Journal of Algorithms and Computation

journal homepage: http://jac.ut.ac.ir

Efficient Preprocessing of 3D Data for Convex Hull Computation

M. Heydari*[1] and A. Khalifeh[2]

[1]Department of Computer Science, Khansar Campus, University of Isfahan, Iran
[2]Department of Mathematical and Physical Sciences, College of Arts and Sciences, University of Nizwa, Nizwa 616, Sultanate of Oman

ABSTRACT

Computing the convex hull of a set of points is a fundamental problem in computer science that has applications in various scientific and engineering domains. This paper presents a preprocessing algorithm, named Tiling, that can be utilized before any desired algorithm for computing the convex hull of a set of n points randomly distributed in $\mathbb{R}^3$ by uniform distribution. The key contributions of this work are threefold. First, we provide a complete preprocessing algorithm with detailed implementation. Second, we present rigorous experimental validation showing $2-2.6\times$ performance improvements over the widely used Qhull implementation when applied to uniformly distributed point sets in space. Third, our algorithm demonstrates the ability to eliminate approximately $95-97\%$ of input points while maintaining convex hull correctness, significantly reducing the computational burden for subsequent hull computation.

Keyword: Convex hull, Algorithm, Computational geometry, Uniform distribution.

ARTICLE INFO

Article history:
Research paper
Received 05, November 2025
Accepted 18, November 2025
Available online 04, December 2025

AMS subject Classification: 68U05, 68W40, 52B55.

*Corresponding author: M. Heydari. Email: m.heydari@khc.ui.ac.ir

1 Introduction

Given a set P of n points in 3D, the convex hull of the set P is the smallest convex polyhedron enclosing the set P. Computing the convex hull is a fundamental problem in various engineering and scientific applications, including image processing [4, 16], neuroscience [8], biology [5], pattern recognition [13] and it servers as a base for many geometric algorithms [17].

Computing the convex hull for a large set of points, particularly in 3D, is in general computationally expensive. Furthermore, most convex hull algorithms tend to achieve more efficient performance when applied to a smaller set of points rather than a larger one [11]. As a result, algorithms have been developed that incorporate a preprocessing stage. During this stage, some points that are guaranteed to lie inside the convex hull are discarded, while the remaining points are used as input for any convex hull algorithm. Therefore, a smaller set of points is derived, whose convex hull is identical to the convex hull of the input points set. The main rationale for adopting this approach is the considerably reduced computational cost associated with identifying and excluding points inside the convex hull compared to computing the convex hull for all input points. Additionally, in certain practical and engineering applications where the points come from a specific statistical distribution, such as Gaussian or uniform, it is possible to determine an upper bound on the expected number of the points residing on the convex hull. This is because, in such applications, the expected number of discarded points is related to the probability distribution of the points.

Despite the significant attention given to the convex hull problem and the numerous algorithms proposed for its construction, recent years have seen the development of novel algorithms aimed at enhancing the performance of existing ones. In some studies and research, attempts have been made to provide highly efficient algorithms by imposing certain constraints on the points under specific conditions. Akl and Toussaint [1] proposed one of the earliest algorithms for point elimination, which has a time complexity of $O(n \log n)$. However, their work does not provide a theoretical result that indicates the order of eliminated points. The work of Singh et al. [15] concentrated on a recursive point elimination algorithm for 2D points. The running time of their algorithm is $O(n \log n)$, but they did not present the order or percentage of eliminated points. In the Gomez [12] paper, an algorithm called TORCH is introduced for points in $\mathbb{R}^2$. TORCH is essentially a sorting algorithm that reduces geometric computations. In this algorithm, in the first stage, an approximate convex hull is constructed with several concave points, followed by the removal of these concave points in the second stage. The algorithm has a time complexity of $O(n \log n)$, and practical experiments have demonstrated that it is 1.17 times faster than the Quickhull algorithm. Oswaldo et al. proposed two preprocessing algorithms for a set of n points in $\mathbb{R}^2$ with a time complexity of $O(n)$ in [7] in 2016 and [6] in 2019. In the former, they assumed that the point coordinates are integers, and the points reside within a rectangular box defined by the dimensions $p \times q$. Furthermore, the performance of the algorithm in point removal varies proportionally to the ratio of $\min(p, q)$ to n. However, in their subsequent work [6], they improved their previous algo-

