EUROGRAPHICS '94 / M. Daehlenand L. Kjelldahl (Guest Editors), Blackwell Publishers

Volume 13, (1994), number 3

The Mesh Propagation Algorithm for Isosurface Construction

C.T. Howic and E.H. Blake

Department of Computer Science, University of Cape Town, Private Bag, Rondebosch 7700, South Africa.
Email: cdwin@cs.uct.ac.za

Abstract

A new algorithm, Mesh Propagation, *is presented for the generation of isosurfaces from three-dimensional discrete data sets. While producing the same surface mesh as that generated by a corrected* Marching Cubes *algorithm, its characteristic is that it constructs an isosurface using connected strips of dynamically triangulated polygons. This compact data structure speeds up surface construction and reduces surface storage requirements. The surface can also be displayed more quickly, particularly where there is hardware support for rendering triangle strips.*

With engineering as well as medical imaging applications in mind, the algorithm can be used with both irregular and rectilinear grids of data, the primitive volume elements need not be hexahedral only, and volumes of heterogeneous polyhedral elements are supported without traversal complications.

The algorithm propagates through the cells in the grid and uses the same lookup table topologies as Marching Cubes *to determine patches of surface-element intersection; additional tables are used for non-hexahedral elements. The surface patches are dynamically coded into triangle strips which are then concatenated and linked to construct the surface. The data structures used for propagating through the volume overcome the topological ambiguities associated with table-based methods of surface construction and no holes are generated in the final mesh.*

Keywords: scientific visualization, isosurface construction, triangle strips, irregular and rectilinear grids.

1. Introduction

Several algorithms have been puhlished for constructing isosurfaces from three-dimcnsional (3-D) data sets; in Computer Graphics these algorithms are also known as implicit surface tilers (see [4] for a recent survey). The field of Scientific Visualization makes extensive use of these techniques to give scientists and engineers insight into the data gathered from engineering simulations and medical imaging applications, among others. The algorithms can also provide convenient polygonal approximations to continuous geometric models. The *Marching Cubes* algorithm [3] is based on the principle of building isosurfaces using a linear interpolation technique within cells (or 'cubes') of the data set. Eight points in a rectilinear 3-D grid are used to define a hexahedral cell. By examining a cell's vertices, it is possible to determine whether or not the cell intersects with a specified isosurface. A surface of intersection can be approximated using up to four polygons. These polygons are usually non-planar and so are triangulated for rendering purposes.

Marching Cubes was designed to construct surfaces from rectilinear grids for applications in medical imaging. Wilhelms and Van Gelder [6] use an *octree* data structure to bypass empty cells in the rectilinear grid to speed up surface construction at the affordable cost of creating and storing the octree. Such a data structure repeatedly subdivides the volume into octants and is most efficient on grids of dimension $2^N \times 2^N \times 2^N$. For non-empty cells, they use the *Marching Cubes* method of determining surface polygons. In engineering, simulation grids

(like those used *forfinite element* analysis) are usually irregular and sometimes make use of heterogeneous cell topologics. An example of such a grid is shown in Figure 1.

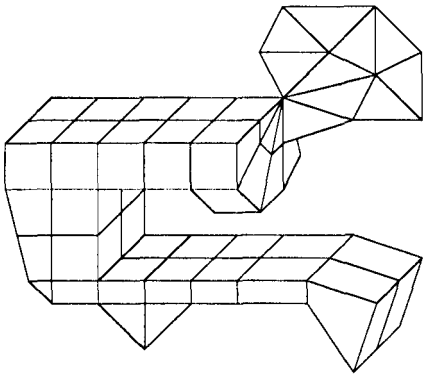

Figure 1. *Example of an irregular grid. These polyhedral cells have four, five and six faces.*

Irregular grids inhibit the coherence optimization used by *Marching Cubes* and complicate the construction of a spanning octree data structure. Wilhelms and Van Gelder note that use of octrees with irregular grids requires further research. We expect practical difficulties there, particularly where the polyhedral cells are heterogeneous in composition, like the grid above. Moreover, the nodes in an octree must have their *maximum* and *minimum* values updated for every increment where results are generated for multiple time steps (common in finite element analysis); this will add pre-proccssing overhead when building a sequence of time-variable isosurfaces for animation purposes.

