Journal of Ubiquitous Systems & Pervasive Networks
Volume 3, No. 1 (2011) pp. 00-00

A Filtering Method for Fast Convex Hull Construction

Fatimah A. Alshehri*, Reham Alshamrani*

Computer Science Department, King Saud University, Riyadh, Saudi Arabia

Abstract

Computing the convex hull of a set of points is a fundamental issue in many fields, including geometric computing, computer graphics, and computer vision. This problem is computationally challenging, especially when the number of points is past the millions. In this paper, we propose a fast filtering technique that reduces the computational cost for computing a convex hull for a large set of points. The proposed method preprocesses the input set and filters all points inside a four-vertex polygon. The experimental results showed the proposed filtering approach achieved a speedup of up to 77 and 12 times faster than the standard Graham scan and Jarvis march algorithms, respectively.

Keywords: *Convex hull, filtering, priority queue.*

1. Introduction

The convex hull of a set of points (*S*) is defined as the smallest convex polygon that contains all of the points in *S* [1]. Computing convex hulls is a fundamental issue with a wide range of applications in computer sciences, mathematics, statics and economics. Some of the applications in which the convex hull is used include image processing, pattern recognition, medical simulations, geometric modeling, geographical information systems (GIS), pathfinding, and computer visualization [2]. It is also applied invideo games, simulations of many bodies physically interacting, engineering design, and recently, in autonomous driving, where computations must be strictly fast and in real time [3].

Many classic algorithms have been introduced for computing convex hulls, including the Graham scan (1972) [4], Jarvis's march (1973) [5], the divide-and-conquer algorithm (1977) [6], Andrew's monotone chain (Andrew 1979) [7], the incremental approach (1984) [8], and QuickHull (1996) [9]. However, one aspect can further improve the performance of these algorithms: adding a preprocessing stage to reduce computation time and memory space. Fig. 1. Shows example of convex hull.

One of the most widely used algorithms for finding convex hulls is the Graham scan algorithm, which calculates the convex hull of a set of points in $O(n \log n)$. The algorithm usesa stack to store points. It starts by sorting the given set of points on the x-axis. After that, it checks the orientation at each point with the two most recently selected points. The required hull results from joining the points stored in the stack [2].

Although convex hulls have been found using Graham's method, there is still an aspect that can be improved. When computing a convex hull, a Graham scan uses the whole set of input points, both inside and outside; thus, lead to the problem that the algorithm works quite slowly. To overcome the drawbacks of the Graham scan algorithm and to further improve the performance of convexhull algorithms, we propose a filtering technique that discards the interior points that forming extra overhead, subsequently to enhance the Graham scan algorithm.

The suggested solution presented in this research is to preprocess the Graham scan algorithm [5] by adding filtering techniques to enhance the convex hull computing. The important stage that is added to Graham's algorithm is discarding points from the approximate convex hull and inserting them into priority queues. The proposed algorithm was designed for serial computing, so it does not need any graphics processing unit (GPU) resources. However, the challenge is to find an efficient way to discard these points for faster computation. This was implemented by employing a preprocessing method to efficiently filter unnecessary points, which resulted in a smaller input set $S' \subseteq S$. This reduction directly reduced the computation time, and experimental results of the proposed algorithm outperformed well-known convex hull algorithms: Jarvis march and Graham scan.

The remainder of the paper is organized as follows. Section 2 provides an overview of the prior work on convex hull algorithms. Section 3 describes the proposed algorithm step by step. Section 4 compares the proposed algorithm to other well-

* Corresponding author. Tel.: +966536327743
E-mail: Reham.Alshamrani@gmail.com
© 2011 International Association for Sharing Knowledge and Sustainability.
DOI: 10.5383/JUSPN.03.01.000

known convex hull algorithms. Section 5 concludes the paper and sheds some light on possible future research directions.

