

A filtering technique for fast Convex Hull construction in $\mathbb{R}^2$ Héctor Ferrada^{a,*}, Cristóbal A. Navarro^a, Nancy Hitschfeld^b^a Instituto de Informática, Facultad de Ciencias de la Ingeniería, Universidad Austral de Chile, Chile^b Departamento de Ciencias de la Computación, Universidad de Chile, Chile

ARTICLE INFO

Article history:

Received 14 December 2018

Received in revised form 25 April 2019

Keywords:

Convex Hull

Filtering Technique

Priority queues

ABSTRACT

This work presents an optimization technique that reduces the computational cost for building the Convex Hull from a set of points. The proposed method pre-processes the input set, filtering all points inside an eight-vertex polygon in $O(n)$ time and returns a reduced set of candidate points, ordered and distributed across four priority queues. Experimental results show that for a normal distribution of points in two-dimensional space, the filtering approach in conjunction with the Graham scan is up to $10\times$ faster than the *qhull* library, and between $1.7\times$ to $10\times$ faster than the Convex Hull methods available in the CGAL library. Results on the worst case scenario (when all points lie in the circumference) show that a slight random radial displacement of the points make this method the fastest one. Moreover, when increasing the magnitude of this displacement, the performance of the proposed method scales at a faster rate than the other methods. In terms of memory efficiency, the proposed implementation manages to use from $3\times$ to $6\times$ less memory than the other methods. The reason behind this memory improvement is because the proposed method stores indices of the input arrays, avoiding duplicates of the original floating points. Furthermore, the approach extends the problem size up to $n \leq 2^{40}$ by employing 5-byte indices (instead of 8-bytes) when $n \geq 2^{32}$. The optimization technique presented in this work has shown to be significantly useful in accelerating the computation of the Convex Hull, and it is not limited just to the combination with the Graham scan, but it can also be used in conjunction with other Convex Hull algorithms.

© 2019 Elsevier B.V. All rights reserved.

1. Introduction

Given a set of $S = \{p_1, p_2, \dots, p_n\}$ points in the plane, the convex hull of S is by definition the smallest convex region containing all of the points in S . For two-dimensions, we will refer to this region as the polygon $CH[1..h] \subseteq S$, where $h \leq n$ is the number of points of the minimal convex region enclosing S .

The convex hull has several applications in fields of science and technology. For instance, finding the smallest rectangular region that can enclose a complex two-dimensional concave shape (useful for doing efficient packing in industrial processes such as optimizing warehousing space) can be achieved by working with its convex representation, which is simpler in geometry [1]. In principle, working with the convex hull not only can reduce the number of points, but it can also bring the possibility to use important geometrical properties and assumptions that simplify further computations on the object. Another application of the convex hull is detecting if two complex objects are free of collisions or not, which can be done faster by checking their convex hulls [2] instead of their original more complex shapes. Collision

* Corresponding author.

E-mail address: hferrada@inf.uch.cl (H. Ferrada).

detection is important for physic engines in video games, simulations of many bodies interacting physically, engineering design and lately in autonomous driving where computations must be strictly in real-time.

The time for computing the convex hull of a set $S = \{p_1, p_2, \dots, p_n\}$ is lower bounded by the sorting problem (in a computing model based on numerical comparisons), i.e., $T(n) = \Omega(n \log n)$ [3]. This is because the convex hull can be used to sort a list of $x_1, x_2, \dots, x_n$ numbers if they are placed in the plane as $S = \{(x_1, x_1^2), (x_2, x_2^2), \dots, (x_i, x_i^2), \dots, (x_n, x_n^2)\}$. Given that sorting has been proved to be at least $\Omega(n \log n)$ in time, the computation of the convex hull also obeys this lower bound, otherwise it would lead to a contradiction.

Several convex hull algorithms have been proposed in the last decades, some of them are optimal, $O(n \log n)$, such as the Graham scan [4] (which is used by the CGAL library), Divide and Conquer [5], Monotone chain [6] and incremental [7]. There is also the Chan's algorithm [8] and the Kirkpatrick–Seidel algorithm [9], which are optimal output-sensitive algorithm with costs of $O(n \log h)$. These algorithms can strongly benefit when $h = o(n)$ being much faster than the classic $O(n \log n)$ ones. For some specific scenarios, non-optimal algorithms have proven to be as fast as optimal ones (and sometimes even faster); two well known ones are Gift Wrapping [10,11], which is $O(nh)$, and quickhull [12] which has an expected running time of $O(n \log n)$. Both of these algorithms have a worst case time of $O(n^2)$, but are still highly effective when chosen for the right scenario.

Despite the fact that the theory of the convex hulls has been extensively studied in asymptotic terms, there is still an aspect that can be exploited to further improve the performance of convex hull algorithms; this is to discard points that are guaranteed to lie inside the convex hull. Such task can be done as a pre-processing stage using an efficient filtering algorithm on the input set S . As a result, one obtains a smaller input set $S' \subseteq S$, for which a known convex hull algorithm can work on. The motivation behind this approach, is that the overall computational cost of filtering S and computing $CH(S')$ can be significantly lower than computing $CH(S)$ directly from the original input set. Indeed, the reduction of the problem size is directly related to the distribution of points, and the worst case scenario produces a set $S' = S$. Nevertheless, practical problems in science and engineering rarely fall into these cases, as many of these data inputs come from Gaussian distributions or random distribution of points. For this, it is interesting to know how much of an improvement can the filtering approach be on different types of input sets.

This work presents a new filtering approach for the convex hull problem in two dimensions, that pre-processes an input set S of points and discards all points that fall inside an eight-vertex polygon built at runtime. For random point sets, the proposed method indeed reduces the set to $S' \subseteq S$, and when run in conjunction with the Graham Scan algorithm, it is able to compute $CH(S)$ up to $10\times$ faster than the QHull library, and between $1.7\times$ to $10\times$ faster than CGAL running with their two fastest Convex Hull methods.

The rest of the paper is organized as follows: Section 2 describes relevant related works within the scope of filtering and accelerated convex hulls. Section 3 presents an overview of the pruning algorithm proposed in this work. Then, Section 4 presents the performance results of the new method, compared to QHull and CGAL libraries. Lastly, Section 5 concludes our work and identifies some future works that can further improve the results presented here.

2. Related work

Two of the most reliable and efficient implementations of convex hull are the ones found in *qhull* [12] and CGAL [13] libraries. They can even be considered standard references for comparing (by transitivity) to future solutions if they also compare against these two libraries. The present work relies on this logic when comparing its performance.