rithm. They removed the constraint of integer point coordinates, and through practical experiments and implementation, they demonstrated that the preprocessing phase effectively eliminates approximately 90% of the points. Ferrada et al. [10] introduced another preprocessing technique for reducing the number of points in $\mathbb{R}^2$. This technique effectively filters all points within an eight-vertex polygon in $O(n)$ time complexity. They demonstrated through experiments that if the points have a normal distribution, their algorithm performs 1.7 to 10 times faster than Quickhull and 10 times faster than Graham scan. As another preprocessing method for points in $\mathbb{R}^2$, one can refer to the work of Alshamrani et al. [2]. In their paper, they compared their algorithm to the original Graham scan method and showed that their algorithm executed up to 77 times faster.

The construction of a convex hull for random points has a long history, dating back to 1864 when it was introduced by Sylvester, followed by Erfon's study [9] in 1965. Later on, Bentley et al. [3] and Golin and Sedgewick [11] presented preprocessing algorithms for points uniformly distributed in $\mathbb{R}^2$, which eliminates all but $O(\sqrt{n})$ of points in $O(n)$. Recentry, Heydari et al. [14] improved the previous results and presented a preprocessing algorithm for points uniformly distributed in $\mathbb{R}^2$ that eliminates all but $O(\log n)$ of points in $O(n)$.

This paper aims to extend the previous work discussed in [14] by introducing a preprocessing algorithm, named Tiling, to eliminate interior points for a set of uniformly distributed points in $\mathbb{R}^3$. The proposed algorithm, called *Tiling*, first identifies in $O(n)$ time a subset of $O(\log n)$ points, referred to as *extreme* points. The convex hull of these points is then computed in $O(\log n \log\log n)$, and all points lying inside this hull are discarded. The remaining points, referred to as *candidate* points, are subsequently used for the computation of the final convex hull.

To validate the effectiveness of the algorithm, we implemented it in C++ and tested it on datasets containing more than one million uniformly distributed points in $\mathbb{R}^3$. The experimental results demonstrate that *Tiling* discards, on average, about $95-97\%$ of the input points, thereby reducing the size of the problem while preserving the correctness of the final convex hull. Moreover, the preprocessing significantly accelerates the overall computation, achieving runtime speedups of approximately 2–2.6$\times$ compared to the standard Qhull implementation. These findings highlight the potential of *Tiling* as a practical and efficient preprocessing step for large-scale convex hull problems. Additionally, this paper explores the generalization of this algorithm for points $\mathbb{R}^d$.

The rest of the paper is organized as follows: Section 2 describes preliminary concepts necessary throughout the paper. Section 3 introduces and analyzes the algorithm. Section 4 is devoted to study the problem in $\mathbb{R}^d$. Section 5 reports the experimental results and provides a comparison with the Quickhull algorithm, using the standard Qhull library implemented in C++ for this purpose. Finally, the paper concludes in Section 6.

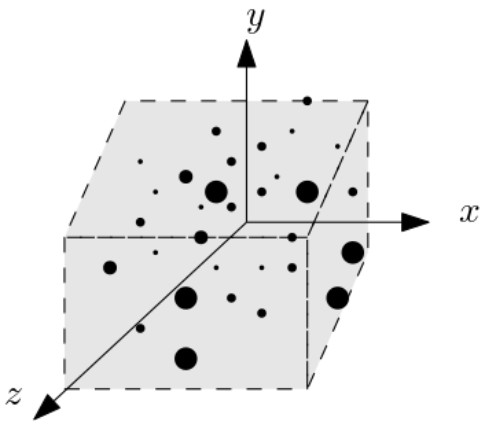

Figure 1: Illustration the bounding box of a set of points.

2 Preliminary Investigation

In this section, we will cover the essential lemmas for the entire paper. Throughout the paper, we will assume that the input set P consists of n points uniformly distributed in $\mathbb{R}^3$ across the x-y and z-coordinates. Let us define an extreme point within a set of points as a point whose x-, y-, or z-coordinate is the minimum or maximum value within the set. A bounding box of a set of points in $\mathbb{R}^3$ is a rectangular prism that completely encloses the set, aligned with the x, y, and z axes and defined by its minimum and maximum values along each axis (see Figure 1). In Figure 1, larger points are closer, and smaller points are farther away.

