


Article

Algorithmic Efficiency in Convex Hull Computation: Insights from 2D and 3D Implementations

Hyun Kwon ¹ , Sehong Oh ¹ and Jang-Woon Baek ^{2,*}
¹ Department of Artificial Intelligence and Data Science, Korea Military Academy, Seoul 01815, Republic of Korea; hkwon@kma.ac.kr (H.K.)

² Department of Architectural Engineering, Kyung Hee University, Gyeonggi 17104, Republic of Korea

* Correspondence: baekjw@khu.ac.kr; Tel.: +82-10-5185-1101

Abstract: This study examines various algorithms for computing the convex hull of a set of n points in a d -dimensional space. Convex hulls are fundamental in computational geometry and are applied in computer graphics, pattern recognition, and computational biology. Such convex hulls can also be useful in symmetry problems. For instance, when points are arranged symmetrically, the convex hull is also likely to be symmetrically shaped, which can be useful for object recognition in computer vision or pattern recognition. The focus is primarily on two-dimensional algorithms, including well-known methods like Gift Wrapping, Graham Scan, Divide and Conquer, QuickHull, TORCH, Kirkpatrick–Sediel, and Chan’s algorithms. These algorithms vary in terms of time complexity and scalability to higher dimensions. This study is extended to three-dimensional convex hull algorithms, such as NAW, randomized insertion, and parallelized versions, such as CudaHull and CudaChain. This study aimed to elucidate the operational principles, step-by-step procedures, and comparative time complexities of each algorithm. The implementation in Python facilitates a detailed comparison of the algorithmic performance through stepwise analysis and graphical outputs. The ultimate goal is to provide insights into the strengths and weaknesses of each algorithm under various scenarios, thereby offering a comprehensive guide for practical implementation.

Keywords: gift wrapping; Graham scan; divide and conquer; quickhull; TORCH; Kirkpatrick–Sediel; Chan’s algorithm



Citation: Kwon, H.; Oh, S.; Baek, J.-W. Algorithmic Efficiency in Convex Hull Computation: Insights from 2D and 3D Implementations. *Symmetry* **2024**, *16*, 1590. <https://doi.org/10.3390/sym16121590>

Academic Editor: Jie Yang

Received: 25 October 2024

Revised: 17 November 2024

Accepted: 20 November 2024

Published: 28 November 2024



Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Given n points in dimension d , the smallest convex set containing n points is called a convex hull. This convex hull [1–4] is composed of n points. To obtain it, various algorithms have different time complexities depending on the dimensions the algorithm can process and whether the algorithm is parallelized. Such convex hulls can also be useful in symmetry problems [5–7]. For instance, when points are arranged symmetrically, the convex hull is also likely to be symmetrically shaped, which can be useful for object recognition in computer vision [8,9] or pattern recognition [10–12].

Convex hull algorithms operating in two dimensions have been extensively studied since the 1970s; therefore, many algorithms are known. The Gift Wrapping algorithm is the simplest. It selects a point that is included in the convex hull. It sequentially determines the points following that point on the convex hull. This has the disadvantage of requiring $O(nh)$ time when the number of points on the convex hull is h ; therefore, it takes a long time if h is large, but it has the advantage of being simple to implement and scalable in multiple dimensions. Graham Scan [13], Divide and Conquer [14], QuickHull [15], and TORCH algorithms [16] take $O(n \log n)$ time and either use the divide-and-conquer method or sort the points in a way that can be processed sequentially and then processed. Each is implemented using different methods, and the Divide and Conquer and QuickHull algorithms have the advantage of being scalable in multiple dimensions. The Kirkpatrick–Sediel [17]

and Chan's algorithms [18] theoretically show the optimal time complexity of $O(n \log h)$. They commonly estimate h as small at first and then estimate h when it is judged to be larger than the expected value. One can use the method of increasing values by squaring.

Several convex hull algorithms run in 3D. In most cases, they are extensions of the 2D convex hull algorithm to 3D. In addition to the previously mentioned Gift Wrapping [19], Divide and Conquer [20], and QuickHull algorithms [21], there is the Newton Apple Wrapper (NAW) algorithm [22] and the randomized insertion algorithm [23]. The same algorithms exist, and the theoretical optimal time complexity for a non-output-sensitive algorithm is $O(n \log n)$. These algorithms can also be parallelized, and in the case of QuickHull, CudaHull [24], and Divide and Conquer [25], many parallelized versions exist. Also, there are specialized algorithms [26] that are premised on running on parallel processors.

This study contributes to the field by implementing and analyzing multiple convex hull algorithms to determine the most efficient approach. Additionally, the study provides a detailed evaluation of computational efficiency through direct implementation and analysis. It explores the components of various models and conducts a comparative analysis of different algorithms on diversely shaped datasets. This analysis includes meaningful insights into performance metrics, offering valuable information for understanding and selecting the most suitable algorithm for specific applications. This study aims to explain the operating principles of these algorithms, examine how they work step by step, compare the time complexities of each algorithm, and determine which algorithm shows the best performance in each situation. Each algorithm is implemented in Python to compare step-by-step algorithm execution times, and a graphical output is provided so that anyone can easily understand the operating principles of the algorithm.

The rest of the paper is organized as follows. Section 2 provides a detailed explanation of the 2D convex hull, including the implementation and experimental analysis of various methods. Section 3 focuses on the 3D convex hull, offering a detailed explanation, implementation, and experimental analysis of the methods. Section 4 presents a discussion of the proposed approach. Finally, Section 5 concludes the paper.

2. Two-Dimensional Convex Hull

2.1. Two-Dimensional Convex Hull Algorithm

2.1.1. Jarvis's March Algorithm

Jarvis's March algorithm [27], also known as the Gift Wrapping algorithm, is a method used to compute the convex hull of a set of points in a two-dimensional plane. The convex hull is defined as the smallest convex polygon that encloses all the given points. This algorithm follows a greedy approach, iteratively identifying the outermost points to form the boundary of the convex hull.

In Algorithm 1, the algorithm begins by taking as input a set of n points, denoted as $P = \{p_1, p_2, \dots, p_n\}$, which are scattered across the plane. The output of the algorithm is the convex hull H , a sequence of points that represent the vertices of the convex polygon enclosing all the input points, ordered counterclockwise.

Initially, the algorithm finds the leftmost point in the set P . This point is considered the starting point of the convex hull, denoted as p_{left} . The algorithm then sets this leftmost point as the current point, $p_{\text{current}} = p_{\text{left}}$, and initializes an empty list H to store the points that form the convex hull.

For each iteration, the algorithm adds the current point p_{current} to the list H since it is part of the convex boundary. The next step is to determine the next point that will be part of the hull. To perform this, the algorithm considers all points in the set P and checks which point is the most counterclockwise relative to the current point p_{current} . This check is performed by comparing the orientation of the current point with respect to all other points, selecting the point that is the most counterclockwise.

The comparison of orientations is conducted by calculating the cross product of vectors formed by the current point and the candidate points. If a point is more counterclockwise than the current candidate, it is selected as the next point. This ensures that the algorithm

moves along the boundary of the convex hull in a counterclockwise direction, progressively wrapping around the points that form the convex boundary.

Once the next point is identified, it becomes the new current point, and the process repeats. The algorithm continues to add points to the convex hull and selects the next most counterclockwise point until it returns to the leftmost point p_{left} , at which point the convex hull is complete. The final sequence of points stored in H represents the vertices of the convex polygon that enclose all the input points.

In conclusion, the Jarvis's March algorithm is a simple and intuitive approach to constructing the convex hull. While its time complexity is $O(nh)$, where n is the number of input points and h is the number of points on the convex hull, it is particularly efficient for smaller datasets or cases where the convex hull contains only a small number of points. The algorithm's simplicity makes it a valuable tool for solving the convex hull problem, especially when the number of boundary points is relatively small. The above steps are shown in Figure 1 below.

Algorithm 1 Jarvis's March Algorithm.

Input: A set of n points in the plane, $P = \{p_1, p_2, \dots, p_n\}$

Output: The convex hull H , a sequence of points ordered counterclockwise

Initialize an empty list H to store the convex hull points

Find the leftmost point $p_{\text{left}} \in P$. Set $p_{\text{current}} = p_{\text{left}}$

while $p_{\text{current}} \neq p_{\text{left}}$ **do**

 Add p_{current} to H

 Set $p_{\text{next}} = p_{\text{current}}$

for each point $p \in P$ **do**

if $p \neq p_{\text{current}}$ and p is more counterclockwise than p_{next} with respect to p_{current}

then

 Set $p_{\text{next}} = p$

end if

end for

 Set $p_{\text{current}} = p_{\text{next}}$

end while

Return H

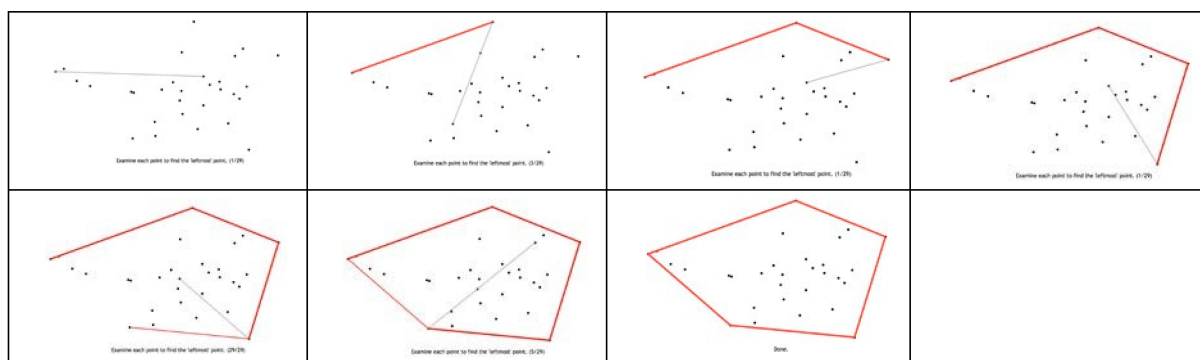


Figure 1. Implementation of the Jarvis's March algorithm.