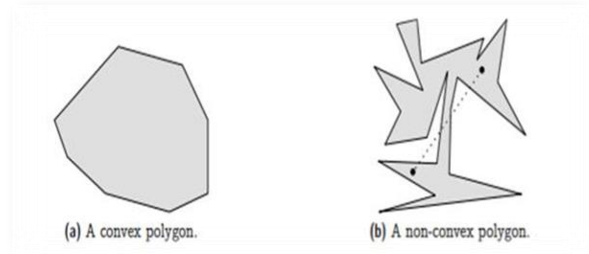

Fig. 1. Example of convex and non-convex polygon [10]

2. Literature Review

In recent years, important efforts have been made to improve the performance of algorithms to compute the convex hull, mostly through two approaches: (1) by proposing a parallel variant of a known algorithm and (2) by filtering the points of S as a preprocessing stage. Additionally, some works focus specifically on the 2D or 3D case.

2.1. Parallel Computing

For the case of parallel solutions, there has been a growing interest in using the GPU for computational geometry in recent years; several GPU-based algorithms have been proposed for the 2D and 3D convex hull as well. A speed up to 14 times faster over a standard CPU-based convex hull solution wasachieved in [10], which proposed a GPU-based algorithm for the 2D convex hull. The authors used a GPU-based algorithm for interior-point filtering, while they used CPU-based exact convex hull computation. The remaining points were processedat *O (n/p log n)* time complexity.

A sorting-based preprocessing approach that reached a similar high speedup was proposed in [11], which runs in a linear time *O(n).* The algorithm was an alternative CUDA- based algorithm for the convex hull called CudaCHPre2D, using GPU-accelerated implementation on single and dual GPUs. The author achieved speedups of approximately 6 ~ 9 times on average and 9 ~ 14 times faster in the best cases in single GPUs, when the data size reaches 20M. The GPU- accelerated implementation developed on dual GPUs was 1.18 ~ 1.35 times faster than on a single GPU. For point sets, they used randomly distributed points in square, circle, and 3D mesh models.

Other works have focused on the 3D version of the convex hull, achieving important speedups thanks to the parallel performance of GPUs. In [12], the authors presented a CPU-GPU hybrid convex hull computation algorithm that runs in *O(n).* It is simple to implement and offered significant speedups (up to 27 times faster and 46 times faster for static and deforming point sets, respectively). They ran it on (10 M to30 M) points randomly distributed within a spherical 3D space;the data transfer time (I/O part) represents the overall running time.

The first algorithm for the 3D convex hull problem that is fully accelerated on the GPU was proposed in [13]; while producing exact results, the experiment on different test cases showed that the implementation on CUDA is more than an order of magnitude faster than the best sequential CPU implementations. Also, it is 3 to 4 times faster than CGAL, where n was in the range of 10^6 to 10^{13} points.

2.2. Filtering Technique

The second approach for improving convex hull computation is to use a preprocessing technique that can help reduce the input set *S* and transform the problem into a smaller one, without affecting the output result. The method is basedon two stages: (1) discard points inside a quadrilateral and (2) compute the convex on the remaining set of points.

A proposed algorithm that runs in *O (m log m)* in [14] is based on the fact that only the points existing in the boundaryof a region are covered by a randomly distributed set of points. The number of points varied from 1000 to 100,000, real-time simulation results that preprocessing improved the overall execution time.

Although Quickhull is the algorithm used for the Qhull library, Qhull was implemented in [11], which was mentioned as a parallel solution; the researchers also included a filtering stage. This filtering stage was performed with the help of sorting algorithms.

Algorithms that used these approaches achieved high speedup performance in [15]; they enhanced the known Graham algorithm by adding the Quickhull algorithm in the first stage to discard interior points. The number of points was quite small (5000-25,000 points). Furthermore, they proved that their proposed method outperforms the two algorithms (Graham and Quickhull).

A similar approach was presented in [16]. Whereas the algorithm reduced the number of interior points, experimental results showed that for a normal distribution of points in two-dimensional space, the filtering approach is up to 10 times faster than the Qhull library. The number of points varied in the range [104...109] in a normal distribution. Their algorithm runs in O (n log n). However, the reduction method executes in time within O(n).

The authors in [17] presented a method that reached linear time, with a computational time within *O(n).* Also, in terms of time performance, the authors reported a speedup of at least a factor of two when using Chan's algorithm.

On the other hand, a sorting-based algorithm to compute the convex hull of a set of points in R2 was presented in [18], referred to as the TOURCH algorithm. The algorithm outperforms the fastest algorithm Quickhull, 1.17 times faster. The TORCH algorithm has *O (n log n)* time complexity. The authors implemented it with 3 multisets varied from one million to 128 million points.

