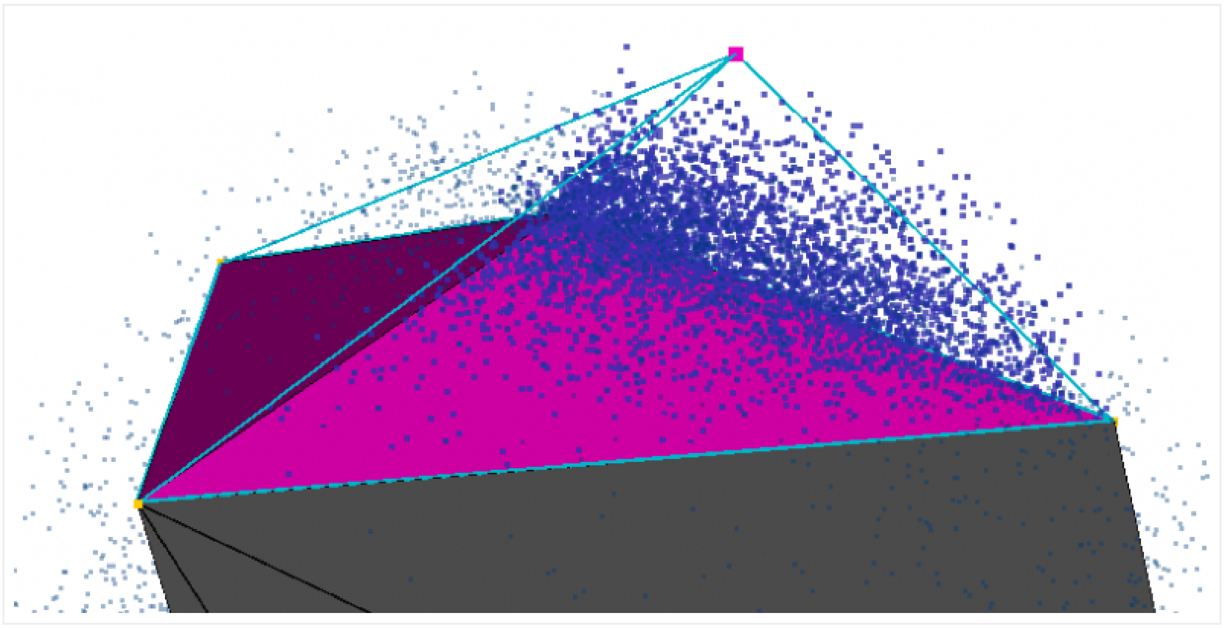
THOMAS DIEWALD

MENU

Convex Hull 3D – Quickhull Algorithm



Quickhull Algorithm

A fast and quite simple algorithm to create a convex hull of a huge number of 3d points is the quickhull algorithm. Here's an short overview of my implementation. For mesh-representation i use the **Half Edge** data structure. It allows fast queries on any kind of neighboring mesh-elements and also fast mesh-modifications.

Source: https://github.com/diwi/QuickHull-3D

Runnable Jar: QuickHull3D_diewald.zip

Online-Demo (Java Applet): http://www.openprocessing.org/sketch/94632 (broken)

1) Initial phase

1. **Create initial simplex** (tetrahedron, 4 vertices). To do this, the 6 Extreme Points [EP], min/max points in X,Y and Z, of the given pointcloud are extracted. From those 6 EP the two most distant build the base-line of the base triangle. The most distant point of EP to the base line is the 3rd point of the base-triangle. To find the pyramids apex, the most distant point to the base-triangle is searched for in the whole point-list. Now having 4 points, the inital pyramid can easily be created.
2. **Assign points to faces**. Each point in the point-list is assigned to the first face the point is in front of ("point can see face"). So each point is assigned to only one face, and each face contains its own point-set. Points that are behind all faces, are therefore automatically

ignored and not used in the further process. I use a separate 2 dimensional dynamic structure for the faces point-sets.

3. **Push the 4 faces on the stack.** Faces without points are ignored.