When the number of points on the convex hull is h , it takes $O(n)$ time to calculate the angle formed by each point forming a convex hull with the remaining $n - 1$ points. Because the h points that constitute the convex hull repeat the process, the time complexity of Jarvis's March algorithm is $O(nh)$.

2.1.2. Graham Scan Algorithm

The Graham Scan algorithm [28] uses a geometric method called incremental construction in which the order of insertion is important. If points are inserted randomly, a separate procedure is required to check whether the point is inside or outside the hull. Therefore,

an insertion method is used to eliminate this procedure after sorting each point in order of angle from largest to smallest, based on the point with the smallest y -axis.

In Algorithm 2, the algorithm starts by taking as input a set of n points, denoted as $P = \{p_1, p_2, \dots, p_n\}$, and the goal is to output the convex hull H , a sequence of points ordered counterclockwise.

Initially, the algorithm finds the point with the smallest y coordinate, denoted as p_{\min} , and sets it as the starting point. The algorithm then sorts the remaining points based on the angle they make with the point p_{\min} , from smallest to largest. This sorting step takes $O(n \log n)$ time.

After sorting, the algorithm proceeds by iterating over the sorted points and maintaining a stack. As each point is processed, the algorithm ensures that the turns made between consecutive points are always left turns. If a right turn is detected, the previous point is removed from the stack, ensuring that only left turns are kept, thus maintaining the convex boundary.

Once all points have been processed, the points remaining in the stack form the vertices of the convex hull. The final sequence of points is outputted as the convex hull, ordered counterclockwise.

In conclusion, the Graham Scan algorithm is an efficient method for finding the convex hull, with a time complexity of $O(n \log n)$ due to the sorting step. It is particularly effective for larger datasets, as it avoids the need for checking inside or outside the hull for each point. The above steps are shown in Figure 2 below.

Algorithm 2 Graham Scan Algorithm.

Input: A set of n points in the plane, $P = \{p_1, p_2, \dots, p_n\}$

Output: The convex hull H , a sequence of points ordered counterclockwise

Find the point p_{\min} with the smallest y -coordinate in P

Sort the remaining points in increasing order of the angle with respect to p_{\min}

Initialize an empty stack S to store the points on the convex hull

for each point $p \in P$, starting from the sorted list **do**

While the size of S is greater than 1 and the turn from the second-to-last point in S to the last point in S and p is a right turn

pop the last point from S

 Push p onto the stack S

end for

Return the points in S , which represent the convex hull H

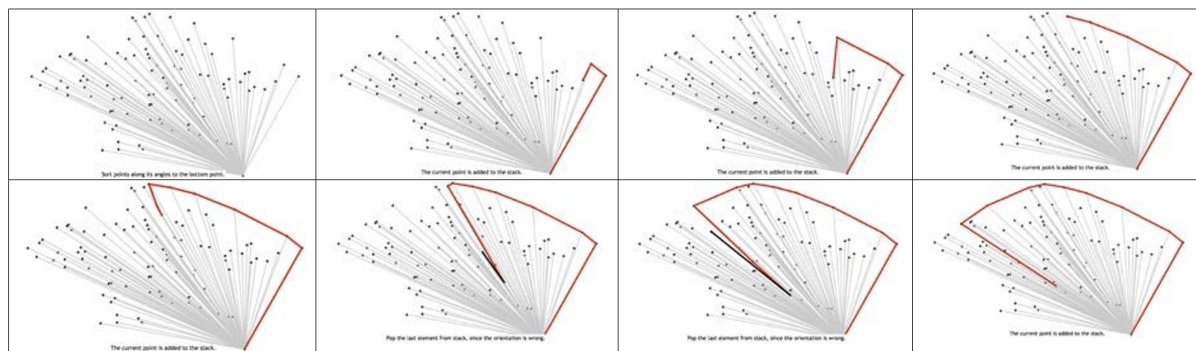


Figure 2. Implementation of the Graham Scan algorithm.

Calculating the time complexity takes $O(n \log n)$ time to sort the first point. It takes $O(1)$ time to perform an orientation test for each point after sorting. An origination test is performed on n points, and once a point is deleted, it is not deleted again; therefore, the maximum is $O(n)$. Therefore, the time complexity of Graham's algorithm is $O(n \log n) + O(n) = O(n \log n)$.

2.1.3. Divide and Conquer Algorithm

The Divide and Conquer algorithm [14] can be viewed as a generalization of the MergeSort sorting algorithm. A rough method involves first aligning each point along the x -axis. The left and right upper hulls are then calculated recursively after determining the median value of x for each sorted point set. The two upper hulls are then merged into one upper hull. The lower hull can be obtained using a similar method.

In Algorithm 3, the algorithm starts by taking as input a set of n points, denoted as $P = \{p_1, p_2, \dots, p_n\}$, and the goal is to output the convex hull H , a sequence of points ordered counterclockwise.

The algorithm begins by sorting all the points based on their x coordinates. The sorted points are then divided into two halves, and the convex hull is computed recursively for each half. After computing the upper hulls for both halves, the two hulls are merged into one upper hull. The lower hull is constructed similarly, and the final convex hull is obtained by combining the upper and lower hulls.

The divide-and-conquer approach is efficient because it reduces the problem size at each step and merges the solutions of smaller subproblems. The time complexity of this algorithm is $O(n \log n)$, which is the same as the time complexity of the MergeSort algorithm. The above steps are shown in Figure 3 below.

Algorithm 3 Divide and Conquer Algorithm.

Input: A set of n points in the plane, $P = \{p_1, p_2, \dots, p_n\}$

Output: The convex hull H , a sequence of points ordered counterclockwise

Sort the points in P based on their x -coordinates

Divide: Split the sorted points into two halves: P_{left} and P_{right}

Conquer:

Recursively compute the upper hull of P_{left} and P_{right}

Merge the upper hulls from P_{left} and P_{right} into one upper hull

Recursively compute the lower hull of P_{left} and P_{right}

Merge the lower hulls from P_{left} and P_{right} into one lower hull

Return the merged upper and lower hulls as the convex hull H

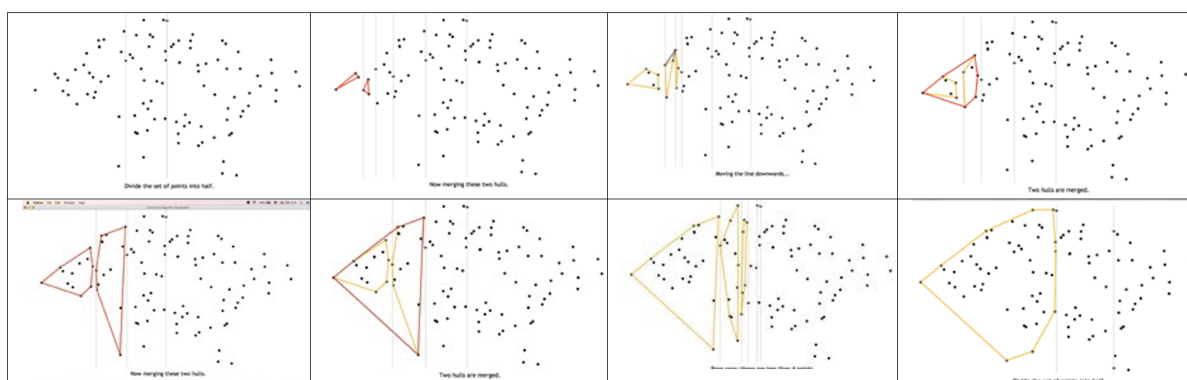


Figure 3. Implementation of the Divide and Conquer algorithm.

