CISC868

Implementations of 3D Convex Hull Algorithms

Clemens Oertel

July 7, 2003

1. Introduction

Convex hulls are one of the major structures in computational geometry. They are involved in a vast variety of applications, including robotics (especially collision detection), cluster analysis, image processing and statistics. Many geometrical problems can be reduced to the convex hull — the relationship between Delaunay triangulations, Voronoi diagrams and convex hulls has been discussed in class.

It is obvious that for many of these applications, a precise as well as efficient technique to calculate the convex hull is desirable.

My presentation in class covered a randomised incremental algorithm that constructs the convex hull of a set of points in three dimensions, requiring $\mathcal{O}(n^2)$ time. The code of O'Rourke's implementation of the algorithm is provided separately. For comparison, this report also includes an overview of QuickHull, which is output sensitive and runs in $\mathcal{O}(n \log v)$ time (v begin the number of output vertices).

2. Incremental Algorithm

The first approach to be described is a standard randomised incremental algorithm. As the name denotes, it iterates over all points in random order.

The underlying principles of the algorithm are quite simple: At each iteration, i.e. for each point p_i, the location of p_i relative to the hull H_{i-1} that has been constructed so far is determined. If p_i is found to be outside the hull, it is added to the hull. Otherwise, p_i is disregarded.

Figure 1 shows one iterative step. In part (a), a point p_i is shown as well as the convex hull H_{i-1} that has been created during previous iterations. In this case, p_i is outside the hull, therefore it is added to the hull, as can be seen in part (b).

How do we test whether a point p_i is inside the current convex hull H_{i-1}? Using the volume sign of a tetrahedron consisting of a face F of the hull and p_i,

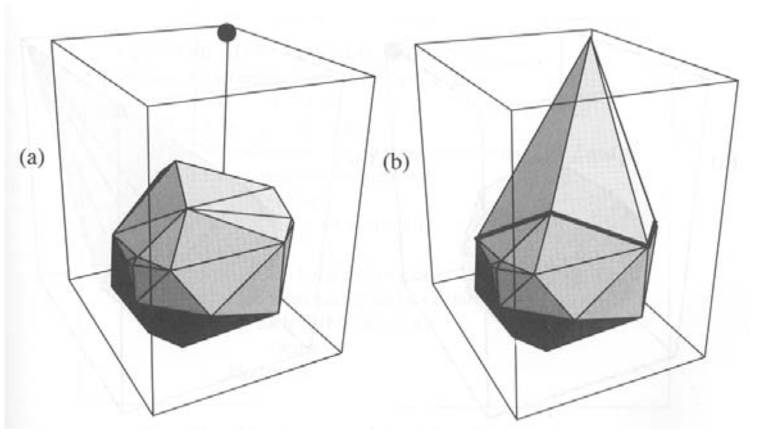

Figure 1: Incremental adding of a point to the existing hull (O'Rourke)

we can define a positive and a negative side of this face F.

Corollary 1 *A point is on the positive side of a triangular face iff the volume of the tetrahedron consisting of the point and the face in counterclockwise orientation is positive.*

This "left-of-triangle" test is similar to "left-of-line" test in 2D[*]. If we ensure a consistent orientation of the faces, a point inside a convex hull will be on the positive side of each face.

Corollary 2 *A point p_i is within the current hull H_{i-1} iff it is on the positive side of each face of the hull.*

Once a point is determined to be inside the hull, it can be entirely disregarded, as it does not contribute to the final convex hull. If however a point is found to be outside the current convex hull, i.e. it is on the negative side of at least one face, it must be incorporated into the hull. In 2D, this is done by adding two line segment that originate in p_i and are tangent to the hull. The corresponding structure in 3-dimensional space is a cone that has its apex in p_i. The triangular faces of the cone therefore consist of p_i and one edge on the hull

[*]See O'Rourke

H_{i-1}. All edges of the hull that participate in the cone form a closed circular shape (see dark line in figure 1), called the horizon. Each vertex strictly inside this closed horizon is not part of the final convex hull and can thus be removed.

To construct this cone that combines p_i and H_{i-1}, it is sufficient to determine the edges of the triangular side faces of the cone. It is obvious that some faces of H_{i-1} are visible from p_i, while others are not. If one thinks of p_i as a light source that shines on the hull H_{i-1}, only some faces will be illuminated — the visible faces. In terms of volume signs, a face is visible from p_i iff p_i is on its negative side. There exists a continuous set of edges (the horizon) separating the visible from the invisible faces. These horizon edges form the base of the cone. All the faces that are visible from p_i can be removed from the hull, as they lie within the boundary of the convex hull. After adding the cone consisting of p_i and the horizon edges and the removal of visible edges, H_{i-1} has been transformed to H_i.