However, the previous works that discussed above didn't reach reasonable speedup that enhancing traditional algorithms. While the proposed filtering method outperforms the original Graham scan and reach a speed up of 77 times without using of GPUs or any parallelization techniques.

Table 1. Comparison of previous work

Paper	Algorithm	Execution time Results	Time Complexity	Dimension	Point set
CudaHull [10]	GPU-based for interior point filtering, CPU-based exact convex computation	14 times faster than standard CPU-based Convex Hull	O (n/p log n)	2D	deforming point sets
CudaCHPre2D [11]	GPU-accelerated implemented on single and on dual GPUs.	In single GPU: 9 × ~ 14× On dual GPUs: 1.18 ~ 1.35	O (n)	2D	randomly distributed points in a square, circle and 3D mesh models
[12]	CPU-GPU hybrid	46 times faster than single CPU	(I/O part)	3D	randomly distributed in 3D space. (10 M to 30 M)
gHull [13]	GPU and Using CUDA programming model.	implementation on CUDA is faster than the best sequential CPU implementations	Not mentioned	3D	n in the range of 106 to 107
[14]	divided the point set into four rectangular regions	real-time simulation results improve the overall execution time	O (m log m)	2D	randomly distributed set of points (1000- 100000)
CudaCHPre2D [11]	Preprocessing method to discard interior points	More than 99% input points are discarded	O (n)	2D	randomly distributed points in a square, circle and 3D mesh models.
[15]	Combining Graham with Qiuck hull algorithm	proposed method outperforms the two algorithms	Not mentioned	2D	(5000-25000) points
[16]	Preprocessing method using priority queue	up to 10 times faster than the qhull library	O (n log n)	2D	n varies in the range (104-109)
[17]	Linear time preprocessing (no sorting step)	speedup of at least a factor of two than Chan's algorithm	O(n)	2D	n size is hundred points and up
TOURCH [18]	Sorting based preprocessing	1.17×times faster than Quickhull	O (n log n)	2D	1 M to 128 M For each 3 multisets

3. Overview of the algorithm

In this work, we present a variant of a filtering technique for reducing the input set S. It is based on a fast filtering process that occurs inside a four-sided polygon [16]. A priority queueis employed to hold the remaining points in the four regions outside the polygon. It also sorts points during the filtering stage, which can then be used to produce an ordered extraction of points when executing a convex hull algorithm using the Graham scan algorithm. The proposed algorithm differs in howthe hulls between the points are calculated. In general terms, the proposed algorithm consists of the following steps, as presented in Fig. 2.

- Find extreme points.
- Filter and group the points into priority queues
- Construct the convex hull.

In the following sections, we described the relevant parts of the algorithm in further detail.

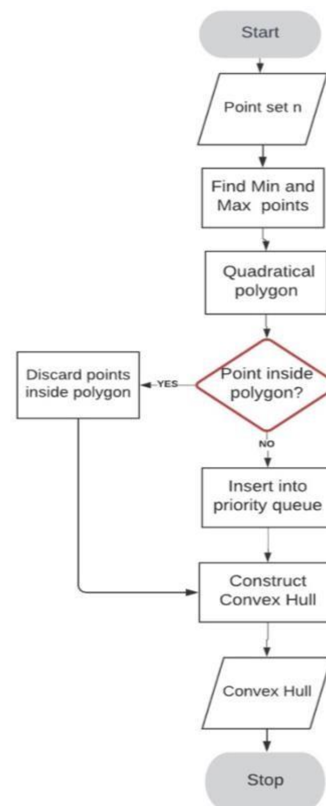

Fig. 2. Flowchart of the proposed algorithm

3.1. Find Extreme Points

In the first stage, a four-sided polygon from the input set is built, which is then used to filter out the inside points and keep the ones outside the polygon as candidates. For each coordinate, in linear time, we identify the set of right-most, upper-most, left-most, and lowest points, denoted as $P = \{X_{MIN}, Y_{MIN}, X_{MAX}, Y_{MAX}\}$. None of the points inside the polygon formed by these points belongs to the convex hull, as illustrated in Fig. 3.

