Implementation of Convex Hull Algorithms in 3D space

Anand Balan	Gunjan Munjal	Tejas Ravi Rao
axb173130	gxm171430	txr171830

ABSTRACT

We attempt to implement three proposed and well-established 3D convex hull algorithms – divide and conquer, randomized incremental construction and quick hull and study the practical issues while implementing these algorithms. We also discuss the practical complexity against the theoretically guaranteed complexity and the bottlenecks in the implementation.

1. Motivation

Convex hull is a well-studied problem in computational geometry and a large number of algorithms have been proposed to solve the problem. There exist multiple algorithms, both deterministic and non-deterministic, that determines convex hull in optimal time. However, there is no study on the practical feasibility and the limitations of implementing these algorithms.

3D Convex hull is a much more complicated problem to solve than its 2D counterpart. The most efficient algorithms devised for both 2D and 3D run in *O(nlogn)* time. In this paper, we study three 3D convex hull algorithms that have the most efficient theoretical time complexity – namely, Preparata, F.P. and Hong, S.J. (1977)'s [1] divide and conquer-based algorithm, randomized incremental construction [6] and quick hull[2] .

2. Background

The convex hull, or convex closure for a set of points X in the Euclidean space is the smallest convex set that contains X. We may visualize convex hull to be a shape enclosed by a rubber band stretched around X. [10]

The convex hull is a ubiquitous structure in computational geometry. Even though it is a useful tool in its own right, it is also helpful in constructing other structures like Voronoi diagrams, and in applications like unsupervised image analysis.

The application domain of convex hull is enormous. Some of the areas in which convex hull plays a critical role include:

- Computer Visualization (e.g. video games, replacement of bounding boxes)
- Path Finding (e.g. embedded AI of Mars mission rovers)
- Geographical Information Systems (GIS) (e.g. computing accessibility maps)
- Visual Pattern Matching (e.g. detecting car license plates)
- Geometry (e.g. diameter computation)

3. Approach

In this section, we provide a simplified description of the algorithms under consideration for this project and the approach we took to implement these algorithms. We also discuss the practical time complexity that these algorithms achieve and the limitations, if any, that we faced while implementing these algorithms.

3.1 Divide and Conquer

The algorithm, as the name suggests, works by dividing a set of points into two halves and solving both halves separately and recursively and then by merging the two convex hulls into a single convex hull. We suggest readers to refer the original paper on divide and conquer by Preparata and Hong[1] for a more comprehensive description and validity of the algorithm. This paper will focus more on the implementation aspects of the algorithm.

For the purpose of this project, the algorithm is implemented in Java. We represent the input to the algorithm as a set of distinct *Point*. Each *Point* comprises of three properties denoting the x, y, and z coordinates in 3D space. The output and the intermediate convex hulls are represented by a *HashMap* with Point (or vertex) as the key and a set of Points (or vertices) as the value. Particularly, we use *LinkedHashSet* for the set to maintain ordering while also achieving *O(1)* access, insertion and deletion time. This representation of convex hull basically is an adjacency list of all edges that are present in the hull.

3.1.1 Divide Phase

In the divide phase of the algorithm, we sort the given set of points based on the y-coordinate. Storing the set of points as an array provides an ideal way to sort and split the points. Our implementation, instead of splitting the points array, uses two pointers that indicates the range of points to be considered and each recursive call works only on this range of the points in the original array. The sorted set of points are then split into upper and lower halves and processed separately. We keep splitting the points until the number of points in the set is less than or equal to 4, in which case we have a tetrahedron (4 points) or triangle (3 points) or a line (2 points) or a point.

3.1.2 Merge Phase

The whole bulk of the algorithm lies in the merge phase. The algorithm aims for a runtime on the order of *O(n)* during the merge phase which, consequently, guarantees an overall time complexity of *O(nlog(n))*.

The inputs to the merge phase are two convex hulls formed from the previous step. Merge phase works by using a generalization of gift-wrapping algorithm and obtaining a cylindrical triangulation between the two hulls and then using this triangulation as a bridge between the two hulls.

