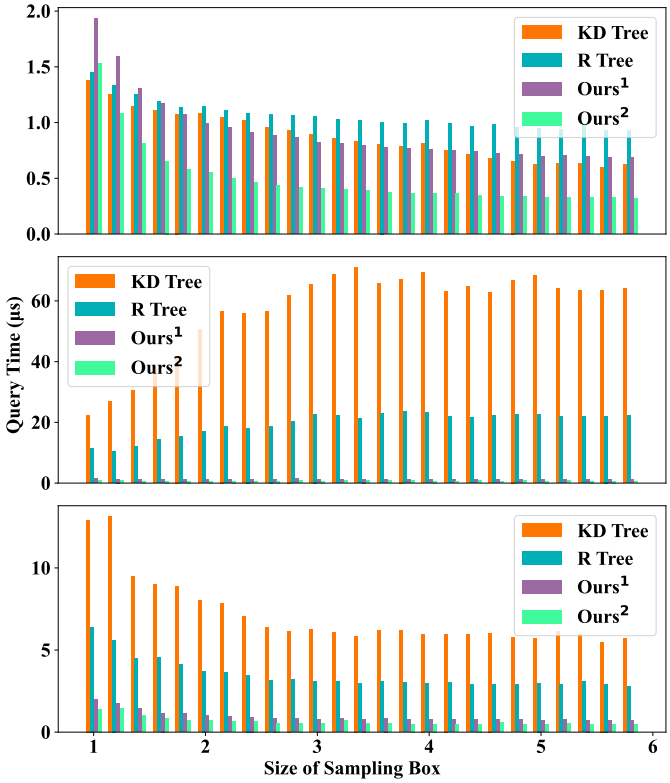


Fig. 13. Average query time with varying sizes of the sampling box (arbitrary axis ratio, with volume ratio calculated as the cube of the axis ratio). Top: Uniform point cloud. Middle: Earth city data. Bottom: Dragon point cloud.

proposed algorithm are both simpler and more intuitive.

VI. Desirable Properties

In light of its speed advantages, we believe that the current method harbors substantial potential for further advancements. In the following, we will delineate several pivotal aspects pertaining to its applications, aiming to elucidate the breadth and depth of its prospective utility in various domains.

TABLE IV
Comparison of Sampling Times (s) between QuickFPS and Ours at a 25% Sampling Rate.

	Camel	Bunny	Dragon	Kitten	Armadillo	Lucy	Sponza
Vertices	28934	72911	100313	291023	726367	1018219	1313504
QuickFPS	0.016	0.094	0.156	1.109	2.391	2.875	3.969
Ours	0.079	0.25	0.328	1.484	5.421	8.453	11.25

A. Nearest Neighbor Among the First k Points

One notable property of our algorithm is that, given any query point q and integer k, we can efficiently identify the nearest point among $\{p_i\}_{i=1}^{k}, k \leq n$. Specifically, the order of the given points is used to incrementally construct the Delaunay. As discussed in Section III, during the nearest-neighbor search process, when querying the nearest neighbor, we only need to compute $\Phi_n(q)$. However, in practice, $\forall\ i \in [1, n]$, $\Phi_i(q)$ is already incorporated into our query process. When a specific k is provided, as shown in Algorithm 2, our query process can terminate early. To the best of our knowledge, no existing algorithm is capable of achieving this.

The Density Peak Clustering algorithm [9] is a well-known method for clustering. One critical step in this algorithm is the calculation of σ_i, which is defined as:

$$\delta_i = \min_{j:\rho_j > \rho_i} (d_{ij}), \tag{9}$$

ρ_i is defined as the density of point p_i, and d_{ij} represents the distance between points p_i and p_j. Generally, computing $\{\delta_i\}_{i=1}^{n}$ requires calculating the distance between every pair of points, which has a complexity of $O(n^2)$. In contrast, our algorithm can efficiently solve this problem in $O(n \log n)$ time, assuming the data points are two-dimensional or three-dimensional. However, for higher-dimensional data, the time complexity of our algorithm increases due to the computationally expensive Delaunay triangulation process.

