Volumes From Overlaying 3-D Triangulations in Parallel

Wm Randolph Franklin*
Rensselæ r Polytechnic Institute
Troy, NY, 12180 USA

Phone: +1 (518) 276-6077, *Fax:* +1 (518) 276-6261
Internet: wrf@ecse.rpi.edu

Mohan Kankanhalli
Institute for Systems Science
National University of Singapore
Internet: mohan@iss.nus.sg

November 1992

Abstract

Consider a polyhedron that is triangulated into tetrahedra in two different ways. This paper presents an algorithm, and hints for implementation, for finding the volumes of the intersections of all overlapping pairs of tetrahedra. The algorithm should parallelize easily, based on our experience with similar algorithms. One application for this is, when given data in terms of one triangulation, to approximate it in terms of the other triangulation. One part of this algorithm is useful by itself. That is to locate a large number of points in a triangulation, by finding which tetrahedron contains each point.

Keywords: tetrahedron, triangulation, overlay, uniform grid, finite element model, mass property, uniform grid, parallel, point location

Contents

*Until Dec 30, 1992: c/o Prof. Leila De Floriani, Dipartimento di Informatica e Scienze dell'Informazione, Universitá di Genova, Viale Benedetto XV, 3, 16132 GENOVA, ITALY, *Phone:* +39-10-3538033, *fax:* +39-10-3538028

List of Figures

1 Introduction

A given polyhedron may be partitioned, or triangulated, into tetrahedra in more than one way. No one triangulation is best since it may be easier to gather data for one set of tetrahedra, while another set might be more compatible with finite element analysis. However, then we must transfer some approximation of the data from the first triangulation to the second. If the data are smoothly varying one reasonable method might be as follows.

Let the input be two different triangulations, $\mathcal{A}$ and $\mathcal{B}$, of the same 3-D object. Assume that we can calculate a set of the pairs of tetrahedra which intersect each other, and the volumes of the intersections. Assume, e.g., that we wish to approximate the mass of each tetrahedra of $\mathcal{B}$ from the mass of each tetrahedron of $\mathcal{A}$, where the density is not constant, so the mass is not simply the volume.

Let m_i be the mass of the i-th tetrahedron of $\mathcal{A}$.

Let u_i be the volume of the i-th tetrahedron of $\mathcal{A}$.

Let v_j be the volume of the j-th tetrahedron of the $\mathcal{B}$.

Let w_{ij} be the volume of the intersection of the i-th tetrahedron of $\mathcal{A}$ with the j-th tetrahedron of $\mathcal{B}$. Note that $u_i = \sum_j w_{ij}$ and $v_j = \sum_i w_{ij}$.

Then the approximate mass of the j-th tetrahedron of $\mathcal{B}$ is

$$\sum_i \frac{w_{ij}}{u_i} m_i$$

Another application, if the tetrahedra are generalized, is to transfer data between input k-cells and the octree obels when building an octree.

We will see how to do this in several stages: the mathematical foundation, finding the volume of the intersection of two tetrahedra, intersecting two complete triangulations, efficient point location, how to implement on a parallel computer, actual implementations of related algorithms, and handling degenerate input.;

The ideas of this paper were first developed in 2D in the context of cartographic map overlay in Geographic Information Systems[Fra90, FS90, Sun89, Siv90]. For some algebraic topological considerations in 2-D overlaying, see [Saa91]. Topological consistency and topology in 3-D are discussed in [ES92, Pig92]. For the latest on plane-sweep overlaying, see [vR91].

This paper is focussed on a specific problem; it takes the triangulations as given, and does not concern itself with their initial generation or later application, although that is clearly necessary. For an treatment of some 2-D triangulation algorithms, see [Hel90]. For another application of tetrahedra in GIS, see [KV92].

2 Mathematical Foundation

Volume, and other mass properties, of a polyhedron can be determined using only the location and local neighborhood of each vertex. The polyhedron can contain multiple nested disjoint components and holes. It may be non-manifold. The only requirements are that it be finite, that there be a boundary that separates the inside from the outside, and the given data correctly represent it.

