Obtaining the Convex Hull from a Set of Points using the Gift Wrapping Algorithm

Paul Nathan 10/04/2015

In the present context, the convex hull is used to represent an "extruded" version of the traverse for the purpose of collision detection during route checking of a linear translation. Rather than advancing in small steps (which carries a risk of missing obstacles smaller than the step size), a single check carried out with an "extruded" traverse guarantees no false negatives and only requires a single overall test rather than possibly hundreds. The convex hull is formed from the vertices that make up the moving traverse at its initial and final positions.

The convex hull of a set of points is defined as the smallest convex set that contains all the points. The convex set is one in which a straight line joining any pair of points within the set is also within the boundary of the convex set. There are many algorithms for computing the convex hull, each with its own advantages and disadvantages. The QuickHull algorithm (originally by Barber et al.[1]) has $O(n \log n)$ complexity on average but, although simple in 2D, in 3D involves some rather involved data management for face merging and horizon edge tracking steps. A much simpler algorithm that has $O(nh)$ complexity, with n being the number of vertices and h being the number of points on the convex hull, is the gift wrapping algorithm. This works, in 2D, by finding the set of lines that envelope all points in a convex manner, and, in 3D, by finding the set of surface triangles that envelope all points with a resulting convex polyhedron. This earliest form of this algorithm, limited to 2D and based on computing angles between successive candidate points, was by Jarvis (1973)[2]

The 3D algorithm essentially involves creating edges between points, and testing whether candidate triangles formed by these edges and other points in the set satisfy the criterion for being part of the convex hull. This criterion is simply that all points must be in the negative half-space of the triangle (i.e. all have a negative signed distance from the triangle's plane). Each accepted triangle then provides two new edges from which new candidate triangles may be formed, and so the process continues until the surface is closed (when no new triangles can be formed anyway).

There are multiple websites that describe the gift wrapping algorithm for computer graphics applications[3], but it seems with never quite enough of the specific details for a practical working implementation. The algorithm detailed overleaf provides full step-by-step implementation and is based on this author's own working code, created from scratch.

[1] Barber, B. C.; Dobkin, D. P.; Huhdanpaa, H (1996): "The quickhull algorithm for convex hulls". ACM Transactions on Mathematical Software **22** (4): 469–483

[2] Jarvis, R. A. (1973): "On the identification of the convex hull of a finite set of points in the plane". Information Processing Letters **2**: 18–21

[3] A good example being http://www.cse.unsw.edu.au/~lambert/java/3d/giftwrap.html (last accessed 10/04/2015)

THE 3D GIFT WRAPPING ALGORITHM

1. *Selection of first point on convex hull*
   a. Flatten all points onto the X-Y plane
   b. Find the point with either the maximum or minimum squared distance from the origin (which can be arbitrarily set to (0, 0, 0)). The squared distance is used to avoid the square root operator.

2. *Selection of first edge on convex hull*
   - Begin outer loop through all flattened points
     a. If the current point's index is equal to the index of the first point selected in step 1 then skip this point and continue onto next point
     b. Create an edge vector joining the current point and the first point selected in step 1 and compute its outward-facing unit normal vector
     c. Begin inner loop through all flattened points
        - Compute the signed distance of each point from the edge vector formed in step 2b
        - If the point returns a positive value then this edge vector cannot be on the convex hull, exit this inner loop and try the next point (return to step 2a)
        - If the point returns only negative signed distances, then this point (and its associated line) is on the convex hull. Exit inner and outer loop and return the index of the point that formed the accepted edge vector.

3. *Initialisation of main loop*
   - Initialise a stack of custom data type "edge". The data type contains the vertices of the end points of the edge as well as their associated indices (linked to the original array of points, not the temporarily flattened set)
   - Create a new edge with start point being the point found in step 1 and end point being the point found in step 2. Similarly store the indices of these points
   - Push this edge onto the stack
   - Add this edge to a dynamic array of edges that will be used to check for duplicated edges.