B. Farthest Point Sampling.

Farthest Point Sampling (FPS) is a widely used sampling algorithm because it ensures uniform point distribution, making it highly applicable across various tasks. For instance, FPS is used in the PointNet++ [35] framework for 3D point cloud deep learning to ensure uniform sampling and clustering of points, which contributes to building an effective receptive field. Although the initial purpose of our method was not specifically for farthest point sampling, it naturally lends itself to a simple and fast FPS strategy. Compared to existing methods, our proposed FPS strategy is more straightforward and easier to implement, offering an intuitive solution for efficient point sampling.

C. Various Primitives & Distance Metrics

Many existing nearest neighbor search algorithms are designed for specific types of primitives and distance metrics,

which limits their generalizability across diverse scenarios. In contrast, our method, as described in Sections III, IV-A, and IV-B, requires only the incremental construction of the Voronoi diagram for the primitives and a distance query function from any point in space to the primitives, making it straightforward to integrate into our nearest neighbor query framework. For instance, by supplying only the algorithm for incrementally constructing the Voronoi diagram of triangles under the Manhattan distance, along with a function to query the Manhattan distance between a spatial point and a triangle, our framework can efficiently determine the nearest primitive to a given point under the Manhattan distance. We believe this is a significant advantage of our algorithm. Of course, we acknowledge that the incremental construction of Voronoi diagrams for various primitives is quite challenging; however, as outlined below, we also provide alternative options.

D. Fast Preprocessing with Compromising Query Performance

In what follows, we aim to enhance the preprocessing efficiency at the expense of query performance. As elaborated in Section IV-A, during the construction of the Query Table, it is essential to identify the set ω_m upon inserting p_m, as illustrated in Algorithm 1. The set is defined by:

$$\omega_m = \{p_i \mid p_m \vdash \text{Cell}(p_i; \mathcal{V}_{m-1})\}. \tag{10}$$

Indeed, when confronted with complex high-dimensional or intricate scenarios, computing ω_m becomes exceedingly challenging. Consequently, we opt to compute ω_m^*, which merely needs to satisfy $\omega_m \subseteq \omega_m^*$, while still ensuring accurate nearest neighbor results. This approach opens up avenues for executing rapid nearest neighbor searches in high-dimensional spaces and diverse primitive and distance metric contexts. As an inevitable outcome, the computed set may harbor redundancies, thereby causing a marginal decrease in the speed of the nearest neighbor search.

Moreover, even if ω_m does not fall within ω_m^*, it could furnish valuable insights conducive to the development of approximate nearest neighbor search algorithms.

E. Fast Query with Elaborated Preprocessing

Another encouraging finding is that the number of comparisons in our query process, as shown in Table I, can be further reduced, suggesting potential for additional algorithmic acceleration. Specifically, as previously outlined in our algorithm, when we focus on the time point $\Phi_k(q) = p_k$ during the query process, the current approach only utilizes the information that $q \in \text{Cell}(\Phi_k(q); \mathcal{V}_k)$. However, we can infer that $q \in \cap_{\Phi_i(q)=p_i, 1<i\leq k} \text{Cell}(\Phi_{i-1}(q); \mathcal{V}_{i-1}) \cap \text{Cell}(\Phi_k(q); \mathcal{V}_k)$ from the query process.

As illustrated in Figure 14, (a) and (b) depict a simple incremental construction process, with the red dot representing the query point q. In this example, we have only utilized the information shown in (c), namely that $q \in \text{Cell}(p_3; \mathcal{V}_3)$. As shown in brown in (c). However, based on the query process, we can infer that $q \in \text{Cell}(p_1; \mathcal{V}_2) \cap \text{Cell}(p_3; \mathcal{V}_3)$, as is shown in (d). This enables us to design a more rigorous and precise evaluation criterion. Our method currently does not leverage this information during the query process, even though it can be obtained through complex preprocessing, which would, of course, involve substantial preprocessing overhead.