The data structure is a set of quadruples of vectors: $(\boldsymbol{P}, \boldsymbol{T}, \boldsymbol{N}, \boldsymbol{B})$, with one tuple for each case of a face, edge, and vertex all adjacent. (E.g., consider the cube in Figure 1, which has 48 tuples.) The components have the following meanings.

- $\boldsymbol{P}$ Cartesian coordinates of the vertex.
- $\boldsymbol{T}$ Unit tangent vector along the adjacent edge of this vertex-edge-face adjacency.
- $\boldsymbol{N}$ Unit vector perpendicular to $\boldsymbol{T}$, in the plane of the face.
- $\boldsymbol{B}$ Unit vector perpendicular to both $\boldsymbol{T}$ and $\boldsymbol{N}$, pointing into the interior of the polyhedron. $\boldsymbol{B}$ adds only one extra bit of information, but this is a convenient form for it.

Now, the mass properties, $\boldsymbol{L}$, $\boldsymbol{A}$, and $\boldsymbol{V}$, can be calculated as follows. $\boldsymbol{L}$ is the total edge length, counted once per each face adjacent to each edge, $\boldsymbol{A}$ is

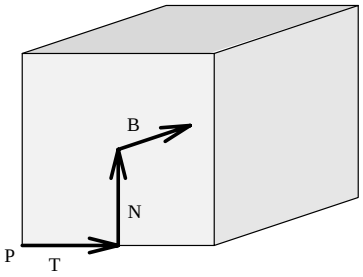

Figure 1: The Local Data Topological Data Structure for a Polyhedron

the total face area, and V is the total volume.

$$L = \sum P \cdot T$$

$$A = \sum P \cdot T \; P \cdot N$$

$$V = \sum P \cdot T \; P \cdot N \; P \cdot B \tag{1}$$

The difficult step is finding the volumes of the intersection polyhedra. Note that we do not need the intersection polyhedra themselves, but only their volumes, which is simpler. We assume that our input data structure includes all the vertices, edges, faces, tetrahedra, and adjacency information.

3 The Volume of the Intersection of Two Tetrahedra

To apply equation 1 to finding the volume of particular intersection polyhedron, we need its vertices, and their local neighborhoods. Such an *output vertex* is one of two types:

1. an input vertex, from one of the intersecting tetrahedra, or
2. the intersection of an edge of a tetrahedron of $\mathcal{A}$ with a face from $\mathcal{B}$ (or vice-versa).

If v is a type-1 vertex, then its adjacent edges and faces are the same as in the original tetrahedron. If v is type-2, the situation is more complicated, as shown in Figure 2. E and F are the input edge and face from $\mathcal{A}$ and $\mathcal{B}$, respectively. $G1$ and $G2$ are the two faces of the $\mathcal{A}$ tetrahedron that are adjacent to E. Now, what is the neighborhood of the new vertex, V?

V has three adjacent edge rays and face planes. Call them $E1$, $E2$, $E3$, $F1$, $F2$, and $F3$, respectively. $E1$ is just the input edge E, properly oriented. $E2$

and $E3$ are the intersection lines $F \times G1$ and $F \times G2$, respectively. $F1$ and $F2$ are $G1$ and $G2$, resp. Finally, $F3 = F$. This gives us the tangents and normals for the neighborhood; all we need now are the binormals.

Since we are intersecting the two polyhedra, the interior of the output polyhedron is interior to both input polyhedra. Therefore the binormals of the output faces are exactly the binormals of the corresponding input faces. Now we have all the data to apply equation 1 back on page 4.

Therefore to find the volume of the intersection of two tetrahedra, we proceed thus:

1. Initialize the subtotal, S, for the volume calculated so far of the output polyhedron, to zero.
2. Repeat for each vertex of the first tetrahedron:
   (a) Test whether it is contained in the other tetrahedron, e.g., by testing whether it is on the inside of all four faces.
   (b) If the vertex is inside, then do this:
      i. Note that there are six vertex–edge–face adjacencies here. Calculate the (P, T, N, B) for each, from the description of the tetrahedron, if they are not already known.
      ii. Calculate a term of equation 1 and add the result to S.
3. Repeat for each pair of an edge from the first tetrahedron and a face from the second.
   (a) Test whether they intersect. For example, this holds if both of the following conditions are true.
      i. The endpoints of the edge are on opposite sides of the face plane.
      ii. The intersection of the infinite line containing the edge with the face plane is a point that is inside the face, considered as a 2-D point-containment problem.
   (b) If they intersect, then calculate (P, T, N, B) for each of the six vertex–edge–face adjacencies here and sum into S.
4. Repeat from step 2 for with the first and second tetrahedra swapped.
5. If $S \neq 0$, then record it.