The Upper Tangent method is used for the merging procedure. In other words, the rightmost vertex (p) is found in the left upper hull, and the leftmost vertex (q) is found in the right upper hull. Using the orientation test, we fix the rightmost vertex (p) in the left upper hull and orient between the leftmost vertex and the next vertex in the right upper hull, moving the right upper hull vertex one step until it becomes clockwise. It is tested by pushing back individually. Similarly, based on the leftmost vertex in the right upper hull, we orient the rightmost vertex in the left upper hull and the vertices in the upper hull behind it and test by pushing the vertices in the left upper hull one step at a time until they turn counterclockwise. do. Using this method, the vertex is found

in the left upper hull, and the vertex is found in the right upper hull. Each operation is $O(1)$, and thus requires $O(n)$. Therefore, the entire method considers $O(n \log n)$ time as $T(n) = n + 2T(n/2)$. ($n > 3$)

2.1.4. Chan's Algorithm

Chan's algorithm [18] is an output-sensitive method for computing the convex hull of a set of points. This algorithm combines Graham Scan and Jarvis's March algorithms, aiming to achieve a more efficient time complexity. Graham Scan has a time complexity of $O(n \log n)$ because it starts by sorting all points. On the other hand, Jarvis's March algorithm works without sorting the points. By combining these two approaches, Chan's algorithm achieves a time complexity of $O(n \log h)$, where h represents the number of points on the convex hull.

In Algorithm 4, the process begins by dividing the set of n points into n/h subsets, where each subset has h points. The convex hull for each subset is calculated using Graham Scan, which takes $O(h \log h)$ time. The total time for calculating the convex hulls of all subsets is $O(n \log h)$. Since the value of h is not known initially, it needs to be predicted. If the predicted value h^* is too large, the overall time complexity will become $O(n \log n)$, but the algorithm ensures that h^* remains within the range $h \leq h^* \leq 2h$.

Once the mini-hulls are computed, they are merged into a single convex hull using Jarvis's March algorithm. In this step, the tangent is calculated using a binary search method in $O(\log h)$ time. Given that there are n/h mini-hulls, the total time for this merging step is $O(n \log h)$. Thus, the total time complexity for Chan's algorithm is $O(n \log h)$.

In summary, Chan's algorithm efficiently combines the strengths of the Graham Scan and Jarvis's March algorithms to compute the convex hull in $O(n \log h)$ time. The above steps are shown in Figure 4 below.

Algorithm 4 Chan's Algorithm.

Input: A set of n points in the plane, $P = \{p_1, p_2, \dots, p_n\}$

Output: The convex hull H , a sequence of points ordered counterclockwise

Divide the set P into n/h subsets, each containing h points

For each subset, calculate the convex hull using Graham's Scan in $O(h \log h)$ time

Merge the mini-hulls using Jarvis's March algorithm

Return H

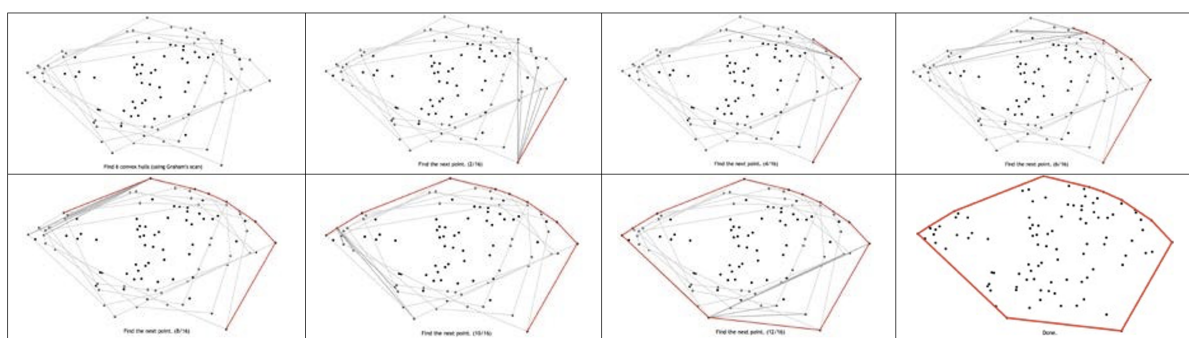


Figure 4. Implementation of Chan's algorithm.

2.1.5. QuickHull Algorithm

QuickHull [29] is a Divide and Conquer algorithm similar to Quicksort. This algorithm divides a set of points, ignores internal points, and recursively determines a convex hull only for external points. The algorithm operates by finding extreme points and progressively refining the convex hull.

In Algorithm 5, the algorithm begins by finding two points with the smallest and largest x coordinates. These two points form a line segment that is part of the convex hull. The remaining points are divided into two subsets based on which side of the line segment

they lie on. Next, for each subset, the algorithm identifies the point that is furthest from the line segment, which forms a triangle with the two points. This step is recursively applied to find external points for the convex hull.

The process continues by ignoring the points inside the triangle and recursively finding the point that is furthest from the newly drawn line segment. This procedure ensures that the points forming the boundary of the convex hull are identified. The algorithm stops when there are no more external points to be added to the convex hull, resulting in the final convex polygon.

The time complexity of QuickHull is $O(n \log n)$, similar to Quicksort. However, in cases where the point set has a special configuration and partitioning is inefficient during the recursive steps, the total recursive time can degrade to $O(n^2)$. In the worst case, the time complexity becomes $O(n^2)$, but the algorithm performs efficiently for general cases.

In conclusion, QuickHull is an efficient Divide and Conquer algorithm for computing the convex hull. Its average time complexity is $O(n \log n)$, making it faster than algorithms like Jarvis's March for larger datasets. The algorithm's efficiency, combined with its simplicity, makes it a popular choice for convex hull computations. The above steps are shown in Figure 5 below.

Algorithm 5 QuickHull Algorithm.

Input: A set of n points in the plane, $P = \{p_1, p_2, \dots, p_n\}$

Output: The convex hull H , a sequence of points ordered counterclockwise

Find the two points with the smallest and largest x-coordinates, p_{\min} and p_{\max}

Draw a line segment between p_{\min} and p_{\max}

Divide the remaining points into two subsets: points to the left and points to the right of the line segment

for each subset **do**

 Find the point p_{furthest} that is furthest from the line segment

 Form a triangle with p_{\min} , p_{\max} , p_{furthest}

 Recursively find the points that are furthest from the sides of the triangle and discard points inside the triangle

end for

Return H s

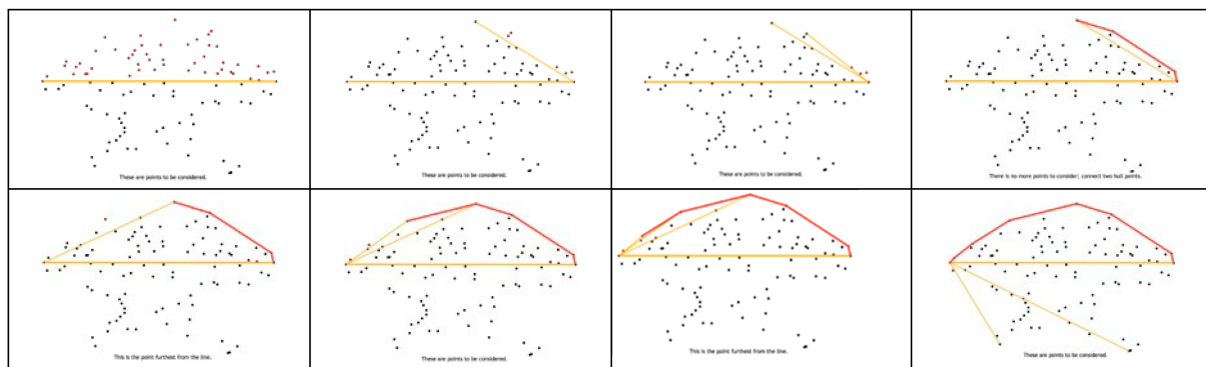


Figure 5. Implementation of the QuickHull algorithm.

2.1.6. Kirkpatrick–Seidel

The Kirkpatrick–Seidel algorithm [17], also known as the “ultimate planar convex hull algorithm”, employs a divide-and-conquer strategy and is recognized for its simplicity and optimal performance in two-dimensional convex hull computations. The Kirkpatrick–Seidel algorithm is a divide-and-conquer method used to compute the convex hull of a set of points in a two-dimensional plane. Unlike traditional Divide and Conquer algorithms that first compute the convex hull of two halves of the point set and then combine them,

this algorithm first finds the edge of the convex hull connecting the two halves before computing the convex hull for both sets.

In Algorithm 6, the algorithm begins by taking as input a set of n points, denoted as $P = \{p_1, p_2, \dots, p_n\}$. The output of the algorithm is the convex hull H , a sequence of points that represent the vertices of the convex polygon enclosing all the input points, ordered counterclockwise.

The algorithm follows these main steps:

- First, the set of points is divided into two halves based on the middle value of the x coordinate.
- Then, the tangent of the upper convex hull between the two sets is found.
- Points below the tangent line are ignored.
- The process is repeated recursively for both the upper and lower convex hulls.

In conclusion, the Kirkpatrick–Seidel algorithm is efficient with a time complexity of $O(n \log n)$, where n is the number of input points. It is particularly useful for large datasets due to its divide-and-conquer approach and efficient handling of the convex hull's edge detection. The algorithm is effective for computing the convex hull in a computationally efficient manner, especially when the number of boundary points is large.

Algorithm 6 Kirkpatrick–Seidel Algorithm.