A *hypercuboid* is a generalization of a rectangle (in 2D) and a rectangular prism (in 3D) to any dimensional spaces, and is formally defined as the cartesian product of $d, d \geq 2$, closed intervals $[a_i, b_i] \subset \mathbb{R}$, where $a_i \leq b_i$, for $i = 1, 2, \cdots, d$.

In the following subsection, we introduce the concept of the Extremal Bounding Hypercuboid, which serves as a foundational building block for the proposed algorithm.

2.1 Extremal Bounding Hypercuboid

The Extremal Bounding Hypercuboid (EBH) for a set of points in d-dimensional space is defined as the axis-aligned bounding cuboid constructed using exactly d extreme points. Each selected point must represent an extreme value (either a maximum or a minimum) along a distinct coordinate axis. Importantly, when defining an EBH, only one extreme value must be chosen from each axis. This means that it is not permissible to select two points where one represents the maximum value and the other represents the minimum value along the same axis.

The process of constructing the EBH is both efficient and straightforward. First, we select the minimum and maximum values for each coordinate axis from the d extreme points. Once these extremal values are identified, the bounding cuboid is constructed by drawing hyperplanes at these minimum and maximum values, perpendicular to the respective

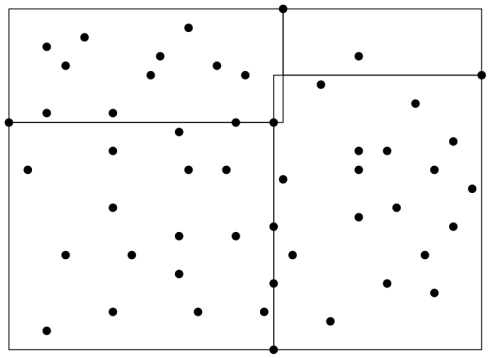

Figure 2: Illustration the EBH of a set of points in 2D.

coordinate axes. These hyperplanes define the boundaries of the EBH, effectively enclosing all the given points within an axis-aligned cuboid. Every hypercuboid in $\mathbb{R}^d$ has exactly 2^d EBHs. As an example, given three extreme points in 3D space: $p_1 = (3, 5, 7)$, $p_2 = (-1, 3, 5)$, and $p_3 = (4, -5, 4)$. The minimum and maximum values for each axis are:

- For x: min = -1, max = 4
- For y: min = -5, max = 5
- For z: min = 4, max = 7

The EBH is constructed by drawing hyperplanes at these values, resulting in an axis-aligned cuboid with boundaries at $x = -1$, $x = 4$, $y = -5$, $y = 5$, $z = 4$, and $z = 7$. For a clearer illustration, four possible EBHs for a set of points in 2D are shown in Figure 2.

Lemma 2.1. *Given d extreme points $p_1, \cdots, p_d$ and n input points in $\mathbb{R}^d$. The time complexity of detemining the EBH of p_is is $\Theta(1)$.*

Proof. Given that dimension d is a constant value, it is evident that the running time is $\Theta(1)$. □

Lemma 2.2. *Consider n points randomly distributed with a uniform distribution inside a 3D hypercuboid defined in the region $(0, a) \times (0, b) \times (0, c)$. If one point is randomly selected from each face of the hypercuboid, the expected number of points contained within each EBH is $n/8$.*

Proof. We prove for one EBH, specifically the EBH formed by the maximum of x, the maximum of y, and the minimum of z. The proof for other EBHs follows similarly. The boundaries of this EBH are determined by the following points

$$(0, 0, 0), (X, 0, 0), (X, Y, 0), (0, Y, 0), (0, 0, Z), (X, 0, Z), (X, Y, Z), (0, Y, Z),$$

where X, Y, and Z are random variables uniformly distributed over $(0, a)$, $(0, b)$, and $(0, c)$, respectively. The volume of this EBH, denoted by V_{trn}, is

$$V_{trn} = XYZ.$$

Since X, Y, and Z are independent random variables, we compute the expected value as follows

$$E[V_{trn}] = E[X]E[Y]E[Z].$$

For a uniform random variable U on (l, u), one has

$$E[U] = \frac{l+u}{2},$$

applying this property results

$$E[X] = \frac{a+0}{2},$$
$$E[Y] = \frac{b+0}{2},$$
$$E[Z] = \frac{c+0}{2}.$$

Therefor we obtain

$$E[V_{trn}] = \frac{abc}{8}.$$

Since the volume of the bounding box is abc and points are uniformly distributed, thus the expected number of points within the EBH is $\frac{n}{8}$. □

The next section will focus on the preprocessing algorithm, addressing its correctness and running time.

