Mass Properties of the Union of Millions of Polyhedra

Wm Randolph Franklin
Rensselaer Polytechnic Institute
`wrf@ecse.rpi.edu`
`http://www.ecse.rpi.edu/Homepages/wrf`

July 15, 2003

Abstract

We present UNION3, a fast algorithm for computing the volume, area, and other mass properties, of the union of many polyhedra. UNION3 is well suited for parallel machines. A prototype implementation for identical random cubes has processed 20,000,000 polyhedra on a dual processor Pentium Xeon workstation in about one hour.

UNION3 processes all the polyhedra in one pass instead of repeatedly combining them pair by pair. The first step finds the candidate output vertices. These are the 3-face intersections, edge-face intersections, and input vertices. Next, the candidates are culled by deleting those inside any polyhedron. The volume is the sum of a function of each survivor. There is no statistical sampling. Input degeneracies are processed with Simulation Of Simplicity. Since UNION3 never explicitly determines the output polyhedron, messy non-manifold cases become irrelevant. No complicated topological structures are computed. UNION3's simple flat data structures permit it to fork copies of itself to utilize multi-processor machines. The expected time is linear in the number of input, even when the number of intersections is superlinear.

The principal data structure is a 3-D grid of uniform cells. Each cell records overlaps of itself with any edge, face, or polyhedron. Intersection tests are performed only between objects overlapping the same cell. However, if a cell is completely contained in some polyhedron, ("covered"), then no intersection tests are performed in it, since none of those intersections would be visible. Indeed, altho there may be a cubic number of intersections, all but a linear number occur in these covered cells, and are never computed. Therefore the final time is linear.

Keywords: boolean operation, union, polyhedron, volume, mass property, uniform grid, parallel geometry computation.

Contents

1 Introduction **2**

2 Volume Determination **2**

3 Time Analysis **5**

4 Memory Limits and Parallel Implementation **7**

5 Implementation Tests **8**

6 Summary **9**

1 Introduction

3-D geometric applications sometimes create an object as the union of a large set of polyhedra. For example, in CAD/CAM, the volume swept by a cutting tool may be approximated by one polyhedron for each segment of the tool path. On occasion we don't need the resulting polyhedron itself, but only certain mass properties. This presents several opportunities for optimization.

First, the usual method for computing the volume of the union of many polyhedra proceeds by forming a $\log N$ level computation tree that combines successively larger intermediate polyhedra pair by pair. The execution time is probably $N(\log N)^2$ to $N^2(\log N)^2$ depending on the size of the intermediate polyhedra and the efficiency of the polyhedron intersections. There may be many intermediate vertices even when the output polyhedron has few.

Next, efficiently combining polyhedra is more difficult than combining polygons. Sweeping space with a plane, while keeping order among all the active faces, is more intricate than sweeping a plane with a line that keeps track of active edges.

Third, if we need only the volume, or similar mass property, of the computed polyhedron, then it is unnecessary completely to compute the union polyhedron first. Indeed, we need only its vertices with their neighborhood geometries.

This paper presents an algorithm with several optimizations addressing the above issues. Then it describes an implementation for the restricted input case of identical cubes, which has processed test cases as large as $20,000,000$ cubes.

This algorithm is well suited to parallel machines, and this implementation forks copies of itself for large inputs. This contrasts favorably to topological sweep methods, which are harder to parallelize.

2 Volume Determination

The volume of a cube with vertices (x, y, z) may be expressed as $\mathcal{V} = \sum_i s_i x_i y_i z_i$, where $s_i = \pm 1$. For any particular vertex, $s = +1$ iff an odd number of the three faces adjacent to that vertex are on the high side of the cube.

The volume of any rectilinear polyhedron, whose edges and faces are all aligned with the coordinate axes, is still $\mathcal{V} = \sum_i s_i x_i y_i z_i$, if the definition of s is generalized for every possible local vertex geometry. Our implementation uses this formula.

The above formula easily extends to other mass properties, such as moments of inertia of any order.

Likewise, lower dimensional mass properties, such as face area and edge length may be computed as $\mathcal{A} = \sum sxy$ and $\mathcal{L} = \sum sx$ with the s in each case a function of the local geometry of the vertex. Thruout this paper "volume" includes area and length.

All above extends to general polyhedra, as described in Franklin (1987); Narayanaswami and Franklin

(1991), altho the implementation is considerably harder. Indeed, there are varying classes of formula, depending on how much topology is available.