3.2. Point Filtering Grouping into Priority Queues

The next stage of the algorithm is a complete pass on the input S. For each point $p_i \in S$, the algorithm checks if it is inside the polygon or not. If the current point lies inside the polygon, then this point is filtered by the algorithm. Importantly, 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 will be discarded in linear time.

If the point lies outside P, then the algorithm will insert the point into priority queues, which are implemented using min/max heaps. The queue Q holds the points of the four regions formed by P and the corner points, as illustrated in Fig. 3.

3.3. Construct the Convex Hull

The construction of the convex hull follows the Graham method, with the hull construction exploiting this sorting between the points. The algorithm extracts each point from Q. To establish a counterclockwise order, we set Q to be a maximum priority queue—based on the X coordinate—that stores points from right ($Xmax$) to up ($Ymax$), as illustrated in Fig. 4. The same procedure is repeated with the other three regions.

For each point (p_j) from queue Q, we determine at which side of the line P_{last}, P_{ext} it is, where P_{ext} is the last extremepoint in the area handled by the queue and P_{last} is the last point added to the current segment of the partial hull from the points stored in the queue. Accordingly, if p_j is outside of the line (i.e., on the right side in a counterclockwise direction), thenthis point is added to the potential convex hull *CH [1 ... h]*. This verification utilizes the Graham algorithm test, which checks whether triplets of consecutive vertices are in counterclockwise order, simply by computing the slopesbetween points. Suppose the three points (a, b), (c, d), and (e,f), given in that order. Suppose that the first point is farthest to the left, so $a < c$ and $a < f$. Then, the three points are in counterclockwise order if and only if the line (a, b) (c, d) is lessthan the slope of the line (a,b) (e, f):

$$Counterclockwise \Longleftrightarrow ((d-b) / (c-a)) < ((f-b) / (e-a)) \quad (1)$$

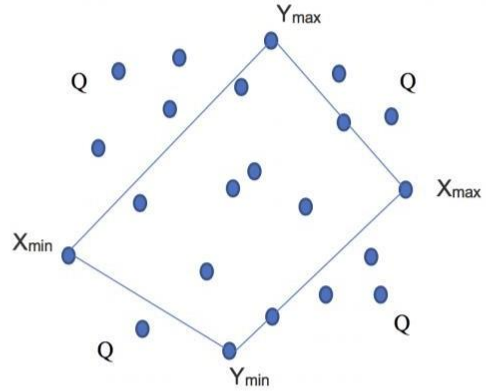

Fig. 3. Finding extreme points from the polygon

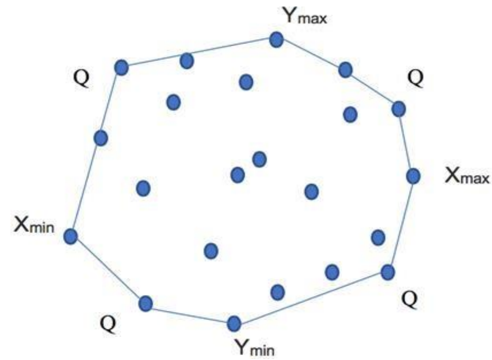

Fig. 4. Finding extreme points from the polygon

Because both denominators are positive, we can rewrite this inequality as follows:

$$\text{Counterclockwise} \Longleftrightarrow (f-b)(c-a) > (d-b)(e-a) \quad (2)$$

The resulting polygon of this stage is the convex hull *CH [1 ... h]* of P, which is ordered in a counterclockwise direction, starting at the rightmost point in P. The final output form is illustrated in Fig. 4.

Algorithm 1 above describes the process to compute the convex hull. Lines 2-9 describe the filtering stage, and lines 10-14 are the convex hull computation over the reduced set of $n' \leq n$ points. At line 2, the process findExtremes() finds, in linear time, the four extreme points that define the filtering convex polygon *CH(E)*. The for loop at lines 4-9 iterates for all n points to discard the ones inside *CH(E)* (the polygon), through the conditional at line 5. The remaining points for the convex hull are those outside *CP(E)*, for which the priority queue is set to insert the point into it. The final stage, at lines 10-14, constructs the convex hull *CH [1... h]*, with $h \leq n'$, using the Graham method, which computes the slopes between points in counterclockwise order [19].