4 Using Triangulations, not Single Tetrahedra

The above section described how to process two single tetrahedra, but we have two whole triangulations of tetrahedra, $\mathcal{A}$ and $\mathcal{B}$. Although we could apply the above to all pairs of tetrahedra with one from each triangulation, that would be silly since it would take quadratic time.

Using some culling operation to select pairs of tetrahedra likely to intersect would be better, but still quite suboptimal. It ignores optimizations resulting from the sharing of vertices, edges, and faces by adjacent tetrahedra. A much better solution is as follows.

1. Initialize a hash table to store the partial volumes of all the output polyhedra, as they are calculated. The key is to be (a, b) for the volume of the intersection of tetrahedron a with tetrahedron b. We choose a hash table since we don't know in advance which intersection polyhedra will be nonzero, and our abstract operations are to test the existence of a key, read the associated record (partial volume) if the key exists, and write a replacement record for that key.
2. Repeat for each vertex, v, of $\mathcal{A}$.
   (a) Determine which tetrahedron, b, of $\mathcal{B}$ contains v. Store this information with v.
   (b) Find all the adjacent tetrahedra, a_i, in $\mathcal{A}$, to v.
   (c) Since v is a vertex of a_i that is inside b, calculate the resulting partial volume for each (a_i, b) and add into the hash table.
3. Repeat for each edge e of $\mathcal{A}$.
   (a) Get the tetrahedra of $\mathcal{B}$ containing each end of e.
   (b) If they are the same, then go on to the next edge. Otherwise:
   (c) Get the tetrahedra a, of $\mathcal{A}$ adjacent to e.
   (d) Trace through $\mathcal{B}$ to determine which faces of $\mathcal{B}$, e intersects.
   (e) Repeat for each face such face, f :
      i. Find the intersection of e and f.
      ii. Apply the equation.
      iii. Add the resulting partial volumes into the hash table.
4. Repeat the above steps 2 through 3 with $\mathcal{A}$ and $\mathcal{B}$ swapped.

The resulting volumes in the hash table should all be nonnegative. A negative quantity indicates a error, either in the implementation or in the input data.

5 Point Location

Given a vertex v and a triangulation of tetrahedra, $\mathcal{A}$, how do we determine which tetrahedron contains v? We recommend an extension of the uniform grid[FKN89, FNK+89]. For an analysis of the uniform grid on a transputer, see [HH90]. For a comparison of the grid to other methods, and related issues, see [Pul90, HHW92].

Build the data structure as follows.

1. Choose a grid-cell length, λ, proportional to the average edge length. The optimal value is a subject of future research; however in 2-D, being off the optimal by a factor of three either way has never increased the time more than 50%.

2. Construct a 2-D grid on the XY plane.

3. Project the vertices onto the grid to determine which cells they fall in. Use this to determine which PROJECT DATA which cells each edge and face intersect. The accurate method uses Bresenham's algorithm.

   Another method is to find the smallest enclosing box around the edge or face and then write the edge or face into all the cells that the box intersects. This will put the edge or face into too many cells, which will make the later point location slower. However this step will be faster, so the question is which part of the algorithm dominates the total time. Note that with this method, the result is still correct, since the grid is used later merely to cull objects.

4. Repeat for each cell, c: test each face intersecting c to see whether the projected face completely covers c. Such *blocking faces* partition the objects in the cell into those above and those below the blocking face. The resulting data structure for each cell will be an ordered list of blocking faces, with an unordered set of edges and other faces between each pair of adjacent blocking faces.

Since the number of objects in any cell will vary widely, a good data structure is an *expandable array*:

1. Initially, each cell, c, is a null pointer.

2. When we find the first object in c, allocate a block sufficient to hold a counter, and, say, five objects, perhaps with the C language `malloc` routine.