4. *Begin main loop*
   - Pop an edge from the stack
   - Begin outer loop through all points
     a. Check if the index of the current point is equal to either index of the edge vertices of the current edge. If true, then this is a degenerate case, do nothing and move onto next point in the array, else continue to next step
     b. Create a candidate triangle comprising the vertices of the current edge and the current point
     c. Compute the outward-facing unit normal
     d. Begin inner loop through all points
        - Compute the signed distance of each point from the plane of the triangle formed in step 3b
        - If the point returns a positive value, then it cannot be a point on the convex hull, exit the inner loop and move onto the next point in the outer loop
        - If the point returns only negative signed distances, then this point (and thus its associated candidate triangle) is on the convex hull
          - Create two new edges based on the new edges on the candidate triangle (those that share the current point indexed out in the outer loop), however with the very important difference that the new edge vectors must point in the opposite direction to those of the triangle. This ensures that subsequent triangles formed from the new edges

will have anti-clockwise positive winding, essential for obtaining the outward-facing unit normal vector
    - Check that these two new edges are not duplicates of existing edges (stored in the dynamic array of edges initialised in step 3). This is done by checking that the pair of indices of the edge end-points does not match any pair of indices of existing edge end-points. It is necessary to also perform this check with the end-points in reversed order to take into account the possibility of the duplicated edges having opposite direction
    - If any edge is not a duplicate, then push this new edge onto the edge stack and also add it to the dynamic array of edges.
    - Exit the outer loop and add the candidate triangle to a dynamic array of convex hull triangles (the required end product)
- Continue main loop until the edge stack is empty.

<u>Elucidation of Key Steps in the Algorithm</u>

<u>Step 1:</u>

The point set $\{\mathbf{P}\}$ is flattened to $\{\mathbf{P}'\}$ simply by setting all the Z coordinate values to zero.
The points with maximum and minimum squared distance from the origin are simply $\max\left(\mathbf{P}'_i \bullet \mathbf{P}'_i\right)$ and $\min\left(\mathbf{P}'_i \bullet \mathbf{P}'_i\right)$ for all *i*.
At this stage it is convenient to compute the value of epsilon that is scaled to the geometry at hand.

$$\varepsilon = \varepsilon_{\text{machine}} \cdot \left(\sqrt{\max\left(\mathbf{P}'_i \bullet \mathbf{P}'_i\right)} - \sqrt{\min\left(\mathbf{P}'_i \bullet \mathbf{P}'_i\right)}\right)$$

Testing against epsilon rather than zero is used to ensure that points on a plane are counted as "on the negative side" such that they can be valid convex hull points when taking into account numerical truncation errors.

<u>Step 2:</u>

Given two end-point vertices $\mathbf{V}_0$ and $\mathbf{V}_1$, the associated edge vector $\mathbf{E}_{01}$ is given by

$$\mathbf{E}_{01} = \mathbf{V}_1 - \mathbf{V}_0$$

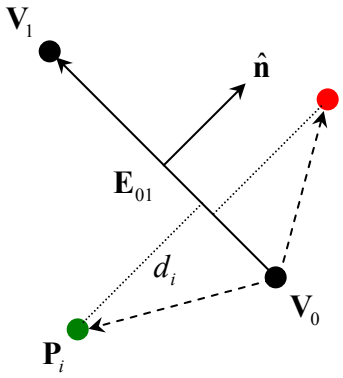
In 2D, the outward-facing unit normal vector for this edge can be found by taking the cross product of the normalised edge vector with the positive unit z-axis, $\hat{\mathbf{z}} = \begin{pmatrix} 0 & 0 & 1 \end{pmatrix}$

$$\hat{\mathbf{n}} = \frac{\mathbf{E}_{01}}{\left\|\mathbf{E}_{01}\right\|} \times \hat{\mathbf{z}}$$