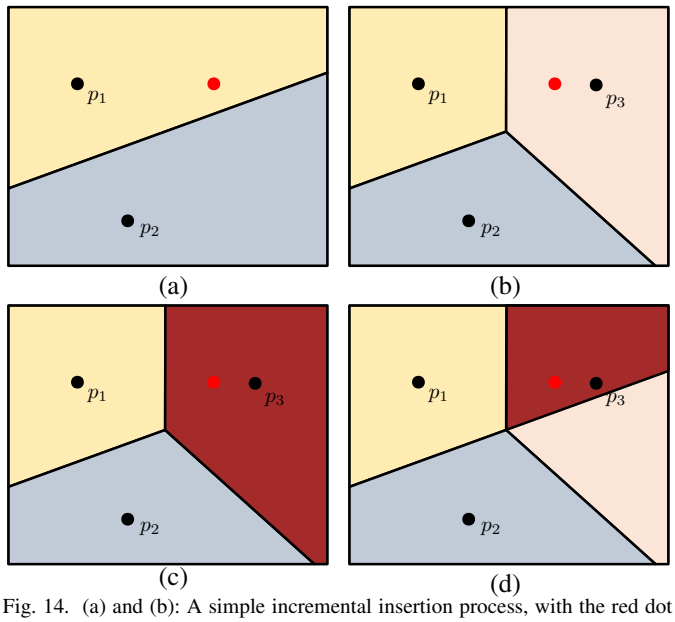


Fig. 14. (a) and (b): A simple incremental insertion process, with the red dot representing the query point q. (c) Based on the query process of (a)(b), we know the information $q \in \text{Cell}(p_3; \mathcal{V}_3)$, as shown in brown. (d) In fact, the complete information we can obtain is $q \in \text{Cell}(p_1; \mathcal{V}_2) \cap \text{Cell}(p_3; \mathcal{V}_3)$, as shown in brown.

However, we also recognize that utilizing all of the aforementioned information would result in significantly increased preprocessing complexity. To balance this, we implemented a simplified approach that compromises between the algorithm described in this paper and the information mentioned in this section. In brief, in addition to $q \in \text{Cell}(p_i; \mathcal{V}_i)$, we incorporated $q \in \text{Cell}(\Phi_{i-1}; \mathcal{V}_{i-1}) \cap \text{Cell}(p_i; \mathcal{V}_i)$. This region is used as the fundamental unit for occupancy evaluation, rather than a whole Voronoi cell.

We were pleasantly surprised to find that, despite implementing only a simplified version, we achieved a significant reduction in the number of comparisons compared to Table I. As shown in Table V, $Ours_*^1$ and $Ours_*^2$ represent the number of comparisons without and with farthest point sampling, respectively, under the strategy outlined in this section. Remarkably, only around 100 pairwise distance comparisons are required to find the exact nearest neighbor for a query point among 1 million points. However, we must acknowledge that while the number of comparisons has decreased, the actual query time has increased. We speculate that this may be due to increased memory usage by the data structure during the query phase, which likely results in more cache misses, thereby reducing efficiency. Nevertheless, we believe this is a highly meaningful direction for further exploration.

VII. Limitations and Future Work

The primary limitation of our algorithm is its high preprocessing cost, which restricts its applicability, particularly in large-scale scenarios or with high-dimensional datasets where efficiency is crucial. Therefore, in future work, we plan to

TABLE V
THE NUMBER OF PAIRWISE COMPARISONS DURING THE QUERY PROCESS USING DIFFERENT CONSTRUCTION STRATEGIES.

	Camel	Bunny	Dragon	Kitten	Armadillo	Lucy	Sponza
Vertices	28934	72911	100313	291023	726367	1018219	1313504
Ours^1	171.783	166.726	209.516	182.719	234.742	276.549	418.285
Ours^2	144.754	135.831	161.822	155.806	197.042	235.085	358.413
Ours$^1_*$	93.218	97.49	108.545	107.708	127.760	134.123	124.615
Ours$^2_*$	75.453	74.055	88.089	82.812	94.953	106.269	108.203

couple the construction of our algorithm more closely with Delaunay computations, or alternatively, develop an approximate Delaunay construction to achieve faster preprocessing. These approaches aim to reduce the preprocessing burden and improve scalability for large-scale and high-dimensional applications.