Suppose that we have only the set of incidences of output vertices, edges lines, and face planes, together with which side is inside. For instance, for a cube, each vertex would induce six such incidences. Then if P is the vertex position, $\hat{E}$ is a unit vector along the edge incident on it, $\hat{F}$ is a unit vector normal to $\hat{E}$ in the plane of the face, and $\hat{B}$ is a unit vector normal to both $\hat{E}$ and $\hat{F}$ pointing into the polyhedron, then the volume is $\mathcal{V} = -\frac{1}{6} \sum P.\hat{E}\, P.\hat{F}\, P.\hat{B}$. Similar formulae obtain for other mass properties.

Mirtich (1996) also describes efficient polyhedron formulae.

There are three classes of output vertices resulting from uniting a set of polyhedra: (1) a vertex of one of the input polyhedra, (2) an intersection of an edge with a face, and (3) an intersection of 3 faces.

An output vertex must not be contained in any polyhedron. Therefore, the process goes as follows. (1) Generate a candidate output vertex of one of the above types. (2) Test to see if it's outside all the polyhedra. (3) If so, then compute its sign based on its neighborhood, and add another term into the running total of the volume.

We define "intersection" to exclude components of the same polyhedron. That is why edge-face and 3-face intersections are different cases, even tho every edge might be considered as the intersection of two adjacent faces.

The naive algorithm would test all triples of faces for intersection, requiring $T = N^3$ time. However we use a uniform grid data structure to reduce that, as follows.

1. Superimpose a 3-D grid with $G \times G \times G$ cells on the universe. A reasonable cell size is half the average polyhedron size.
2. For each cell, maintain a list of items, to be determined later, overlapping it.
3. For each polyhedron, determine which grid cells it completely encloses. Mark those cells as *covered*.
4. For the next several steps, whenever an item would be inserted into a cell, do not insert it if that cell is covered.
5. For each polyhedron, determine which cells it overlaps with. Add the polyhedron to those cells' lists.
6. Ditto for each face and edge.
7. Iterate thru the cells. For each cell:
   (a) Test all triples of faces, which are from three different polyhedra, for intersection. Three faces intersect if the intersection point of their three planes is contained in each face polygon. For each triple that does intersect, say at point $\mathcal{P}$, test if $\mathcal{P}$ is outside all polyhedra. If it is, then look up its sign in a table, based on the directions of its three faces, and add a term to the running total for the volume.

   (b) Test all pairs of edges and faces, from two different polyhedra, for intersection, and process the intersections as before. A face and edge intersect if the intersection point of the face plane and the edge line is inside both the face and edge. The intersection's sign is a complicated function of the directions of the two faces adjacent to the edge, and of the intersection face.

8. Iterate thru the vertices, testing whether each is outside all the polyhedra. For each outside vertex, then determine its s, and add $sxyz$ to the volume running total.

Determining whether point $\mathcal{P}$ is contained in any polyhedron proceeds as follows.

1. Compute which cell, $\mathcal{C}$, contains $\mathcal{P}$.
2. If $\mathcal{C}$ is covered, then $\mathcal{P}$ is contained in some polyhedron.
3. Otherwise, test whether any polyhedron in $\mathcal{C}$'s list contains $\mathcal{P}$.

Our implementation handles only identical cubes, altho the theory extends to general polyhedra. Allowing only cubes removed numerical roundoff problems and simplified the implementation, while still demonstrating the algorithm.

This implementation handles degeneracies via Simulation of Simplicity (SoS), (Edelsbrunner and Mücke, 1988) for volume computation. For instance, whenever a point is tested against a face, the three polyhedra whose faces intersected to create that point are brought along. In cases of a coordinate equality, the polyhedron numbers are compared to break the tie. However, then the computed area and length differ from the result of a regularized set union. Say that two polyhedra share a face. With SoS, either both faces will be counted, or neither will, depending on which polyhedron has the lower number.

The importance of the covering cell concept requires emphasis. The bad case for any intersection algorithm is when the objects are large, so that there are a cubic number of 3-face intersections. However we don't need all the intersections, but only those that are outside all polyhedra, i.e., *visible*. The number of visible intersections grows much more slowly than the total number of intersections. Indeed, under some reasonable assumptions, it grows only linearly. Therefore, the covering cell concept can reduce the intersection time from cubic to linear. It also reduces the query time to test point containment from linear time per point down to constant time per point. Indeed, the average number of polyhedra contained in each cell is constant, with reasonable cell sizes.