Input: A set of n points in the plane, $P = \{p_1, p_2, \dots, p_n\}$

Output: The convex hull H , a sequence of points ordered counterclockwise

Divide the set P into two halves based on the middle value of the x -coordinate

Find the tangent of the upper convex hull between both sets

Ignore all points below the tangent line found in the previous step

Recursively apply the algorithm to the upper and lower sets

Return H

At this point, if process 2 takes $O(n)$ times three, this process is repeated $O(h)$ times, not $O(n)$, so a total of $O(n \log h)$ is required to implement this algorithm. It can be observed that this process requires time. In other words, like Chan's algorithm, this algorithm is also an optimal output-sensitive algorithm.

It is unclear how the process of finding the upper convex hull tangent between both sets can be performed in $O(n)$. However, it can be performed by considering the relationship between the slope of an arbitrary line and the upper supporting point of the set of points for that line. You can obtain this idea by doing so. When the slope of the straight line decreases from ∞ to $-\infty$, the upper supporting point moves from the extreme left to the extreme right. Therefore, for slope s of a straight line, if the upper supporting point is located to the left of the line dividing the set of points, the slope of the upper convex hull tangent must be less than or equal to s . This is because if the slope of the upper convex hull tangent is greater than s , one endpoint of the tangent cannot be located on the right. Similarly, if for a line's slope s the upper supporting point is located to the right of the line dividing the set of points, then the slope of the tangent to the upper convex hull is greater than or equal to s .

Therefore, if we randomly pair points and know that for a given slope s , the upper supporting point is located on either of the two point sets, the points on the left or right side of the pair among the paired points are the upper convex hulls. They could not be located along a tangential line. Here, if s is taken as the median of the slopes of randomly paired points, $1/4$ of the points are removed each time this process is performed, and because it takes $O(n)$ time to perform this process once, there are two total times that $O(n + (3/4)n + (3/4)2n + \dots) = O(n)$ is consumed until a point remains. This process can be written as Algorithm 7. The above steps are shown in Figure 6 below.

Algorithm 7 Upper Convex Hull Tangent Process.**Input:** A set of n points in the plane, $P = \{p_1, p_2, \dots, p_n\}$ **Output:** The upper convex hull tangent for the set of points

Randomly match the points in pairs. If only one point remains, leave it alone.

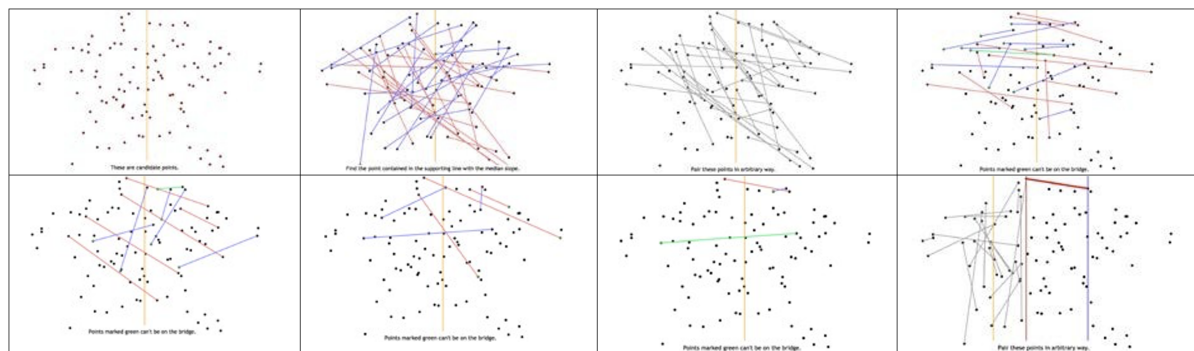
Find the median of the slopes of the pairs.

Find the upper supporting point for the median slope.

If the upper supporting point is to the left of the baseline, remove all pairs to the right with a slope greater than the median.

If the upper supporting point is to the right of the baseline, remove all pairs to the left with a slope smaller than the median.

Repeat steps 1 to 4 until only two points remain.

**Figure 6.** Implementation of the Kirkpatrick–Seidel algorithm.**2.1.7. TORCH Algorithm**

The total-order convex hull (TORCH) algorithm [30] refers to an algorithm developed for efficient computation of convex hulls in higher dimensions, making it particularly suitable for large datasets where computational efficiency is essential. The TORCH algorithm is a heuristic-based convex hull algorithm that efficiently determines the convex hull by focusing on the leftmost, rightmost, topmost, and bottommost points of a given set of points. This algorithm improves computational efficiency by constructing an approximate convex hull before performing the final convex hull calculation.

In Algorithm 8, the process begins by aligning all points based on the x-axis. This sorting ensures that the points are organized in sequential order for easier analysis. Once aligned, the algorithm identifies the rightmost and leftmost points of the set. Next, the lowest and highest points are also determined. These points serve as the corners of the convex hull.

Using the information obtained from the previous step, the TORCH algorithm generates four lateral hulls between the turning points. For instance, it calculates the lateral hull between the leftmost and topmost points by determining the range of x values from the leftmost point to the highest x value, and then drawing edges to connect points that increase along the y-axis until the highest point is reached. This method results in four lateral hulls being constructed.

These lateral hulls represent an approximate convex hull that encloses the points. Once the approximate convex hull is constructed, the algorithm converts it into a convex hull using the Jarvis's March algorithm. This final step efficiently calculates the true convex hull by focusing only on the points within the approximate convex hull, rather than all the original points.

The TORCH algorithm is particularly advantageous when dealing with larger datasets because it reduces the number of points that need to be processed to construct the convex hull. Although the time complexity remains $O(n \log n)$, the computational speed is faster compared to other 2D convex hull algorithms due to the preliminary approximation step. The above steps are shown in Figure 7 below.

Algorithm 8 TORCH Algorithm**Input:** A set of n points in the plane, $P = \{p_1, p_2, \dots, p_n\}$ **Output:** The convex hull H , a sequence of points ordered counterclockwise

Align all points based on the x-axis

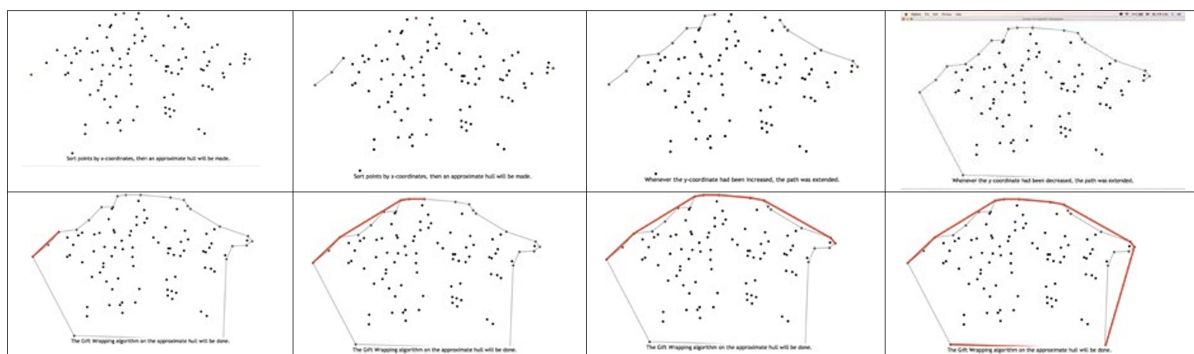
Find the rightmost point p_{right} and the leftmost point p_{left} Find the lowest point p_{bottom} and the highest point p_{top}

Construct four lateral hulls between the turning points

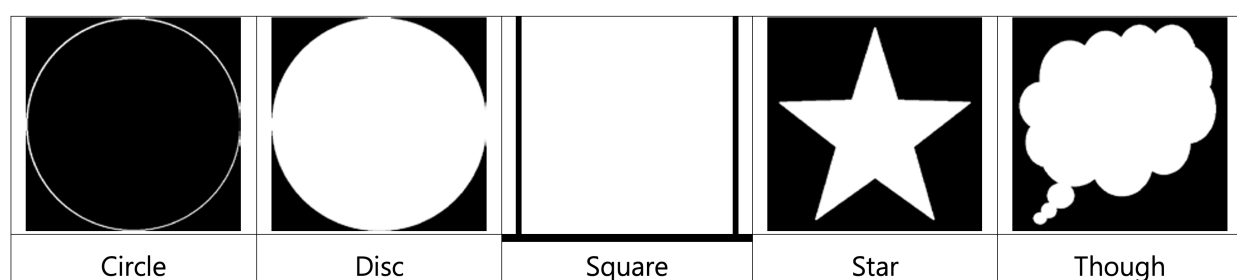
For each lateral hull, draw edges connecting points along the y-axis from the leftmost to the uppermost point

Construct an approximate convex hull by merging the four lateral hulls

Convert the approximate convex hull into a convex hull using the Jarvis's March algorithm

Return H **Figure 7.** Implementation of the TORCH algorithm.**2.2. Analysis of 2D Convex Hull****2.2.1. Environment**

The programming language was implemented using Python 3. There were five shapes for each dot: Circle, Disc, Square, Art, and Thought, as shown in Figure 8. In the experiment, the time taken was measured as the number of points increased by 1000 points from 1000 to 10,000 points.