This paper presents a new isosurface construction algorithm, *Mesh Propagation,* which constructs the same surface mesh as a corrected *Marching Cubes* (with respect to surface 'holes') but with design objectives to

- support irregular as well as rectilinear volume grids;
- use a data structure which will facilitate faster rendering than is possible with discrete polygon storage;
- reduce the storage space required for the isosurface;
- construct the surface in the same order of time as *Marching Cubes* and its *Octree* extension.

The term 'clement', and not cell, is used to refer to volume primitives since this accords with engineering usage, as docs the term 'node' when referring to an element vertex. Elements usually have four *(tetrahedron* element), six *(wedge* element) or eight *(brick* element) nodes. A 'border' is the edge of an element face, and a 'vertex' is a point of intersection between a border and the isosurface. An 'edge' connects two vertices. The term 'mesh' is here defined as a sequence of N triangles stored using N+2 vertices where three consecutive vertices define a triangle of surface-clement intersection. While such a data structure is also known as a triangle 'strip', we reserve this latter term to mean a short mesh of triangles within a single volume element.

The isosurface comprises many triangle meshes, constructed under propagation through the grid. Triangle meshes are significantly more compact than discrete triangle data structures, and the triangles can be displayed more quickly, especially where hardware support is availahle for rendering triangle strips. These features improve exploration and animation of isosurfaces. Reduced storage requirements allow more surfaces to be generated prior to animation, while faster rendering improves the display frame rate.

The triangle meshes of the surface are assembled by propagating through the cells of the volume, connecting triangulated polygons of surface-clement intersection. These polygons arc determined using the *Marching*

Cube method of binary node inspection and linear interpolation except that additional tanles arc used for non-hexahedral elements, and the polygons are *dynamically* coded into triangle strips. An inter-clement data structure allows the algorithm to overcome the topological inconsistency inherent in table-based algorithms which results in obtrusive holes in the polygonalized isosurfaces.

2. Node and Element Connectivity

In irregular volumes, explicit node and element indices are required for grid definition purposes. The algorithm stores this information for the nodes and elements in linear arrays, denoted as Node [] and Element[] respectively. Each element records the indices of the nodes which define it, allowing access to nodal information when needed. For rectilinear grids, these data structures are not required as element and nodal information is implicit for any clement offset in the volume.

The algorithm uses the reference codes of Figure 2 to identify every node, face, and border for a brick element. Similar codes can be derived for other element types.

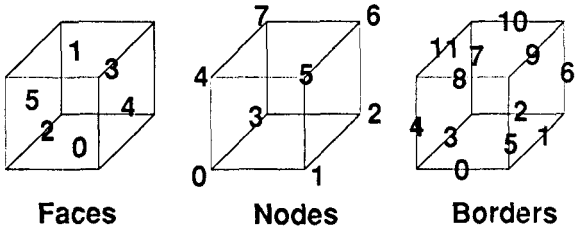

Figure 2. Brick *element reference codes.*

In order to support propagation from element to element, the connectivity between all the elements has to be determined. This information is stored in nextElement[] and nextFace[] arrays for each element. The nextFace values arc used by the algorithm to identify the *entryface* and *exitface* when propagating through an element.

For triangle shading purposes, the result gradient must be computed at clement nodes. In irregular grids, a node can he connected to more than six neighbouring nodes. Computation of the result gradient for such nodes requires a technique like the 'least squares' method used by Gallagher and Nagtegaal [2], while gradients for nodes in rectilinear grids can be computed using finite differences..

To determine nodal and clement connectivities, a hash table (called the *FaceTable),* of pointers to linked lists of element faces is created once per volume. Among the record fields for cach face in a list arc:

```
integer e1,e2;          /* indices to Element[] */
integer face1,face2;    /* element face numbers */
```

The value of *e2* is *null* if the face is not shared between two elements. Using the FaceTable, clement connectivity is easily resolved:

```
for each face F in FaceTable do begin
      Element[F.e1].nextElement[F.face1] := F.e2;
      if ( F.e2 <> null ) then begin
            Element[F.e1].nextFace[F.face1] := F.face2;
            Element[F.e2].nextElement[F.face2] := F.e1;
            Element[F.e2].nextFace[F.face2] := F.face1;
      end;
end;
```

The geometry of an clement face allows determination of which nodes neighbour a given node in the face. For example, the first node in a quadrangular face must be connected to the second and fourth nodes. By considering all faces in the FaceTable, connectivity lists which identify neighbouring nodes can be created for every node in the grid.