2.1. Data Structures

When constructing convex hulls in 3D, one encounters 3 structural entities, vertices, edges and faces (or facets), which must be represented in the code.

As container structures circular linked lists are used. In circular linked lists, the `next`-pointer of the last element points to the first element, and the `previous`-pointer of the first elements points to the last element. The order of the elements in the lists is irrelevant.

As the lists have no predefined start element, a pointer to one element in each list is maintained (these pointers are called `faces`, `edges` and `vertices`). It is important to note that these pointers may be moved along the elements of a list — their only guaranteed property is that they point to any one element in the list.

The links needed to traverse the lists are stored within the elements themselves.

Vertices: Along with each vertex, its coordinates `v` and an identifier `vnum` are stored. A boolean field `is_processed` is used to state whether a vertex has been tested for addition already. In addition to that, various hooks to associate temporary information are provided (`newface` and `do_delete`).

Edges: Each edge contains pointers to its two adjacent faces (`adjface`) and its two endpoints (`endpt`). Two additional temporary fields are provided as well.

Faces: A face mainly consists out of pointers to its three edges (`edge`) and to its three vertices (`vertex`). This redundance is introduced on purpose, as it greatly simplifies the handling of the faces. The boolean flag `is_visible` is used for temporary purpose only.

2.2. The Algorithm

Although the basic ideas of the algorithm are very straightforward, the implementation proves rather tricky. Not only does one have to be aware of possible precision errors[†], but the implementation is also supposed to work efficiently.

First, an initial data structure using 3 points is created. Then, one point after the other is added to the hull.

2.2.1. Helper Functions

The function `collinear` is given three points and checks whether they are in collinear position. To do so, the points are tested for collinearity in the three two-dimensional subspaces: xy, yz and xz. The points are collinear in 3D if and only if they are collinear in each of the 3 subspaces.

`volume_sign` is given a face and a point and returns the sign of the volume of the tetrahedron described by the face and the point. The function uses double values for its internal computations, as double allow for larger values without

[†]As the algorithm is laid out to work with integer input, the main issue with precision is the calculation of the volumes of the tetrahedrons

overflow, compared to integers. To calculate the volume, the determinate of the following matrix is used:

$$\begin{pmatrix} a_x & a_y & a_z & 1 \\ b_x & b_y & b_z & 1 \\ c_x & c_y & c_z & 1 \\ d_x & d_y & d_z & 1 \end{pmatrix}$$

where a , b, c and d are the for vertices of the tetrahedron. The implementation reduces the number of arithmetic operations by using a translation in which p_i becomes the origin.

The procedure `read_vertices` reads the vertices' coordinates from standard input. The constructor `make_empty_vertex` handles the inclusion of the newly created vertices in the vertex list.

2.2.2. Initial Structure

`create_initial_triangles` finds the first three non-collinear vertices in the list of vertices. These 3 vertices are then marked as processed (`is_processed = true`) and are used to create two triangular faces. Thus, these two faces consists of exactly the same points and the same edges. However, for one of the two faces, the vertices are ordered clockwise, whereas for the other face the vertices are ordered counterclockwise.

The rationale behind this strategy is that the next point to be inserted will be on the positive side of one the two faces, but on the negative side of the other face. Thus, by choosing the "negative" face and discarding the other one, the correct orientation of the vertices can be determined indirectly.

The actual creation and allocation of the edges and faces as well as their insertion into the respective lists is handled by the function `make_face`.

2.2.3. Incremental Step

`construct_hull` is the central routine of the incremental part of the algorithm. For each point that has not yet been processed, it calls `add_one`, followed by a call to `clean_up`.

Within `add_one`, the faces' visibility with respect to the given point is checked. If no face's outer (negative) side is visible from the point, the point must be inside the hull constructed so far. In this case, the function terminates.

If at least one face's outer side is visible, all of the horizon's edges are determined and a cone face is created for the point and each of those edges.

The creation of the cone faces is handled by `make_cone_face`. This function ensures that every edge from the point to the vertices on the horizon is created exactly ones (multiple creation could occur as each of these point-vertex edges participates in two cone faces). A new face is created consisting of the point-vertex edges and one horizon edge, and the newly created face is associated with the participating edges.

The orientation of the points in the newly created face is managed by `make_ccw`, that uses the orientation of already existing faces (one of the faces associated with the horizon edge) to determine the new face's orientation. This technique ensures consistent orientation of the faces throughout the algorithm.