3 Overview of the Algorithm

The algorithm described in this section is essentially a preprocessing technique that for a given set P of n uniformly distributed points in $\mathbb{R}^3$, will select a set $C \subseteq P$ so that the convex hull of P and C are the same, and the expected size of C is $O(\log n)$. For any given set S of points in $\mathbb{R}^3$, we define the extreme points with maximum values for the y-coordinate, x-coordinate, and z-coordinate as p_t^S (top), p_r^S (right), and p_f^S (far), respectively. Similarly, the points with minimum values for the y-coordinate, x-coordinate, and z-coordinate are denoted as p_b^S (bottom), p_l^S (left), and p_n^S (near), respectively. We may omit the superscript S when the set is specific and no ambiguity arises. We refer to the subscripts b, t, r, l, f, and n as *plane index identifiers*. If there are multiple points with maximum or minimum values along any of the axes, one of them is arbitrarily selected. Furthermore, for simplicity and without loss of generality, we assume there are six unique extreme points, and no point exists with more than one coordinate being the maximum or minimum value. If a point violating such a condition is found, it is added to the set of extreme points, and the recursive procedure is repeated accordingly.
The algorithm constructs the set C of points recursively. Assuming there are six extreme points in the input set P, the algorithm places these six points inside C in the first step,

and forms the bounding box. It considers eight triplets of points: (p_t^P, p_r^P, p_n^P),(p_t^P, p_r^P, p_f^P), (p_t^P, p_l^P, p_n^P), (p_t^P, p_l^P, p_f^P), (p_b^P, p_r^P, p_n^P), (p_b^P, p_r^P, p_f^P), (p_b^P, p_l^P, p_n^P), (p_b^P, p_l^P, p_f^P) to create eight EBHs. We assign an abbreviated label to each of the eight EBHs, based on a plain index identifier derived from the extreme points defining it. For instance, the EBH defined by the top, right, and near points is labeled as *trn*, as shown in Figure 3.

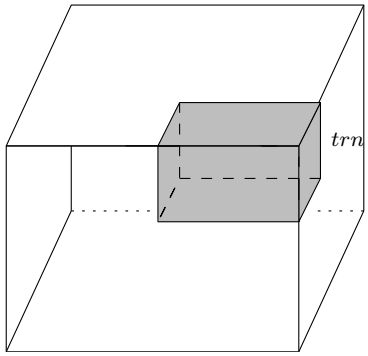

Figure 3: Illustration of an EBH.

In the next step, the algorithm identifies the points within each EBH. Then, the recursive process is initiated. For each set of these points, the algorithm locates three extreme points with plane index identifiers similar to their corrsponding EBH. These three extreme points are added to the set C. Then, the algorithm recursively identifies the extreme points for each EBH whose plane index identifiers match those of the EBH they belong to, adds them to C, and disregards the points outside the EBH. The Figure 4 shows the recursive procedure for an EBH.

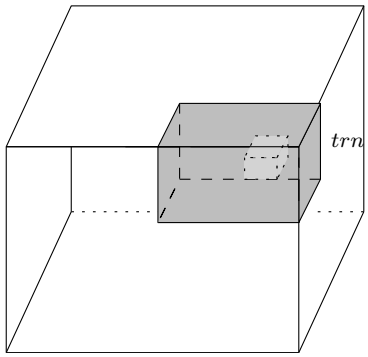

Figure 4: Illustration of recursive procedure in 3D.

For clarity, the algorithm is demonstrated on 2D points in Figure 5, as a 3D representation is less practical for visualization. The gray points represent the candidate points identified by the algorithm, and the dotted rectangles (EBHs) illustrate the recursive process at each step.

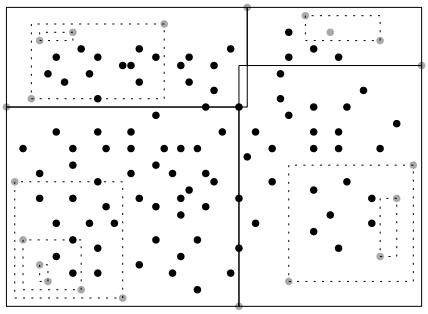

Figure 5: Illustration of the recursive procedure for 2D points.

