

# The Quickhull Algorithm

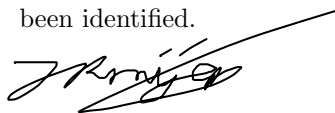
Student number: 700046371

Word Count: 1481

## Abstract

**Abstract** This paper examines the quickhull algorithm. The quickhull algorithm is used to calculate the convex hull of a set of vertices. It examines the main principals of the algorithm, pseudo code of the algorithm , conducts a time and space complexity analysis, the algorithms limitations and real life applications.

I certify that all material in this dissertation which is not my own work has been identified.

A handwritten signature in black ink, appearing to read 'J. Krüger', is written over the certification text.

# 1 Introduction

Quickhull is an algorithm that is used to compute the convex hull of a finite set of points in n-dimensional space(1). A convex hull is the smallest set of vertices that contains each vertex in a convex set(2). A convex set is a set of vertices that contains the complete straight line segment between any two given points in the set. An example of a convex set is a triangle or pyramid. An example of a non convex set is a hollow or indented object such as a donut or a crescent moon. It derives its name from the quicksort algorithm because it uses a similar method.

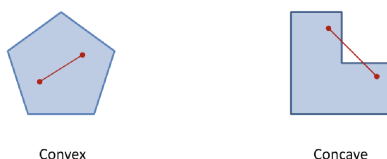


Figure 1: Convex vs concave shape

## 2 Main Principals

It is an iterative and recursive algorithm and uses a divide and conquer technique similar to quicksort(1; 3). It is iterative because it adds one point at a time. It is recursive because it calls itself with inputs decreasing in size or complexity until it reaches a base case. The algorithm then recursively iterates using the base case as the first result and performing operations on each result. As the number of dimensions increases the complexity of the code increases.

First the algorithm selects the points with the minimum and maximum x coordinates to be part of the convex hull (xmin and xmax). In the case of multiple points satisfying that criteria the points with the minimum and maximum y should be chosen. The reason these are chosen is because they will always be part of the convex hull. A line (L) is drawn between these 2 points. This line splits the set into 2 sections. The algorithm then recursively handles each section independently.

Calculate a point P such that it has the maximum distance from L. P then forms a triangle with the initially selected points xmin and xmax. Any points that are inside of this triangle cannot be part of the convex hull and are therefore ignored. This is because if they were to be included the set of points would not be a convex hull, no longer being the smallest set of vertices containing the set. The line joining P and xmin and the line joining P and xmax are newly formed lines. The points outside the triangle are the new set of points. Repeat this passing through the new set of vertices and calculating a new furthest point

P each time until there are no more points left. This is the base case being reached. The function will then return the convex hull.

### 3 Pseudo code

```
def quickHull(Set):
    convex hull = []
    if length of Set < 2:
        return Set

    Find farthest left point (xmin) and right point (xmax) and add them to the
    convex hull
    Create a line segment (L) from these 2 points
    Split the set of points into a set above and below the line. S1 is those above
    and S2 is those below the line
    recursivequickhull(S1, xmin, xmax, above)
    recursivequickhull(S2, xmin, xmax, below)
    Add the results to the convex hull
    return convex hull

def recursivequickhull (S, P1, P2):
    if S is empty:
        return
    Find the point (Q) furthest from the line segment P1-P2
    Find and remove the points inside Q,P1,P2 from S
    Add point Q to the convex hull
    Separate the points based on where they are compared to the line segments
    Q-P1 and Q-P2. S1 is those above and S2 is those below.
    recursivequickhull(S1, P1, Q)
    recursivequickhull(S2, Q, P2)
    add the results to the convex hull    return convex hull
```

## 4 Time and Space complexity analysis of the algorithm

### 4.1 Time complexity

The quickhull algorithm greatly increases the time complexity to find the convex hull of a shape compared to brute force.

The quickhull algorithm needs to perform  $n$  computations on average on every point in a set. This is because the sets are split based on a line segment that pass through 2 extreme points. Finding these extreme points requires  $O(n)$  computations. Therefore the time complexity is dependent on the size of the

sets passed through in each recursion. In the best case scenario the set is split in two with each recursion with  $n$  operations being done on a set so  $T(n) = 2 * T(n/2) + n$ . This is a common time complexity for a recursion algorithm and has been solved to be  $O(n \log n)$ .

However we will demonstrate this fact. We do so using case 2 of the master theorem which solves equations that have the format of  $T(n) = aT(n/b) + f(n)$  (4). Where  $n$  = size of input,  $a$  = number of subproblems,  $f(n)$  = the work done aside the recursive call and both  $a$  and  $b$  are greater than 1 and  $f(n)$  is asymptotically positive. In the second case of the master theorem if  $f(n) = \Theta(n^{\log_b(a)})$  then  $T(n) = \Theta(n^{\log_b(a)} \log(n)) = \Theta(\log n)$ .