In the last years there have been important efforts at improving the performance of the known algorithms that compute the convex hull, mostly through two approaches; (1) through the proposal of a parallel variant of one of the known algorithms and (2) through the filtering of points of S as a pre-processing stage. Additionally, some works focus specifically on the 2D case, while others on the 3D one. Although our work corresponds to a filtering approach without considering parallelism yet, we still include a description of the state of the art regarding parallel solutions, as it will be useful for connecting our results with future directions aimed at improving the algorithm's performance via massively parallel architectures, such as GPUs.

For the case of parallel solutions, Miller and Stout presented a series of parallel approaches using different parallel models of computation [14]. Their conclusion, that the convex hull cannot be solved faster than sorting, agrees with the lower bound of the convex hull problem. Selim G. Akl proposed a theoretical $O(1)$ time parallel algorithm for computing the convex hull [15], although using a polynomial number of processors and some theoretical assumptions such as no three co-linear points and that the AND gate can operate through multiple inputs in constant time. The algorithm assigns all possible combinations of a point p_i and a triplet $\{p_j, p_k, p_q\}$ to different processors. Each processor would write on a table, at the corresponding column for p_i , in a unique row for that processor, the value of 1 if p_i lies outside the triangle formed by $\{p_j, p_k, p_q\}$, otherwise 0. This algorithm uses the fact that points of the convex hull have the property of being outside of the triangle formed by any possible triplet of the input set S . Each of these point in triangle tests can be done in constant time and the integration of all results within a column can be reduced by an AND operator in constant time as well (while the other columns are reduced as well in parallel). Although the algorithm indeed can run in $O(1)$ time, in practice, it will be rapidly limited by the number of available processors, as it must be at least $\Omega(n^4)$ since it is n times all triplets, which corresponds to a tetrahedral number, i.e., $n \times n^3 = \Omega(n^4)$.

Several GPU-based algorithms have been proposed for the 2D and 3D convex hull as well. Srungarapu et al. proposed a GPU-based algorithm for the 2D Convex Hull [16]. The authors manage to minimize the impact of irregular memory access

patterns on the GPU (which are known to have a negative impact on performance), and achieve up to $14\times$ of speedup over a standard CPU-based Convex Hull solution. Mei presented an alternative CUDA based algorithm for the convex hull [17], based on a sorting pre-processing approach. The method is based on two stages: (1) discard points inside a quadrilateral and (2) compute the convex on the remaining set of points. The author reports up to $6\times$ of speedup over the Qhull implementation, when dealing with problems of 20M points. Other works have focused on the 3D (or higher dimensions) version of the convex hull, achieving important speedups thanks to the parallel performance of GPUs [18–20].

The second approach for improving convex hull computation is to propose pre-processing techniques that can help reduce the input set S and transform the problem into a smaller one, without affecting the output result. This reduction on the input set S has an immediate effect in performance, as there is less work to be done for the convex hull algorithm hence it will take less time (more performance). The challenge however is to find an efficient way to discard these points, that is faster than what it would take for the convex hull algorithm when processing them. Our work is directly related to this challenge, and the act of discarding points is also known as *filtering* the input set S . The idea of filtering was already used by Barber et al. in their QuickHull algorithm [12], where they discard all points that lie inside a polygon that grows in edges as the recursion advances, starting from a line of extreme points. Quickhull is the algorithm used for the Qhull library. The work of Mei [17], which was mentioned as a parallel solution, also includes a filtering stage, based on the construction of a quadrilateral polygon where all points that fall inside are discarded. This filtering stage is done with the help of sorting algorithms, which will facilitate the computation for the next stage of the algorithm.

In this work we present a new variant of filtering technique for reducing the input set S , one that is based on a fast filtering process that occurs inside an eight-side polygon. Four priority queues are employed to hold the remaining points on the four regions that fall outside the polygon (one queue for each region). The usage of priority queues produces a partial sorting of points during the filtering stage, which can then be used in favor for an ordered extraction of points when employing a convex hull algorithm such as the Graham Scan.

3. Overview of the algorithm

The algorithm proposed in this work is essentially a filtering technique that generates a set of points $S' \subseteq S$, in the plane, for which a convex hull algorithm can work on afterwards. This section focuses on explaining the filtering part, which is the main contribution in this work. This filtering process with the convex algorithm make up four relevant stages, (1) building an octagon, (2) points filtering and grouping into priority queues, (3) Partial Ordering and (4) Convex Hull computation. The first three stages take $O(n)$ time, while the convex hull stage takes $O(n' \log n')$ in our current approach, where n' is the size of the filtered set $S' \subseteq S$.

Algorithm 1 describes the process to compute the convex hull. Lines 2..10 describe the filtering stage and lines 12..14 correspond to the convex hull computation over the reduced set of $n' \leq n$ points. At line 2, the procedure `FINDEXTREMES()` finds, in linear time, the 8 points that define the filtering convex octagon $CP(E)$. The for loop at line 4 iterates for all n points and discards the ones inside $CP(E)$, through the conditional at line 5. The points that are candidate for the convex hull are the ones outside $CP(E)$, for which one of the four priority queues is chosen (`findQueue(...)`) to insert the point. The final stage, between lines 11–14, computes the convex hull $CH[1..h]$, with $h \leq n'$, for all the points stored into the priority queues Q_1, Q_2, Q_3, Q_4 using the Graham method [4] which runs in $O(n' \log n')$ time in its worst case for the n' candidate points.

An important aspect to keep in mind is that these candidate points end up being partially sorted by the priority queues. Therefore it is important for the convex hull algorithm to take advantage of this effort already made in advance. The following sections describe more in detail the relevant parts of the algorithm.

3.1. Building the filtering convex octagon

This stage builds an eight-side polygon from the input set, which is then used to filter the points inside and keep the remaining ones (the ones outside the polygon) as candidates. For each coordinate axis, in linear time, we identify the set of right-most, upper-most, left-most and lowest-most points, named $P = \{ri, up, le, lo\}$ respectively. All points inside the quad formed by these points are guaranteed to not belong to the convex hull. Although one could already begin filtering from this stage, the proposed method goes one step further and converts the quad into an octagon before filtering. For this, the algorithm will find four new points from S and add them to the set P , making it an octagon. These new points, denoted m_1, m_2, m_3, m_4 , will help the polygon cover a larger filtering surface than in its quad version. The placement of m_1 considers the line L_{ri} , which is parallel to the y -axis and passes through point ri , and the line L_{up} , which is parallel to the x -axis and passes through point up . Then, the location of m_1 will be the point from S that is closest to the corner intersection between L_{ri} and L_{up} , denoted c_1 . The process is analogous for m_2, m_3 and m_4 , just changing L_{ri} and L_{up} to the corresponding lines and corners as shown in Fig. 1.

