Efficient Subdivision of Finite-Element Datasets into Consistent Tetrahedra

Guy Albertelli
(albertel@cis.ohio-state.edu)

Roger A. Crawfis
(crawfis@cis.ohio-state.edu)

Department of Computer and Information Science
The Ohio State University

Abstract

This paper discusses the problem of subdividing unstructured mesh topologies containing hexahedra, prisms, pyramids and tetrahedra into a consistent set of only tetrahedra, while preserving the overall mesh topology. Efficient algorithms for volume rendering, iso-contouring and particle advection exist for mesh topologies comprised solely of tetrahedra. General finite-element simulations however, consist mainly of hexahedra, and possibly prisms, pyramids and tetrahedra. Arbitrary subdivision of these mesh topologies into tetrahedra can lead to discontinuous behavior across element faces. This will show up as visible artifacts in the iso-contouring and volume rendering algorithms, and lead to impossible face adjacency graphs for many algorithms. We present here, various properties of tetrahedral subdivisions, and an algorithm for determining a consistent subdivision containing a minimal set of tetrahedra.

***Keywords**: tetrahedralization, mesh subdivision, volume rendering, flow visualization, isosurfaces, metrics, irregular grids.*

1. Introduction

A tetrahedron is the most basic of solid primitives. It has several attractive features for visualization. It is convex. It is defined by four vertices, which can usually be specified in an independent order. A function sampled at these four vertices leads to a unique linear function throughout the tetrahedra. This is a very useful property for interpolation and reconstruction. For these reasons, many author's have developed visualization algorithms for tetrahedral meshes. Shirley and Tuchman [Shirley90], describe an efficient algorithm for volume rendering tetrahedra in their Projected Tetrahedra algorithm. Kenwright and Lane [Kenwright96] describe a technique for efficient particle tracing through tetrahedral meshes. They split the curvilinear cells in their CFD data into five tetrahedra on the fly using an odd/even scheme on the computational coordinates. Yagel, et. al. [Yagel96] describe a volume rendering technique that calculates slices through a finite-element mesh consisting of strictly tetrahedra. Several additional authors describe algorithms that work only on tetrahedral meshes (e.g., [Cignoni96], [Knight96]). Many simulations however, use mesh topologies consisting of primarily hexahedra, with occasional prisms, pyramids and tetrahedra.

The problem of subdividing a finite-element mesh into tetrahedra is currently unknown. An inconsistent subdivision will have the adjacent face of two primitives split differently for each primitive. This inconsistency emanates as a discontinuity in the underlying data field when using tri-linear interpolation (or many other interpolation schemes). This discontinuity is readily visible when taking an iso-contour of the data, as illustrated in Figures 1 and 2. In Figure 1, a regular mesh is subdivided by splitting each voxel into five tetrahedra randomly. Several holes and shading artifacts are clearly visible. Figure 2, shows the same data set with a consistent subdivision. Similar artifacts or numerical instabilities occur using other visualization techniques. An efficient and robust algorithm for subdividing irregular meshes into tetrahedra is needed in order to allow us to use these visualization algorithms.

We will first discuss similar work in Section 2. Section 3 of this paper will present our labeling scheme for discussing subdivisions and present the possible set of subdivisions (without adding points or edges) for a hexahedron. This is perhaps the heart of the paper, and several interesting observations will be presented. Section 4 will discuss the possible subdivisions of pyramids and prisms. Section 5 will then show some characterization experiments we performed to determine what constraints we could impose on a subdivision. Section 6, gives an overview of a simple greedy algorithm we developed for consistently subdividing meshes into tetrahedra. Section 7 presents results on both test data and some real data sets. Finally, we conclude with some future research directions in Section 8.

2. Previous Work

Calculating a 3D tetrahedralization from scattered data points is a well know problem, and the 3D extension to the Delauney triangulation algorithm [Preparata85] is the most prevalent solution. This technique can be applied to unstructured meshes, by simply throwing out the mesh topology. Not only does this destroy the local topology of the mesh, but also ignores the boundary of the original mesh, leading to representations of data outside of the normal problem domain. What is needed is a technique that can produce a consistent tetrahedralization while preserving the original mesh. This implies that no edges or data points can be removed, but only added. Furthermore, any points added need to be within the original volume. Techniques to constrain the Delauney triangulation do exist, but these are usually only applied at the boundary.