Additionally, our method currently only supports nearest neighbor search. In future work, we aim to extend the algorithm to support k-nearest neighbor queries, which would enhance its versatility. Moreover, addressing all the desirable properties outlined in Section VI remains a significant long-term goal for us.

VIII. CONCLUSION

In this paper, we observe that traditional nearest neighbor search algorithms exhibit decreased efficiency when dealing with point clouds distributed over 2D manifolds in 3D space, or when query points are distant from the target points. In certain special cases, the complexity can degrade to $O(n)$. Inspired by the incremental Delaunay construction process, we propose a novel nearest neighbor search algorithm specifically designed for 2D manifold point cloud data. Our method achieves a speedup of 1-10 times compared to the KD Tree approach. We validated the efficiency advantages of our approach through extensive experiments. The high performance of our algorithm is also supported by evidence showing that it requires only about 100 pairwise distance comparisons to identify the point closest to the query point from a set of 1 million points.

Additionally, our algorithm demonstrates significant potential. For instance, it supports querying the closest point to q among any of the first k points. This capability facilitates the application of peak clustering algorithms to large-scale data scenarios. Moreover, we have implemented a farthest point sampling algorithm with $O(k \log n)$ complexity in 3D space. We believe that the algorithm presented in this paper holds significant potential for application and will make a substantial contribution to the community.

REFERENCES

[1] R. Xu, Z. Wang, Z. Dou, C. Zong, S. Xin, M. Jiang, T. Ju, and C. Tu, "Rfeps: Reconstructing feature-line equipped polygonal surface," *ACM Transactions on Graphics (TOG)*, 2022.

[2] K. Pearson, "Liii. on lines and planes of closest fit to systems of points in space," *The London, Edinburgh, and Dublin philosophical magazine and journal of science*, vol. 2, no. 11, pp. 559–572, 1901.

[3] C. Zong, J. Xu, J. Song, S. Xin, S. Chen, W. Wang, and C. Tu, "P2m: A fast solver for querying distance from point to mesh surface," *ACM Transactions on Graphics (TOG)*, pp. 1–11, 2023.

[4] M. P. Corso, F. L. Perez, S. F. Stefenon, K.-C. Yow, R. García Ovejero, and V. R. Q. Leithardt, "Classification of contaminated insulators using k-nearest neighbors based on computer vision," *Computers*, vol. 10, no. 9, p. 112, 2021.

[5] T. Liu, C. Rosenberg, and H. A. Rowley, "Clustering billions of images with large scale nearest neighbor search," in *2007 IEEE workshop on applications of computer vision (WACV'07)*. IEEE, 2007, pp. 28–28.

[6] P. Quin, G. Paul, and D. Liu, "Experimental evaluation of nearest neighbor exploration approach in field environments," *IEEE Transactions on Automation Science and Engineering*, vol. 14, no. 2, pp. 869–880, 2017.

[7] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, p. 509–517, sep 1975. [Online]. Available: https://doi.org/10.1145/361002.361007

[8] A. Guttman, "R-trees: a dynamic index structure for spatial searching," in *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '84. New York, NY, USA: Association for Computing Machinery, 1984, p. 47–57. [Online]. Available: https://doi.org/10.1145/602259.602266

[9] A. Rodriguez and A. Laio, "Clustering by fast search and find of density peaks," *science*, vol. 344, no. 6191, pp. 1492–1496, 2014.

[10] M. Muja and D. G. Lowe, "Scalable nearest neighbor algorithms for high dimensional data," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 36, no. 11, pp. 2227–2240, 2014.