The pseudo-code of the above procedure is given in Algorithm 1.
In what follows, we analyze this algorithm. In Theorem 3.1 the correctness of the algorithm is discussed, while Theorem 3.2 demonstrates that the expected number of elements in the set C is $O(\log n)$. Finally, in Theorem 3.3, we conclude this section by showing that the running time of the Tiling procedure is $O(n)$.

Theorem 3.1. *Let S be a set of points in $\mathbb{R}^3$, and let D be the set of points returned by the Tiling preprocessing algorithm. Then D contains all vertices of the convex hull of S.*

Proof. The algorithm only removes points that lie strictly inside the convex hull of intermediate subsets. All extreme points and candidate points that can appear on the final convex hull are retained throughout the recursive procedure. Therefore, every vertex of the original convex hull of S is included in D.

□

Theorem 3.2. *The expected size of the set C is $O(\log n)$*

Proof. Based on Lemma 2.2, the expected value for number of points to enter the next recursion for each EBH is asymptotically bounded by $O(n_r/8)$, where n_r is the number of points at the rth recurrence inside the EBH. Consequently, the expected number of iterations of the procedure for an EBH is $O(\log_8 n)$. Given that only 3 points are stored in each recursive iteration, the expected value of the total number of points saved is equal to

$$O(8 \times 3 \log_8 n) = O(\log n).$$

□

Lemma 3.3. *The Tiling procedure has time complexity of $O(n)$.*

Proof. The computational complexity of the procedure involves an initial step with a time complexity of $O(n)$, along with recursive steps at each EBH. For each EBH, the expected number of iterations is $O(\log_8 n)$. Finding extreme points and determining whether a point is inside the EBH takes $\Theta(n)$. Consequently, the running time for each recursion is as follows:

$$T(n) = T(n/8) + \Theta(n),$$

Algorithm 1 Tiling Algorithm

Input: Set S of n points

Output: Set C of points

IF $|S| \leq 3$ **THEN RETURN** S

Find extreme points $p_{k_1}, p_{k_2}, p_{k_3}, p_{k_4}, p_{k_5}, p_{k_6}$ with plane index identifiers t, b, l, r, f, n, respectively.

Let

S_1 : points in the EBH labeled trn, S_5 : points in the EBH labeled brn

S_2 : points in the EBH labeled tln, S_6 : points in the EBH labeled bln

S_3 : points in the EBH labeled trf, S_7 : points in the EBH labeled brf

S_4 : points in the EBH labeled tlf, S_8 : points in the EBH labeled blf

$C = \{p_{k_1}, p_{k_2}, p_{k_3}, p_{k_4}, p_{k_5}, p_{k_6}\}$

Tiling(S_1, "t", "r", "n")

Tiling(S_2, "t", "l", "n")

Tiling(S_3, "t", "r", "f")

Tiling(S_4, "t", "l", "f")

Tiling(S_5, "b", "r", "n")

Tiling(S_6, "b", "l", "n")

Tiling(S_7, "b", "r", "f")

Tiling(S_8, "b", "l", "f")

Let F = the convex hull of C

Return all points that are not strictly inside F

procedure TILING(S, pii_1, pii_2, pii_3)

    IF $|S| \leq 3$ **THEN**

        $C = C \cup S$

        RETURN

    Let p_i, p_j, p_k, $1 \leq i, j, k \leq n$ be the extreme points with plane index identifiers pii_1, pii_2, pii_3 in set S.

    $C = C \cup \{p_i, p_j, p_k\}$

    Let S =$\{$The points inside the EBH constructed by $p_i, p_j, p_k\}$

    Tiling(S, pii_1, pii_2, pii_3)

end procedure

which equals to $O(n)$. Therefore, the total running time is

$$T(n) = O(n) + O(n) = O(n).$$

□

In the next step, the algorithm constructs the convex hull F of the set C. Since the expected number of points in C is $O(\log n)$, the expected time complexity for constructing the convex hull is $O(\log n \log\log n)$.
In the next section, we will study the generalization of the algorithm to $\mathbb{R}^d$.

4 Algorithm Generalization to $\mathbb{R}^d$

In this section, we generalize the results obtained in the previous sections and study the algorithm in $\mathbb{R}^d$.