3.1.2.1 Triangulation Initialization

As the initial step of this phase, the two 3D hulls are projected on the xy-plane and corresponding 2D hulls are obtained. We use graham scan algorithm to get the 2D hull of

the projected points. The issue here is that since the complexity of graham scan is O(nlogn) in the worst case, and since we run this at each merge step, our overall time complexity cannot be guaranteed to be O(nlogn) anymore.

Graham scan returns a 2D hull with points sorted in certain order. From the two 2D hulls obtained, we find the right tangent connecting the two hulls which forms the first edge of the triangulation. The original paper [1] describes this step in depth. We now form the initial triangulation by creating a triangular face parallel to the z-axis. This can be easily done by taking the upper end (t_u) or lower end(t_l) of the tangent and by setting the z-coordinate to *min* ($z(t_u)$, $z(t_l)$)*-1*.

3.1.2.2 Advancing Mechanism

The next step in the merge phase is to advance the triangulation cylindrically around the two convex hulls, described as "advancing mechanism" by Preparata and Hong. Let *u* and *l* be the two points from upper and lower hulls that are recently added into the triangulation. We maintain two pointers for these points. We also maintain the most recently added face *f*. The most recently added face can be obtained by maintaining a third pointer (t). This third pointer gets updated as follows –

i. If the next point in triangulation is added from upper hull, the third pointer gets updated to *u* and *u* gets updated to the next point.
ii. If the next point in triangulation is added from lower hull, the third pointer gets updated to *l* and *l* gets updated to the next point.

To find the next point in triangulation, we take all edges incident on *u* and on *l*. We find the convex angle between faces (*u*, *l*, *next*) and *f* and sort the edges in the clockwise direction with respect to the face *f*. However, we cannot use a comparison-based sorting algorithm such as merge sort or quick sort as the time complexity of these algorithms in the worst case can be O(nlogn) or $O(n^2)$ respectively. We use bucket sorting to place points into one of the 360 buckets (one bucket per degree angle) and sort each bucket separately. Since bucket sorting runs in linear time, we can get the clockwise ordering of the edges in linear time.

Since sorting the edges is a core part of the algorithm, we discuss the implementation of finding angle between two faces in 3D in further detail. In our implementation, each face is represented by three points. We find normal vector to the face, for both faces, by constructing two vectors from any two sides of the face. We then apply cross product to these vectors to obtain the face's normal vector. Now, finding the angle between faces is as simple as finding the angle between their corresponding normal vectors. This can be done by calculating the dot product of the normalized vectors and taking the inverse cosine.

$$\cos(\theta) = \frac{v1.v2}{|v1|*|v2|} \qquad \text{radians, where } \theta \text{ is angle between v1 and v2.}$$

$$\theta = \arccos\left(\frac{v1.v2}{|v1|*|v2|}\right) * \frac{180}{\pi}$$

However, the problem here is that in 3D space, angle between two 3D vectors are always in the range [0,180]. However, our program requires the convex angle in the clockwise direction in the range [0, 360).