2) Iteration Phase

1. **If Stack is not empty Pop Face from Stack** . … and check if it has a point-set, otherwise continue next iteration. Although in fact empty faces are not pushed to the stack in the first place.
2. **Get most distant point of the face's point-set.**
3. **Find all faces that can be seen from that point**. Those faces must be adjacent to the current face. I call them light-faces in my implementation, and therefore the point can be seen as a point-light. All found light-faces are labeled as such and also temporarily saved to a heap for later use.
4. **Extract horizon edges of light-faces and extrude to Point.** Clearly there is exactly one closed and convex horizon from the points view that encloses all light-faces. Now each horizon-segement and the current point build a new triangle. So the horizon is somehow projected to the point. The new faces are build and attached to the mesh (and also temporarly saved to a heap) while iteration through the horizon-edges, which automatically detaches all light-faces.
5. **Assign all points off all light-faces to the new created faces.** This is extacly the same procedure as in 1.2. Each point is assigned to the first face it can see. I tried different assigning priorities, but it didn't help much. But again, points behind all faces, are ignored in the further process.
6. **Push new created faces on the stack, and start at (2.1).** Faces without points are ignored.

My implementation mainly works as described in this paper:

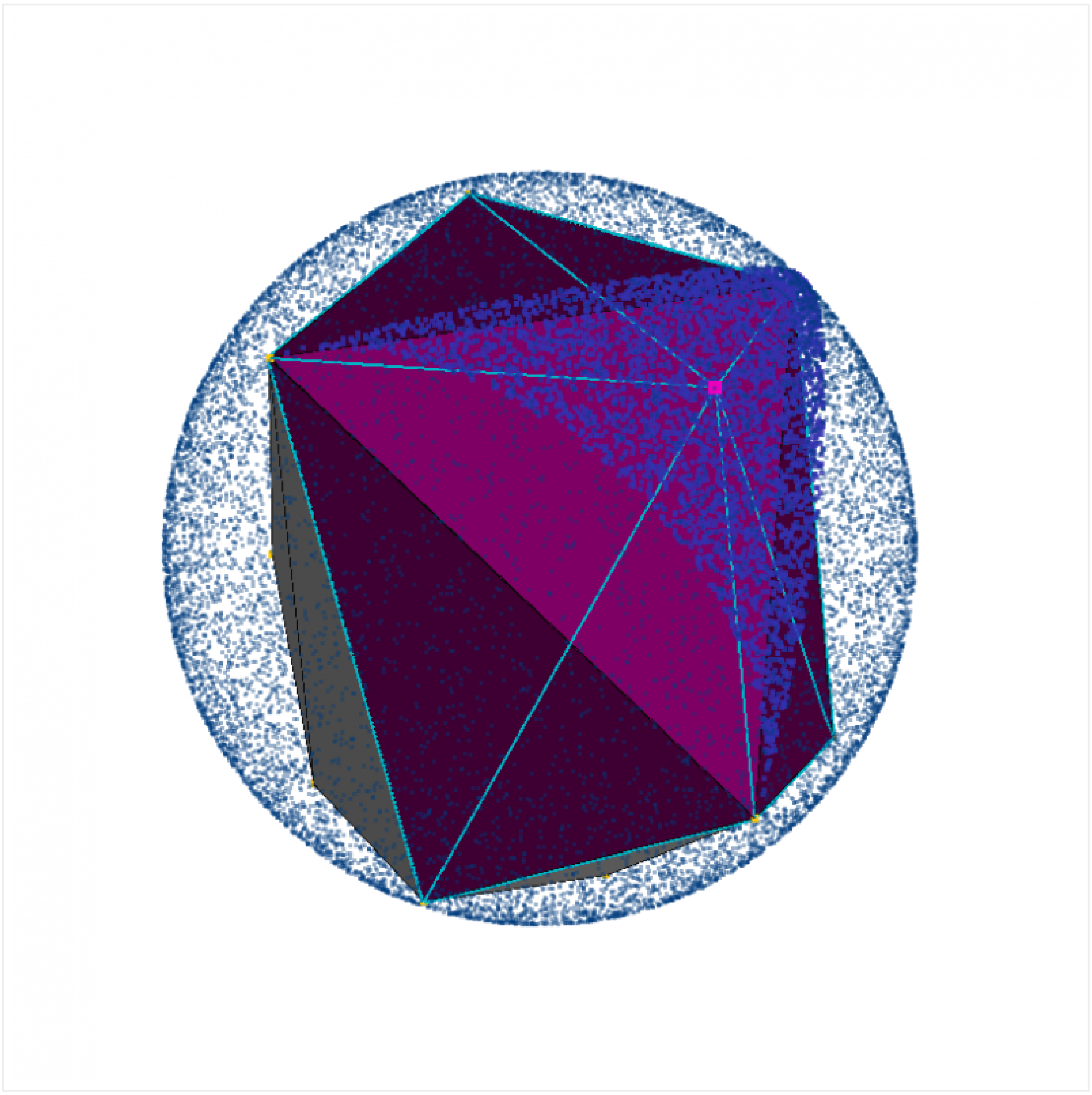
http://www.cise.ufl.edu/~ungor/courses/fall06/papers/QuickHull.pdf