**Figure 8.** Overview of the general case.**2.2.2. Result**

The complexities of the convex hull algorithms in a two-dimensional plane are compared below. Jarvis's March: $O(nh)$ (h : number of vertices), Graham Scan, Divide and Conquer, QuickHull; TORCH: $O(n \log n)$, Kirkpatrick–Seidel; Chan's algorithm: $O(n \log h)$. It appears to be the most optimal among the Kirkpatrick–Seidel and Chan algorithms. However, in actual programming, the experimental results for each algorithm differed from the theoretical complexity. Therefore, we analyzed the time required to calculate the convex hull for each algorithm using programming. The assumptions for the experiment are as follows:

1. As the number of points increases, the calculation time increases.
2. Each algorithm will have significant differences depending on the scattered shape of the given points.
 - A. The number of dots increases from 1000 to 10,000 in increments of 1000.
 - B. The following five types of point distribution are applied and we calculate the convex hull in each case. The time is measured.
 - (a) In the case of the Disc, Square, and Though, most of the points are placed on the inner white plane, and some of the points form a convex hull.
 - (b) In Circle, the points are placed only on the edges of the Circle, so most of the points form a convex hull. In contrast, in the Star, most of the points are placed on the inner white plane as follows, and the points are placed on the five most protruding vertices of the Star. Only five points are used to create the convex hull.

How much performance will each algorithm show, on average, in a general case like (a) (Disc, Square, Though), and what performance will each algorithm perform if there is a drastic difference in the results, as in (b)? This experiment aimed to investigate whether there were any differences in performance between the algorithms. First, the results for distribution type (a) were as follows.

Figure 9 illustrates the performance of convex hull algorithms when applied to general point distributions, specifically Disc, Square, and Though. The results highlight several critical observations regarding the practical performance of these algorithms. First, the TORCH algorithm consistently outperformed all others, achieving the shortest computation time across all tested point quantities. Its performance superiority became even more pronounced as the number of points increased. This finding underscores the practical efficiency of the TORCH algorithm, making it a strong candidate for real-world applications where computational efficiency is critical. Second, while algorithms like Kirkpatrick–Seidel (KS) and Chan’s algorithm are theoretically optimal with a complexity of $O(n \log h)$, their actual performance lagged behind simpler algorithms such as TORCH and Graham Scan. This discrepancy can be attributed to the constants hidden within their theoretical complexity, as well as the computational overhead introduced by their more intricate implementation details. Consequently, the theoretical advantage of these algorithms was not fully realized in practice, especially under general point distributions where the number of points forming the convex hull (h) is relatively small compared to the total number of points (n). Third, the characteristics of the point distributions significantly influenced the performance outcomes. In cases like Disc, Square, and Though, where most points are concentrated inside the convex hull, algorithms optimized for $h \ll n$ scenarios did not showcase their theoretical strengths. Instead, algorithms such as TORCH and Graham Scan, which rely more heavily on n -centered computations, exhibited better performance. Finally, the results reveal the importance of implementation efficiency. Despite the theoretical strengths of KS and Chan’s algorithm, the computational complexity of the functions used during their implementation had a substantial impact on their overall performance. This observation highlights the need to consider both theoretical complexity and practical implementation details when selecting algorithms for real-world use.

Figure 10 provides an analysis of convex hull algorithms under special point distributions, specifically Circle and Star, which offer distinct characteristics compared to general distributions. The results offer insights into the behavior of output-sensitive algorithms in these unique scenarios. In the case of the Circle distribution, where most points form the convex hull ($h \approx n$), algorithms such as Jarvis’s March, Chan’s algorithm, and KS exhibited the slowest computation times. These output-sensitive algorithms are designed to excel when $h \ll n$, but their computational overhead becomes a limitation when h approaches n . Conversely, algorithms like TORCH and Graham Scan demonstrated superior performance under this distribution, as their designs are better suited to scenarios where the convex hull encompasses a large portion of the points. In contrast, the Star distribution presents a situation where only a small number of points (h) form the convex hull, while the majority

are located inside it ($h \ll n$). In this scenario, output-sensitive algorithms such as Jarvis's March, Chan's algorithm, and KS capitalized on their design advantages, showing relatively better performance. However, TORCH continued to deliver competitive results, reinforcing its versatility and adaptability across varying distributions. These results underscore the strong relationship between point distribution characteristics and algorithm performance. Output-sensitive algorithms exhibit clear advantages in cases where the convex hull is formed by a small subset of the total points. However, in distributions where most points form the convex hull, their performance can be significantly hindered.

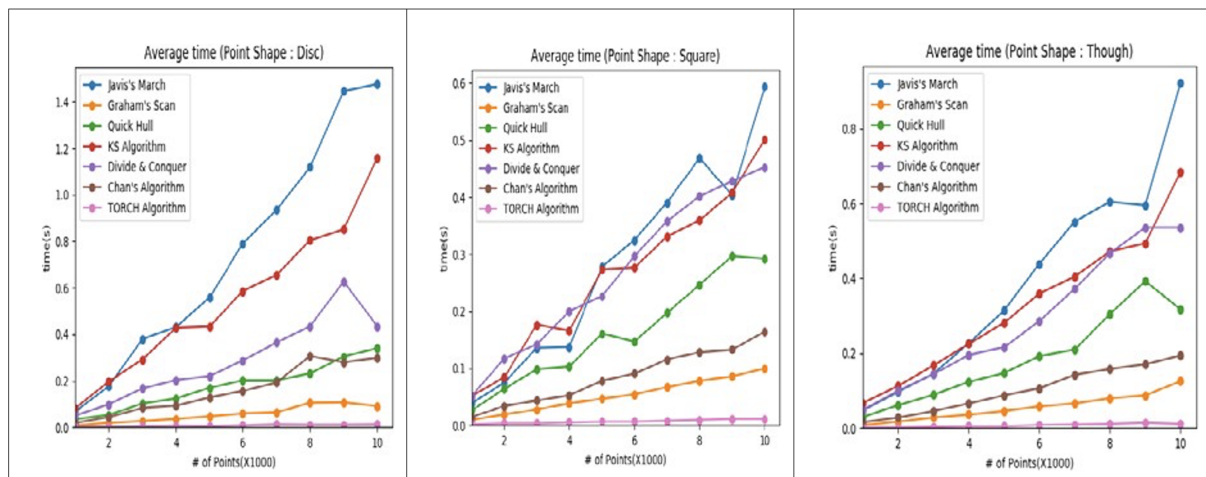


Figure 9. Average time for several 2D algorithms for each point (point shapes: Disc, Square, and Though).

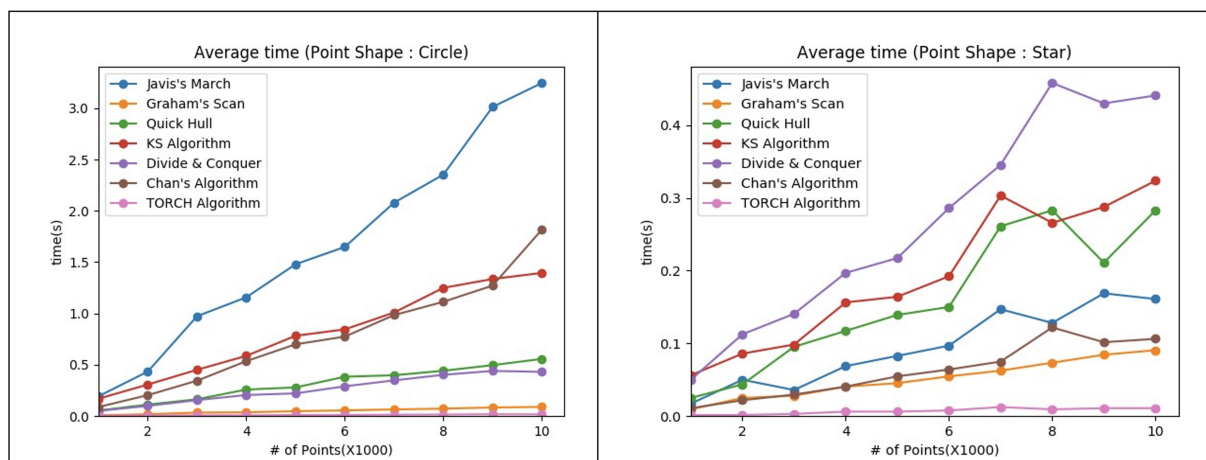


Figure 10. Average time for several 2D algorithms for each point (point shapes: Circle and Star).

3. Three-Dimensional Convex Hull Algorithms

As in the two-dimensional case, the three-dimensional convex hull is defined as the smallest convex set containing all given points. However, when calculating the convex hull in 3D, several points that are different from when calculating the convex hull in 2D must be considered. Most importantly, polyhedra in three dimensions are not uniquely determined, and reconstructing a convex hull from a set of points on a convex set is not a simple task. Therefore, the 3D convex hull algorithm must create a convex hull and return a set of lines and faces along with a set of points on the hull. In addition, the complexity of the convex hull, which is the number of lines and faces on the convex hull, must be considered when analyzing the convex hull algorithm. In 2D, a polygon with n vertices has n edges.