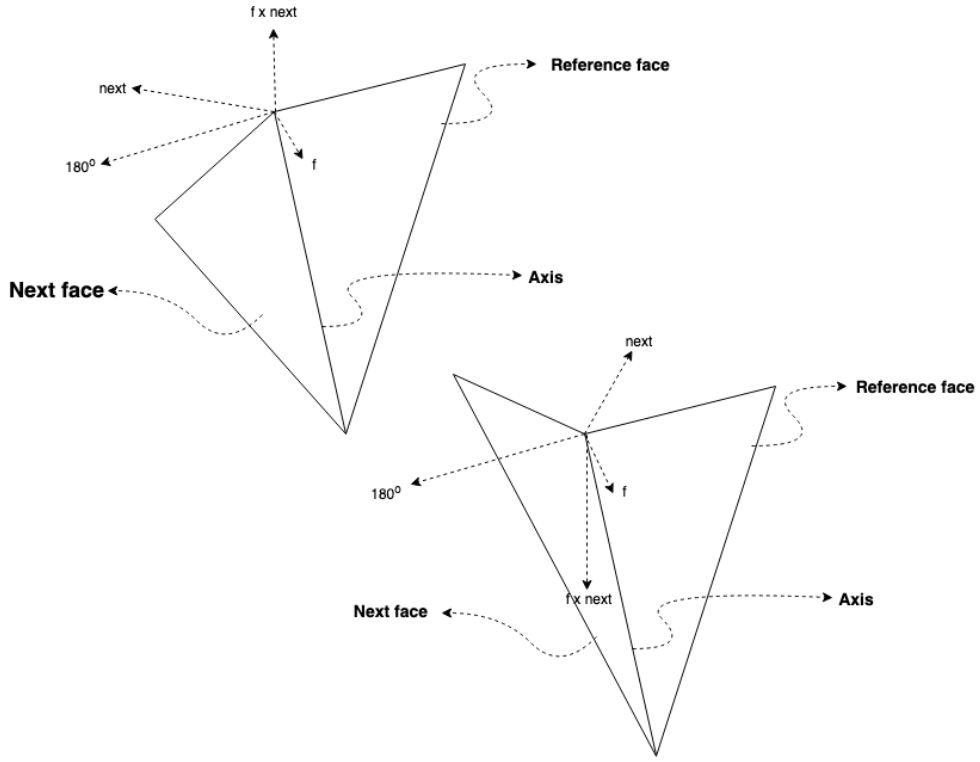


Figure 1*: The cross product of the normal vector changes direction every 180°. Due to this, the direction of measured angle also inverts resulting in convex angle range of [0,180].*

To solve this problem, we see the sign of the cosine angle between the normal vectors. If the sign is negative, we know the convex angle is calculated in anticlockwise direction. So, we subtract the θ from 360 to obtain the clockwise angle. If the sign is positive, no other work is needed.

From the sorted set of edges, we then find the edge (u_{max}) that forms the largest convex angle with the current face f. In the process, any edge in between u and u_{max} are internal to the hull and they will not be processed again. We perform a similar operation on lower hull as well and determine l_{max}. We then perform a "run-off" comparison between u_{max} and l_{max} and the vertex that forms maximum convex angle is chosen as the next point. The internal vertices of the corresponding hull, as determined in the previous step, are completely isolated from any other vertex in the hull. We repeat this process until both upper and lower hulls run out of edges.

One caveat while implementing previous step is that the edge-sorting must be performed for each step in the advancing mechanism. In the worst case, no vertex might become an inner vertex and the first clockwise-ordered vertex could be the maximum convex angle vertex to f. In such a scenario, the time complexity increases to O(EV) where E is the number of edges in the hull and V is the number of vertices in the hull.

3.1.2.3 Isolating Internal Vertices, Adding Triangulation Edges and Merging Hulls

In the final step of the merge phase, we use the two convex hulls and the triangulation obtained from previous step to construct a single merged convex hull. In our implementation, all the edges of the two convex hulls are first added into a single merged resulting convex hull. Then, we recreate faces from the triangulation and again sort the edges with respect to the face in the current iteration and isolate all internal vertices from the resulting convex hull. This process could again take O(VE) time to isolate all internal vertices.

After the internal vertices are isolated, we run through the triangulation and add all edges from triangulation that connect the upper and lower hulls. We did not remove edges within a hull that are a part of triangulation, so we do not have to add them.

3.2 Random Incremental Algorithm

The random incremental algorithm is a 2-step algorithm. It starts by choosing four points that do not lie in the same plane, thereby forming a tetrahedron. Taking this tetrahedron as the base, more points are added, maintaining the convex hull as we proceed. The implementation is based on the algorithm presented in the book "Computational Geometry: Algorithms and Applications, third edition. Mark de Berg, Otfried Cheong, Marc Van Kreveld, Mark Overmars".