Extending an existing Hull

To extend an existing convex-hull by another point-cloud, the inital phase starts at point 2, which can take a while when the existing hull already has a lot of faces. So this is only efficient when the extending point-cloud has a reasonable (big) number of points, otherwise it may be faster, to merge the point-cloud with the vertices of the hull and build a completely new hull.

One Iteration Explained

The following image shows one iteration. The light blue points are the given point-cloud. The light-pink triangle shows the current face from the stack. The dark bluish-pink points are the point-set of that face. The light-pink point is the most distant point of the face's point-set. The dark-pink triangles are all (neighboring) faces, seen from that point. The cyan lines show the horizon and therefore the new faces that are going to be created during this iteration. All pink faces are removed then. The interactive online-demo (link above) shows the process much better.



Stack Traversal

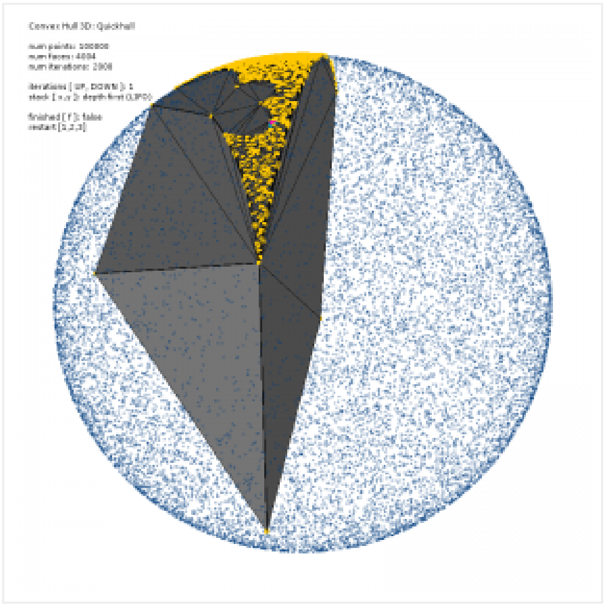
An interesting and important thing to mention is the different behaviour (performance and convergence) when changing the stack-traversal type (depth-first or breadth-first).

Depth-First, LIFO:

in general this seems to be the faster way of generating the convex hull. I guess basically because Stack-Size is always kept at minimum, although it takes more iterations than breath-first to build the hull. In a worst case-scenario, which is, when most poinst or even all, are convex-hull vertices depth-first was about 2-3 times faster than breadth-first while the number of iterations is the same. The drawback i noticed is, that sometimes there were errors during horizon-creation. Especially when the input points are distributed in a cubic volume, some lightfaces were not recogniced as such, and therefore, the horizon cannot be build in a proper way. The reason for this is basically due to the fact that all my calculations are based on floats and not double.

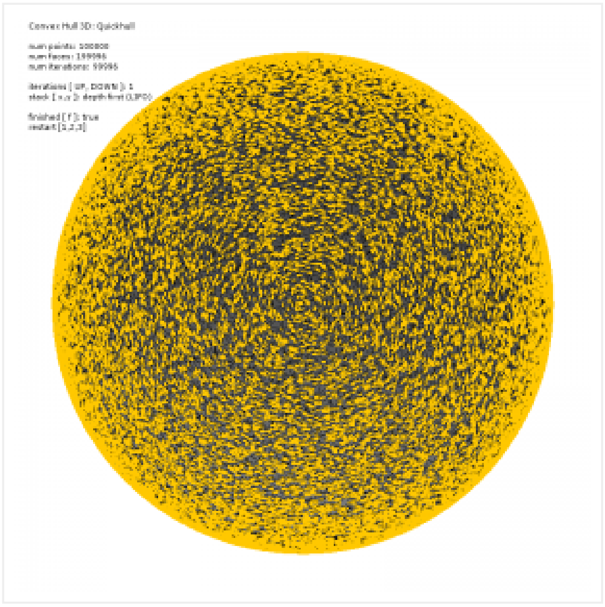
The pictures shows, that each face gets further and further subdivided bevore the other faces are processed. So in generall, details are modelled first.