Algorithm 1: To compute convex hull *S* [1….*h*] in 2D

Input: *n* points, in 2D, sorted in an array *P* [1…*n*]

Output: an array *CH* [1…*h*] with h points of the hull in counterclockwise order.

1. Procedure Algorithm(*P*,*n*)
2. *E* ← FIND EXTREME POINTS(*P*,*n*) ▷ **find the 4 extreme points in $O(n)$ time**
3. Leftmost, Rightmost, Uppermost, Lowermost;
4. **For** *j*=1 to *n* do
5. If *P*[*j*] is outside the convex polygon
6. Store *j* in the priority queue *Q* ▷**insert $P[i]$ in Q in $O(1)$ time**
7. **else**
8. Discard point
9. **end for**
10. Construct **convex** hull
11. **For** i=1to *n* do
12. Construct Convex **hull** (*CH*) from **Queue** ▷ **CH for points of Q in $O(n' \log n')$ time**
13. **end for**
14. Return *CH* ▷ **the convex hull $CH[1...h]$, with $n' \leq n$**
15. End procedure

3.4. Time complexity analysis

Recall that the algorithm is given a set *P [1... n]* and outputs another set *CH [1... h]* ⊆ *P*. The first stage is finding the four extreme points in *O(n)* time (linear time). In the filtering stage, checking whether a point is inside a polygon is atask that takes *O(n)* time. The process of filtering and grouping points into the priority queue takes *O(n)* time. The final stage atwhich to extract the convex hull is *O (n log n)* according to the Graham algorithm. Taking into consideration the previous analysis, we conclude that the proposed algorithm has *O (n log n)* running time [20].

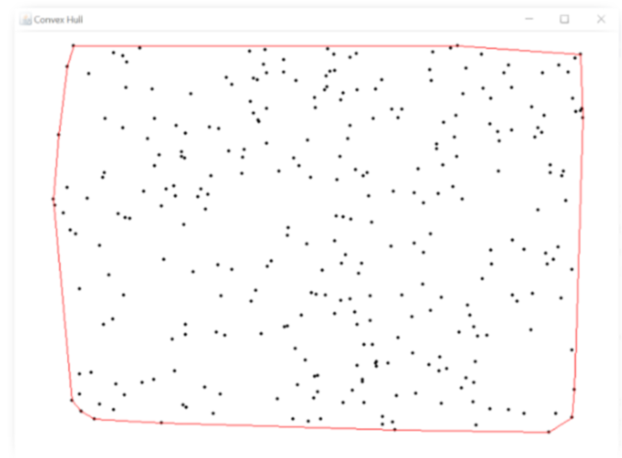

Fig. 5. The resulting convex hull

4. Results and discussion

In this paper, we measured the proposed algorithm's execution time to determine if filtering the input points before computing the convex hull would enhance the original Graham scan algorithm [5].

We implemented the proposed algorithm using the Java programming language in NetBeans IDE 8.2. No parallel programming was used in our implementation. All of the experiments were conducted on an HP Pavilion with a 1.80 GHz Intel® Core™ i7-8565U CPU and 8 GB RAM, running the Windows 10 Home operating system, version 1903 (OS Build 18362.418). The resulting output is illustrated in Fig. 5, with a random normal distribution point set.

For benchmarking, the proposed algorithm was compared to two algorithms: Graham scan [5] and Jarvis march [6]. For the input point set, we randomly generated points in a two-dimensional domain in the range [100,000... 1000,000], as shown in Table 1. Each point was defined as a pair of floating-point numbers. The execution times were measured in milliseconds.

According to Fig. 6, the proposed algorithm has a faster execution time than the Graham scan and Jarvis march algorithms. Additionally, as the number of points increased, the execution time of the Graham scan algorithm also increased, while the proposed algorithm presented better performance than even the Jarvis march algorithm. The original Graham scan algorithm halted at 300,000 points because it has large computation overheads and thus could not handle a dense number of points. However, the proposed algorithm showed a significant enhancement and successfully processed up to one million points in a short time. As shown in Table 2, the execution time of the proposed algorithm was around 9 ms for 100,000 points, but the execution time gradually decreased as more points were added.