The signed distance *d* of a point **P** from this edge is obtained as follows

$$d = \hat{\mathbf{n}} \bullet \left(\mathbf{P} - \mathbf{V}_0\right)$$

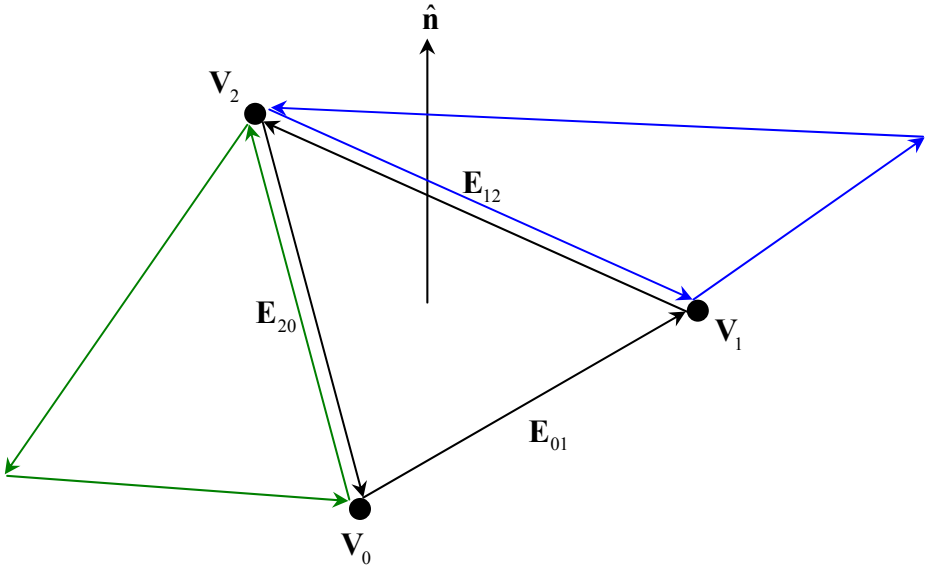
The point cannot be on the convex hull if $d > \varepsilon$.



In the above illustration, the edge cannot be on the convex hull because a point exists on the side of the edge that has a positive signed distance (the red point).

Step 4:

A triangle with anti-clockwise positive winding is illustrated below (black)



The outward-facing unit normal vector is obtained as follows

$$\mathbf{E}_{01} = \mathbf{V}_1 - \mathbf{V}_0$$
$$\mathbf{E}_{12} = \mathbf{V}_2 - \mathbf{V}_1$$
$$\mathbf{E}_{20} = \mathbf{V}_0 - \mathbf{V}_2$$

$$\hat{\mathbf{n}} = \frac{\mathbf{E}_{01} \times \mathbf{E}_{12}}{\left\| \mathbf{E}_{01} \times \mathbf{E}_{12} \right\|} \quad \text{or} \quad \hat{\mathbf{n}} = \frac{\mathbf{E}_{20} \times \mathbf{E}_{01}}{\left\| \mathbf{E}_{20} \times \mathbf{E}_{01} \right\|}$$

The signed distance test is the same as in step 2. The green and blue triangles illustrate the importance of reversing the direction of the newly created edges for future candidate triangles so that they themselves will have the same anti-clockwise positive winding.

Additional notes

In the inner loop of step 4d, three of the points checked will be the candidate triangle's vertices. Although a check on indices could allow these points to be skipped (as they will always pass the test), code profiling has shown that it is actually faster to simply let these point be checked anyway.

One additional (and seemingly remarkable) use of the convex hull is in the computation of the Delaunay triangulation of a set of points on a plane. Supposing the points to be on the X-Y plane, by setting the z coordinate of each point as $z_i = x_i^2 + y_i^2$ and then obtaining the convex hull of this paraboloid, the Delaunay triangulation is simply the projection back onto the X-Y plane of all the triangles that have their outer faces directed away from the viewer when looking into the positive z axis. These triangles can be selected simply by taking the dot product of their normal vectors with the positive z axis vector and keeping only those that return a negative value. This link between the convex hull of a paraboloid and the Delaunay triangulation can be explained[1] by observing that the intersection between the plane of any triangle on the convex hull of the paraboloid and the paraboloid itself is an ellipse which, when projected onto the plane turns out to be the circumcircle of the three points. If this is done for all triangles, it can then be observed that no points lie within any of the circumcircles, which is the criterion of a Delaunay triangulation.