In rendering the isosurface, an effective approach is to render an opaque surface within a translucent volume boundary. This is particularly useful in engineering visualization to identify the position of the isosurface within the volume since surfaces are less obviously featured than those constructed from medical data. Those faces in the FaceTable which have an *e2* value of *null* lie on the boundary of the volume, so rendering these faces using translucent properties produces the desired effect.

When the element and nodal connectivities have been established, the FaceTable can be deleted, after boundary element faces have been defined (if required) to free up memory space as it is only needed once per volume and serves no further purpose in constructing isosurfaces.

3. Propagation Principles

Each triangle mesh is constructed by starting at a non-empty element and continuing until the volume boundary is reached, or until encountering an element which has already been visited (each element has a 'visited' flag which is set once it has been processed). A *for-loop* is used at the highest level to ensure every element is visited:

```
for i := 1 to NumberOfElements do
     if ( Element[i] has not been visited ) then
          if ( Element[i] is not empty ) then
               Propagate mesh starting at Element[i];
```

When entering an element (or initialising a mesh in a non-empty element), the LookupTable dynamically constructs a sequence of vertices (called a *strip)* for the triangulated polygon in the element. *Mesh continuity* requires that the sequence of mesh vertices must be correct after coupling of two strips from neighbouring elements — any three consecutive vertices must still represent a triangle defined by one of the two original strips. Where a mesh is already under construction, the LookupTable refers to the face of entry into the clement in order to arrange a correct vertex sequence supporting concatenation of the strip to the current mesh. However, if propagation is starting in the clement, the polygon can be coded arbitrarily into one of several possible strips. For example, a quadrangle of consecutive vertices A, B, C and D could be coded as ABDC, ADBC, DCAB etc. but if the strip must start with edge CB, say, for concatenation with a current mesh ending . . . CB, then the only valid sequence would be CBDA.

In propagating from one element to another, the algorithm keeps a record of the last vertex of the mesh under construction to enable the LookupTable to maintain mesh continuity. Because border codes are not portable across clement boundaries (borders arc labelled locally within an clement), the last vertex must be tracked using the two node numbers (these arc global with respect to the volume) of the border on which it lies.

A polygon of surface-element intersection, once coded as a triangle strip, defines a single *exitface* through which the propagation process will leave the element. This face number is determined by referring to the last two vertices in the strip as these represent the last triangle edge. The element face in which this edge lies is obtained easily with a lookup table mapping the element code numbers for the two borders (on which the edge's vertices lie) to an element face number.

The propagation process will on occasion need to place a strip on a stack. This will occur when it enters an element where there is more than one strip of triangles defined by surface-element intersection, an example of which is given in Figure 3. The algorithm can only make use of one of the strips in the element and the other(s) must be stacked for later use.

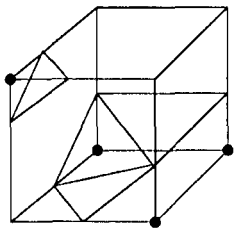

Figure 3. *Example of surface-element intersection where there is more than one strip of triangles.*

Once the exitface has been established, the next element for processing is obtained from the element connectivity information: Element [current] , nextElement [exitface] . If this clement value is *null,* the boundary of the volume has been reached and the current mesh is terminated. If the neighbouring element has not been visited, the algorithm enters it and propagation continues, otherwise the current mesh is terminated and the stack (if not empty) is flushed, each stacked triangle strip providing the source for a new mesh. Once the stack is empty, propagation returns to thefor-loop to find a new element not yet processed. Upon termination of the loop, surface construction is complete.

Ambiguous Element Faces

Certain polygon topologics in the Lorenson and Cline lookup table can lead to a quadrangular clement face having a polygon vertex on each of its borders. The topologics require the generation of two polygon edges using the four vertices. Figure 4 shows how ambiguity arises because either the dashed or the solid edges could be generated.