3. If more space is needed, then allocate a larger block and copy the old data, say with `realloc`.

The advantages of this method are as follows.

1. Variable numbers of objects per call can be accommodated without artificial limits.

2. Less space is wasted for pointers than with a linked list.

3. The objects in a cell can be accessed randomly with array indexing. If the cell is an array of pointers to integers, i.e., `int ***grid`, then the k-th object of `grid[i][j]` is simply `grid[i][j][k]`.

4. Again unlike with linked lists, the information for one cell is stored contiguously, which makes virtual memory managers perform better. (However, since real memory is becoming so inexpensive, about $40 per megabyte as this is written, this factor will become less important in the future.)

The creation cost of expandable arrays depends on how much the array is grown each time it overflows. Suppose, for example, we double the array each time, and add one million elements to it, one-by-one. How expensive is this? The average element will be copied only once, there will be 20 reallocations, and a total of 2 000 000 words of memory will be allocated and 1 000 000 words freed. This is quite good performance.

With this data structure, locating a point, v, goes as follows.

1. Project v onto the XY plane, and determine which cell, c, contains it.

2. If c contains blocking faces, then locate v between an adjacent pair of them. Recommended procedures for this include a binary search or an interpolation search[PIA78]. The latter uses an expected $(\log \log N)$ queries if the statistical distribution of the elements is known. $(\log \log N)$ is effectively a small constant.

   Only the faces in the cell between these two blocking faces, and the upper blocking face, are relevant to what follows. See Figure 3.

3. Run a vertical semi-infinite ray up from v and intersect it with all the relevant faces.

4. Find the lowest such intersection point. The tetrahedron on the bottom side of this face contains v.

5. If there is no intersection, then v is not in any tetrahedron.

How does this perform? The only serious uncertainty has to do with the number of faces between adjacent blocking faces in each cell. For efficient performance, the imput data must not be distributed "too" unevenly, which means no more than, say, a factor of ten variation in density throughout the space. The exact characterization of acceptable data is a research topic. However, in 2-D, we have never seen bad data.

If the data are reasonably random, and if the cell size is proportional to the average face size, then the probability that a particular face blocks a cell, given that it is in the cell, is constant. Therefore the expected number of faces between each pair of adjacent faces is constant, independent of the total number of faces in the cell.

Therefore, the expected time to locate a point is effectively constant.

6 Parallelizability

The algorithm presented here is clearly parallelizable for a data-parallel machine. Related algorithms on a uniform grid, such as finding all intersections among a large set of small edges, an object-space hidden-surface removal algorithm, determination of mass properties of polygonal CSG objects, and finding a boolean combination of polygons have been parallelized successfully[Kan90, Nar91, NF92a, NF92b, FK90, FNK+89]. Consider the point-location algorithm for example.

When we insert an object into the uniform grid to build the point-location data structure, the only clash might occur if two objects are inserted into the same grid cell. If the grid is $G \times G$, so that there are G^2 cells, and there are P processors, then the probability that, in parallel write, two processors somewhere try to write to the same cell is

$$1 - \left(1 - \frac{1}{G^2}\right)^P \approx \frac{P}{G^2}$$

if $P << G^2$. This is simply the coincident birthday problem. There are different solutions for this problem, depending on the hardware.

- On a machine with hardware semaphores, such as a Sequent, we lock an cell before appending to it.
- On a CM-2 Connection Machine, if we attempt two simultaneous writes to the same word, then one succeeds and one loses. Therefore we must read back a datum after writing, and rewrite it if the write did not succeed. The average number of necessary repetitions can be calculated, and is tolerable.

Performing the actual point location in parallel is even simpler since we only read the data structure, but don't write it.

7 Observed 2-D Performance for Map Overlay Areas

Some indication of the possible performance might be gained from seeing the efficient 2-D performance of the map overlay area program. The largest input data. had these statistics.

Database	Vertices	Edges	Faces
US counties	55068	46116	2985
Hydrography	76215	69835	2075

When executed on a Sun IPC (25 MHz, 10 MIPS, ≈1990), we observed the following performance.