[11] J. Wang, J. Wang, G. Zeng, R. Gan, S. Li, and B. Guo, *Fast Neighborhood Graph Search Using Cartesian Concatenation*. Cham: Springer International Publishing, 2015, pp. 397–417. [Online]. Available: https://doi.org/10.1007/978-3-319-14998-1_18

[12] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 42, no. 4, p. 824–836, Apr. 2020. [Online]. Available: https://doi.org/10.1109/TPAMI.2018.2889473

[13] G. Ruiz, E. Chávez, M. Graff, and E. S. Téllez, "Finding near neighbors through local search," in *Similarity Search and Applications*, G. Amato, R. Connor, F. Falchi, and C. Gennaro, Eds. Cham: Springer International Publishing, 2015, pp. 103–109.

[14] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov, "Approximate nearest neighbor algorithm based on navigable small world graphs," *Information Systems*, vol. 45, pp. 61–68, 2014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0306437913001300

[15] ——, "Scalable distributed algorithm for approximate nearest neighbor search problem in high dimensional general metric spaces," in *Similarity Search and Applications*, G. Navarro and V. Pestov, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 132–147.

[16] N. Wang, B. Wang, W. Wang, and X. Guo, "Computing medial axis transform with feature preservation via restricted power diagram," *ACM Transactions on Graphics (Proc. SIGGRAPH Asia)*, vol. 41, no. 6, pp. 1–18, 2022.

[17] A. Bowyer, "Computing Dirichlet tessellations*," *The Computer Journal*, vol. 24, no. 2, pp. 162–166, 01 1981.

[18] H. Si, "Tetgen, a delaunay-based quality tetrahedral mesh generator," *ACM Trans. Math. Softw.*, vol. 41, no. 2, feb 2015. [Online]. Available: https://doi.org/10.1145/2629697

[19] S. Hert and M. Seel, "dD convex hulls and delaunay triangulations," in *CGAL User and Reference Manual*, 5.6.1 ed. CGAL Editorial Board, 2024. [Online]. Available: https://doc.cgal.org/5.6.1/Manual/packages.html#PkgConvexHullD

[20] C. R. Qi, L. Yi, H. Su, and L. J. Guibas, "Pointnet++: deep hierarchical feature learning on point sets in a metric space," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 5105–5114.

[21] J. Li, J. Zhou, Y. Xiong, X. Chen, and C. Chakrabarti, "An adjustable farthest point sampling method for approximately-sorted point cloud data," in *2022 IEEE Workshop on Signal Processing Systems (SiPS)*, 2022, pp. 1–6.

[22] M. Han, L. Wang, L. Xiao, H. Zhang, C. Zhang, X. Xu, and J. Zhu, "Quickfps: Architecture and algorithm co-design for farthest point sampling in large-scale point clouds," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 11, pp. 4011–4024, 2023.

[23] G. Voronoi, "Nouvelles applications des paramètres continus à la théorie des formes quadratiques. premier mémoire. sur quelques propriétés des formes quadratiques positives parfaites." *Journal für die reine und angewandte Mathematik (Crelles Journal)*, vol. 1908, no. 133, pp. 97–102, 1908. [Online]. Available: https://doi.org/10.1515/crll.1908.133.97

[24] S. Lo, "Parallel delaunay triangulation in three dimensions," *Computer Methods in Applied Mechanics and Engineering*, vol. 237-240, pp. 88–106, 2012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0045782512001570

[25] C. M. Nguyen and P. J. Rhodes, "Delaunay triangulation of large-scale datasets using two-level parallelism," *Parallel Computing*, vol. 98, p. 102672, 2020.

[26] K. Buchin and W. Mulzer, "Delaunay triangulations in o (sort (n)) time and more," *Journal of the ACM (JACM)*, vol. 58, no. 2, pp. 1–27, 2011.

[27] K. L. Cheung and A. W.-C. Fu, "Enhanced nearest neighbour search on the r-tree," *SIGMOD Rec.*, vol. 27, no. 3, p. 16–21, sep 1998. [Online]. Available: https://doi.org/10.1145/290593.290596

[28] A. Papadopoulos and Y. Manolopoulos, "Performance of nearest neighbor queries in r-trees," in *Database Theory — ICDT '97*, F. Afrati and P. Kolaitis, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 394–408.