The task of finding the points m_i takes linear time as it only requires one pass on the input set S . For the distance function we chose the *Manhattan distance* [21], that is, for each corner c_i , $1 \leq i \leq 4$, we find the point in S that minimizes the sum of the vertical and horizontal distances to this corner. Indeed, in some cases, the Manhattan distance will not provide the closest point to the corner as the Euclidean distance would [22], but these cases tend to occur with low frequency, and when they occur, it can be seen that the chosen point still lies very close to the closest one using Euclidean distance. Furthermore, Manhattan distance compensates its behavior by offering simpler computation, i.e., no square roots are required.

Algorithm 1 *heaphull*. To compute the convex hull $CH[1..h]$ in 2D.

Input: n floating points, in 2D, stored in an array $P[1..n]$.

Output: an integer array $CH[1..h]$ with the h points of the hull in counter-clockwise order.

```

1: procedure HEAPHULL( $P, n$ )
2:    $E \leftarrow \text{FINDEXTREMES}(P, n)$                                 ▷ find the 8 extreme points in  $O(n)$  time
3:    $n_1 = n_2 = n_3 = n_4 = 0$                                 ▷ counter for the 4 priority queues  $Q_i$ 
4:   for  $j = 1$  to  $n$  do
5:     if  $P[j]$  is outside the convex octagon  $CP(E)$  then
6:        $i \leftarrow \text{FINDQUEUE}(P, n, j)$                                 ▷ find  $Q_i$  for  $P[i]$  ( $O(1)$  time)
7:       store  $j$  in the priority queue  $Q_i$ 
8:        $n_i \leftarrow n_i + 1$ 
9:     end if
10:  end for
11:   $CH \leftarrow \emptyset$ 
12:  for  $i = 1$  to 4 do
13:     $CH \leftarrow CH \cup \text{HULL}(Q_i, n_i)$                                 ▷ CH for points of  $Q_i$  in  $O(n_i \log n_i)$  time
14:  end for
15:  return  $CH$                                 ▷ the convex hull  $CH[1..h]$ , with  $h \leq \sum_{i=1}^4 n_i = n' \leq n$ 
16: end procedure

```

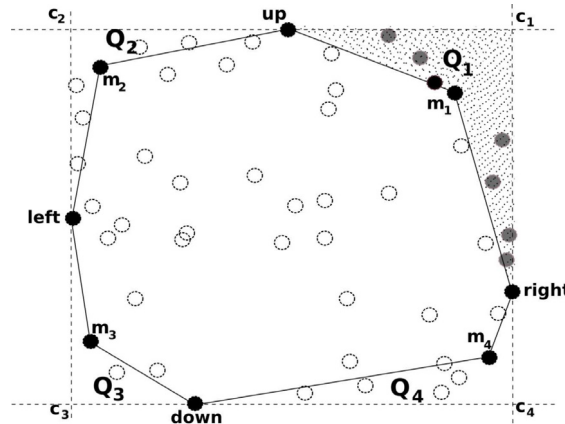


Fig. 1. The Filtering Octagon. The queue Q_1 handles the area from *right* to *up* (the rightmost and uppermost points in S) in counter-clockwise order; and stores all points in the hatched area (green points), which is itself a polygon formed by the points *right*, m_1 , *up* and the corner c_1 . Note that, the vertices of the convex hull from *right* to *up* will be in decreasing order given the X -coordinate; for that reason Q_1 is a maximum priority queue in X -component. The others queues work analogously. Q_2 also is a maximum priority queue given the Y -coordinate. Q_3 and Q_4 are minimum priority queues given the X and Y component respectively. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

3.2. Point filtering and grouping into priority queues

The next stage of the algorithm does a complete pass on the input S , and for each $x_i \in S$, it checks if it is inside P or not. If x_i lies inside P , then this point is filtered by the algorithm. Checking if a point is inside a polygon is a task that takes $O(|P|)$ time where $|P|$ is the number of vertices of the polygon P . In this case, $|P| = 8$, therefore the task takes $O(1)$ constant time. It is important to note that in many practical cases, where the distribution function of the input set S is similar to the normal or uniform distribution, most of the points are inside of the filtering polygon, and these will be discarded in linear time.

If the point lies outside P , then the algorithm will insert the point into one of four possible priority queues which are implemented using min/max heaps, denoted Q_1, Q_2, Q_3, Q_4 . Each heap Q_i holds the points of one of the four regions formed by P and the corner points. These heap insertions do not heapify the structure yet, they only group the points into different regions.

In order to establish a counter-clockwise order, we set Q_1 to be a maximum priority queue, based on the X coordinate, and stores points from *right* to *up*. The other queues work analogously, as illustrated in Fig. 1.

3.3. Partial ordering

Once the filtering process finishes, the stage ends with a set of four heaps that hold the candidate points for the convex hull. At this stage, the algorithm does the heapify operation on each heap. The heapify process consists of re-arranging the points inside the heap so that for every node, the heap property is satisfied. In the case of the convex hull, it means that each node appears before its children when doing a counter-clockwise sweep from the center of the octagon.

When the heapify process is done after all points are inserted (instead of doing it separately at each insertion), the amortized cost becomes $O(n)$ [23] instead of $O(n \log n)$ time. The $O(n)$ cost can be obtained by adding the costs of heapify at each level of the heap, i.e.,

$$\sum_{i=0}^{\log(n)} \frac{n}{2^{i+1}} O(i) \leq n \sum_{i=0}^{\log(n)} \frac{i}{2^i} \quad (1)$$

where $O(i)$ is the cost of doing heapify at a node that is i levels away from the leaves of the tree. By using the following property of geometric series

$$\sum_{i=0}^n i \cdot c^i = \frac{nc^{n+2} - (n+1)c^{n+1} + c}{(c-1)^2} \quad (2)$$

we have that in the limit $n \rightarrow \infty$, the cost of heapify converges to

$$\lim_{n \rightarrow \infty} n \sum_{i=0}^{\log(n)} \frac{i}{2^i} = \lim_{n \rightarrow \infty} 4n \left(\frac{n}{n^n} - \frac{n+1}{4^n} + \frac{1}{2} \right) \quad (3)$$

$$= 2n \in O(n) \quad (4)$$

3.4. Convex hull computation

The computation of the convex hull follows a process similar than Andrew's method [24] to construct the hull exploiting this sorting between the points. It starts with Q_1 , and extracts point by point, and moving to the next heaps in order Q_2, Q_3, Q_4 as their points are all extracted.

For each point p_j extracted from queue Q_i , we determine at which side of the line $\overline{P_{last}, P_{ext}}$ it is, where P_{ext} is the last extreme point in the area handled by the queue (for Q_1 it is *up*), and P_{last} is the last point added to the current segment of the partial hull from the points stored in the queue. For the first extracted point there is no P_{last} point. Accordingly, if p_j is in the *external side* of the line (i.e on the right side in a counter-clockwise direction) we add this point to the potential convex hull $CH[1..h]$ and check if some previous points recently incorporated would need to be deleted or not. This verification is similar to the Graham's algorithm test, which can be made checking whether triplets of consecutive vertices are in counter-clockwise order, simply by computing slopes between points. The resulting polygon of this stage is indeed the convex hull $CH[1..h]$ of S , which is ordered in a counter-clockwise direction starting at the rightmost point in S .