Operation	CPU Time
Read map	99.32
Scale vertices	1.03
Extract edges from chains	2.38
Calculate input polygon areas	3.65
Make grid	1.28
Add map to grid	8.60
Intersect edges	6.50
Locate map 0 points in map 1	5.83
Locate map 1 points in map 0	8.17
Accumulate output areas	14.35
Print areas	17.23
TOTAL TIME	168.35

After building the data structure, locating a point took 100μseconds on the average. Simply reading the input data from an ASCII file took more time than everything else combined.

8 Observed 2-D Performance for Parallel Object Mass Property Determination

We might obtain some idea of this algorithm's performance on a parallel machine by considering a the performance of a related problem. This is to determine the 2-D mass properties edge length and area of the union of many rectangles. Kankanhalli implemented this on a $\frac{1}{2}$ CM-2 Connection Machine, with 32786 processors. The highlights of the implementation are as follows.

1. Each processor serves first as an edge processor, then as a cell processor.
2. We distribute the edges to the processors.
3. Each (edge) processor finds the cells its edges pass thru, and sends messages to the cell processors.
4. Because of collisions, it reads back to check success, and retransmits if necessary. We can analyze expected number of retries. Experimentally, the maximum was 1–11, and the average number 1–4.
5. Each (cell) processor then intersects its edges to find some possible output vertices.
6. We do a point inclusion to select which are the actual output vertices.
7. Finally, we apply the formula.

The test data were part or all of a VLSI chip containing isothetic edges in rectangles. Notable observations included these.

1. The time to distribute the two million edges to processors is 216 secs.
2. If data size $<$ P, then the time is rather constant, else it grows linearly.
3. Some actual times for various sizes of input are as follows.

# Edges	Grid size	# Procs	# Virt. procs	Time CM-2	Time 4/280
1000	90	8k	8k	1.42	
10000	90	8k	8k	1.41	3.3
100000	512	32k	32k	1.59	34
200000	512	32k	64k	3.42	73
400000	512	32k	128k	7.48	213
1819064	512	32k	512k	36.21	1066

The Sun 4/280 execution times are given for comparison since that is a much less expensive machine.

This suggests that overlaying tetrahedra in 3-D should also be quite parallelizable.

9 Degenerate Data

Degeneracies, such as a vertex falling exactly on a face of another tetrahedron, can be handled by Simulation of Simplicity[EM90b, EM90a]. This requires that calculations be exact. One solution is to scale the data so that edge and face equations are exact, and that tests against them can be performed without roundoff. Assume that the input coordinates are scaled to the range $[-M, M]$. Then in a face equation, $Ax + By + Cz + D = 0$, we have $-2M \leq A, B, C \leq 2M$ and $-6M^2 \leq D \leq 6M^2$. Testing a point against this can generate temporary numbers up to $12M^2$. With single-precision integers on a 32-bit machine, $M \approx 14000$. This may appear imprecise but might well be adequate for volume calculations.

For greater precision, we can use double-precision floats, considered as integers, to get seven digits of precision. However, many, though not all, workstations calculate with floating point much more slowly than with integers.

10 Summary

Transferring data between two triangulations of the same polyhedron is possible by finding the intersection volumes of the tetrahedra. We do not need to find the intersection polyhedra themselves. This algorithm is parallelizable, and, based on past experience, should parallelize well.

11 Acknowledgements

Parts of this work were supported by NSF grant CCR-9102553, by the Directorate for Computer and Information Science and Engineering, NSF Grant CDA-8805910, and by the Gruppo Nazionale Informatica Matematica of the Italian National Research Council.

References

[BCC92] P. Bresnahan, E. Corwin, and D. Cowen, editors. *Proceedings of the Fifth International Symposium on Spatial Data Handling*. International Geographical Union, Commission on GIS, 3-7 August 1992. ISBN 0-9633532-2-5.

[BK90] Kurt Brassel and H. Kishimoto, editors. *4th International Symposium on Spatial Data Handling*, Zürich, 23-27 July 1990.

[EM90a] H. Edelsbrunner and E. P. Mücke. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Trans. Graph.*, 9:66–104, 1990.

[EM90b] Herbert Edelsbrunner and Ernst Peter Mücke. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM T. Graphics*, 9(1):66–104, January 1990.