Several authors describe how to decompose a uniform or curvilinear mesh into tetrahedra [Garrity90], [Max90], [Shirley90], [Kenwright96]. Here, each voxel or hexahedra is subdivided into five tetrahedra. An alternating pattern of two subdivisions is used to ensure consistency. Max [Max92] employs a subdivision of six tetrahedra per curvilinear cell for a global climate simulation to handle non-planar faces in the data. In their

flow volumes paper [Max93], they also describe a technique for generating a complex unstructured mesh with prisms, such that the prisms can be consistently subdivided into tetrahedra. For curvilinear data sets, care must be taken when the mesh folds back upon itself. Here, a simple alternating scheme fails at the merged seam when the periodic length of the cells is odd. As can be seen

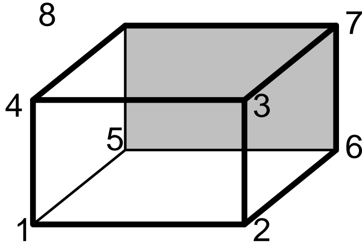

Figure 3. Hexahedron numbering scheme.

in Section 3, this can easily be fixed by using a subdivision into six tetrahedra at the seam. For finite-element meshes, with arbitrary topologies, a robust algorithm is needed that can consistently subdivide the mesh into tetrahedra.

If we are only interested in consistency, then a subdivision of a hexahedron into 24 tetrahedra would ensure consistency. Here, each face is split into four triangles about either the face centroid, or the diagonal intersections, and each triangle is then connected to the hexahedron's centroid to construct tetrahedra. Since each face produces four tetrahedra, and there are six faces, a total of 24 tetrahedra are produced. The draw back, of course, is that the efficient algorithms we are trying to use have a cost proportional to the total number of tetrahedra. Our goal therefore is to construct a tetrahedralization that is consistent and has as few tetrahedra as possible.

3. Possible Hexahedron Subdivisions

It is only possible to subdivide a hexahedron into either five or six tetrahedra without adding additional data points. By adding the hexahedron centroid as a data point, we can produce a subdivision into 12 tetrahedra, where each face is still split by a single diagonal. From here, we can progressively add face centroids, splitting a face into four triangles to produce 14, 16, 18, 20, 22 or 24 tetrahedra. Ideally, we would like to be able to subdivide a mesh using splittings into either five or six tetrahedra. This avoids the large jump to 12, but more importantly, avoids the difficulties in adding new data points to a mesh. This section will examine possible splittings of a single hexahedron.

Consider a single face of a hexahedron. There are two possible diagonals along which the face can be split into tetrahedra. We can encode the diagonal direction in a one bit entity, with a zero indicating the bottom-left to upper-right diagonal and a one indicating the upper-left to lower-right diagonal. For a hexahedron therefore, we have a six bit entity that can encode all of the diagonal directions. Let's order the bits (or faces) such that opposing faces have adjacent bits, say {front | back | left | right | bottom | top}. Figure 3 shows a hexahedron with eight numbered vertices. Our bit assignment is thus:

Slice		
Face	*0*	*1*
Bottom	1 to 6	2 to 5
Top	4 to 7	3 to 8
Left	1 to 8	4 to 5
Right	2 to 7	3 to 6
Front	1 to 3	2 to 4
Back	5 to 7	6 to 8

Table 1. Diagonal slice labeling

This table states that the "front" face has a zero bit for the diagonal slice from node 1 to node 6 and a one bit for the diagonal slice from node 2 to node 5. The other faces are similarly labeled. This six-bit vector leads to 64 possible diagonal sets. Of these 64, it can be shown (Table 2) that 46 can easily be subdivided into either 5 or 6 tetrahedra. The remaining 18 configurations present problems or are configurations we need to avoid. There are exactly two possible configurations that lead to subdivisions into five tetrahedra. These are labeled 010101 and 101010. The two alternating bit patterns. The remaining 44 "good" configurations can be subdivided into 6 tetrahedra (2 prisms each subdivided into three tetrahedra).

Examining all 64 of these cases leads to some interesting insights. All of the 18 "bad" cases can actually be converted to a "good" case with a single bit change. In fact, eight of these can take a single bit change in all but one face, and changing the bit of the appropriate face will produce one of the five tetrahedra configurations. Four other "bad" cases can take a bit change in any of four faces to produce a "good" case, but require three bit changes to produce one of the five tetrahedra configurations. Finally, six of the "bad" cases will take a bit change in only two of the faces, and require 2 bit changes to produce a five tetrahedra configuration.

We can also classify the "bad" cases into two distinct classes. Twelve of the cases actually produce two prisms that can be subdivided. The problem arises on the interior face, where an inconsistent diagonal is chosen for the two prisms. This is annotated as "interior" in Table 2. The remaining 6 bad cases have bit patterns such that opposite faces have diagonals in opposite directions. Alternating directions would thus have a 2-bit pattern of either 01 or 10 for each of the three sets of opposing faces. This leads to 8 possible configurations where the opposing faces have different diagonals. Two of these are the valid subdivisions into 5 tetrahedra, and the remaining 6 are undividable. These six require the 2 bit changes to produce a five tetrahedra configuration.