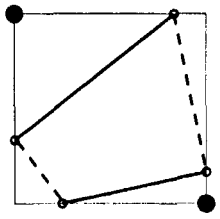

Figure 4. *Ambiguous element faces present two possible polygon edge configurations.*

The triangular faces of tetrahedral and wedge elements create no ambiguity problems. Ning anti Bloomenthal [4] and Wilhelms and Van Gelder [5] describe ambiguous topologics and various disambiguation methods used to arrive at topologically consistent and correct polygon connectivities.

An ambiguous face can lead to an error in the continuity of concatenated triangle strips as mesh continuity is achieved by referring to the last vertex in the strip of the previous element occupied. Figure 5 shows how reference to the last vertex only is insufficient to detect a continuity error caused by an ambiguous face.

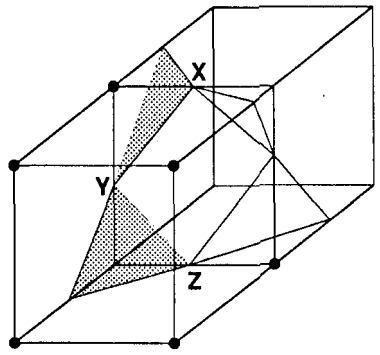

Figure 5. *An ambiguous element face requires two vertex checks to detect for correct mesh continuity. When trying to concatenate the two shaded triangles through the ambiguous face, vertex Y is common to both triangles but vertices X and Z are not.*

The two edges, XY and *YZ*, cannot be connected during mesh propagation becauce the ambiguous face prevents the required vertex continuity. Overcoming ambiguity, therefore, requires the propagation algorithm to keep track of the last *two* vertices in the current mesh and compare these with the first two vertices in the strip returned from the LookupTable. If the two vertices do not match, the propagation process terminates the mesh and starts a new one with the current strip in the second element.

Holes In The Isosurface

The triangle topologies shown in Figure 5 illustrate how ambiguous faces can cause holes in the constructed isosurface, as pointed out by Dürst [1]. Where there is a vertex mismatch, the entryface must be ambiguous and it must contain a hole. The algorithm closes the hole by adding the interior quadrangle of the ambiguous face (as two triangles) to the end of the terminated mesh. This automatic detection of a hole in the surface only applies to an clement entryface or exitface. Other faces in an element must be checked deliberately in case the propagation process does not pass through them.

To reduce this overhead, the LookupTable returns a binary code number indicating which, if any, faces of the current element arc ambiguous. The entryface and exitface are masked from this code and any remaining faces arc checked as follows: the node values on the two faces opposite the ambiguous face arc used to compute a binary coded index into a table, as done for the LookupTable. The table returns *true* if the quadrangle in the ambiguous face must be generated. This hole reparation method satisfiecboth test functions used by Wilhelms and Van Gelder [5], while some disambiguation methods described by them do not.

4. Algorithm Optimization

This section describes two methods which can improve the meshes used to construct the isosurface.

'Lookahend' Mesh Orientation

When construction of a mesh hac just started, the entryface to the current element is *null,* so the LookupTable could select an arbitrary triangle vertex sequence — there is no continuity to satisfy and polygon vertices can be meshed in any of several valid sequences. The sequence chosen might then define an exitface such that propagation will ceace immediately once propagation leaves the element becauce either the next element has been visited already or the boundary of the volume hac been reached.

'Lookahcad' optimization leads to better mesh construction by ensuring, wherever possible, the propagation process is started in such a direction that the process leaves the initial clement through an exitface leading to a non-empty cell which has not been visited. Each face of the clement is considered and the first one which leads to a suitable clement is used by the LookupTable to produce a vertex sequence which will result in that face being the exitface. If no suitable exitface can be found, face 0 is used as the default. The LookupTable only generates optimized strips for one polygon in the current clement. Additional polygons are stacked, so orientating them for improved propagation is pointless. However, when propagation starts with a stacked strip, that strip is oriented using 'lookahcad'.