In the best case scenario  $f(n) = n$ ,  $a = 2$  and  $b = 2$ . Therefore  $f(n) = n^{\log_b(a)}$ . This is because  $\log_b(a) = \log_2(2) = 1$  therefore  $n^{\log_b(a)} = n^1 = n = f(n)$ . Therefore the time complexity in the best case is  $T(n) = n \log n$ .

The worst case the quickhull algorithm is where every single vertex is part of the convex hull. This means that there are no vertices contained within the hull that can be ignored and do not have to be computed, decreasing the time complexity. This is the worst case because it requires an  $O(n)$  calls to quickhull each of which requires an  $O(n)$  computation on average. Therefore the time complexity is  $O(n^2)$  in the worst case.

## 4.2 Space Complexity

The convex hull problem has a space complexity of  $O(n)$  where  $n$  is the number of vertices. This is because each call to the recursive function in quickhull partitions the existing set  $S$  into 3 separate sets. The first two are the different sets that are being recursively passed to the quickhull algorithm and the remaining set is the vertices that are within the convex hull. Between these three sets there is no duplication of any vertices. Therefore at any point in time the computer only needs to have enough memory to record  $O(n)$  vertices. This can be reduced by discarding the points that are within the convex hull. However in the worst case scenario for this each vertex is on the convex hull and no points are discarded.

## 5 Limitations

The quickhull algorithm is only suitable for a set of vertices in 2D space. This means it cannot be used for 3D objects. The quickhull algorithm can however be modified to  $n$  dimensions using the same divide and conquer principles. However it has a decreased time and space complexity (3; 1).

The algorithm is also most suitable to calculate points that distributed uniformly, far away from one another and are small. This is because the algorithm depends on separating the vertices depending on which side of a line they are located. If the points are clustered together then with each division only a small number of points may be inside the convex hull, making the algorithm inefficient. Whereas if they were more spaced out with each division more points

would be eliminated. The algorithm needs to divide numbers to decide which side of a line a point is on which means that there may be floating point precision errors. Therefore the algorithm may produce incorrect results if the coordinates are very large. It can also produce errors for this reason if the coordinates are very close to one another.

This can be solved by analysing the set of vertices and determining whether quickhull is the best algorithm to use for this set or if another algorithm is more suitable.

The worst case time complexity for the quickhull algorithm is  $O(n^2)$ . For very large sets of points this may be very slow. This can also be addressed by determining whether it is likely that a lot of points will be inside or part of the convex hull.

## 6 Application

The quickhull algorithm is very useful in any field where convex hulls need to be found.

This is especially prevalent when producing computer graphics(5). This means that it has many applications in any field that utilise this such as gaming or animation. In gaming complex objects are usually approximated using the convex hulls of simpler shapes for collision. This is because it is much more efficient to do so than using render geometry. Quickhull provides a fast and simple way to calculate these convex hulls.

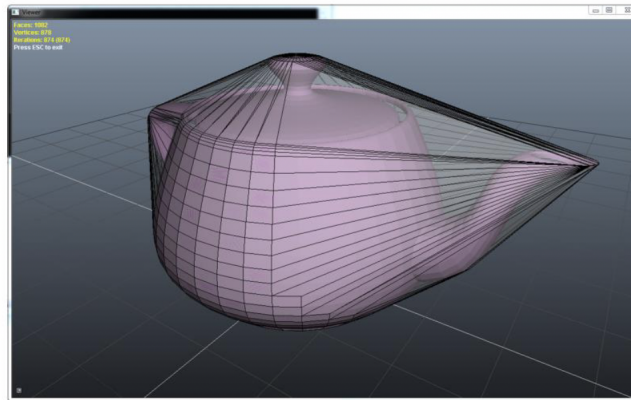


Figure 2: 3d convex hull of a teapot

The quickhull algorithm is also used in various fields of mathematics and computational geometry(1; 3). This can be to help construct voronoi diagrams or draw a Bézier curve. This is because a Bézier curve must lie within the convex hull of its control points.

The quickhull algorithm is used in many more fields such as economics, robotics and statistics as it is a widely documented and well known algorithm that is efficient and simple to implement. (3)

## References

- [1] E. Mucke, “Computing prescriptions: Quickhull: Computing convex hulls quickly,” *Computing in Science and Engineering*, vol. 11, no. 5, p. 54–57, 2009.
- [2] “Convex hulls,” *Algorithmic Geometry*, p. 125–126, 1998.
- [3] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa, “The quickhull algorithm for convex hulls,” *ACM Transactions on Mathematical Software*, vol. 22, no. 4, p. 469–483, Jan 1993.
- [4] A. Bawa, “Master theorem for time complexity analysis,” Jan 2022. [Online]. Available: <https://iq.opengenus.org/master-theorem-time-complexity/>
- [5] D. Gregorius. [Online]. Available: [https://steamcdn-a.akamaihd.net/apps/valve/2014/DirkGregorius\\_implementingQuickHull.pdf](https://steamcdn-a.akamaihd.net/apps/valve/2014/DirkGregorius_implementingQuickHull.pdf)