Lemma 4.1. *The volume of a hypercuboid in d-dimensional space with vertices $v_0, v_1, \cdots, v_d$ is given by*

$$|\det\begin{pmatrix} v_1 - v_0 & v_2 - v_0 & \cdots & v_d - v_0 \end{pmatrix}|,$$

where each column of the $d \times d$ determinant is a vector that points from vertex v_0 to another vertex v_d.

Lemma 4.2. *Given a d-hypercuboid with vertices $v_0 = (0, 0, \cdots, 0)$, $v_1 = (a_1, 0, \cdots, 0)$, $v_2 = (0, a_2, \cdots, 0)$, $\cdots$, and $v_d = (0, 0, \cdots, a_d)$. Let there be n uniformly distributed points in $\mathbb{R}^d$ inside it. Suppose d points $v_1^{'} = (A_1, 0, \cdots, 0)$, $v_2^{'} = (0, A_2, \cdots, 0)$, $\cdots$, and $v_d^{'} = (0, \cdots, 0, A_d)$ are chosen on each edge such that A_i is independent random variable following uniform distribution on the interval $(0, a_i)$. Then the expected number of points lying inside the d-hypercuboid $v_0 v_1^{'} \cdots v_d^{'}$ is $\dfrac{n}{2^d}$.*

Proof. The ratio of the volumes of d-hypercuboids $\Delta_d = v_0 v_1 \cdots v_d$ and $\Delta_d^{'} = v_0 v_1^{'} \cdots v_d^{'}$ is equal to the ratio of the determinants of these two, as presented in Lemma 4.1. Let us denote the matrix corresponding to vectors of Δ_d as

$$A = \begin{bmatrix} x_1 & x_2 & \cdots & x_d \\ x_{d+1} & x_{d+2} & \cdots & x_{2d} \\ \vdots & \vdots & \vdots & \vdots \\ x_{d^2-d+1} & x_{d^2-d+2} & \cdots & x_{d^2} \end{bmatrix},$$

where $x_i \in \mathbb{R}$, and the matrix corresponding to vectors of $\Delta_d^{'}$ as

$$B = \begin{bmatrix} X_1 & X_2 & \cdots & X_d \\ X_{d+1} & X_{d+2} & \cdots & X_{2d} \\ \vdots & \vdots & \vdots & \vdots \\ X_{d^2-d+1} & X_{d^2-d+2} & \cdots & X_{d^2} \end{bmatrix},$$

where X_is are uniform random variables such that

$$X_i \sim U(0, x_i).$$

We can write $X_{ij} = x_{ij}U_{ij}$ with U_{ij} uniform on $[0, 1]$ and independent. Therefore

$$\begin{aligned} E[\det(B)] &= E[\sum_{\sigma} \epsilon(\sigma) \prod_{i=1}^{d} U_{i,\sigma(i)} x_{i,\sigma(i)}] \\ &= \sum_{\sigma} \epsilon(\sigma) E[\prod_{i=1}^{d} U_{i,\sigma(i)}] \prod_{i=1}^{d} x_{i,\sigma(i)} \\ &= \frac{1}{2^d} \sum_{\sigma} \epsilon(\sigma) \prod_{i=1}^{d} x_{i,\sigma(i)} = \frac{1}{2^d} \det(A), \end{aligned}$$

where σ describes the whole set of of the $d!$ permutations of $\{1, 2, \cdots, d\}$ and $\epsilon(\sigma) = \pm 1$. Since the points are uniformly distributed, thus the expected number of points lying inside the $\Delta_d^{'}$ is $\frac{n}{2^d}$ and it completes the proof. □

The pseudo-code of the algorithm in $\mathbb{R}^d$ is based on the idea of Algorithm 1. First, we need to identify the extreme points in each dimension. Then, we form the EBHs and recursively find the corresponding extreme points within each EBH, storing them in a list C. The recursion is stopped when there are at most d points left at each EBH. Finally, the convex hull of C is computed, and all points lying strictly inside the hull are discarded.