3.5. Correctness of the algorithm

We ensured the correctness of our algorithm in each stage. First, when we evaluate all point from S and discard points from the input set, we check that these points are inside of our filtering polygon. Consequently, any point enclosed by a closed polygon, which vertices are in S too, cannot be considered as part of the convex hull. In the second stage, each time that we incorporate a new vertex in the current convex hull, we have ensured that it is in the right side (given a counter-clockwise order) of the line from the last point insert and the final extreme point corresponding to the area handled by the queue. Therefore, we incorporate only convex edges to the hull. Also, given the implicit order in the queues, when we extract points from them, we have ensured to obtain edges in the correct order.

3.6. Running time analysis

We recall that we are given a set $S[1..n]$ and we output another set $C[1..h] \subseteq S$. We saw that the optimal worst case time to construct the convex hull is $O(n \log n)$, and there are some output sensitive algorithms which achieve $O(n \log h)$ time. Our algorithm also achieves $O(n \log n)$ optimal time. Moreover, we can refine our complexity to $O(n + n' \log n')$ time, where $h \leq n' \leq n$. n' represents the total points inserted in the queues; that is, the points not discarded at the *filtering stage*.

3.7. Improvements to the memory space

One of the contributions of this work is the efficient memory usage. In order to save memory, when $n \geq 2^{32}$, we employ 5 bytes of memory, instead of a built-in 8-byte data type, to reference any index $i \in [0..2^{40} - 1]$ in the heaps. In the other case, i.e., $n < 2^{32}$, we only use 4 bytes per index in the queues. It is important to note that the approach indeed can be extended to 6-bytes in case problem sizes of $n > 2^{40}$ points are being handled. By using less bytes per value, the implementation supports an input set of up to $n = 2^{40}$ points (x_i, y_i) where x_i and y_i are 32-bit floating point values. With this approach the queues require $n' \log n$ bits, and the final convex hull needs $h \log n$ bits. Finally, the data structure requires $O(n' \log n)$ bits in total.

Although it is also possible to implement re-sizable arrays, we decided to avoid it given the additional time needed to allocate new arrays and copy their values. Moreover, although $2^{40} \sim 10^{12}$ points are theoretically possible to handle with this approach, in practice they would require roughly 5 TB of memory for the input data, which to the present date is not possible to handle on a single computer node (the computer used in the tests has 128 GB of RAM). Nevertheless, the performance results presented in this work still cover problem sizes that are 5 times larger than the largest size that CGAL and QHull could handle in our test machine.

3.8. Filtering in N -dimensional space

The filtering algorithm can be extended to 3D space or even a higher dimensional one. The intuition is that the extreme points of the initial filtering structure can correspond to different axis-aligned hyper-planes. The intersections of these hyper-planes define a bounding-box (orthotope) for which the closest points to the corners, in conjunction with the original extreme points, are sufficient to build the filtering structure.

In the case of two-dimensions, four intersections are found between the parallel lines to the coordinate axes, and four additional corner points are added to the polygon (the closest ones to each corner), giving as a result an Octagon. In general, for N dimensions, the extreme points define $2N$ axis-aligned hyper-planes which compose an N -dimensional axis-aligned bounding box. As a result, in a N dimensional space, the filtering *polytope* has $2N + 2^N$ points (assuming the extreme points are not co-linear).

It is important to note that although the filtering indeed can work for higher dimensions, one would need to choose a convex hull that is suitable for the corresponding dimensions. For instance, the Graham Scan does not work for 3-dimensions, therefore it may be possible that the partial ordering done through heaps could be modified to another strategy that will eventually be more useful to the corresponding convex hull algorithm for N -dimensional space, such as *quickhull*.

4. Experimental performance results

We implemented the proposed algorithm in C++ which will be referred as *heap-hull*¹ for the remainder of this manuscript. Our implementation uses simple floating point arithmetic, therefore roundoff error occurs with each calculation, especially when the points are very close each other or nearly at the lines of the convex hull edges.

We compared its performance against *qhull*.² and *CGAL*.³ libraries. In the *Computational Geometry* community, these are two of the most used and cited libraries. From these libraries, we considered the following functions to compute the 2D-convex hull:

- **QHull**: the *qh_new_qhull* function, which implements the quick hull algorithm [12]. It requires $O(n^2)$ time in the worst case for n input point and $O(n \log n)$ in the average case.
- **CGAL**: the *convex_hull_2()* function. It uses the best option between two algorithms: the Bykat algorithm [25], which has a worst-case running time of $O(nh)$ and the Akl–Toussaint algorithm [26], which requires $O(n \log n)$ time, where n is the number of input points and h is the number of vertices of the final convex hull.
- **CGAL**: the *ch_graham_andrew()* method. This function implements Andrew's variant of the Graham scan algorithm [4] and follows the representation of Mehlhorn [24]. The algorithm requires $O(n \log n)$ time in the worst case for n input points.

Although CGAL library contains other functions to obtain the convex hull, we did not include any other method from it. The reason is because the performance of the selected methods was superior.

The two libraries, *qhull* and *CGAL*, can output fictitious points in their results. This means that a point p' could be given as a vertex of the convex hull with p' not belonging to the original input S . By contrast, our implementation does not produce new points for the convex hull. In fact, the *heap-hull* method limits its result to a subset of the input S . This is because our queues store indexes of the original input arrays instead of approximated or rounded floating point values. For this, we can then retrieve the original points whenever they are required. Despite this difference between the

¹ Available at <https://github.com/hferrada/heap-hull>.

² Available at <http://www.qhull.org> Accessed: October 1, 2018.

³ Available at <http://www.cgal.org> Accessed: October 1, 2018.