Therefore, it is not necessary to seriously consider the complexity of a convex hull. However, in a general d -dimensional case, the complexity of the convex hull can increase faster than linearly with respect to n , which affects the execution time of the convex hull algorithm. In fact, according to the upper bound theorem, the number of faces of the convex hull in the d dimension is $O(n^{\lfloor d/2 \rfloor})$, and when d is four or more, it becomes $O(n^2)$ or more; therefore, the time taken to calculate the convex hull is $O(n^2)$. The complexity of the convex hull governs this. Fortunately, in the 3D case, the complexity of the convex hull is linear, which means that it can be drawn in 2D without any edges intersecting the 3D convex hull. Thus, the 3D convex hull is a triangulation of n points in 2D. This can be seen from the fact that the number of edges and faces increases linearly with n . Therefore (because the 3D convex hull algorithm must be able to calculate the 2D convex hull), the fastest execution time is $O(n \log n)$; as we will see later, in the case of the Divide and Conquer and QuickHull algorithms, this complexity can be confirmed. In addition, the output-sensitive $O(n \log h)$ algorithm is known among the halfplane-intersection algorithms. The halfplane-intersection algorithm is the dual of the convex hull algorithm, so in the case of 3D, the output-sensitive algorithm is at least O as in 2D. It can be observed that it takes $(n \log h)$. In this study, we cover three 3D convex hull algorithms based on the 2D convex hull algorithm, the Jarvis's March [31], Divide and Conquer, and QuickHull algorithms; how these three algorithms are extended to 3D; and how 3D, unlike 2D, is used. We explain the additional work required for each algorithm.

3.1. Jarvis's March (Gift Wrapping) Algorithm

Similar to the 2D Gift Wrapping algorithm, the basic principle of the 3D Gift Wrapping algorithm is to expand the convex hull by side in Figure 11. However, unlike in the case of a 2D Circle where the 'next' edge can be easily found, in 3D, it is not intuitive to determine the direction, so the 2D algorithm cannot be directly applied to the 3D algorithm. The next step must be that you need to consider how to find it.

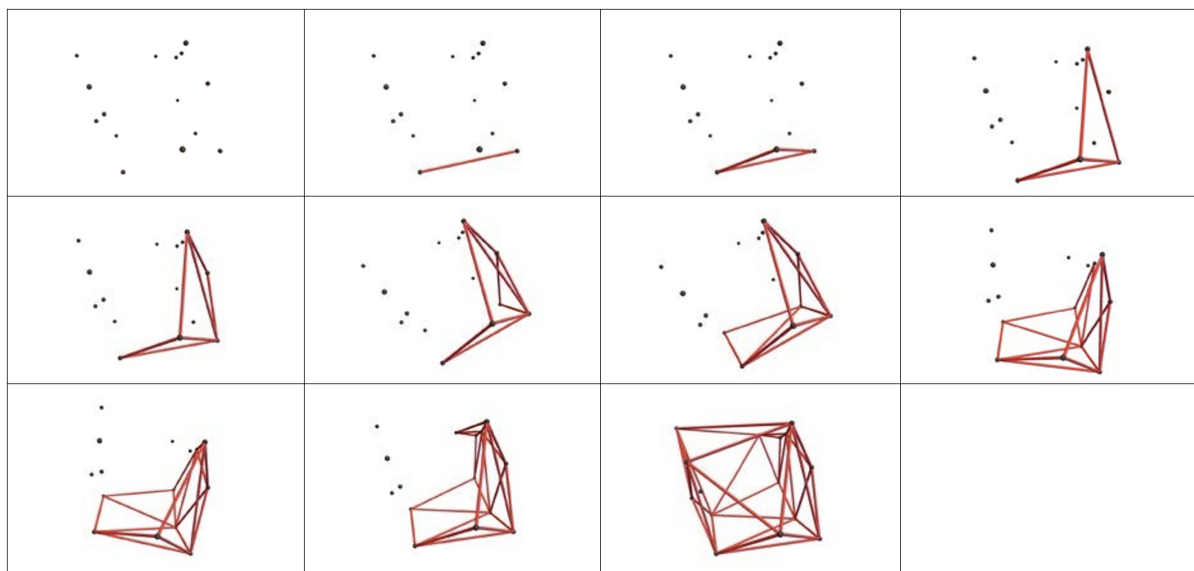


Figure 11. Implementation of the Jarvis's March (Gift Wrapping) algorithm.

Similar to the 2D case, in 3D, given a line and a point, the direction of the point relative to the line can be determined. Additionally, given a face and a point, the direction of the point relative to the surface can be determined. This is performed by calculating the sign of the dot product between the normal vector of the surface and the perpendicular vector from the point to the surface.

In 3D, given an edge and two points, we can determine which of the two points lies further ‘outside’ the edge. When searching for a new face, we consider edges that have not yet been explored. From these, we find the outermost points and create new faces. The process begins by finding the first edge to create the first face. This is carried out by selecting the point with the lowest z coordinate, followed by selecting a second point that forms the largest angle with the z -axis based on the first point. A search algorithm can then be used to identify these points.

The process can be summarized in pseudocode as in Algorithm 9.

Algorithm 9 Jarvis’s March Algorithm (3D Version).

Input: A set of n points in 3D space
Output: The convex hull H , a set of faces that form the convex polyhedron
Step 1: Find the point p with the smallest z -coordinate (the bottom point)
Step 2: Find the point q that makes the largest angle with the z -axis from p
Step 3: Initialize the search queue $Q = \{(p, q)\}$
while Q is not empty **do**
 Randomly select an edge (x, y) from Q
 Determine the two outermost points v and w relative to edge (x, y)
 if Faces (x, y, v) and (x, y, w) are not already added to the convex hull **then**
 Add faces (x, y, v) and (x, y, w) to H
 end if
 Add the edges (x, v) , (y, v) , (x, w) , and (y, w) to the search queue Q
end while
Return H

The steps in lines 1 and 2 each require $O(n)$ time. Line 4 is repeated h times, where h is the total number of edges or faces in the convex hull. Each iteration of loop 4 (particularly step B) requires $O(n)$ time, leading to an overall time complexity of $O(nh)$, similar to the 2D case.

While the 3D Jarvis’s March algorithm follows a simple principle and can be implemented relatively easily, it becomes more complex in three dimensions due to the increased number of edges and faces. In 3D, the convex hull tends to have more edges compared to the 2D case, as it includes the sides of the convex polyhedron. This complexity makes the algorithm less efficient in 3D spaces with a large number of points.

3.2. Divide and Conquer Algorithm

The basic form of the 3D Divide and Conquer algorithm follows the same principles as in the 2D case. The steps are as follows in Algorithms 10 and 11:

Algorithm 10 Three-Dimensional Divide and Conquer Algorithm for Convex Hulls.

Input: A set of n points in 3D space
Output: The convex hull of the set of points
Step 1: Find the median value for one axis coordinate
Step 2: Divide the points into two sets based on the median value
Step 3: Recursively apply the algorithm to each of the two sets
Step 4: Combine the two convex hulls obtained in step 3

Since steps 1 and 2 take linear time, and step 4 takes $O(n)$ time, the overall time complexity of the algorithm is $O(n \log n)$. However, combining two convex hulls in 3D is more complex than in 2D. In 3D, when combining two convex hulls, a tube with a ring at both ends is added to the ring formed by certain edges of the two convex hulls, and the surfaces inside the tube are discarded. This step requires an efficient method for creating the tube.

Steps 1, 2, and 3 in the process of combining the two convex hulls take $O(n)$ time. Therefore, the overall time complexity of the Divide and Conquer algorithm remains $O(n \log n)$.

Algorithm 11 Combining Two Convex Hulls in 3D.

Input: Two convex hulls H_1 and H_2 from two subsets of points

Output: The combined convex hull H_{combined}

Step 1: Find the first edge of the tube by finding the tangent of the 2D projection of both convex hulls

Step 2: Repeat the following process until the first edge is reached again:

Find the next edge of the tube from the neighbors of both endpoints of the current edge

Connect the outermost point among the neighbors to the corresponding point in the opposite convex hull

Step 3: Discard all the surfaces inside the tube and combine the tube with the remaining surfaces to form the final convex hull

3.3. QuickHull Algorithm

The 3D QuickHull algorithm follows the same basic principles as the 2D QuickHull algorithm. The steps are as follows in Algorithms 12 and 13:

Algorithm 12 Three-Dimensional QuickHull Algorithm.

Input: A set of n points in 3D space

Output: The convex hull of the set of points

Step 1: Select three points on the convex hull to form a plane

Step 2: Find the point that is furthest from the plane

Step 3: Connect the selected point and the plane

Step 4: Recursively repeat steps 2–4 for the new faces formed

Algorithm 13 Connecting Points and Planes in 3D QuickHull.