[ES92] Max Egenhofer and J. Sharma. Topological consistency. In Bresnahan et al. [BCC92], pages 335–343. ISBN 0-9633532-2-5.

[FK90] Wm Randolph Franklin and Mohan Kankanhalli. Parallel object-space hidden surface removal. In *Proceedings of SIGGRAPH'90 (Dallas, Texas) in Computer Graphics*, volume 24, August 1990.

[FKN89] Wm Randolph Franklin, Mohan Kankanhalli, and Chandrasekhar Narayanaswami. Geometric computing and the uniform grid data technique. *Computer Aided Design*, 21(7):410–420, 1989.

[FNK+89] Wm Randolph Franklin, Chandrasekhar Narayanaswami, Mohan Kankanhalli, David Sun, Meng-Chu Zhou, and Peter YF Wu. Uniform grids: A technique for intersection detection on serial and parallel machines. In *Proceedings of Auto Carto 9: Ninth International Symposium on Computer-Assisted Cartography*, pages 100–109, Baltimore, Maryland, 2-7 April 1989.

[Fra90] Wm Randolph Franklin. Calculating map overlay polygon' areas without explicitly calculating the polygons — implementation. In *4th International Symposium on Spatial Data Handling*, pages 151–160, Zürich, 23-27 July 1990.

[FS90] Wm Randolph Franklin and Venkatesh Sivaswami. OVERPROP — calculating areas of map overlay polygons without calculating the overlay. In *Second National Conference on Geographic Information Systems*, pages 1646–1654, Ottawa, 5-8 March 1990.

[Hel90] Martin Heller. Triangulation algorithms for adaptive terrain modeling. In Brassel and Kishimoto [BK90], pages 163–174.

[HH90] Sara Hopkins and Richard G. Healey. A parallel implementation of Franklin's uniform grid technique for line intersection detection on a large transputer array. In Brassel and Kishimoto [BK90], pages 95–104.

[HHW92] Sara Hopkins, R.G. Healy, and T.C. Waugh. Algorithm scalability for line intersection detection in parallel polygon overlay. In Bresnahan et al. [BCC92], pages 210–218. ISBN 0-9633532-2-5.

[Kan90] Mohan Kankanhalli. *Techniques for Parallel Geometric Computations*. PhD thesis, Electrical, Computer, and Systems Engineering Dept., Rensselaer Polytechnic Institute, October 1990.

[KV92] Menno-Jab Kraak and E. Verbree. Tetrahedrons and animated maps in 2d and 3d space. In Bresnahan et al. [BCC92], pages 63–71. ISBN 0-9633532-2-5.

[Nar91] Chandrasekhar Narayanaswami. *Parallel Processing for Geometric Applications*. PhD thesis, Electrical, Computer, and Systems Engineering Dept., Rensselaer Polytechnic Institute, 1991. UMI no. 92-02201.

[NF92a] C. Narayanaswami and W. R. Franklin. Boolean Combinations of Polygons in Parallel. In *Proceedings of the 1992 International Conference on Parallel Processing*, volume tbd, page tbd, August 1992.

[NF92b] Chandrasekhar Narayanaswami and Wm Randolph Franklin. Determination of mass properties of polygonal CSG objects in parallel. *International Journal on Computational Geometry and Applications*, 1(4), 1992.

[PIA78] Y. Perl, A. Itai, and H. Avni. Interpolation search – a log log n search. *Comm. ACM*, 21(7):550–553, July 1978.

[Pig92] Simon Pigot. A topological model for a 3-d spatial information system. In Bresnahan et al. [BCC92], pages 344–360. ISBN 0-9633532-2-5.

[Pul90] David Pullar. Comparative study of algorithms for reporting geometrical intersections. In Brassel and Kishimoto [BK90], pages 66–76.

[Saa91] Alan Saalfeld. An application of algebraic topology to an overlay problem of analytical cartography. *Cartography and Geographic Information Systems (formerly The American Cartographer)*, 18(1):23–36, 1991.

[Siv90] Venkateshkumar Sivaswami. Point inclusion testing in polygons and point location in planar graphs using the uniform grid technique. Master's thesis, Rensselaer Polytechnic Institute, Electrical, Computer, and Systems Engineering Dept., May 1990.