4. Pyramids and Prisms

Of course, many finite-element meshes consist of other solid primitives (and even non-solid primitives). An analysis of prisms and pyramids was also conducted, and produced similar promising results. Pyramids are especially easy to deal with. They have a single quadrilateral face, which can be split about either

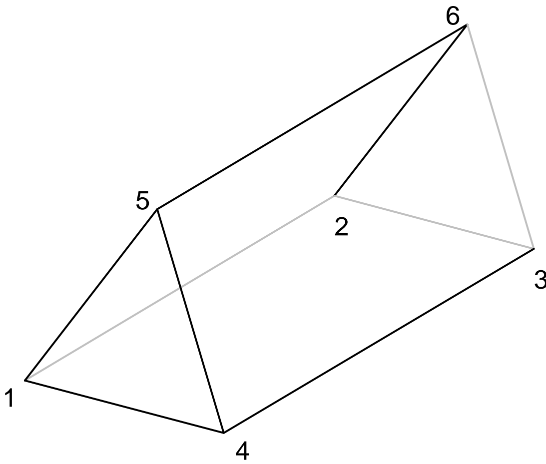

Figure 4. Simple Prism Labelling

diagonal to produce two tetrahedra. Thus, a pyramid abutted against a hexahedron face, will not impose any additional constraints on the system. Note, that the authors are assuming triangular faces are not abutted against quadrilateral faces in any *real* finite-element meshes.

As pointed out in Max, et. al. [Max93], a prism can be split into three tetrahedra. There are three quadrilateral faces that need to be split, bottom, left and right. If we choose diagonals, such that none of the them share a common vertex, then an inconsistent state exists, and a valid subdivision does not exist. A similar scheme of subdividing about the prism centroid can be used for *bad* configurations. This would produce eight tetrahedra, rather than three, and again, leave us with the additional centroid point to manage. Of the eight possible configurations, there are only two bad cases where the diagonals do not touch. Note, that if two diagonals touch, we have a tetrahedra containing the shared vertex. Separating this from the prism, yields a pyramid which can be split arbitrarily into two tetrahedra. Therefore, we can constrain any two faces of a prism and pick the appropriate diagonal for the remaining face. Since two out of eight possible configurations (25%) are *bad* choices, prisms impose perhaps more constraints on our system.

5. Constrained Subdivisions

Since a substantial number of cases can be subdivided into 5 or 6 tetrahedra, our next investigation was examining the effect that flipping a diagonal on one of the bad cases would have on the adjacent hexahedra. Our first question was whether given a 2x2x2 set of hexahedra, in which all of the 24 external faces had their diagonals constrained, could a consistent subdivision always be found. Amazingly, all 16 million possible external face assignments led to consistent internal subdivisions. In fact, on average each configuration could be subdivided in over 290 different sets of tetrahedra. The twelve unconstrained internal faces gives us plenty of freedom in choosing a subdivision.
Further constraining the systems, we examined a 2x2x1 set of hexahedra. Here, we have 16 external faces, over 65 thousand possible diagonal assignments, and only 4 internal faces whose diagonals can be selected for a possible tetrahedralization. Of these possible configurations, only 1520 or 2.32% could not be subdivided into 5 or 6 tetrahedra. Similarly, we also examined a 2x1x1 set of hexahedron, having only a single internal face. Of the 1024 possible external diagonal settings, 110 or 12% led to configurations that could not be consistently subdivided.

In a single hexahedra, if five faces are randomly assigned, for only sixteen of the 192 cases (8.3%), it is impossible to choose the diagonal direction on the remaining face, such that a good configuration results. Additionally, for more than half (100 of 192) of the cases, either choice of the remaining diagonal leads to a good subdivision.

6. A Simple Greedy Algorithm

Our initial idea was to mark each face to be split by the shortest diagonal. This provides well shaped tetrahedra and ensures consistency. We could then use this as a starting point to determine a tetrahedralization, changing diagonal choices as needed. A further refinement to this would be to associate weights with each diagonal, such that those faces that are really skewed would show a strong preference to be split by the shortest diagonal. The preceding analysis illustrates that any 2x2x2 set of hexahedra with exterior constraints can be consistently subdivided, so if we employ a greedy algorithm to assign subdivisions, we can always re-coupe in a relatively small area. In other words, a configuration can not be produced, that can not be corrected within a small localized area.