A major objection to the uniform grid is that it will fail when presented with real world data, since the real world is not uniform. This point is obvious, but *wrong*. Altho it is counterintuitive, it can be shown that, in these applications, the grid structure tolerates nonuniform input as well as a hierarchical structure such as the quadtree, and the implementation is simpler.

The bad case for both a grid and a quadtree would be many close edges that do not intersect. A topological sweep handles that, but requires a complicated data structure that is difficult to parallelize and takes $O(N \log N)$ time. However, first, $\log N$ is nontrivial for current N. Second, topological sweeps are much harder in 3D. Finally, and most important, the sweep finds all intersection vertices, including all $\Omega(N^3)$ hidden ones.

3 Time Analysis

Let

$$
\begin{array}{rcl}
N & \triangleq & \text{number of input vertices} \\
L & \triangleq & \text{length of each edge, on a scale where the universe is } 1 \times 1 \times 1 \\
G & \triangleq & \text{number of grid cells on a side}
\end{array}
$$

Then, being only slightly informal,

Number of cells that each edge overlaps	=	$LG + 1$
Number of cells that each face overlaps	=	$(LG + 1)^2$
Number of cells that each polyhedron overlaps	=	$(LG + 1)^3$
Average number of edges per cell, $\triangleq N_{epc}$	=	$NG^{-3}(LG + 1)$
Average number of faces per cell, $\triangleq N_{fpc}$	=	$NG^{-3}(LG + 1)^2$
Average number of polyhedra per cell, $\triangleq N_{ppc}$	=	$N\left(L + G^{-1}\right)^3$
Average number of cells covered by a given polyhedron	=	$(\max(LG - 1, 0))^3$
Probability of a specific cell not being being covered by any polyhedron $\triangleq q$	=	$e^{-N \max(L-G^{-1},0)^3}$
Expected number of 3-face intersection tests, w/o using covered cells	=	$G^3 N_{fpc}^3$
Expected number of 3-face intersection tests, using covered cells, $\triangleq N_{3ft}$	=	$qG^3 N_{fpc}^3$
Expected number of 3-face intersections, excluding those in covered cells. Note that many of the N_{3ft} 3-face tests fail. $\triangleq N_{3fx}$	=	qN^3L^6
Expected number of face-edge intersection tests, w/o using covered cells	=	$G^3 N_{fpc} N_{epc}$
Expected number of face-edge intersection tests, using covered cells, N_{fet}	=	$qG^3 N_{fpc} N_{epc}$
Expected number of face-edge intersections, using covered cells, N_{fex}	=	qN^2L^3
Time to test if one point is outside all polyhedra, w/o using covered cells	=	N_{ppc}
Time to test if one point is outside all polyhedra, using covered cells, $\triangleq T_p$	=	qN_{ppc}
Number of input vertices not in a covered cell, $\triangleq N_{vnc}$	=	qN
Number of points that need testing for inclusion in any polyhedron, $\triangleq N_{pt}$	=	$N_{3fx} + N_{fex} + N_{vnc}$
Total time, $\triangleq$ polyhedron and cell setup time $+$ number of intersection tests $+$ point inclusion testing time, T	=	$N + G^3 + N_{3ft} + N_{fet} + T_p N_{pt}$

Therefore,

$$\begin{aligned}T &= \left(q^2L^9+3\,\frac{q^2L^8}{G}+\frac{q^2L^6}{G^3}+3\,\frac{q^2L^7}{G^2}\right)N^4\\&+\left(6\,\frac{qL}{G^5}+15\,\frac{qL^4}{G^2}+q^2L^6+3\,\frac{q^2L^4}{G^2}+3\,\frac{q^2L^5}{G}+\frac{q}{G^6}+\frac{q^2L^3}{G^3}+20\,\frac{qL^3}{G^3}+qL^6+6\,\frac{qL^5}{G}+15\,\frac{qL^2}{G^4}\right)N^3\\&+\left(3\,\frac{q^2L^2}{G}+q^2L^3+\frac{q}{G^3}+qL^3+\frac{q^2}{G^3}+3\,\frac{qL^2}{G}+3\,\frac{qL}{G^2}+3\,\frac{q^2L}{G^2}\right)N^2+N+G^3\end{aligned}$$