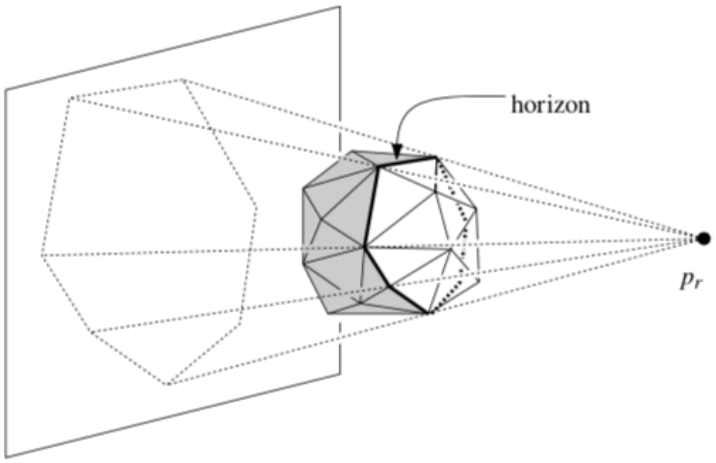
For this project, the implementation of Random Incremental Algorithm is done in Python. The input is a set of distinct points read from a file. Each point is represented in 3-Dimensional space. The class Face, Edge, and Vertex store the faces, edges and vertices present in the convex hull respectively.

3.2.1 Initialization Step

For the construction of the initial tetrahedron, four points are selected from the given set of points such that they do not lie on the same plane. This is done as follows.
Any two points, say p_1 and p_2, are randomly picked. The third point, p_3, is picked such that it is not collinear with the first two points, such that the three points form a plane. We continue searching P until we find the fourth point p_4 such that it does not lie on the plane passing through p_1, p_2, and p_3. If no such point is found, then all points in P lie on a single plane and the program raises an exception saying 3D convex hull is not possible.

3.2.2 Adding a Point

The incremental algorithm successively computes $CH(P_5)$, $CH(P_6)$, . . ., $CH(P_n)$. In each step, we find $CH(P_i)$ by inserting the point p_i into $CH(P_{i-1})$. Each convex hull is stored as a Doubly Connected Edge List (DCEL), representing the planar graph $G(CH(P_i))$.



Before inserting the next point to the current convex hull, we determine the facets of the hull that are visible to the point. Only the facets which are visible to the point plays a role in updating of the convex hull. All the facets which are visible to p must be deleted when p is inserted.

3.2.3 Visibility Criteria

To check the visibility of a face of the convex hull from a point p, the volume of the tetrahedron formed by joining the point p with the face vertices is computed. The volume is positive if the counterclockwise normal to the face points outside the tetrahedron, which means that the point is outside the convex hull and the face is visible to the point. If the volume is negative that means the face is either not visible to the point or the point lies inside the convex hull. In either case, no processing for the face is required and hence the visibility flag of the face remains negative. Position of the point w.r.t to the convex hull is computed using the right-hand thumb rule.
If there is no visible face to the point p_i, then p_i lies inside of $CH(P_{i-1})$, and nothing has to be done. We mark the point as "Processed" and randomly pick another unprocessed point. Otherwise, the facet visible to the point p_i must be deleted. We check for conflict of the point with all the faces of the hull and mark the visibility property of the faces as "true". After checking for all the faces, the faces marked as "true" are deleted. If the face on both sides of an edge is deleted, the edge in not a part of the horizon and is removed from the set of edges of the convex hulls. If only one face on either side of an edge is removed, that means the edge is a part of the horizon and a new face from the edge to point p is to be added to the list of convex hull faces. This concludes the description of one insertion. This process is repeated until all points have been processed i.e. $CH(P_n) = CH(P)$.

3.3 Quick hull

The Quick hull algorithm is used to compute the convex hull of a finite set of points in 3D space. It is derived from the original Quick hull algorithm devised to compute the convex hull of a set of points on a 2D plane. The algorithm uses a similar approach like the quicksort. The average case time complexity of quick hull is *O(nlogn)*, whereas the worst-case complexity is $O(n^2)$.