Theorem 4.3. *The expected size of the set C is $O(\log n)$*

Proof. The hypercuboid in $\mathbb{R}^d$ has 2^d EBHs, where d is a fixed number. The procedure recursively finds d extreme points on each EBHs. Based on Lemma 4.2, the expected number of iterations for each EBH is $O(\log_{2^d} n)$. Therefore, the expected size of C is:

$$O(d \times 2^d \times \log_{2^d} n) = O(\log n).$$

□

Theorem 4.4. *The running time of the tiling procedure is $O(n)$.*

Proof. For each EBH, an upper bound on the expected number of iterations is $O(\log_{2^d} n)$. Finding extreme points takes $O(n)$, thus the total running time for recursive procedures is

$$T(n) = T(\frac{n}{2^d}) + O(n) = O(n).$$

□

4.1 Scalability and Performance as Dimensionality Increases

The scalability of the proposed algorithm is handled by the number of computational steps derived in Theorem 4.3:

$$S(d) = d \times 2^d \times \log_{2^d} n,$$

where $S(d)$ represents the computational steps for a given dimensionality d, and n is the size of the input. When d increases by 1, the number of steps becomes:

$$S(d+1) = (d+1) \times 2^{d+1} \times \log_{2^{d+1}} n.$$

The ratio of computational effort between dimensions $d+1$ and d can be expressed as:

$$\frac{S(d+1)}{S(d)} = \frac{(d+1) \times 2^{d+1} \times \log_{2^{d+1}} n}{d \times 2^d \times \log_{2^d} n} = \frac{2(d+1)d}{(d+1)d} = 2.$$

Thus, the theoretical expectation is that for an input size n, the computational cost approximately doubles when the dimensionality d increases by 1.

5 Experimental Results

The preprocessing algorithm was implemented in **C++**. The experimental evaluation was performed on uniformly distributed 3D point sets, with input sizes ranging up to 100 million points. The results, summarized in Table 1, demonstrate that the algorithm discards on average about **95–97%** of the input points, thereby significantly reducing the problem size while preserving all points on the final convex hull.

To compare efficiency, we measured the execution time of the proposed preprocessing algorithm combined with the Quickhull algorithm against the standard Quickhull algorithm applied directly to the entire input point set. For Quickhull, the standard `Qhull` library in C++ (see `http://www.qhull.org`) was employed. Table 1 reports the results for 3D uniformly distributed point sets of varying sizes. For each input size, the experiments were repeated 100 times with independently generated random points, and the average running time was recorded. The reported results correspond to a stable implementation of the algorithm. The experiments show that our preprocessing algorithm combined with Quickhull achieves a speedup of approximately 2–2.6× compared to running Quickhull alone on the entire dataset. The implementation of the proposed algorithm is publicly available at https://github.com/drmheydari/tiling-preprocessing.

6 Conclusion

In this paper, we presented an efficient preprocessing algorithm for the convex hull problem in three-dimensional space. The algorithm identifies a small subset of the input points whose convex hull is guaranteed to be identical to the convex hull of the entire input

#	Tiling + Quickhull (s)	Quickhull (s)	Speedup (x)	Pruning(%)
1,000,000	0.082815	0.191212	2.31x	95.5%
2,000,000	0.149860	0.299275	2.00x	94.97%
3,000,000	0.233400	0.484391	2.07x	95.8%
4,000,000	0.335705	0.773731	2.30x	95.3%
5,000,000	0.358603	0.808824	2.26x	96.1%
6,000,000	0.504307	1.209625	2.40x	96.1%
7,000,000	0.514367	1.280801	2.49x	96.0%
8,000,000	0.601844	1.319946	2.19x	95.8%
9,000,000	0.747324	1.864289	2.49x	96.0%
10,000,000	0.799999	1.885906	2.36x	95.94%
20,000,000	1.414985	3.293460	2.33x	97.1%
30,000,000	2.495870	5.596905	2.24x	96.5%
40,000,000	3.205192	6.810925	2.13x	95.2%
50,000,000	4.275561	11.231107	2.63x	95.4%
60,000,000	5.303428	11.008735	2.07x	95.32%
70,000,000	5.805455	12.239942	2.11x	96.2%
80,000,000	6.417896	13.174330	2.05x	95.9%
90,000,000	7.322517	18.128958	2.48x	94.7%
100,000,000	8.826596	19.691600	2.23x	96.3%

Table 1: Comparison of execution times, speedup ratio, and pruning percentages

set. This reduced subset can then be provided as input to any convex hull algorithm, substantially decreasing both the problem size and the running time. This work extends our earlier study in [14], which focused on two-dimensional point sets, and generalizes the approach to arbitrary dimension d.