The following images show some stats for the worst case scenario for LIFO, when all points are on the convex hull.



```
Convex Hull 3D : Quickhull

num points: 100000
num faces: 4004
num iterations: 2000

iterations ( UP, DOWN ): 1
stack ( x,c ): depth first (LIFO)

finished ( f ): false
restart (1,2,3)
```

LIFO, 2000 iterations



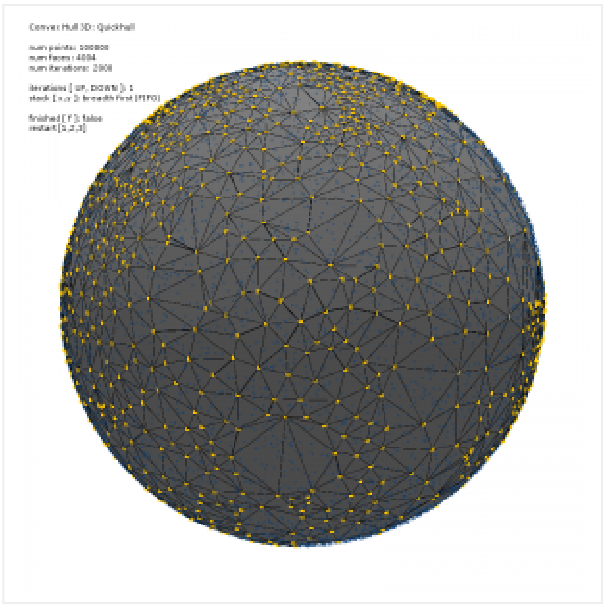
```
Convex Hull 3D : Quickhull

num points: 100000
num faces: 199996
num iterations: 99996

iterations ( UP, DOWN ): 1
stack ( x,c ): depth first (LIFO)

finished ( f ): true
restart (1,2,3)
```

LIFO, finished

Breadth-First, FIFO:

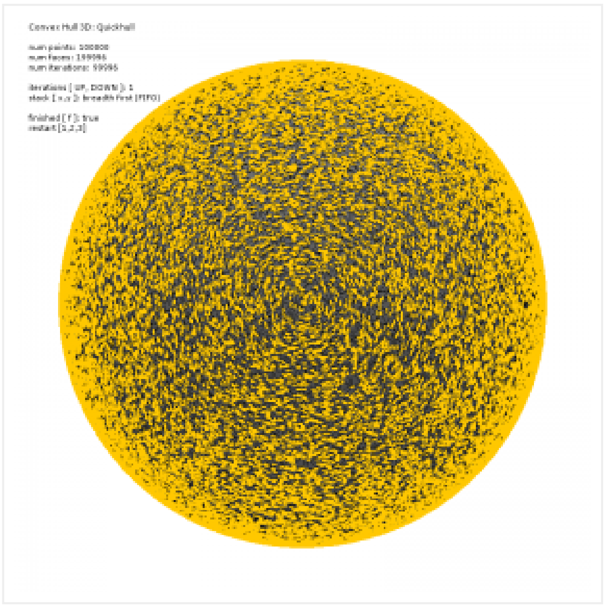
slightly slower in general, but therefore stable in all cases. Breadth-first also needs the lowest number of iterations, but the overall needed stack-size is bigger too. This methods also has the advandage that, even after a few iterations, you get a coarse approximation of the final convex hull, which can be usefull in certain situations.

The following images show some stats for the worst case scenario for FIFO, when all points are on the convex hull.



```
Convex Hull 3D : Quickhull

num points: 100000
num faces: 4004
num iterations: 2000

iterations [ UP, DOWN ]: 1
stack [ x,y ]: breadth first (FIFO)

finished [ F ]: false
restart [1,2,3]
```

FIFO, 2000 iterations



```
Convex Hull 3D: Quickhull

num points: 100000
num faces: 199996
num iterations: 99996

iterations [ UP, DOWN ]: 1
stack [ s,q ]: breadth first (FIFO)

finished [ f ]: true
restart [1,2,3]
```

FIFO, finished

March 29, 2013 In algorithms, processing / JAVA, rendering & modelling

← PREVIOUS POST NEXT POST →

Comments are closed.