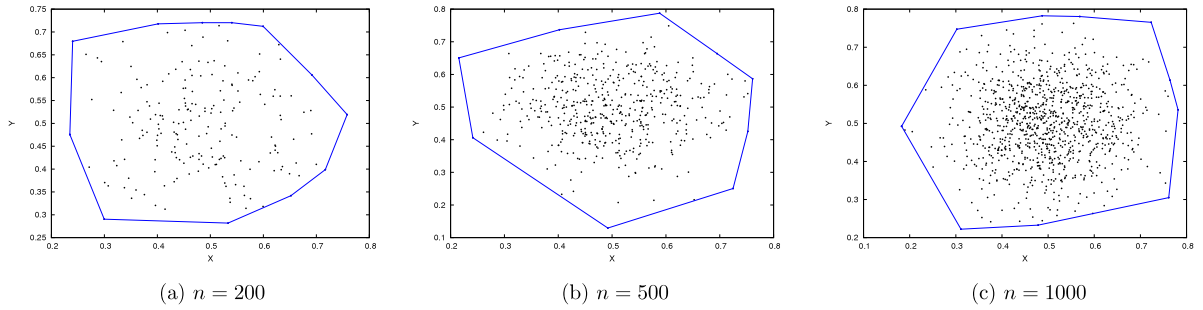


Fig. 2. Illustration of the normal case, when the points came from a normal distribution with $\mu = 0.5$ and $\sigma^2 = 0.1$.

proposed method and the others, we have noted that the number of points for the resulting convex hull is still the same on all three cases (CGAL, Qhull and Heap-hull).

The machine in which we executed all the experiments has an Intel Core i7-6950X CPU with 10 cores at 3.50 GHz and 25M Cache L3, and 128 GB DDR4 RAM. It runs Arch Linux, and we use g++ with $-O3$ optimization.

4.1. Description of the typical and worst Case tests

For all tests, the input consists of n points in the plane, whose coordinates are float random values from two probability distributions. The tests consist of returning the convex hull from the input set. In order to better characterize the proposed algorithm, we have established two types of tests, one that represents the typical case and another one that represents the worst case.

1. **[Typical] Points following a Gaussian distribution.** The first test represents a typical case where points occupy the space using a normal probability distribution. This test is a good representation of all data sets that come from physical phenomena or human data coming from interactions or processes, as these events naturally tend to behave similar to a Gaussian distribution. The test is created by generating n random points (x_i, y_i) , $1 \leq i \leq n$, whose x, y coordinates come from a normal distribution with $\mu = 0.5$ and $\sigma^2 = 0.1$. This obtains a set where practically all the points are in the interval $[0, 1]$, which, in addition, produces high accuracy in the computations with floating point arithmetic. Fig. 2 illustrates the distribution of points for this test, from lower to higher input size.
2. **[Worst Case] Points on a circumference from a uniform probability distribution.** The second test places all (or most) of the points on the line of a perfect circumference. This experiment can be considered one of the hardest tests for the proposed *heap-hull* method because only a small fraction of the input set S can be filtered; the rest of the points end up placed into their corresponding heaps and eventually form part of the convex hull or become discarded at later on in the convex hull stage. The test is created by generating n points on a circumference (or very close to it) centered at the origin with radius $r = 0.25$. The test offers a p parameter to be chosen, which produces a displacement probability for each point in the range $[r - rp, r + rp]$. This displacement may move the point inwards to the center, or outwards the circumference, making a band of points surrounding the circumference. The distribution of points is illustrated in Fig. 3 using different values of $p \in [0, 1]$. The expectation of the test is to have $n(1 - p)$ points in the circumference and np points displaced inwards or outwards.

For the two categories of tests the problem size n varies in the range $[10^4..10^9]$. Then, two performance metrics are computed; (1) the average running time with a standard error below 1%, and (2) the peak memory consumption.

4.2. Filtering in the worst Case

Table 1 summarizes the percentages of filtering points in the worst case test. From the table, we recall that p is the probability of a point being displaced from the circumference line. Two sizes n are considered, a large one $n = 10^8$, and a smaller one $n = 10^4$. The behavior of the filtering process is similar for any $n \geq 10^4$ with $p \geq 0.16$. The table only shows up to $p \leq 0.16$, because once $p > 0.16$, the % of discarded points remains stable around $\sim 0.97n$. The red numbers indicate that the construction time of the *heap-hull* loses against the fastest implementation in CGAL. The blue color means that the running times among *heap-hull* and the libraries are practically identical. The rest of the entries, in black color, mean that the performance of *heap-hull*, for the worst case, is significantly better than the other libraries. Performance results are detailed in the upcoming plots within this Section.

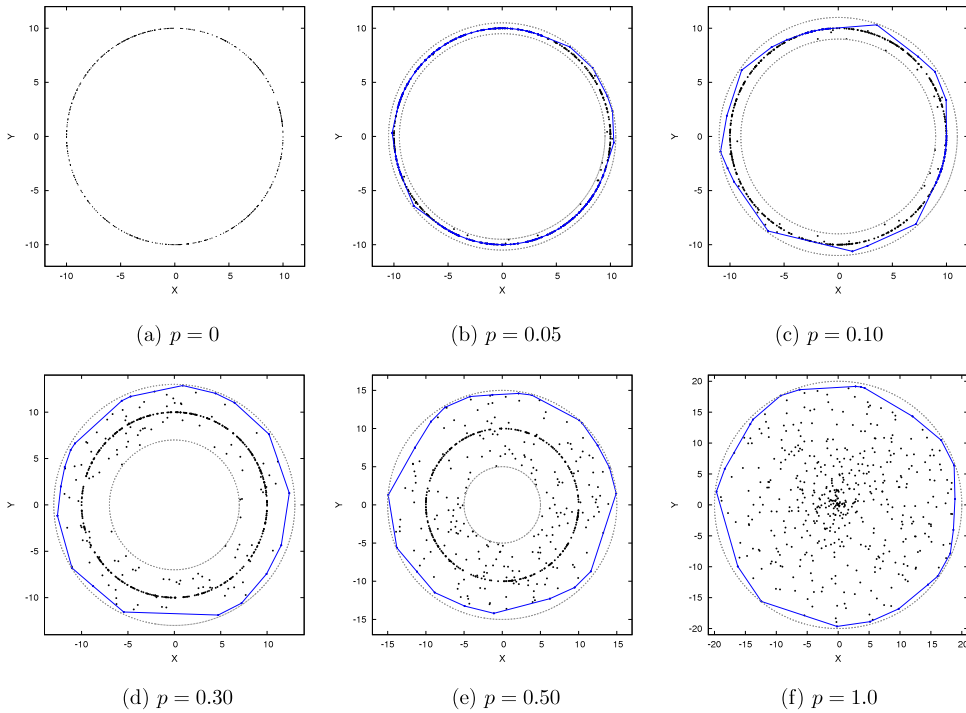


Fig. 3. Illustration of the worst-case test with randomly selected displacement at different values for $p \in [0, 1]$.

Table 1

Average percentage of discarded points at the filtering stage in the *heap-hull* algorithm for the worst case. (For interpretation of the references to color in this table, the reader is referred to the web version of this article.)

p	filtering %, $n = 10^4$	filtering %, $n = 10^8$
0.00	0.01	0.01
0.02	10.04	13.12
0.04	25.86	28.68
0.06	46.63	48.51
0.08	70.36	82.01
0.10	86.97	97.17
0.12	94.66	97.12
0.14	96.41	97.07
0.16	96.65	97.02

4.3. Performance plots

For the first category of tests (normal distribution in the plane), the performance results are presented in Fig. 4.