One easy value for G is $1/L$. In this case,

$$T=8\,q^2N^4L^9+\left(8\,q^2L^6+64\,qL^6\right)N^3+\left(8\,qL^3+8\,q^2L^3\right)N^2+N+L^{-3}$$

If we don't utilize covered cells, effectively $q=1$, so

$$T=16\,N^2L^3+8\,N^4L^9+72\,N^3L^6+N+L^{-3}$$

A further reasonable assumption is that the sum of all the volumes stays constant as N grows, i.e., $L=N^{-1/3}$. Then

$$T=98\,N=\theta(N)$$

How does all this improve when we use covering cells? There are no covering cells unless the cells are smaller than the polyhedra, so assume $G=2/L$. Then

$$q=e^{-N}$$

Then,

$$T=\theta(L^{-3}+N)=\theta(G^3+N)$$

regardless of whether the sum of all the volumes is constant, A more sophisticated analysis with a better choice of G would reduce this to $T=\theta(N)$. (This will be included in a later version of this paper.)

This analysis has a few limitations.

1. If the sum of all volume gets very large, then the union volume approaches 1. If $GL\gg 1$ then all $(G-2)^3$ interior cells will be covered, while the, noncovered, cells adjacent to the universe's exterior will have ever more polyhedra, edges, and faces. The above analysis doesn't capture this unevenness, which becomes important since the processing time in each cell grows as the cube of that cell's contents.
2. All the constant factors are ignored. A more detailed analysis might confirm the experimental observation that G larger than $2/L$, perhaps large enough that the expected number of items per cell is constant, is often better.
3. However, storage grows almost as fast as G^3. For large N, there may not be sufficient storage for the optimal G.

4 Memory Limits and Parallel Implementation

A major limit with the Pentium family of processors is that each process is limited to only 3GiB[1] of virtual memory, altho the real memory can be at least as large as 12GiB. (Utilizing 12GiB would require four parallel processes.) This limit forces any large problem to be decomposed into 3GiB chunks. More compact data structures, such as described here, require fewer chunks. Our storage budget is roughly as follows:

1. 6 bytes per cube, for the lower left corner using short ints.
2. 4 bytes per grid cell, for a pointer to the vector of its contents, or for various flags if the vector is empty.
3. 4 bytes per each overlap of an active cell with an edge, face, or cube. "Active" means that the cell is not covered by a cube, and, if we are multiprocessing, the cell is being handled by this process.
4. 12 more bytes per active cell for the STL vector internals.
5. An unknown amount for unused storage in the vectors and storage fragmentation. These could be eliminated as follows. Process the polyhedra into the cells twice. During the first time, just count how many objects overlap each cell, but don't store which objects overlap each cell. Then, use those counts to allocate exactly each cell's required storage. Finally, process the polyhedra into the cells a second time, this time storing the overlap information. Alternatively, instead of processing the polyhedra twice, the overlap data could be temporarily stored on disk.

Allowing more general polyhedra would increase this storage budget considerably. However storing global topology such as edge loops and face shells would never be necessary. Also, this algorithm would always be much more compact than any data structure with a linked list of the edges around each vertex etc.

Altho decomposition is necessary to process large datasets, many algorithms for this problem cannot be easily decomposed. The input polyhedra overlap each other. So, they cannot just be partitioned into groups since some polyhedra in one group will overlap polyhedra in other groups. Another method might be to disentangle the overlaps by cutting some polyhedra into several pieces, one for each group. However, this will increase those polyhedra's will surface areas, and invalidate the area computation.

The parallel version of our algorithm goes as follows, using only about 100 lines of new code. The cells are partitioned between the processes. When a subprocess would perform any operation on a cell not in its group, that operation is not done. Each subprocess writes its total volume to a pipe. The parent process reads each subprocess's computed volume from its pipe, and sums them.

Lightweight threads are also a future possibility, because most of the execution time is spent traversing the data structure to accumulate the volume. The only variables that are written are loop control variables and the volume running total. If each thread uses its own running totals, then there will be no writes at all to shared variables. This compares quite favorably to the difficult task of maintaining, in parallel, a complicated topological data structure while sweeping thru space.

[1] 1GB=10^9, 1GiB=2^{30}.

5 Implementation Tests

The HW is a dual 2.4 GHz Xeon with 4GiB of real memory. The SW is SuSE 8.2 linux and the Intel C++ compiler. The program is about 1000 lines of code, excluding debugging lines, comments, and blank lines. The input cubes are generated with a combination of three Tausworth random number generators, which is much better than the widely used class of linear congruential generators. One problem with even the best linear congruential generators is that, if we let the generated numbers be x_i, then the 3-D points (x_i, x_{i+1}, x_{i+2}) fall on a relatively small number of parallel planes.

