V.2

AN EFFICIENT BOUNDING SPHERE

Jack Ritter
Versatec, Inc.
Santa Clara, California

This gem is a method for finding a near-optimal bounding sphere for any set of N points in 3D space. It is Order (N), and extremely fast. The sphere calculated is about 5% bigger than the ideal minimum-radius sphere.

The algorithm is executed in two passes: the first pass finds two points that are close to maximally spaced. This pair describes the initial guess for the sphere. The second pass compares each point to the current sphere, and enlarges the sphere if the point is outside. The algorithm is as follows:

1. Make one (quick) pass through the N points. Find these six points:
   The point with minimum x, the point with maximum x,
   The point with minimum y, the point with maximum y,
   The point with minimum z, the point with maximum z.

This gives three pairs of points. Each pair has the maximum span for its dimension. Pick the pair with the maximum point-to-point separation (which could be *greater* than the maximum dimensional span). Calculate the initial sphere, using this pair of points as a diameter.

2. Make a second pass through the N points: for each point outside the current sphere, update the current sphere to the larger sphere passing through the point on one side, and the back side of the old sphere on the other side. Each new sphere will (barely) contain the old sphere, plus the new point, and usually some other outsiders as well. The number of updates needed will be a tiny fraction of N. In testing each point against the current sphere, the square of its distance from the current sphere's

center is compared to the square of the current sphere's radius, to avoid doing a *sqrt()* calculation.

The following pseudo code compares a point (*x*, *y*, *z*) with the current sphere [center = (*cenx, ceny, cenz),* and radius = *rad*]. If (*x*, *y*, *z*) is outside the current sphere, *(cenx*, *ceny*, *cenz)* and *rad* are updated to reflect the new sphere. The current square of the radius is maintained in *rad sq*:

```
given x, y, z, cenx, ceny, cenz, rad, and rad_sq

dx ← x – cenx;
dy ← y – ceny;
dz ← z – cenz;
old_to_p_sq ← dx*dx + dy*dy + dz*dz;
do economical r**2 test before calc sqrt
if (old_to_p_sq > rad_sq)
        then
        Point is outside current sphere. update.
        old_to_p ← $\sqrt{old\_to\_p\_sq}$;
        rad ← (rad + old_to_p)/2.0;
        update square of radius for next compare
        rad_sq ← rad*rad;
        old_to_new ← old_to_p–rad;
        cenx ← (rad*cenx + old_to_new*x)/old_to_p;
        ceny ← (rad*ceny + old_to_new*y) / old_to_p;
        cenz ← (rad*cenz + old_to_new*z) / old_to_p;
        end
```

The following two tests were run on a Sun 3/50 workstation (68020 with MC68881 floating point co-processor).

Case 1

A spherical volume centered at the origin with a radius of 128, was randomly populated with 10,000 points. Five of these were forced to be at the edge of the sphere. This means that the optimal sphere should have a radius of 128. Results: center = (3, 5, 4); radius = 133 (4% > ideal); processor time: 1.8 seconds.

CASE 2

A cubic volume with a half-edge length of 128 was randomly populated with 10,000 points. Included were the eight corner points. This means that the ideal radius $= \sqrt{3} * 128 = 222$. Note: this is close to the worst case for this algorithm, because an orthogonally aligned box means that no corners will be found in the initial guessing phase. (A box rotated by any angle around any axis would allow corners to be found initially.) Results: center = (5, 21, 2); radius = 237 (7% > ideal); processor time: 1.8 seconds.

A full C version of this algorithm can be found in the appendix.

See Appendix 2 for C Implementation (723)