The plots include the construction time and the memory peak for each convex hull method. For the running time we observe that *heap-hull* offers a speedup of $\sim 10\times$ with respect to *cgal-grah* and *qhull*, and $\sim 1.7\times$ of speedup with respect to *cgal-ch2*, which is the best method from CGAL. In terms of the amount of memory usage, the CGAL library reached a maximum input size of $n = 10^9$ points (less than 2^{32}). Both of CGAL's methods, for such value of n , required near 100 GiB of RAM, while *heap-hull* used less than 10 GiB of RAM for the same n . Even for $n = 5 \cdot 10^9$ which is larger than 2^{32} , *heap-hull* still did not surpass the memory usage of CGAL running for $n = 10^9$. In the case of *qhull*, for small data sets, their implementation does not perform well in terms of memory usage. On the other hand, for large problem sizes such as $n \geq 10^6$ it uses less memory than CGAL, however for $n = 10^9$ it caused an overflow and the machine's memory was not enough for the execution.

The results for the second category of tests (worst case) are presented in Figs. 5 and 6. From the results, we observe that in general *heap-hull* will perform better or worse depending if it manages to filter a significant part of the input set S , or not, respectively. The first four plots of Fig. 5 show different variations of the worst case, following the displacement

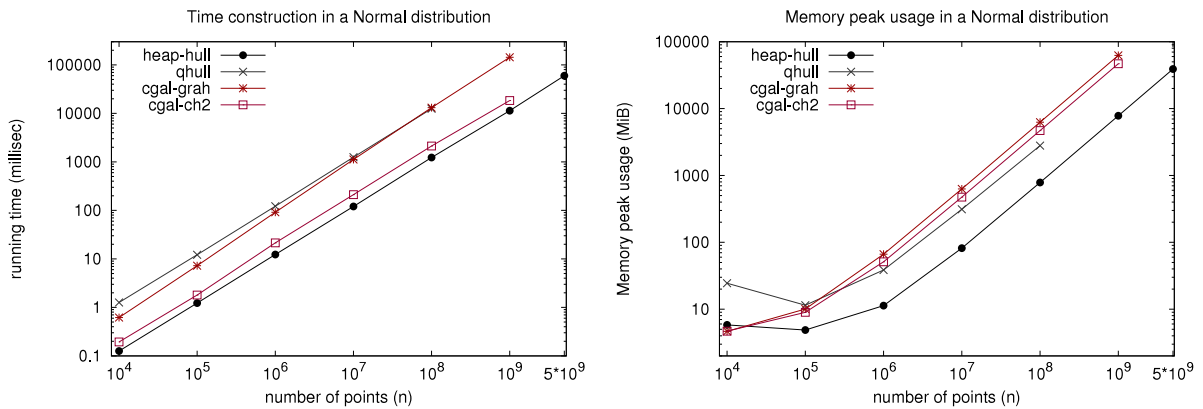


Fig. 4. Time construction (left) and memory peak usage (right) for the *heap-hull* against *qhull* and CGAL's implementations to compute de convex hull.

rule given by the p value. When p is small, such as $p = 0$ and $p = 0.05$, the performance of *heap-hull* is indeed worse than CGAL and QHull for large values of n . In practice $p = 0.0$ implies that no point will be filtered, thus all points will need to be inserted in their corresponding queues as convex hull candidates. For values in the order of $n = 10^4$ (see Fig. 6), the crossing point where *heap-hull* takes the lead in performance occurs at $\sim p = 0.05$, and for values in the order of $n = 10^8$, it occurs at $\sim p = 0.08$. After this crossing points is met, *heap-hull* remains as the fastest one for such order of n . It is interesting to note that for either case, the amount of displacement required to make *heap-hull* efficient is smaller than $p = 0.10$.

In terms of memory efficiency, *heap-hull* uses significant less memory than the CGAL and Qhull implementations. This difference is more notorious for large n , such as $n = 10^8$, where it is up to $6\times$ and $3\times$ more efficient than *cgal-grah* and *qhull*, respectively.

5. Conclusions

This work has presented a fast and memory efficient method for discarding interior points in the 2D convex hull problem. The technique is based on building an eight-vertex polygon in which all points that reside inside it get discarded, and the remaining ones are assigned onto different priority queues, all this in linear time.

The partial sorting of the candidate points is a key aspect for the method because the *heapify* process takes $O(n)$ amortized time, and it facilitates the computation of the actual convex hull (which benefits from any level of ordering among points) afterwards. Furthermore, the implementation associated to this work uses succinct structures, enabling the algorithm to use a reduced amount of memory compared to the other approaches, while at the same time reaching large data sets of up to $n \leq 2^{40}$ input points, which is considerably greater than the limit of $n \leq 2^{32}$ found on Qhull and CGAL.

The benefits of speed and space of the proposed method manifest better when a significant number of the input points lie inside the convex hull. This scenario is a typical case in many applications where phenomena tend to group its points following a normal distribution, which for a two-dimensional problem can be seen as clusters of points that are more dense at their centers. Experimental performance tests using random points following a normal distribution in a two-dimensional space show that the proposed method is between up to $\sim 10\times$ faster than the *qhull* library and between $1.7\times$ to $10\times$ faster than the CGAL library methods.

Indeed, the worst case scenario, where all points lie in a circular convex hull, is not beneficial for the proposed approach, as the computational effort put on discarding points does not give any beneficial return. However, the experimental results on a variation of the worst case scenario have shown that a slight displacement of the points from the circumference makes the proposed method the fastest one with an increasing speedup based on how much displacement is put onto the test (100% of displacement makes the problem a point cloud). This favorable sensitive behavior on the worst case is an important property for the proposed method, as in practice there are very few scenarios that present a perfect worst case input, and in that case it is highly probable that a special heuristic may end up being the best solution.

The filtering approach presented here can be a useful tool for many practical problems that require a fast computation of the convex hull using a reduced amount of memory in the process as well. Furthermore, it was shown the filtering technique can be extended to higher dimensions. As a future work, it would also be interesting to study how this method can further improve its performance through parallel computing, when in the presence of multi-core CPUs as well as with modern GPUs that offer massively parallel computation for highly parallelizable problems. Indeed, some of the GPU approaches mentioned in the related work may be suitable for future integration with our method.