Here are some sample runs.

# cubes	approx inverse edge length	grid resolution	# processes	CPU time per process (sec)	memory per process
10,000	30	60	1	2.5	15M
30,000	50	100	1	7.3	47M
1,000,000	100	200	1	346	938M
1,000,000	200	200	1	139	559M
1,000,000	200	400	2	195+185	982+982M
3,000,000	200	400	2	496+495	2.3+2.3G
3,000,000	200	200	2	350+350	709+709M
3,000,000	200	200	1	856	1.68G
10,000,000	300	300	2	1846+1846	2.4+2.4G
10,000,000	400	400	2	797+793	2.94+2.94G
10,000,000	1,000	400	2	2576+2580	2.13+2.13G
10,000,000	1,000	500	4	476+445+460+403	1.40+1.40+1.40+1.40G
20,000,000	1,000	500	4	2084+2004+2106+2182	2.6+2.6+2.6+2.6G

Here are some stats from the largest case. Lengths are scaled so that the universe is $1 \times 1 \times 1$. The actual edge length is 32768/(approx inverse edge length), rounded to a multiple of 10 (to make the numbers easier to read when debugging).

Number of cubes	20,000,000
Edge length	0.00916
Number of processes	4
Number of grid cells	$125,000,000$
Number cell – cube overlaps	61,890,769
Number cell – face overlaps	254,823,296
Number cell – edge overlaps	349,733,052
Sum of the cubes' volumes	0.015
Volume of the union	0.009
Sum of all the cubes' surface areas	100.583
Surface area of the union	99.053
Sum of all the cubes' edges	219,727
Edge length of the union	219,708
Number of input vertices	160,000,000
Number of those vertices that were outside all cubes	157,561,445
Number of 3-face intersections	37,170
Number face–edge intersections	7,259,255

The hard limit affecting the implementation is the available real memory. (If the total virtual memory of all the processes greatly exceeds the real memory then the amount of paging can hurt performance by more than an order of magnitude.) This limit also inhibits the use of a finer grid, which would reduce the time in some cases. In this largest case, that limitation meant that there were no covered cells.

One solution to the memory problem is to subdivide the problem, then execute the parts sequentially, not in parallel. This is now in testing.

How do we know that the above numbers are correct? Altho errors are always possible, several indicators give us confidence. First, altho the final volume, whose range is $[0, 1]$, intermediate subtotals range up to $5 \,.\, 10^{16}$. Any error is likely to produce a clearly illegal volume. Second, we constructed nasty test cases involving many cubes exactly overlapping or aligning along faces. (This also tested the SoS code.) The computed volume was always correct.

Third, for random input, we can estimate the volume as follows. For N independent and uniformly distributed cubes, each of volume v, the expected union volume is $V_{est} = 1 - (1 - v)^N \approx 1 - e^{-vN}$. The agreement between the predicted and computed volumes is excellent.

6 Summary

This prototype implementation demonstrates that simple data structures, with no global topology, are an excellent method for processing large geometric datasets in 3D. However, this technique supports other operations, not yet implemented, such as online computation of mass properties, as polyhedra are inserted and deleted. Insertion is easy. Deletion requires identifying the candidate output vertices that are now visible since they were hidden only by the deleted polyhedron. Computing properties of more complicated boolean operations is feasible. Producing the explicit output polyhedron is also possible, tho considerably more complicated. We are now considering how to extend this to collision detection of moving objects, each composed of the union of many simple polyhedra.

References

Edelsbrunner, H. and Mücke, E. P. (1988), Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms, *in* 'Proc. 4th Annu. ACM Sympos. Comput. Geom.', pp. 118–133.

Franklin, W. R. (1987), Polygon properties calculated from the vertex neighborhoods, *in* 'Proc. 3rd Annu. ACM Sympos. Comput. Geom.', pp. 110–118.

Mirtich, B. (1996), 'Fast and accurate computation of polyhedral mass properties', *J. Graphics Tools* **1**(2), 31–50.

Narayanaswami, C. and Franklin, W. R. (1991), 'Determination of mass properties of polygonal CSG objects in parallel', *Internat. J. Comput. Geom. Appl.* **1**(4), 381–403.

July 15, 2003, 12:37