Post-construction Mesh Concatenation

The propagation process terminates meshes to avoid entering a previously processed clement. This result creates the possibility for post-construction concatenation of meshes which start and end at a common edge. In fact, the use of a 'swap' vertex allows for further concatenation possibilities. Such a vertex is actually not a triangle vertex but rather an instruction code number, placed in the vertex sequence, to inform the rendering algorithm to swap the previous two vertices around. For example, the strip ABCD defines two triangles: ABC and BCD. Where '$' represents the swap code, the strip AB$CD defines the triangles BAC and ACD as the effective vertex sequence in the rendering pipeline is ABACD. Therefore, for example, a mesh ending . . .FAEB can be concatenated with one starting BACD. . . using the join sequence . . . FAESBCD. . .

In finding potential meshes to concatenate with a given mesh, a linked list of pointers to meshes is compiled for each vertex after the surface has been constructed. Only the first three and last three vertices in a mesh need contain a reference to the mesh. A concatenation algorithm then compares the heads and tails of likely meshes using the vertex lists for the first and last three vertices in the given mesh. Following Concatenation, the relevant vertex lists *are* updated to reflect the existence of the new mesh.

5. Algorithm Evaluation

This section assesses the algorithm with respect to some of its major design objectives: storage space, rendering speed and construction speed. Propagation through irregular volumes was tested using fictitious hand-coded volumes and actual engineering data sets, one of which is used below to demonstrate the algorithm's application on an irregular grid with heterogeneous clement topologies. The effectiveness of post-construction mesh concatenation is also discussed.

Surface Storage Requirements

Assuming a compact storage scheme where a triangle is stored as three pointers to vertex data, the number of triangle vertices generated is the same whether the triangles are stored in a mesh or discretely (polygons of surface-element intersection must be triangulated for rendering purposes). However, meshes significantly reduce the number of pointers to vertex information. Results of surfaces constructed with the new algorithm showed a reduction in polygon storage of about SO%, with some meshes containing 500+ triangles and a typical average being between 8 and 10 triangles/mesh. For total surface storage (combined vertex and triangle data) meshes provide a reduction of around 20%.

Surface Rendering Speed

Measurements of rendering speed were performed on a Silicon Graphics Indigo2 (with *Extreme* hardware support) to compare a meshed storage scheme with discrete triangle storage. Meshed surfaces were rendered 2.1 to 2.2 times faster than discrete triangle surfaces, achieving rendering rates in excess of 275K Gouraud-

shaded triangles per second. This faster rendering improves surface exploration and animation. Surfaces from engineering data sets, where animation greatly aids result interpretation, were explored and animated smoothly (typical surfaces contained 1K to 2K triangles) Such performance confirms the suitability of the new algorithm for use in engineering result visualization. No expensive post-processing is required for vertex reduction (as is desirable with medical imaging surfaces) and the compact and fast-to-render meshes support animation sequences well. Most engineering data sets contain fewer than 10K elements, so a surface of 20K triangles would be considered huge but could still be animated at rates greater than 12 frames per second.

Surface Construction Speed

On a Sun SPARCstation 1+, a rectilinear grid of 63 x 63 x 63 elements (the iron protein data set, courtesy of L.Noodleman and D.Case, Scripps Clinic, distributed by the University of North Carolina) was processed to compare surface construction time performance with the figures reported by Wilhelms and Van Gelder [6](who used a SPARCstation 1) for *Marching Cubes* and its *Octree* extension. They used additional data sets, so surface construction times (CPU times) were converted to propagation specd (eIements/see) for uniformity. The iron protein data set provided good insight into the propagation speed of the new algorithm because the surfaces constructed contained triangles well distributed through the volume, removing the bias of a single, localised region of non-empty elements.

The result gradients at nodes were calculated dynamically and not prior to surface construction. Results *are* shown in Table 1, with construction speed shown to the nearest 500 elements per second. Only a rectilinear grid was used for comparison purposes as Wilhelms and Van Gelder did not measure the *Marching Cubes* and *Octree* algorithms on irregular grids.