For this project, the Quick hull algorithm has been implemented in Python. The implementation, although not entirely, derives certain design cues obtained from a research paper based on said algorithm by Barber, Dobkin & Huhdanpaa[2] , from a proof of the algorithm by Greenfield[3] and from a presentation at Game Developers Conference,2014[4].

As mentioned earlier, we represent the input to the algorithm as a set of distinct Points/Vertices. We represent each point denoted by their respective x, y and z coordinates in 3D space. The output however is represented as a *Set* of *Vertices* and *List* of resultant *Faces*/*Planes* formed by these Vertices. Together, both the structures describe the final output for the 3D Convex Hull.

3.3.1 Overview

Our implementation initially constructs the tetrahedron with 4 extreme points. We maintain a Set of vertices called the *Outside Set* for each of the 4 faces of the tetrahedron. We pick the point from the *Outside Set* that is farthest from the respective face known as the *Apex point.* The next step is determining the *Horizon* of the hull that would facilitate in identifying the visible faces in front of the *Apex* point. A Depth First Search is performed to find all the edges of the *Horizon*. The Apex point is then joined to each of these edges thus forming new faces of the intermediate hull. We repeat this until all *Apex* points are consumed thus giving us the final Set of Vertices of the Convex Hull.

3.3.2 Initial Construction

The first step in our implementation is to determine the initial extreme points along each of the axes from our set of input vertices. We, therefore, obtain 6 extreme points i.e. 2 points each that make up for the positive and negative axis. These points are obtained by checking against boundary conditions of our given 3D space and are thus determined in O(n) time. We then select two points from these 6 points that are farthest apart and construct a line between them. This step is like the one from the 2D Quick hull Algorithm. This is done in O (1) time.

We then select a point that is the farthest from the above constructed line to form our first plane in O(n) time. Similarly, the next step is to find the farthest point from the constructed face. We join the farthest point to this face to form our initial tetrahedron. In all, we complete the construction in O(n) time.

3.3.3 Outside Set and Horizon

With each new face constructed, we determine an orthogonal vector for the said face. As a result, it becomes feasible to determine the orientation of a new point with respect to a face based on the signed distance of the point to the face.[5] Points that have a positive signed distance (in direction of the orthogonal vector of the face) to the face are part of the *Outside Set* of the face i.e. these points are in front of the face.[2]

After the construction of the initial tetrahedron, the *Outside Set* of each of the 4 faces are updated and we include these 4 faces in our List of resultant faces. Finding the *Outside Set* of each face is an O(n) operation.

Now, iteratively for every face from our List of resultant faces, we determine an *Apex* point in front of the said face. It is imperative to determine the *Horizon* once the *Apex* point for the face is determined. The *Horizon* is determined by performing a Depth First Search along the edges of adjacent faces.[4] The search therefore marks all those edges that form the *Horizon.* The *Horizon* may initially enclose a single face, but as we add more *Apex* points to the hull and more faces are formed, the *Horizon* may enclose a greater number of faces. At any step, the resultant *Horizon* will separate those faces that are visible to the *Apex* point from those that aren't visible to the *Apex* point. The next step is to join the *Apex* point to the edges that make up the *Horizon* thus forming new faces. However, it is imperative to note that the internal faces enclosed by the *Horizon* are no longer needed and are hence removed from our List of resultant faces.

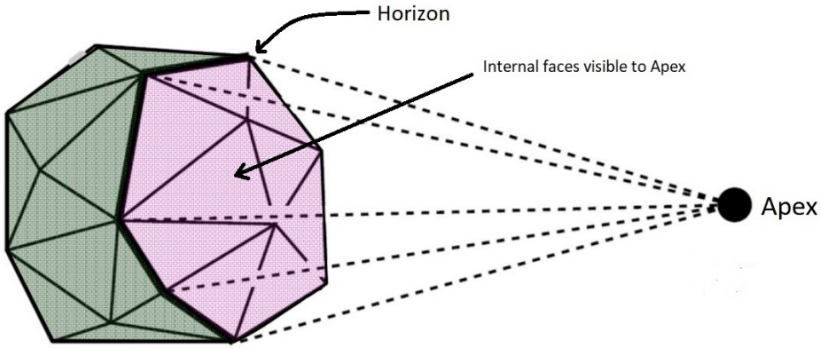


Figure 2*: The Horizon obtained after performing DFS. The marked edges depict the Horizon. The internal faces are visible to the Apex point. Dotted lines indicate new faces being formed.*