We implemented the algorithm in C++ and conducted extensive experiments on large-scale datasets containing up to 100 million uniformly distributed points in $\mathbb{R}^3$. The results demonstrate that the algorithm discards, on average, about $95-97\%$ of the input points, while preserving the correctness of the convex hull. When combined with Quickhull, the preprocessing achieves a runtime speedup of approximately 2–2.6$\times$ compared to applying Quickhull alone. These findings highlight the practical efficiency of the proposed method and its potential as a scalable preprocessing step for convex hull computations in higher dimensions.

References

[1] Selim G. Akl and Godfried T. Toussaint. A fast convex hull algorithm. *Information Processing Letters*, 7(5):219–222, 1978.

[2] Reham Alshamrani, Fatimah Alshehri, and Heba Kurdi. A preprocessing technique for fast convex hull computation. *Procedia Computer Science*, 170:317–324, 2020.

The 11th International Conference on Ambient Systems, Networks and Technologies (ANT) / The 3rd International Conference on Emerging Data and Industry 4.0 (EDI40) / Affiliated Workshops.

[3] Jon Louis Bentley, Kenneth Lee Clarkson, and David B. Levine. Fast linear expected-time algorithms for computing maxima and convex hulls. *Algorithmica*, 9:168–183, 1993.

[4] Praveen Bhaniramka, Rephael Wenger, and Roger Crawfis. Isosurface construction in any dimension using convex hulls. *IEEE Transactions on Visualization and Computer Graphics*, 10(2):130–141, 2004.

[5] Fred L. Bookstein. *Morphometric Tools for Landmark Data: Geometry and Biology*. Cambridge University Press, 1992.

[6] José Oswaldo Cadenas and Graham M Megson. Preprocessing 2d data for fast convex hull computations. *PLoS One*, 14(2), 2019.

[7] José Oswaldo Cadenas, Graham M Megson, and Cris L Luengo Hendriks. Preconditioning 2d integer data for fast convex hull computations. *PLoS One*, 11(3), 2016.

[8] Barna Dudok, László Barna, Marco Ledri, Szilárd I. Szabó, Eszter Szabadits, Balázs Pintér, Stephen G. Woodhams, Christopher M. Henstridge, Gyula Y. Balla, Rita Nyilas, Csaba Varga, Sang-Hun Lee, Máté Matolcsi, Judit Cervenak, Imre Kacskovics, Masahiko Watanabe, Claudia Sagheddu, Miriam Melis, Marco Pistis, Ivan Soltesz, and István Katona. Cell-specific storm super-resolution imaging reveals nanoscale organization of cannabinoid signaling. *Nature Neuroscience*, 18, 2014.

[9] Bradley Efron. The convex hull of a random set of points. *Biometrika*, 52(3/4):331–343, 1965.

[10] Héctor Ferrada, Cristóbal A. Navarro, and Nancy Hitschfeld. A filtering technique for fast convex hull construction in $\mathbb{R}^2$. *Journal of Computational and Applied Mathematics*, 364:112298, 2020.

[11] Mordecai Golin and Robert Sedgewick. Analysis of a simple yet efficient convex hull algorithm. pages 153–163, 1988.

[12] Abel J.P. Gomes. A total order heuristic-based convex hull algorithm for points in the plane. *Computer-Aided Design*, 70:153–160, 2016. SPM 2015.

[13] C. Gope and Nasser Kehtarnavaz. Affine invariant comparison of point sets using convex hulls and hausdorff distances. *Pattern Recognition*, 40(1):309–320, 2007.

[14] Mohammad Heydari, Ashkan Khalifeh, and Laxmi Rathour. A simple and efficient preprocessing step for convex hull problem. *Discrete Mathematics, Algorithms and Applications*, 16(7):2350091, 2024.

[15] Prasenjit Das L. Dolendro Singh and Nirmalya Kar. A pre-processing algorithm for faster convex hull computation. In *Confluence 2013: The Next Generation Information Technology Summit (4th International Conference)*, pages 413–418, 2013.

[16] Apurba Sarkar, Arindam Biswasand, Mousumi Dutt, Partha Bhowmick, and Bhargab B.Bhattacharya. A linear-time algorithm to compute the triangular hull of a digital object. *Discrete Applied Mathematics*, 216(2):408–423, 2017.

[17] Steven Skiena. *The Algorithm Design Manual*. Springer, London, 2010.