Input: A face F , a set of points, and the current convex hull

Output: Updated convex hull with new faces

Step 1: Place the current face in the navigation queue

Step 2: While the queue is not empty, repeat:

Pop face F from the queue

For all unvisited neighbors of F , do:

Compute the dot product of the outer normal vector of F and the perpendicular vector drawn from the face

If the dot product is negative and the edge is shared with F , add the face to the result set

If the dot product is positive, add the neighboring face to the search queue

Step 3: Add the faces from all edges in the result set to the current point and delete all visited faces (those with a positive dot product)

Selecting the three points on the convex hull is simple because the point with the smallest z coordinate is the lowest point, and the point with the largest z coordinate is the highest point. The third point is selected by finding the point furthest from the plane formed by the first two points. Finding the farthest point is straightforward by drawing a perpendicular line from each point to the plane and choosing the point with the largest distance.

The complexity arises in Step 3 when connecting points and planes because, unlike in 2D, multiple polyhedra can be formed from the same set of points in 3D. Thus, there may be cases where an edge must be flipped after the third process to maintain the convex hull. In Step 3, the connection of a point and a plane must be performed carefully by not just connecting the point from the current face but also by looking at the completed convex hull and properly connecting to the border.

To achieve this, we use a search algorithm to identify visible faces from the current face and connect the edges of all visible faces while removing faces inside the edges.

Steps 1, 2, and 3 in the process of connecting points and planes require $O(n)$ time. Therefore, the overall time complexity of the QuickHull algorithm is $O(n \log n)$ on average.

In Step 3, when selecting a point to add next, there is no need to guarantee that the point is already on the convex hull. By inserting points in an order that simplifies finding visible sides, randomization can be employed to optimize the insertion process, and the algorithm can be executed in a similar manner.

3.4. Transformation Algorithm

There are many variations of the 2D and 3D algorithms described so far. Newton Apple Wrapper [22] is a variant of the 3D randomized insertion convex hull algorithm, in which points are pre-sorted in the order of $(z(x(y)))$ coordinates and then sorted sequentially starting from the first point. This method was applied to a convex hull. At this time, since the points are sorted in coordinate order, it has the advantage of easily determining that the next point is outside the convex hull and which side to look at. For the 2D QuickHull algorithm, many parallelization techniques have been devised for the examples [32,33]. A parallelized example of the 3D QuickHull algorithm is the CudaHull algorithm [24]. These parallelization algorithms generally parallelize the process of finding the point furthest from the current face/edge rather than adding a point to the convex hull. For example, in the CudaHull algorithm, the process of finding a point to be added is based on a set of points. It is parallelized by dividing and distributing it to each GPU core. One CPU core is responsible for adding and deleting faces and points. This is because the QuickHull algorithm spends most of its time looking at many points and finding those that are the furthest away.

4. Discussion

4.1. Contribution

This study makes several significant contributions to the field of computational geometry. First, it provides a comprehensive analysis of convex hull algorithms, including Jarvis's March, Graham Scan, Divide and Conquer, Chan's Algorithm, and QuickHull, covering both two-dimensional and three-dimensional cases. Through systematic implementation and evaluation, the study highlights the computational performance, strengths, and limitations of these algorithms under varying conditions.

Additionally, the research enhances understanding through detailed visualization of algorithmic processes. These visualizations illustrate the computational flow and principles behind each method, making complex geometric computations more accessible to researchers and practitioners. Such an approach not only aids in education but also supports more informed decision-making when selecting algorithms for specific applications.

The study also explores parallelized implementations, such as CudaHull and Parallel Divide and Conquer, demonstrating their effectiveness in diversely shaped datasets. This investigation broadens the scope of convex hull algorithms, emphasizing their potential in computationally intensive environments and practical applications requiring high performance.

Furthermore, the study evaluates algorithm performance across diverse scenarios, including varying data distributions and dimensionalities. These evaluations provide actionable insights, such as the suitability of output-sensitive algorithms like Chan's Algorithm for datasets with fewer hull points and the practicality of three-dimensional adaptations for theoretical and applied tasks. By addressing these factors, the research bridges the gap between theoretical complexity and real-world utility.

Overall, this study makes a substantial contribution to computational geometry by systematically addressing the theoretical and practical aspects of convex hull algorithms. It lays a robust foundation for future research, particularly in optimizing algorithms for specific contexts and extending their applications to higher-dimensional spaces.

4.2. Scalability of the Algorithm in Large Dataset and Higher Dimensions

Scalability is indeed a crucial aspect, particularly in computational geometry, and we have addressed the handling of large-scale problems for each of the algorithms as follows:

The Gift Wrapping algorithm, also known as Jarvis's March, has a time complexity of $O(nh)$, where n is the number of points and h is the number of points on the convex hull. While this algorithm performs well for small- to medium-sized datasets, its performance significantly deteriorates as the dataset size increases. This is because the number of iterations grows linearly with the number of points on the hull. For large datasets, Gift Wrapping becomes inefficient and impractical due to its high complexity, especially when h is large.

In contrast, Graham Scan operates with a time complexity of $O(n \log n)$, which makes it more efficient and scalable for large datasets. The dominant factor in the time complexity is the sorting step, and since sorting is performed in $O(n \log n)$ time, this algorithm can handle larger datasets with ease. However, despite its efficiency, Graham Scan may face limitations in terms of memory usage when dealing with extremely large datasets.

The divide-and-conquer approach, which also has a time complexity of $O(n \log n)$, is efficient for large-scale problems. By dividing the problem into smaller subproblems and merging the results, the divide-and-conquer method can handle large datasets effectively. Its performance scales well as the size of the dataset increases, making it a suitable choice for problems involving thousands of points. Furthermore, its memory handling during the merge phase is efficient, which contributes to its scalability.

Quickhull, a Divide and Conquer algorithm, typically operates with an expected time complexity of $O(n \log n)$, although it can degrade to $O(n^2)$ in the worst case. Quickhull performs well on large datasets in practice, but its worst-case performance can be problematic, particularly in cases involving highly irregular or degenerate datasets. Therefore, while QuickHull is a good choice for large-scale problems, caution is needed to handle edge cases to avoid significant performance degradation.

TORCH is an advanced algorithm that is particularly efficient for higher-dimensional datasets. With a time complexity of $O(n \log n)$ in practice, TORCH excels in higher-dimensional spaces compared to traditional 2D or 3D convex hull algorithms. However, as the dataset size grows, the algorithm's memory usage and computational demands increase. Therefore, while TORCH is scalable for large datasets, its performance depends heavily on data distribution and dimensionality.

Kirkpatrick–Sediel is a randomized incremental algorithm with an expected time complexity of $O(n \log n)$. This algorithm performs well for large datasets as it allows for efficient incremental updates of the convex hull. While its design supports scalability, the quality of the randomization process can impact its performance. Nevertheless, Kirkpatrick–Sediel is typically suitable for large-scale problems and can handle reasonably large datasets with proper optimization.

Chan's algorithm combines the divide-and-conquer strategy with a refined approach to convex hull problems, achieving a time complexity of $O(n \log h)$, where h is the number of points on the convex hull. This makes Chan's algorithm particularly efficient for sparse datasets where h is small relative to n . It scales very well with large datasets, especially in cases where the number of points on the convex hull is much smaller than the total number of points. As a result, Chan's algorithm is highly efficient for large-scale datasets, where it can significantly reduce the computational complexity.

The CudaHull [24] algorithm was introduced to improve the scalability of the algorithm in higher dimensions. CudaHull is a parallelized convex hull algorithm designed to run efficiently on CUDA-enabled GPUs. By distributing the convex hull computation

across multiple GPU threads, CudaHull accelerates the process, significantly reducing computation time compared to CPU-based methods. It also optimizes memory usage by utilizing shared memory and minimizing data transfers between the host and GPU. These improvements make CudaHull highly scalable, allowing it to handle large datasets and high-dimensional spaces, where traditional convex hull algorithms become computationally expensive.

5. Conclusions

In this study, we implemented various algorithms to calculate convex hulls and confirmed their principles. In particular, there is a significant gap between the theoretical time complexity of the seven 2D convex hull algorithms (Graham Scan, Divide and Conquer, Jarvis's March, QuickHull, TORCH, Kirkpatrick–Seidel, and Chan's algorithms) and the actual performance. This was confirmed. (This seems to be due to the role of constant values that are not expressed in the Big_O function and the use of complex functions for theoretical optimization.) In addition, the performance depends on the result value for algorithms that are sensitive to the result value (output-sensitive). Statistically significant differences were observed between the groups. This means that when we use the convex hull algorithm in an application, simply using only a theoretically optimized algorithm may decrease performance. Therefore, in each expected situation and environment, variables include how quickly the most frequently called function or command in the algorithm can be called and executed, expected result value, increase in constant value, etc. The algorithm that shows statistically optimal performance through simulation should be selected. In the case of the 3D convex hull algorithm, we looked at how the 2D convex hull algorithm can be applied to 3D cases. Compared with the 2D case, in the 3D case, the polyhedron can be uniquely determined from the points. Several differences, such as examining the direction between the surfaces and points instead of lines and points, were confirmed. It was also confirmed that the time complexities of $O(nh)$ and $O(n \log n)$ remained the same despite the additional steps, owing to this difference.