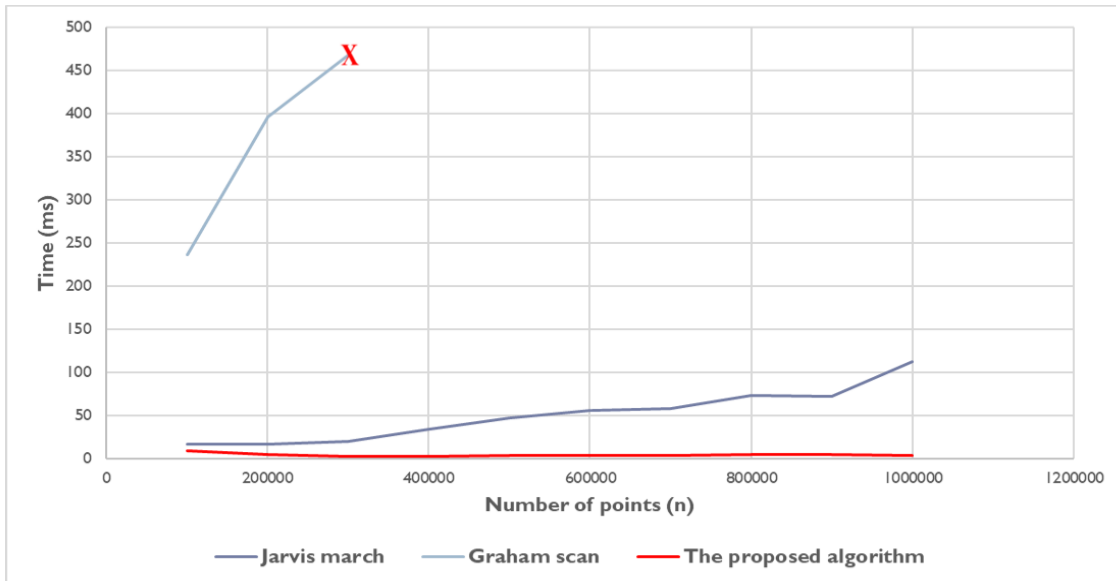

Fig. 5. Execution times of Jarvis march, Graham scan, and the proposed algorithm.

Although the analyzed time complexity of the proposed algorithm is *O (n log)*, which is the same as in the Graham algorithm, the proposed algorithm showed enhanced performance in real execution scenarios, outperforming boththe Graham scan and Jarvis march algorithms.

The results illustrate that the preprocessing approach presented in this paper reduces the execution time of the Graham scan by up to 77 times. Also, it is 12 times faster than the Jarvis march algorithm. Thus, the preprocessing technique can be applied to other algorithms, especially convex hull algorithms.

Table 2. Time performance (s) of Jarvis march and Graham scan algorithms versus the proposed algorithm.

Number of points (n)	Jarvis march (ms)	Graham scan (ms)	Proposed algorithm (ms)
100,000	17.4202	237.1718	9.7401
200,000	16.9774	396.4281	4.717
300,000	20.471	467.7013	2.8273
400,000	34.7065	-	2.8936
500,000	47.7725	-	4.6297
600,000	55.8307	-	3.7483
700,000	58.9525	-	3.7075
800,000	73.7883	-	4.7002
900,000	72.8367	-	4.8417
1,000,000	113.0887	-	4.4846

5. Conclusions

Enhancing the convex hull algorithm by reducing the interior points for fast convex hull computing has been of interest to computer scientists for decades. This work has presented a preprocessing approach for the Graham scan algorithm to compute a convex hull for a random set of points in two-dimensional space.

The main contribution of this paper is enhancing the Graham scan algorithm by adding filtering techniques for fast convex hull computing.

The benefits of speed and memory saved through the proposed method were better when a significant number of the input points lay inside the convex hull. This scenario is a typical case in many applications where points are presented as clusters of points.

Experimental results based on normal distribution point sets showed that the proposed algorithm outperforms the original Graham scan in terms of execution time by up to 77 times. Thus, the preprocessing technique presented here can be a useful approach for many practical problems that require fast computation of a convex hull. Moreover, the filtering technique can be extended to higher dimensions. In future work, it would be interesting to study how this method of performance can be further improved through parallel computing, with modern GPUs that offer parallel computation for highly parallelizable problems, as well as to illuminate geometric computations.