The *Outside Sets* of these removed faces are merged together and are used to update the *Outside Set* of newly formed faces. We then add the newly formed faces to the List of resultant faces. As mentioned earlier, obtaining the *Outside Set* of each face is an O(n) operation and finding the *Apex* point from the *Outside Set* is O(n). In all, we incur O(n) time for each face.

4. Results

4.1 Divide and Conquer

The theoretical algorithm as described by Preparata and Hong guarantees a time complexity of O(nlogn) which is among the most efficient, deterministic 3D convex hull algorithm. But, practical implementation of the algorithm is more difficult and does not ensure the same time complexity due to limitations in computer programs. For example, sorting the edges in clockwise order is required at each iteration of the triangulation because the reference face is different at every iteration and hence the ordering of the edges will differ at each iteration. However, running a linear sorting algorithm for each iteration would increase the time complexity beyond linear time. The time complexity that our implementation achieves is $O((n\log n+VE)\log n) = O(n\log^2 n + VE\log n)$.

4.2 Random Incremental

The time complexity of the algorithm depends on the structural updation of the convex hull and checking the visibility between the point and the faces.

Structural Updation:
Each facet is deleted at most once, so it suffices to bound the number of facets that are created. In the worst case, this can be quadratic.

Checking for conflict:
While checking for visibility of a face to the point, each facet f is traversed at most once for each point. Thus, the total time for the conflict change is proportional to the total number of conflicts that exist between the created facets and the points in P. This makes the run time quadratic.

4.3 Quick Hull

The theoretical algorithm from the research papers guarantee an average case complexity of O(nlogn) which leads to Quick hull being one of the fastest and widely used 3D convex hull algorithms.[2]

However, the practical implementation of the algorithm is limited by the infeasibility in sorting the *Apex* points beforehand with respect to each newly formed face in constant time. Therefore, picking an Apex point does not become an O (1) operation. With every new face formed, we incur time complexity with respect to obtaining the *Outside Set* and determining the *Apex* point. We also incur time complexity with respect to initial construction. If we were to mention *m* to be the total number of faces of our 3D convex hull and every face contributes to O(n) time i.e. each face incurs cost for operations - linear in size of our input vertices, the complexity of our implementation would approximate to *O(mn).*

References

1. Preparata, F.P. and Hong, S.J. (1977) Convex Hulls of Finite Sets of Points in Two and Three Dimensions. Communications of the ACM, 20, 87-93.

2. Barber, C.B., Dobkin, D.P., and Huhdanpaa, H.T., "The Quickhull algorithm for convex hulls," ACM Trans. on Mathematical Software, 22(4):469-483, Dec 1996, http://www.qhull.org.

3. Greenfield, Jonathan Scott, "A Proof for a QuickHull Algorithm" (1990). Electrical Engineering and Computer Science Technical Reports. 65.

4. GDC 2014: Dirk Gregorius - "Physics for Game Programmers: QuickHull"

5. http://mathworld.wolfram.com/Point-PlaneDistance.html

6. https://www.cs.jhu.edu/~misha/Spring16/10.pdf

7. Computational Geometry: Algorithms and Applications, third edition. Mark de Berg, Otfried Cheong, Marc Van Kreveld, Mark Overmars

8. https://michelanders.blogspot.com/2012/02/3d-convex-hull-in-python.html

9. http://www.bowdoin.edu/~ltoma/teaching/cs3250CompGeom/spring17/Lectures/cg-hull3d.pdf

10. https://brilliant.org/wiki/convex-hull/

11. https://en.wikipedia.org/wiki/Convex_hull