2.2.4. CleanUp

Whenever a new face are added to the hull, some vertices (and thus edges and faces) become invisible from the outside — they are not on the current convex hull anymore, and thus do not lie on the final convex hull. These outdated elements must be removed after each addition of a point. The clean up routines also take care of resetting all temporary data fields.

The dispensable edges are taken care of first, as the determination of their status follows from the status of the adjacent faces. Each edge for which both adjacent faces are visible from the previously added point are removed.

Next, all faces that are visible from the previously added point are removed, as they lie on the inside of the hull.

Last, all vertices are are not part of one of the remaining edges are removed.

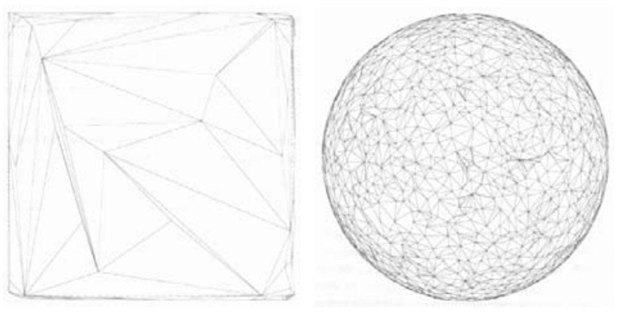

Figure 2: Input point sets with different distribution (O'Rourke)

2.3. Performance

Observe there are at most $\mathcal{O}(n)$ faces and $\mathcal{O}(n)$ edges; this follows from Euler's equation. For each point, the following operation are performed:

- $\mathcal{O}(n)$ faces must be tested for the volume sign
- $\mathcal{O}(n)$ faces and $\mathcal{O}(n)$ edges must be created
- $\mathcal{O}(n)$ faces, $\mathcal{O}(n)$ edges and $\mathcal{O}(n)$ vertices must be processed during the cleanup phase

Thus, the total complexity of each point amounts to $\mathcal{O}(n)$.

The complexity for the entire algorithm comes to $\mathcal{O}(n^2)$, as n points must be processed.

The actual performance does not only depend on its algorithmic complexity, but also on the distribution of the input points. Figure **??**istrex shows two different point sets — one has a spherical distribution with most of the points on or very close to the hull, the other one is of cubic shape and has mostly non-extremal points. Figure **??**istrch demonstrates the runtime results of the convex hull algorithm for the two point sets[‡]. The difference in runtime performance is clearly visible.

[‡]The tests were performed on a Silicon Graphics Indy machine with 133MHz.

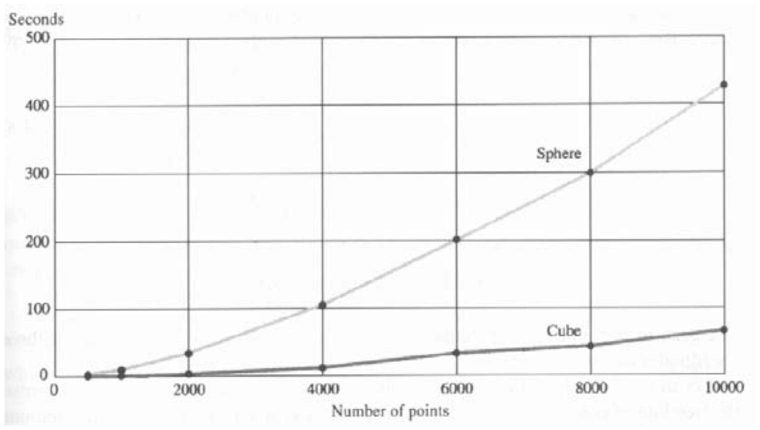

Figure 3: Runtime for different input point sets (O'Rourke)

3. QuickHull

Barber, Dobkin and Huhdanpaa have developed a convex hull algorithm for arbitrary dimension, called QuickHull. This algorithm is an incremental algorithm as well, however points are not chosen at random but according to certain properties. This allows for runtime improvements.

Since QuickHull can be applied to sets of points of any dimensionality, we speak of facets (instead of faces) and ridges (instead of edges).

3.1. The Algorithm

QuickHull is very similar to the incremental algorithm presented above. For each point that is processed, it is determined whether the point is on the outer (or upper) side of a facet. If this is the case, the point is added to the hull by creating a tangent cone.

Points are assumed to be in general position. For QuickHull, a facet is represented by the outward-pointing norm of its hyperplane and the offset from the origin. A point is said to be above a facet iff its distance to the hyperplane is positive (the signed distance is the inner product of the point with the hyperplane's norm, plus the offset). Observe that this definition of above/below according to the signed distance differs from the volume sign used above.