[Sun89] David Sun. Implementation of a fast map overlay system in c. Master's thesis, Rensselaer Polytechnic Institute, Electrical, Computer, and Systems Engineering Dept., May 1989.

[vR91] Jan W. van Roessel. A new approach to plane-sweep overlay: Topological structuring and line-segment classification. *Cartography and Geographic Information Systems (formerly The American Cartographer)*, 18(1):49–67, 1991.

`~/p/tetrahed/tet.tex`

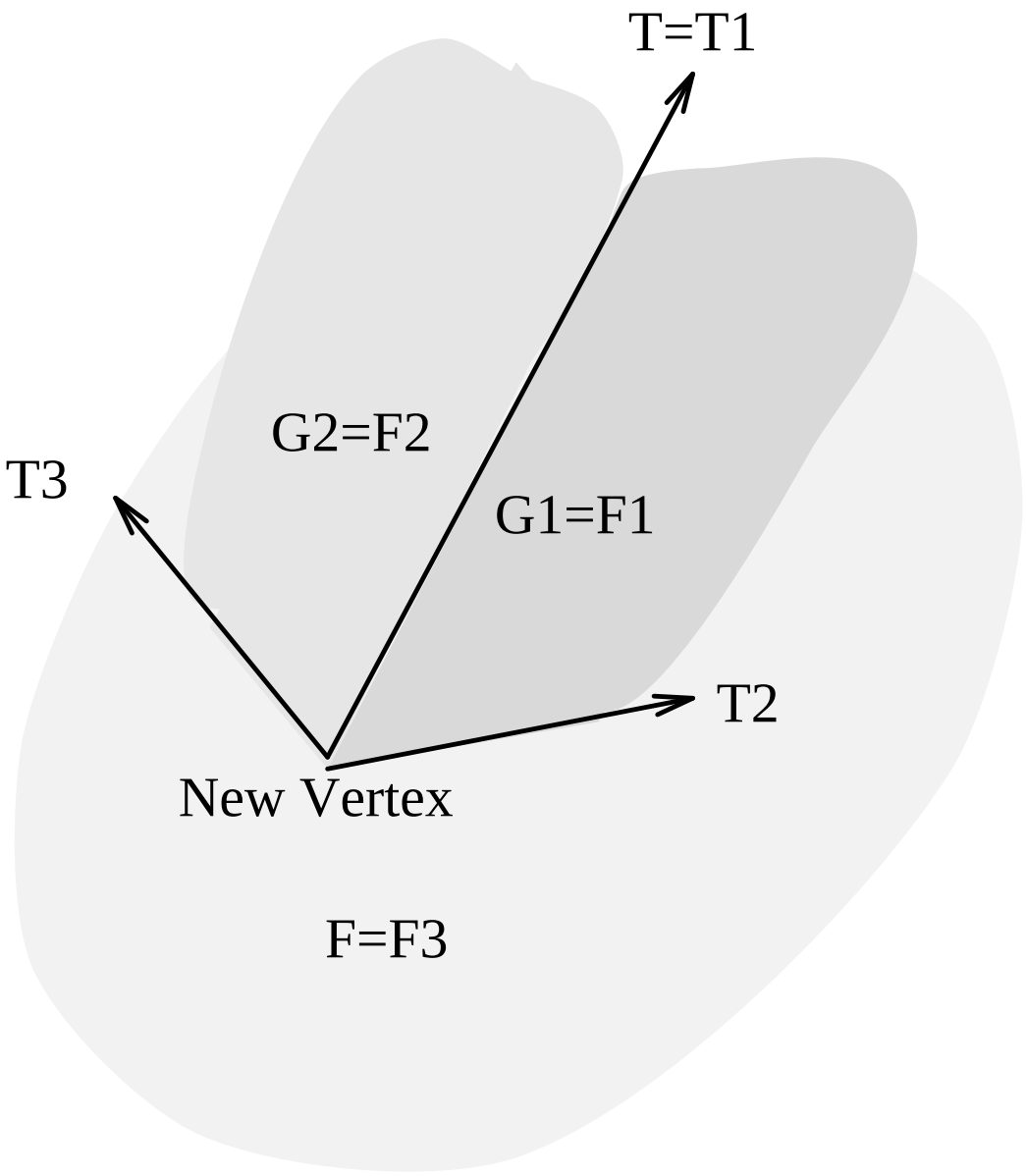

Figure 2: New Type-2 Vertex

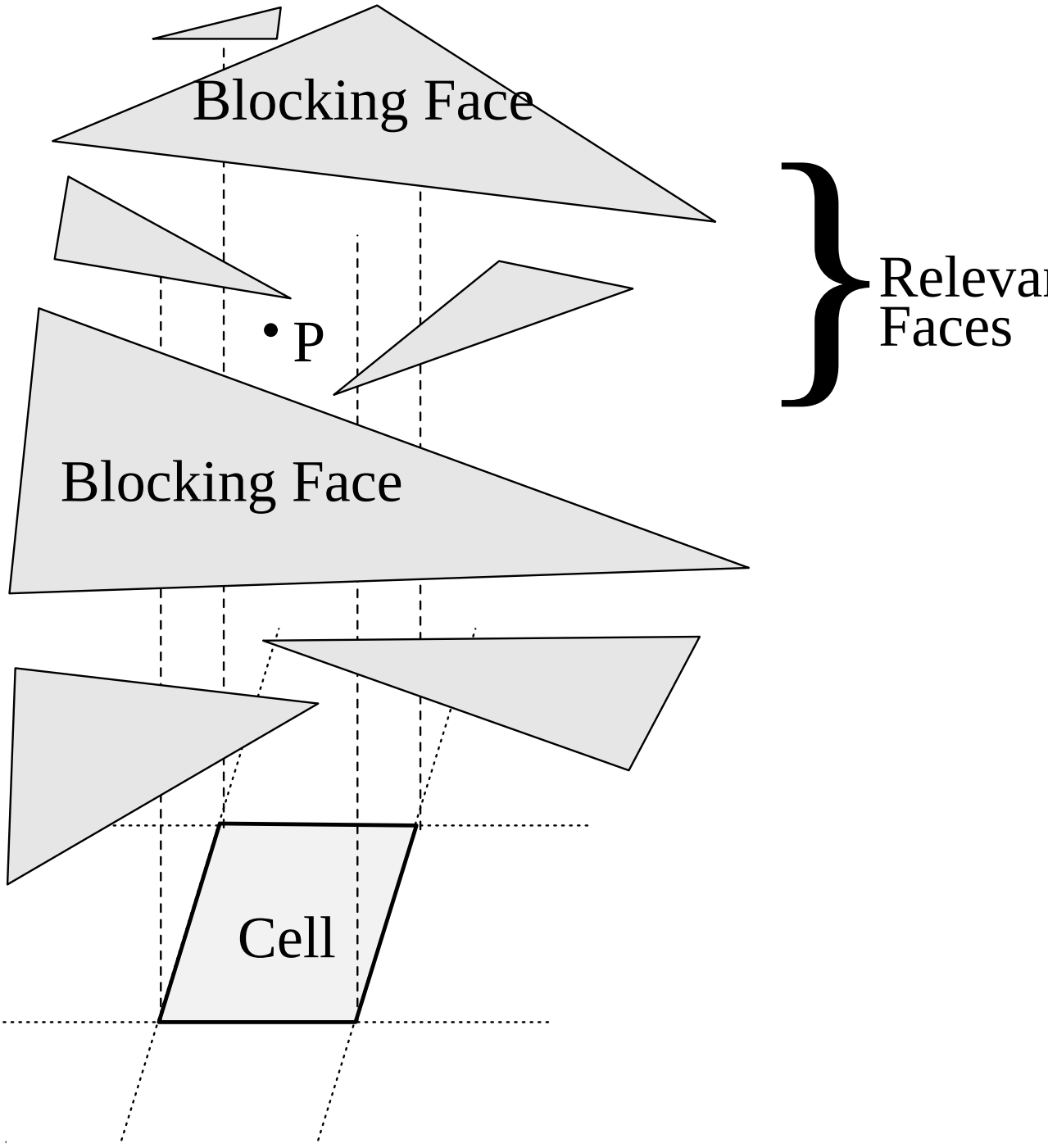

Figure 3: Blocking Face Concept