Our algorithm performs a depth first traversal of a finite-element mesh, starting at a random element. A face adjacency graph is needed to perform the traversal. As we march through the mesh, we mark those zones that have already been processed. If we reach a point where all of the current zone's neighbors have already been processed, we then take the next active cell on the wait list. As we process a cell, we randomly choose a neighbor to process next. Hence, we have a random walk through the mesh. All other neighboring zones that have not been marked are put on the wait list.

A problem arises if we encounter an area where a consistent subdivision can not be achieved without adding centroids. We have two possible solutions for handling this. In the first alternative, we back up to the zone we just came from and try an alternative configuration. In practice, this solves many of the problems. Alternatively, we can pick a good subdivision for the zone giving us difficulty and then try to fix any neighbors that are subsequently in a bad configuration state. We choose to implement the first approach.

Hexahedra List	Encoding Works?	Number of Tets	Reason of failure	Decimal Value
000000	yes	6		0
000001	yes	6		1
000010	yes	6		2
000011	yes	6		3
000100	yes	6		4
000101	yes	6		5
000110	no		Interior	6
000111	yes	6		7
001000	yes	6		8
001001	no		Interior	9
001010	yes	6		10
001011	yes	6		11
001100	yes	6		12
001101	yes	6		13
001110	yes	6		14
001111	yes	6		15
010000	yes	6		16
010001	no		Interior	17
010010	yes	6		18
010011	yes	6		19
010100	no		Interior	20
010101	yes	5		21
010110	no		Undividable	22
010111	no		Interior	23
011000	yes	6		24
011001	no		Undividable	25
011010	no		Undividable	26
011011	yes	6		27
011100	yes	6		28
011101	no		Interior	29
011110	yes	6		30
011111	yes	6		31
100000	yes	6		32
100001	yes	6		33
100010	no		Interior	34
100011	yes	6		35
100100	yes	6		36
100101	no		Undividable	37
100110	no		Undividable	38
100111	yes	6		39
101000	no		Interior	40
101001	no		Undividable	41
101010	yes	5		42
101011	no		Interior	43
101100	yes	6		44
101101	yes	6		45
101110	no		Interior	46
101111	yes	6		47
110000	yes	6		48
110001	yes	6		48
110010	yes	6		50
110011	yes	6		51
110100	yes	6		52
110101	yes	6		53
110110	no		Interior	54
110111	yes	6		55
111000	yes	6		56
111001	no		Interior	57
111010	yes	6		58
111011	yes	6		59
111100	yes	6		60
111101	yes	6		61
111110	yes	6		62
111111	yes	6		63

Table 2. Possible Hexahedron subdivisions

Given an arbitrary mesh, it is not clear if a valid subdivision is possible without adding zone or face centroids. This is a problem that the Computational Geometry community has not seemed to address yet. Ruppert and Seidel [Ruppert92] shows that the problem of subdividing a single concave polyhedra is NP-complete. Our finite-element meshes are certainly concave, but the individual elements are typically convex and of small dimension. The minimal subdivision of an unstructured mesh is still a future area of research. Since it is also unknown whether a valid subdivision should exist, we avoid endless alterations by stopping the subdivision process after several failed attempts. We handle the bad zone by splitting it about its zone centroid, producing 12 tetrahedra in these rare cases.

Storing a valid subdivision of a large mesh would be prohibitively expensive. Instead, we simply store with each zone the resulting six-bit vector that dictates the needed splitting. This allows for the use of algorithms optimized to handle hexahedra to work efficiently, and those that require tetrahedra can quickly and easily subdivide the hexahedra (or prisms and pyramids) into the needed tetrahedra on the fly. The six-bit vector is actual stored as a byte, leaving additional bits to flag zones that need to generate a face centroid. This amounts to a fairly insignificant increase in storage for most finite-element meshes.

7. Results

The real results of this research is more the analysis described in Sections 3 through 5. We tested our algorithm first on regularly gridded data so that we would have an optimal subdivision to compare with. We generated regular grids with several aspect ratios, randomly fixed an increasing number of diagonal slices in order to impose some constraints, and then applied our algorithm. Our algorithm always tries to use one of the subdivisions into five tetrahedra first, before attempting any of the prism subdivisions. With no constraints, we always produce the expected five tetrahedra per cell. The average number of tetrahedra per hexahedra over several runs with varying degrees of constraints was 5.5 tetrahedra per cell. Therefore, half of the hexahedra were split into five tetrahedra and half into six tetrahedra. This amounts to a ten percent increase over the optimal solution without constraints. We also never encountered a bad case that could not be handled by backtracking one zone and trying alternative configurations.