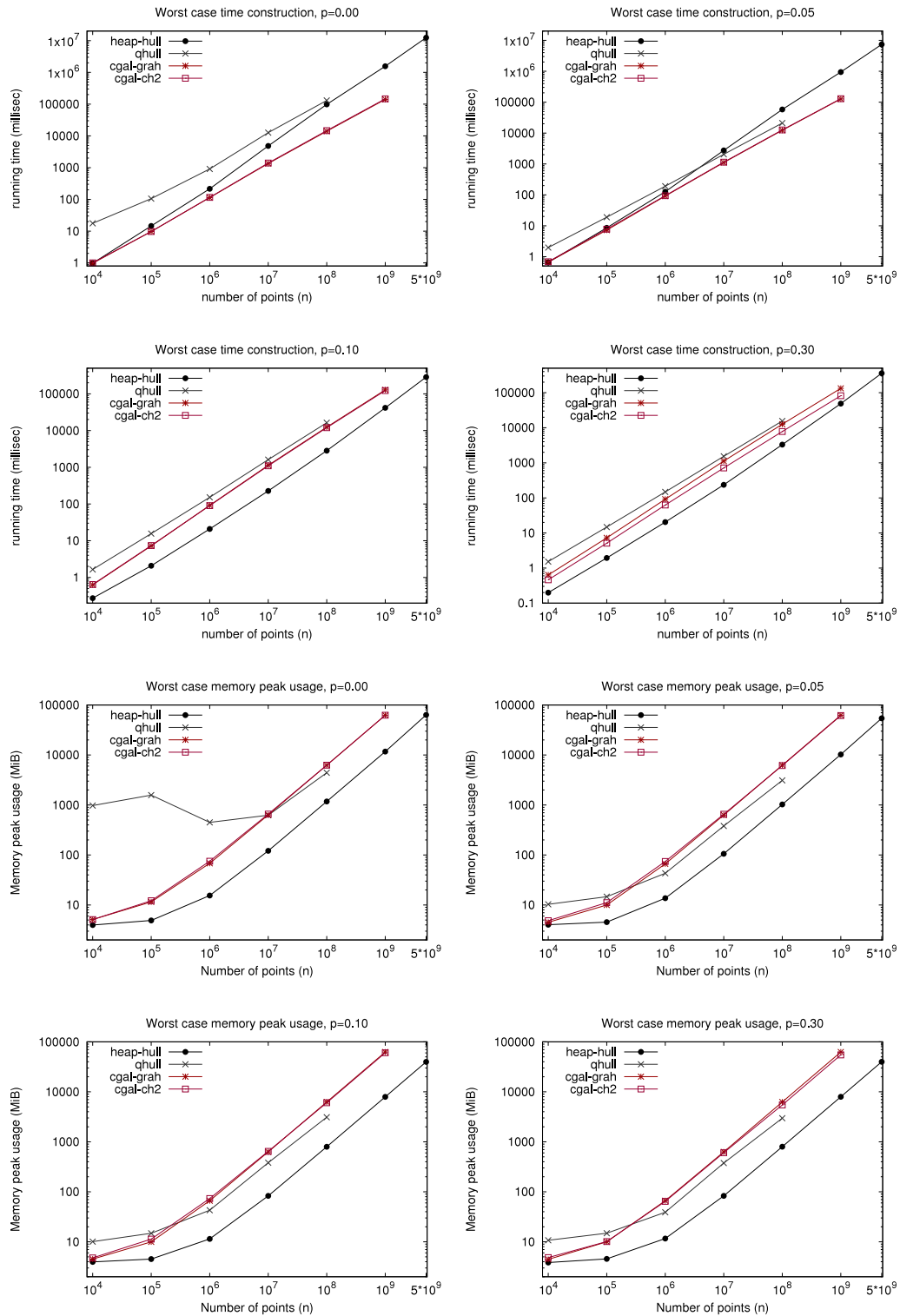


Fig. 5. Construction time (the 4 top figures) and memory peak usage (the 4 bottom figures) for the *heap-hull*'s worst case against *qhull* and *CGAL*'s implementations.

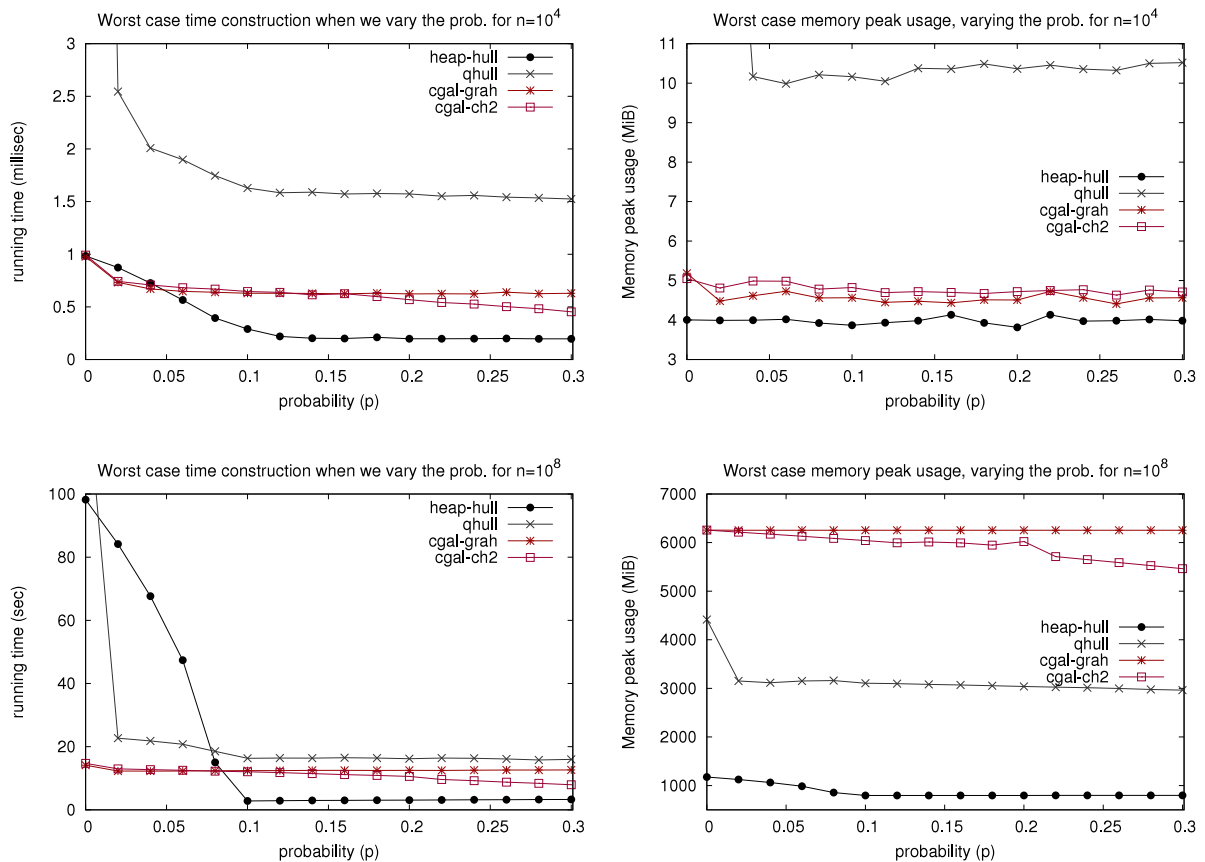


Fig. 6. Running time (left) and memory peak usage (right) for the *heap-hull*'s worst case against *qhull* and *CGAL*.