References

[1] Jayaram, M., Fleyeh, H. (2016) "Convex Hulls in Image Processing: A Scoping Review." American Journal of Intelligent Systems 6 (2): 48-58.

[2] Mei, G. (2016). "CudaChain: an alternative algorithm for finding 2D convex hulls on the GPU." SpringerPlus, 5(1).

[3] SOUTO, N. (2019). "Video Game Physics Tutorial - Part II: Collision Detection for Solid Objects." [online] DEVELOPERS.

[4] Graham, RL. "An efficient algorithm for determining the convex hull of a finite planar set." Inform Process Lett 1972; 1(4).

[5] Jarvis, RA. "On the identification of the convex hull of a finite set of points in the plane." Inform Process Lett 1973; 2(1)

[6] Franco P. Preparata. S.J. Hong. "Convex Hulls of FiniteSets of Points in Two and Three Dimensions," Commun. ACM, vol. 20, no. 2, pp. 87–93, 1977.

[7] A.M. Andrew. "Another efficient algorithm for convex hulls in two dimensions." Information Processing Letters, Volume 9, Issue 5, 16 December 1979, Pages 216-219.

[8] Kallay, M. "The complexity of incremental convex hull algorithms in Rd." Information Processing Letters, Volume 19, Issue 4, 12 November 1984, Page 197.

[9] Barber CB, Dobkin DP, Huhdanpaa H. "The quickhull algorithm for convex hulls." ACM Transactions on Mathematical Software (TOMS), Volume 22 Issue 4, Dec. 19

[10] Stein, A., Geva, E. and El-Sana, J. (2012). "CudaHull: Fast parallel 3D convex hull on the GPU." Computers & Graphics, 36(4), pp.265-271.

[11] Qin, J., Mei, G., Cuomo, S., Guo, S. and Li, Y. (2019). "CudaCHPre2D: A straightforward preprocessing approach for accelerating 2D convex hull computations on the GPU." Concurrency and Computation: Practice and to imExperience, p.e5229.

[12] Tang, M., Zhao, J., Tong, R. and Manocha, D. (2012). "GPU accelerated convex hull computation." Computers& Graphics, 36(5), pp.498-506. 96, Pages 469-483.

[13] Gao, M., Cao, T., Nanjappa, A., Tan, T. and Huang, Z. (2013). "gHull.: A GPU algorithm for 3D convex hull." ACM Transactions on Mathematical Software, 40(1), pp.1-19.

[14] Das, Patita & Singh, Laiphrakpam & Kar, Nirmalya. (2013). "A Pre-processing Algorithm for Faster Convex Hull Computation." IET Conference Publications. 2013. 8.01-8.01. 10.1049/cp.2013.2348.

[15] M. Sharif, “A new approach to compute convex hull,” Innov. Syst. Des. Eng., vol. 2, no. 3, pp. 186–192, 2011.

[16] Ferrada, H., Navarro, C. and Hitschfeld, N. (2019). "A filtering technique for fast Convex Hull construction in R2." Journal of Computational and Applied Mathematics, 364, p.112298.

[17] Cadenas, O. and Megson, G. (2019). "Preprocessing 2D data for fast convex hull computations." PLOS ONE, 14(2), p.e0212189.

[18] Gomes, A. (2016). "A Total Order Heuristic-Based Convex Hull Algorithm for Points in the Plane." Computer-Aided Design, 70, pp.153-160.

[19] D. Cañas, A. Orozco, and L. Villalba, “An Extension Proposal of AntOR for Parallel Computing,” *J. Ubiquitous Syst. Pervasive Networks*, vol. 3, no. 2, pp. 67–72, 2011, doi: 10.5383/juspn.03.02.005.

[20] M. Mansour, A. Ghneimat, R. Alasem, F. Jarray, and A. U.-Jordan, “Performance Analysis and Functionality Comparison of First Hop Redundancy Protocols,” vol. 15, no. 1, pp. 1–9, 2021, doi: 10.5383/JUSPN.15.01.007.