Future work could focus on domain-specific algorithm selection, where the most suitable algorithm is chosen based on the unique characteristics of the dataset and computational environment, thereby ensuring optimal performance in real-world applications. Additionally, there is significant potential for integrating convex hull algorithms with other geometric algorithms, such as Voronoi diagrams, Delaunay triangulations, and nearest-neighbor searches. This integration could lead to more efficient solutions for a wide range of problems in fields like robotics, spatial data analysis, and path planning. Addressing these areas of research will considerably enhance the applicability and efficiency of convex hull algorithms across diverse domains.

Author Contributions: H.K.: Conceptualization, Methodology, Writing—original draft, Software, Validation, Formal analysis, and Visualization. S.O.: Conceptualization, Methodology, Formal analysis, Validation, Writing—original draft, Writing—review and editing, and Visualization. J.-W.B.: Conceptualization, Formal analysis, Writing—review and editing, Funding acquisition, and Supervision. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by a National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (RS-2023-00218832).

Informed Consent Statement: All authors give ethical and informed consent. There are no human or animal experiments in this paper. Also, there are no copyrighted data related to the figures.

Data Availability Statement: The data used to support the findings of this study will be available from the corresponding author upon request after acceptance.

Conflicts of Interest: The authors declare that there are no conflicts of interest regarding the publication of this paper.

References

- Knueven, B.; Ostrowski, J.; Castillo, A.; Watson, J.P. A computationally efficient algorithm for computing convex hull prices. *Comput. Ind. Eng.* **2022**, *163*, 107806. [\[CrossRef\]](#)
- Seidel, R. Convex hull computations. In *Handbook of Discrete and Computational Geometry*; Chapman and Hall/CRC: Boca Raton, FL, USA, 2017; pp. 687–703.
- Kenwright, B. Convex Hulls: Surface Mapping onto a Sphere. *arXiv* **2023**, arXiv:2304.04079.
- Rossignol, H.; Minotakis, M.; Cobelli, M.; Sanvito, S. Machine-learning-assisted construction of ternary convex hull diagrams. *J. Chem. Inf. Model.* **2024**, *64*, 1828–1840. [\[CrossRef\]](#) [\[PubMed\]](#)
- DeFord, D.; Dhamankar, N.; Duchin, M.; Gupta, V.; McPike, M.; Schoenbach, G.; Sim, K.W. Implementing partisan symmetry: Problems and paradoxes. *Political Anal.* **2023**, *31*, 305–324. [\[CrossRef\]](#)
- Meyer, J.J.; Mularski, M.; Gil-Fuster, E.; Mele, A.A.; Arzani, F.; Wilms, A.; Eisert, J. Exploiting symmetry in variational quantum machine learning. *PRX Quantum* **2023**, *4*, 010328. [\[CrossRef\]](#)
- Izsák, R.; Ivanov, A.V.; Blunt, N.S.; Holzmann, N.; Neese, F. Measuring electron correlation: The impact of symmetry and orbital transformations. *J. Chem. Theory Comput.* **2023**, *19*, 2703–2720. [\[CrossRef\]](#) [\[PubMed\]](#)
- Fabrizio, J. How to compute the convex hull of a binary shape? A real-time algorithm to compute the convex hull of a binary shape. *J. Real-Time Image Process.* **2023**, *20*, 106. [\[CrossRef\]](#)
- Guan, Y.; Yan, W.; Li, Y. Convex Hull Collaborative Representation Learning on Grassmann Manifold with Norm Regularization. In Proceedings of the Chinese Conference on Pattern Recognition and Computer Vision (PRCV), Xiamen, China, 13–15 October 2023; pp. 453–465.
- Ansar, H.; Al Mudawi, N.; Alotaibi, S.S.; Alazeb, A.; Alabdullah, B.I.; Alonazi, M.; Park, J. Hand gesture recognition for characters understanding using convex Hull landmarks and geometric features. *IEEE Access* **2023**, *11*, 82065–82078. [\[CrossRef\]](#)
- Huang, Z.; Wu, Z.; Yan, H. A convex-hull based method with manifold projections for detecting cell protrusions. *Comput. Biol. Med.* **2024**, *173*, 108350. [\[CrossRef\]](#) [\[PubMed\]](#)
- Du, N.; Xie, L.; Zhou, M.; Gao, W.; Wang, Y.; Hu, J. Convex Hull Triangle Mesh-Based Static Mapping in Highly Dynamic Environments. *IEEE Trans. Instrum. Meas.* **2024**, *73*, 8500814. [\[CrossRef\]](#)
- Wibowo, A.; Santoso, H.B.; Rachmat, C.A.; Delima, R. Mapping and grouping of farm land with Graham scan algorithm on convex hull method. In Proceedings of the 2019 International Conference on Sustainable Engineering and Creative Computing (ICSECC), Bandung, Indonesia, 20–22 August 2019; pp. 121–126.
- Zhou, T.; Bilmes, J.A.; Guestrin, C. Divide-and-conquer learning by anchoring a conical hull. *Adv. Neural Inf. Process. Syst.* **2014**, *27*, 1242–1250.
- Barber, C.B.; Dobkin, D.P.; Huhdanpaa, H. The quickhull algorithm for convex hulls. *ACM Trans. Math. Softw. (TOMS)* **1996**, *22*, 469–483. [\[CrossRef\]](#)
- Gomes, A.J. A total order heuristic-based convex hull algorithm for points in the plane. *Comput.-Aided Des.* **2016**, *70*, 153–160. [\[CrossRef\]](#)
- Kumar, V.; Mahima; Verma, S.; Nijhawan, N. Convex Hull: Applications and Dynamic Convex Hull. In *Ambient Communications and Computer Systems: Proceedings of RACCCS 2021*; Springer: Cham, Switzerland, 2022; pp. 371–381.
- Chan, T.M. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discret. Comput. Geom.* **1996**, *16*, 361–368. [\[CrossRef\]](#)
- An, P.T.; Hoang, N.D.; Linh, N.K. An efficient improvement of gift wrapping algorithm for computing the convex hull of a finite set of points in R^n . *Numer. Algorithms* **2020**, *85*, 1499–1518. [\[CrossRef\]](#)
- Day, A. Parallel implementation of 3D convex-hull algorithm. *Comput.-Aided Des.* **1991**, *23*, 177–188. [\[CrossRef\]](#)
- Zhao, J.; Jiao, L.; Liu, F.; Fernandes, V.B.; Yevseyeva, I.; Xia, S.; Emmerich, M.T. 3D fast convex-hull-based evolutionary multiobjective optimization algorithm. *Appl. Soft Comput.* **2018**, *67*, 322–336. [\[CrossRef\]](#)
- Sinclair, D. A 3D Sweep Hull Algorithm for computing Convex Hulls and Delaunay Triangulation. *arXiv* **2016**, arXiv:1602.04707.
- Blelloch, G.E.; Gu, Y.; Shun, J.; Sun, Y. Randomized incremental convex hull is highly parallel. In Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual, 15–17 July 2020; pp. 103–115.
- Stein, A.; Geva, E.; El-Sana, J. CudaHull: Fast parallel 3D convex hull on the GPU. *Comput. Graph.* **2012**, *36*, 265–271. [\[CrossRef\]](#)
- Amato, N.M.; Preparata, F.P. The parallel 3D convex hull problem revisited. *Int. J. Comput. Geom. Appl.* **1992**, *2*, 163–173. [\[CrossRef\]](#)
- Mei, G. CudaChain: A Practical GPU-accelerated 2D Convex Hull Algorithm. *arXiv* **2015**, arXiv:1508.05488.
- Alshamrani, R.; Alshehri, F.; Kurdi, H. A preprocessing technique for fast convex hull computation. *Procedia Comput. Sci.* **2020**, *170*, 317–324. [\[CrossRef\]](#)
- Xu, J.; Zheng, Z.; Feng, Y.; Qing, X. A concave hull algorithm for scattered data and its applications. In Proceedings of the 2010 3rd International Congress on Image and Signal Processing, Yantai, China, 16–18 October 2010; Volume 5, pp. 2430–2433.
- Gamby, A.N.; Katajainen, J. Convex-hull algorithms: Implementation, testing, and experimentation. *Algorithms* **2018**, *11*, 195. [\[CrossRef\]](#)
- Alshehri, F.A.; Alshamrani, R. A Filtering Method for Fast Convex Hull Construction. *J. Ubiquitous Syst. Pervasive Netw.* **2011**, *3*, 7.
- Jarvis, D.; Griffiths, P.D. Clinical applications of 3D volume MR imaging of the fetal brain in utero. *Prenat. Diagn.* **2017**, *37*, 556–565. [\[CrossRef\]](#)

32. Srikanth, D.; Kothapalli, K.; Govindarajulu, R.; Narayanan, P. Parallelizing two dimensional convex hull on NVIDIA GPU and Cell BE. In Proceedings of the International Conference on High Performance Computing (HiPC), Kochi, India, 16–19 December 2009; pp. 1–5.
33. Srungarapu, S.; Reddy, D.P.; Kothapalli, K.; Narayanan, P. Fast two dimensional convex hull on the GPU. In Proceedings of the 2011 IEEE Workshops of International Conference on Advanced Information Networking and Applications, Biopolis, Singapore, 22–25 March 2011; pp. 7–12.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.