References

- [1] J. O'Rourke, Finding minimal enclosing boxes, *Int. J. Comput. Inf. Sci.* 14 (3) (1985) 183–199, <http://dx.doi.org/10.1007/BF00991005>.
- [2] M.C. Lin, U.D. Manocha, J. Cohen, *Collision detection: Algorithms and applications*, 1996.
- [3] A.C.-C. Yao, A lower bound to finding convex hulls, *J. ACM* 28 (4) (1981) 780–787, <http://dx.doi.org/10.1145/322276.322289>, URL <http://doi.acm.org/10.1145/322276.322289>.
- [4] R.L. Graham, An efficient algorithm for determining the convex hull of a finite planar set, *Inform. Process. Lett.* 1 (4) (1972) 132–133.
- [5] F.P. Preparata, S.J. Hong, Convex hulls of finite sets of points in two and three dimensions, *Commun. ACM* 20 (2) (1977) 87–93, <http://dx.doi.org/10.1145/359423.359430>, URL <http://doi.acm.org/10.1145/359423.359430>.
- [6] A. Andrew, Another efficient algorithm for convex hulls in two dimensions, *Inform. Process. Lett.* 9 (5) (1979) 216–219, [http://dx.doi.org/10.1016/0020-0190\(79\)90072-3](http://dx.doi.org/10.1016/0020-0190(79)90072-3), URL <http://www.sciencedirect.com/science/article/pii/0020019079900723>.
- [7] M. Kallay, The complexity of incremental convex hull algorithms in rd, *Inform. Process. Lett.* 19 (4) (1984) 197, [http://dx.doi.org/10.1016/0020-0190\(84\)90084-X](http://dx.doi.org/10.1016/0020-0190(84)90084-X), URL <http://www.sciencedirect.com/science/article/pii/002001908490084X>.
- [8] T.M. Chan, Optimal output-sensitive convex hull algorithms in two and three dimensions, *Discrete Comput. Geom.* 16 (4) (1996) 361–368, <http://dx.doi.org/10.1007/BF02712873>.
- [9] D.G. Kirkpatrick, R. Seidel, The ultimate planar convex hull algorithm, *SIAM J. Comput.* 15 (1) (1986) 287–299, <http://dx.doi.org/10.1137/0215021>.
- [10] D.R. Chand, S.S. Kapur, An algorithm for convex polytopes, *J. ACM* 17 (1) (1970) 78–86, <http://dx.doi.org/10.1145/321556.321564>, URL <http://doi.acm.org/10.1145/321556.321564>.
- [11] R. Jarvis, On the identification of the convex hull of a finite set of points in the plane, *Inform. Process. Lett.* 2 (1) (1973) 18–21, [http://dx.doi.org/10.1016/0020-0190\(73\)90020-3](http://dx.doi.org/10.1016/0020-0190(73)90020-3), URL <http://www.sciencedirect.com/science/article/pii/0020019073900203>.
- [12] C.B. Barber, D.P. Dobkin, H. Huhdanpaa, The quickhull algorithm for convex hulls, *ACM Trans. Math. Software* 22 (4) (1996) 469–483.
- [13] S. Hert, S. Schirra, 2D convex hulls and extreme points, in: *CGAL User and Reference Manual*, 4.13 ed., CGAL Editorial Board, 2018, URL <https://doc.cgal.org/4.13/Manual/packages.html#PkgConvexHull2Summary>.
- [14] R. Miller, Q.F. Stout, Efficient parallel convex hull algorithms, *IEEE Trans. Comput.* 37 (12) (1988) 1605–1618, <http://dx.doi.org/10.1109/12.9737>.
- [15] S.G. Akl, A constant-time parallel algorithm for computing convex hulls, *BIT Numer. Math.* 22 (2) (1982) 129–134, <http://dx.doi.org/10.1007/BF01944471>.
- [16] S. Srungarapu, D.P. Reddy, K. Kothapalli, P.J. and Narayanan, Fast two dimensional convex hull on the gpu, in: *2011 IEEE Workshops of International Conference on Advanced Information Networking and Applications*, 2011, pp. 7–12, <http://dx.doi.org/10.1109/WAINA.2011.64>.
- [17] G. Mei, *Cudachain: an alternative algorithm for finding 2d convex hulls on the gpu*, *SpringerPlus* 5 (1) (2016) 697.
- [18] M. Gao, T.-T. Cao, A. Nanjappa, T.-S. Tan, Z. Huang, Ghull: A gpu algorithm for 3d convex hull, *ACM Trans. Math. Software* 40 (1) (2013) 3:1–3:19, <http://dx.doi.org/10.1145/2513109.2513112>, URL <http://doi.acm.org/10.1145/2513109.2513112>.

- [19] M. Tang, J. yi Zhao, R. feng Tong, D. Manocha, Gpu accelerated convex hull computation, *Comput. Graph.* 36 (5) (2012) 498–506, Shape Modeling International (SMI) Conference 2012.
- [20] A. Stein, E. Geva, J. El-Sana, Cudahull: Fast parallel 3d convex hull on the gpu, *Comput. Graph.* 36 (4) (2012) 265–271, <http://dx.doi.org/10.1016/j.cag.2012.02.012>, Applications of Geometry Processing.
- [21] E.F. Krause, *Taxicab Geometry*, Dover, 1987.
- [22] M.M. Deza, E. Deza, *Encyclopedia of Distances*, Springer Berlin Heidelberg, 2009, http://dx.doi.org/10.1007/978-3-642-00234-2_1.
- [23] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, *Introduction To Algorithms*, third ed., The MIT Press, 2009.
- [24] K. Mehlhorn, *Data Structures and Algorithms 3: Multi-Dimensional Searching and Computational Geometry*, Springer-Verlag New York, Inc., New York, NY, USA, 1984.
- [25] A. Bykat, Convex hull of a finite set of points in two dimensions, *Inform. Process. Lett.* 7 (6) (1978) 296–298, URL <http://dblp.uni-trier.de/db/journals/ipl/ipl7.html#Bykat78>.
- [26] S.G. Akl, G.T. Toussaint, A fast convex hull algorithm, *Inform. Process. Lett.* 7 (5) (1978) 219–222, URL <http://dblp.uni-trier.de/db/journals/ipl/ipl7.html#AklT78a>.