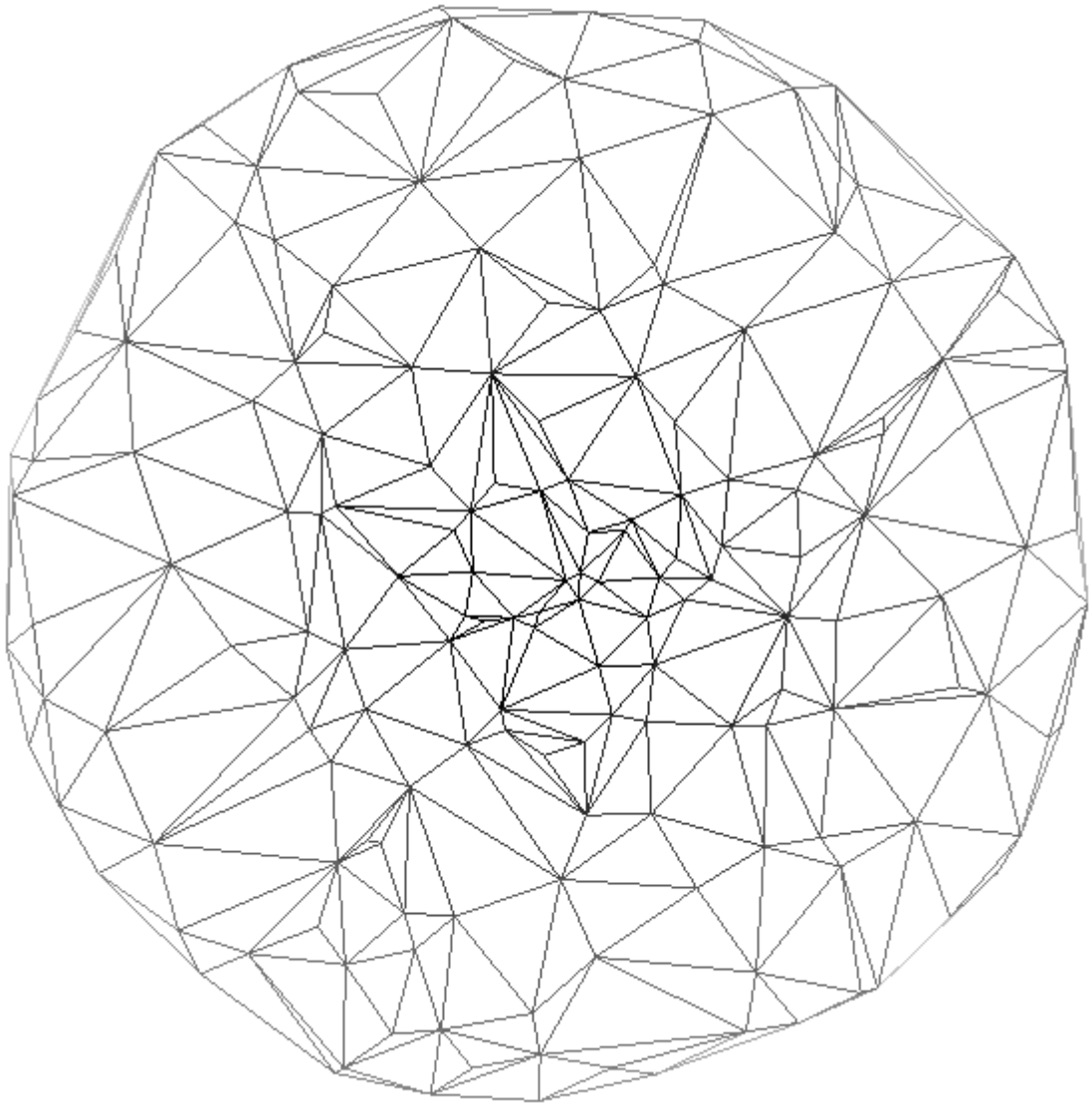


Figure 1: Example Delaunay triangulation computed using the convex hull of a set of 1024 (flattened) spherically distributed random points

[1] See for example http://research.engineering.wustl.edu/~pless/506/l17.html (last accessed 10/04/2015)