We applied the algorithm to several data sets, summarized in Table 3. The shuttle data, is part of the IRIS Explorer distribution, as is the blunt fin data. The blunt fin is actually a curvilinear grid, so produces trivial results. We also applied it to some sample data distributed with AVS (avs.inp, and box10.inp). The submarine data is courtesy of Lawrence Livermore National Laboratory (LLNL). We are still searching for more complex mesh topologies.

The table lists the number of hexahedra, the resulting number of tetrahedra, and the time to build the adjacency graph. The time to actually perform the subdivision was less than a second on all of these data sets, once the adjacency graph was available. A theoretical bound on the minimal number of tetrahedra is five per hexahedra. Both the bluntfin and the box10 produce this minimum, as expected.

Data set	Hex's	Tetras	CPU time
Shuttle	644	3275	7 sec.
Avs.inp	690	3450	7 sec.
Post	13800	69000	365 sec.
Bluntfin	441	2205	4 sec.
Box10.inp	2744	13720	60 sec.

Table 3. Number of tetrahedra generated.

8. Future Work

There is still some work needed in the analysis of the subdivision of a single hexahedron. Several possible symmetries can be employed to reduce the total set of configurations. More enumeration of which bits will turn good cases into bad cases, and visa versa, would also aid in the development of more efficient algorithms.

There are also several theoretical questions that have arisen as part of this investigation. We have already mentioned the question of whether a mesh is subdividable without additional data points. The subdivision problem can be expressed in terms of graph theory. If we start off with the initial adjacency graph of the mesh, the goal then is to expand each node into either five or six new nodes (for a hexahedron) and refine the connections. Other questions are: Can a mesh be subdivided using strictly a five tetrahedra split? What is the optimal (fewest tetrahedra) splitting of a mesh? Hopefully, this research will stimulate interest in these problems.

Finally, many improvements into the simple algorithm presented here are possible. A genetic or simulated annealing algorithm may be ideally suited for determining a (locally) optimal subdivision. Processing the zones in larger blocks may avoid any bad zones and backtracking. Our current algorithm also makes no use of the kernels of insight uncovered in Section 3. More intelligent picking up the splittings would leverage this information.

Acknowledgements

We would like to thank Nelson Max for initially turning the second author onto this problem. Rephael Wenger provided assistance and pointers into the computational geometry literature, as well as many helpful comments. The Post data was provided by Mark Christon from the Lawrence Livermore National Laboratory. We would also like to thank the reviewers who made several suggestions to improve the paper.

References

[Cignoni96] Cignoni, P., C. Montani, et al. (1996). Optimal Isosurface Extraction from Irregular Volume Data. 1996 Volume Visualization Symposium, San Francisco, CA, ACM.

[Garitty90] Garrity, M. P. (1990). "Raytracing Irregular Volume Data." Computer Graphics **24**(5): 35-40.

[Kenwright96] Kenwright, D. N. and D. A. Lane (1996). "Interactive Time-Dependent Particle Tracing Using Tetrahedral Decomposition." IEEE Transactions on Visualization and Computer Graphics **2**(2): 120-129.

[Knight96] Knight, D. and G. Mallinson (1996). "Visualizing Unstructured Flow Data Using Dual Stream Functions." IEEE Transactions on Visualization and Computer Graphics **2**(4): 355-363.

[Max90] Max, N., P. Hanrahan, et al. (1990). "Area and Volume Coherence for Efficient Visualization of 3D Scalar Functions." Computer Graphics **24**(5): 27-33.

[Max92] Max, N., R. Crawfis, et al. (1992). Visualizing Wind Velocities by Advecting Cloud Textures. Visualization '92, Boston, Massachusetts, IEEE Computer Society Press.

[Max93] Max, N., B. Becker, et al. (1993). Flow Volumes for Interactive Vector Field Visualization. Visualization '93, Los Alamitos, CA, IEEE Computer Society Press.

[Preparata85] Preparata, F. P. and M. I. Shamos (1985). Computational Geometry: An Introduction. New York, Springer-Verlag.

[Ruppert92] Ruppert, J. and R. Seidel (1992). "On the difficulty of triangulating three-dimensional non-convex polyhedra." Discrete Computational Geometry **7**: 227-253.

[Shirley90] Shirley, P. and A. Tuchman (1990). "A Polygonal Approximation to Direct Scalar Volume Rendering." Computer Graphics **24**(5): 63-70.

[Yagel96] Yagel, R., D. M. Reed, et al. (1996). Hardware Assisted Volume Rendering of Unstructured Grids by Incremental Slicing. 1996 Volume Visualization Symposium, San Francisco, CA, ACM.