Fast Approximate Convex Hull

Abstract

I present a method to compute an approximation of a convex hull. The complexity is O(N), and it's accurate enough for practical applications.

Introduction

What is Convex Hull?

From Wikipedia: the convex hull of a set X of points in the Euclidean space is the smallest convex set that contains X. For instance, when X is a bounded subset of the plane, the convex hull may be visualized as the shape enclosed by a rubber band stretched around X.

This article focuses on convex hulls of 3D triangular meshes and 3D point clouds, on x86 and AMD64 platforms, with single precision floating-point data on input.

Practical Applications

For a class of problems, a convex hull is just as good as the original data set. For example, a convex hull is enough to build an axis-aligned bounding box with arbitrary-directed axis. It's enough to do collision detection against a plane. Optimizing these two use cases were the main reasons I've developed the algorithm.

A convex hull only contains the outer points of the set. When the set contains many inner points, a convex hull is much smaller than the original set, and these algorithms will work much faster processing the convex hull.

Building the Convex Hull

Unfortunately, for large datasets (millions of points), building a precise convex hull is prohibitively expensive. Theoretically, the complexity is $O(N \cdot \log(N))$. Practically, the constant component is large, and memory access patterns are random, i.e. for large input dataset the algorithm will likely be RAM latency bound.

Approximate Convex Hull

The idea of the algorithm is following.

Pick a center point somewhere inside the hull.

Divide all space into a finite set of small solid angles around the center. In this article, we'll call a single solid angle "a bucket".

Group input points into these buckets.

For each bucket, find the input point with the largest distance from the center.

Drop the rest of the points from the set.

The Algorithm

The algorithm reads input data twice.

On the first pass, it calculates the center of the data set. Both center of bounding box, and center of mass are suitable for the center. Center of bounding box is faster to compute, because for center of mass you'll need to read both vertex buffer and index buffer, and calculate area of each triangle. Center of mass of the vertices won't work because many real-life models have very different triangle density in different parts of the model. For example, a typical human head model is very detailed around eyes and very coarse on the back.

On the second pass, it calculates the approximate hull.

First, calculate bucket ID for a point[i]. Compute a radius vector, i.e. the vector between the point and the center. Normalize the radius vector making it unit-length. Multiply it by a scalar constant R, the reasonable value for R is between 20 and 100. Round all three coordinates of the vector to the nearest integer. Use the integer coordinates vector as the bucket ID.

This bucket ID is very fast to compute, especially with SIMD.

The bucket ID serves as key in the unordered map. If there's no value for the ID, the algorithm inserts a new one, saving original input point, and distance from the point to the center. If there's already some value for the ID, the algorithm compares the current distance with the previous distance, and if the current point is farther away from the center, it replaces the value in the map.

After the second pass is complete, that unordered map will contain the points of the approximate convex hull. If you need topology not just points, producing it is not hard, but outside of scope of this article.

Further Optimizations

Because we know all three components of the bucket ID are between -R and +R, it's possible to optimize the map performance by compacting the key from 3-components integer vector to just a single integer. With uint32_t keys, there's space for 10 bits per component, i.e. it's sufficient for R < 511.

If you're implementing it with SIMD, instead of comparing the distance, better to keep and compare the inverse if that, $\frac{1}{distance}$ The inverse value is just as good at finding out which of the 2 points are closer to the center. The reason for the inverse is `_mm_rsqrt_ps` intrinsic (that stands for RSQRTPS instruction) that calculates $\frac{1}{\sqrt{x}}$ in just a couple of CPU cycles. You need to calculate that inverse value anyway, to normalize the radius vector.

For the best performance, don't use std::unordered_map. Instead, pick a functionally equivalent hash map container that delivers better memory locality. The examples of suitable containers are Microsoft's CAtlMap and Google's dense_hash_map.

How much data it produces?

The result size is independent of the input dataset size. The theoretical limit is about $\frac{3}{2} \cdot 4\pi R^2$ vertices.

How Fast?

Here is some benchmarks against qhull.org. The compiler is Visual C++ 14, 64 bit. The environment is my desktop PC, i5-4460, 16GB RAM, Windows 10.

Model	**Triangles**	**FACH**		**qhull.org**	
		milliseconds	vertices	milliseconds	vertices
Teapot	1024	0.137	468	2.44	164
Bunny	259898	33.2	25427	364	6053
Piedad	969522	96.1	31103	582	1592
Another model	1178726	63.3	13468	1100	8846

As you see, the proposed FACH algorithm is much faster, by an order of magnitude. Especially for larger models. However, it generally produces more vertices on output.

For all these benchmarks, R constant for FACH algorithm was 50.0.

Limitations

While extremely fast, the proposed algorithm is not accurate at all.

One problem is quantization. When multiple input points fall into in the same bucket, and all of them have similar distance to the center, it's perfectly possible all of them would be a part of the precise convex hull. The described algorithm however will only use a single point with maximum distance from the center, dropping the rest of the points. A trivial edge case demonstrating this problem is 1000×1×1 box: the result will only contain 2 points instead of 8. As a workaround, I only use the approximate hull algorithm with large enough input datasets. For small input sets, just a unique set of points is almost as fast.

Another problem is the result isn't actually convex. If you'll visualize the result, for most input sets you'll see a star-like polyhedron instead of a convex thing.

Despite the limitations, for some practical applications the data is just as useful as the true convex hull would be.

[i] You might wonder why I've picked this particular method to partition the space into solid angles.

The naïve approach is calculating latitude + longitude, and quantizing the values into a fixed set of integer values. One problem, this division is very uneven. The buckets near the poles will be much smaller than the buckets near the equator, and I wanted the division to be more or less isotropic. Another thing, inverse trigonometry functions are relatively slow to compute. The scalar FPATAN instruction takes hundreds of cycles, and SSE approximate version isn't trivial either.

The most accurate approach, build an icosphere (an icosahedron with progressively subdivided triangles), project points to the icosphere, and use the icosphere's face index for the bucket ID. While this division is very even, I don't know a fast enough method to calculate that face index given a radius vector of a point.