[29] J. McNames, "A fast nearest-neighbor algorithm based on a principal axis search tree," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 23, no. 9, pp. 964–976, 2001.

[30] J. L. Blanco and P. K. Rai, "nanoflann: a C++ header-only fork of FLANN, a library for nearest neighbor (NN) with kd-trees," https://github.com/jlblancoc/nanoflann, 2014.

[31] J. Vermeulen, A. Hillebrand, and R. Geraerts, "A comparative study of k -nearest neighbour techniques in crowd simulation," *Computer Animation and Virtual Worlds*, vol. 28, p. e1775, 05 2017.

[32] Boost, "Boost C++ Libraries," http://www.boost.org/, 2015, last accessed 2015-06-30.

[33] S. Koch, A. Matveev, Z. Jiang, F. Williams, A. Artemov, E. Burnaev, M. Alexa, D. Zorin, and D. Panozzo, "Abc: A big cad model dataset for geometric deep learning," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

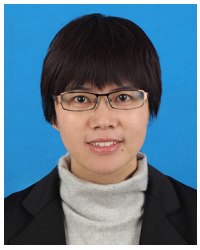
[34] Q. Zhou and A. Jacobson, "Thingi10k: A dataset of 10,000 3d-printing models," *arXiv preprint arXiv:1605.04797*, 2016.

[35] C. R. Qi, L. Yi, H. Su, and L. J. Guibas, "Pointnet++: Deep hierarchical feature learning on point sets in a metric space," *CoRR*, vol. abs/1706.02413, 2017. [Online]. Available: http://arxiv.org/abs/1706.02413

Pengfei Wang is currently working toward the PhD degree in computer science with Shandong University. His research interests include computational geometry, computer graphics, and computer-aided design.

Jiantao Song is currently pursuing a Ph.D. in computer science at Shandong University His research interests are computer graphics, computational geometry, point cloud reconstruction and geometric modeling.

Shiqing Xin is an associate professor at the School of Computer Science and Technology, Shandong University. He received his Ph.D. degree in applied mathematics from Zhejiang University. His research focuses on geometric calculation, geometric modeling, and scene understanding.



Shuangmin Chen received the PhD degree from Ningbo University in 2018. She is currently an assistant professor with the School of Information and Technology, Qingdao University of Science and Technology, China. She has authored or coauthored more than 40 research papers in famous journals and conferences. Her research interests include computer graphics and computational geometry

Changhe Tu received the BSc, MEng, and PhD degrees from Shandong University, China, in 1990, 1993, and 2003, respectively. He is currently a professor with the School of Computer Science and Technology, Shandong University, China. He currently leads the CG-VIS Group, Shandong University. He has authored or coauthored more than 100 papers in international journals and conferences. His research interests include computer graphics, 3D vision, and computer-aided geometric design.

Wenping Wang (Fellow, IEEE) received the PhD degree in computer science from the University of Alberta, in 1992. He is currently a chair professor of Computer Science with the University of Hong Kong. His research interests include computer graphics, computer visualization, computer vision, robotics, medical image processing, and geometric computing. He is associate editor of several premium journals, including the Computer Aided Geometric Design (CAGD), Computer Graphics Forum (CGF), IEEE Transactions on Computers, and IEEE Computer Graphics and Applications, and has chaired more than 20 international conferences, including Pacific Graphics 2012, ACM Symposium on Physical and Solid Modeling (SPM) 2013, SIGGRAPH Asia 2013, and Geometry Submit 2019. He received the John Gregory Memorial Award for his contributions in geometric modeling.

Jiaye Wang graduated from the Dept. of Mathematics of Shandong University in 1959. Now he is a Professor of Dept. of Computer Science and Technology of Shandong University. He has gone to the University of East Anglia, UK as an invited visitor in 1979, and has been a Senior Fellow at GINTIC Institute of Manufacturing Technology of Singapore from 1991. Prof. Wang has led several national, provincial and departmental projects, and published about 50 papers in domestic and foreign major publications and conferences in the related areas.