Let H be a convex hull in $\mathbb{R}^d$ and p be a point in $\mathbb{R}^d - H$. According to a simplified version of Grünbaum's Beneath-Beyond Theorem, F is a facet of $conv(p \cup H)$ iff

1. F is a facet of H and p is below F, or
2. F is not a facet of H, and its vertices are p and the vertices of a ridge of H with one incident facet below p and the other incident facet above p.

The rationale of the first condition is obvious. The second describes a face of the cone that is to be created if p is at least above one face (see incremental algorithm above). The ridge with one incident facet below and the other one above p is the equivalent of the edge in between a visible and an invisible face for the incremental algorithm above.

The efficient determination of visible facets for a given point is crucial to the runtime behaviour of any incremental algorithm. As visible facets are neighbours, once one visible facet is found, the others can be detected easily. The main idea behind QuickHull is to maintain a set for each facet in which points are stored that are outside the respective facet. A point is outside a facet iff the signed distance between the facet and the point is positive. Each unprocessed point only belongs to exactly one outside set. It can be shown that, if a point is on the outside of multiple facets, it does not matter to which of the according outside sets the point belongs. These outside sets represent a partitioning of the set of unprocessed points.

The algorithm itself goes as follows:

```
# Initialization
create a simplex of d+1 points
for each facet F
    for each unassigned point p
        if p is above F
            assign p to F's outside set

# Iterative step
```

```
for each facet F with a non-empty outside set
    select the furthest point p in F's outside set

    # Find facets visible from p
    initialize the visibility set V to F
    for all unvisited neighbours N of facets in V
        if p is above N
            add N to V

    # Create the cone
    the boundary of V is the set of horizon ridges H
    for each ridge R in H
        create a new facet from R and p
        link the new facet to its neighbours

    # Repartition the points of V's outside sets
    for each new facet G
        for each unassigned point q in an outside set
         of a facet in V
            if q is above G
                assign q to G's outside set

    # Eliminate ``inside'' facets
    delete the facets in V
```

Whenever possible, the initial simplex consists of points with an minimal or maximal coordinate.

3.2. Performance

As fewer non-extremal points must be processed, QuickHull runs usually faster than randomised incremental algorithms. With r being the number of processed point and f_r the maximum number of facets for r vertices, the average running time is $\mathcal{O}(n \log r)$ for 3 or less dimensions, and $\mathcal{O}(\frac{nf_r}{r})$ for 4 or more dimensions. This bounds are conjectured to always hold when the input precision is limited to $\mathcal{O}(\log n)$ bits.

As QuickHull does not maintain obsolete facets, it uses less space than most incremental algorithms.

In two cases, incremental algorithms can perform much worse than expected:

1. Each point create many new facets — this happens for example when the points of a spiral are added in order. As QuickHull selects the point with the greatest distance for addition first, this cannot happen.

2. A search for a visible facet could visit many old facets — for QuickHull, this means assigning all points to the outside set of one facet for most partitions. It is conjectured that this case does not occur when the input precision is limited.

Generally, it is conjectured that QuickHull shows average runtime behaviour when certain balance conditions hold (the balance conditions are described in the original paper).

Furthermore, QuickHull is conjectured to be strictly output sensitive whenever the input precision is limited to $\mathcal{O}(\log n)$. In that case, the worst case complexity for QuickHull is $\mathcal{O}(n \log v)$, where v denotes the number of output vertices.

4. Summary

Two algorithms for convex hull determination have been presented. The first one is a randomised incremental algorithm that runs in $\mathcal{O}(n^2)$ time. Its implementation has been discussed in detail.

QuickHull is an output sensitive convex hull algorithm that works in arbitrary dimensions. It uses less space than most other incremental algorithm, and usually performs faster for point sets that contain non-extremal points. The runtime complexity of QuickHull is conjectured to be $\mathcal{O}(n \log v)$ (v is the number of output vertices).

References

- Joseph O'Rourke: *Computational Geometry in C*; 2nd ed., 1998, Cambridge University Press

  Chapter 4 of O'Rourke's book describes the incremental convex hull algorithm, among other algorithms.

- C. Bradford Barber, David P. Dobkin, Hannu Huhdanpaa: *The QuickHull Algorithm for Convex Hulls*; ACM Transactions on Mathematical Software Vol. 22, No. 4, Dec. 1996, pp. 469-483

  This paper describes the QuickHull algorithm, proves its correctness and gives various empirically founded conjectures about QuickHull's runtime behaviour.