Table 1: *Comparison of surface construction speeds on a Sun SPARCstation I (Mesh Propagation was run on a SPARCstation 1 + which has a 25% faster CPU). Only rectilinear grids were used as Wilhelms and Van Gelder [6] did not report figures for the other two algorithms on irregular ones.*

Grid Elements % occupied	Mesh Propagation elements/sec SunSPARC 1+	Marching Cubes elements/sec SunSPARC 1	Octree Extension elements/sec SunSPARC 1
0.8	65000	22500	263000
2.9	57000	17500	65500
4.3	55000	15500	43500
7.0	46000	12500	25000
9.2	42000	14000	24500

Triangle strips specd up surface construction time because there arc fewer vertex pointers to process. While *Marching Cubes* and *Octree* must consider three vertices for each triangle generated, *Mesh Propagation* only considers one vertex (unless the triangleis the first in the mesh being constructed, in which case all three vertices must be processed). Statictics from test runs showed that *Mesh Propugation* will consider around 60% fewer vertices than the other two algorithms. The Octree algorithm is the fastest algorithm through largely empty rectilinear volumes because of its ability to bypass empty elements.

Irregular Grids

To demonstrate the versatility of the *Mesh Propagation* algorithm, an irregular volume of two element types (bricks and wedges) was processed. The engineering data set is a femur and prosthesis, and comprises 5908 elements, 1056 of them wedge elements. An example surface (2078 triangles) is shown in Figure 6.*Construction speed cannot be measured in elements/sec because there is more than one clement type; time for surface construction was 0.23 seconds on the Sun workstation and 0.03 seconds on the Silicon Graphics machine.

Post-construction Mesh Concatenation

Mesh concatenation has an insignificant effect on improving storage reduction. Concatenating two meshes removes, at best, two vertex pointers from the resultant mesh. The use of 'swap' vertices can sometimes increase the resultant mesh size by one, as occurs in the second example in Section 4. The application of post-construction concatenation to several surfaces recorded typical reductions in surface storage of around 2.5% while adding about 10% to the surface construction time. The number of meshes used to define the isosurface was reduced between 10% and 15%. A typical engineering surface of 2K triangles can be constructed in less than 0.1 seconds on the Silicon Graphics Indigo2, so the added time for concatenation is negligible, but the improved storage would only really benefit a very long animation sequence.

concatenating many meshes into fewer and longer ones has no noticeable effect on rendering speed, since there is no change in the surface itself, only in its representation. Rendering software for the Indigo2 supports the compilation of many meshes into one long mesh array, where 'end mesh' delimiters arc used between actual meshes. For this reason, a surface of M meshes can be rendered just as quickly (as perceived by the viewer) as if $0.8M$ meshes had been used to construct it.

6. Conclusions

The new *Mesh Propagation* algorithm offers improved interactive visualization capabilities by constructing isosurfaces using a compact structure based on triangle meshes. Isosurfaces can be constructed from irregular as well as rectilinear grids, making the algorithm very well suited to engineering applications since *Marching Cubes* cannot employ its usual coherence optimization and the *Octree* extension is faced with spanning difficulties and pre-processing overhead for each increment of a time-variable data set. Propagation through a volume is simple, regardless of the clement topologies and their connectivity relationships (which are readily determined). Grids of heterogeneous element types arc easily supported. When the propagation process passes through ambiguous faces, isosurface 'holes' arc patched automatically; deliberate checking of other ambiguous faces closes all holes.

Surface storage requirements arc reduced by approximately 50% and 20%)for the surface polygons and complete surface (vertices included) respectively. With available hardware support, the surface can be rendered in less than half the time required for discrete triangle surfaces, and surface construction is faster because vertex processing is reduced.

The data structure used to determine element and node connectivities can also be used to resolve which clement faces lie on the outside of the volume for translucent rendering effects.

The algorithm is vulnerable to incorrect implementation of lookup tables so automatic code generation should be used to construct the larger tables to negate this weakness. The main LookupTable, in particular, is more complex than that used for *Marching Cubes.*

Additional storage space required for the explicit propagation information (element connectivity data) is only a factor when irregular grids arc processed; rectilinear grids have implicit connectivity information and require no additional storage. Since irregular grids arc characteristic of engineering data sets, and because these sets arc typically small (usually less than 10K elements), the extra memory space is not significant by present day standards.

* *See* **page C–521 for Figure 6.**

Acknowledgements

We wish to thank Greg Starke and Mike Eastman at the CERECAM unit at the University of Cape Town for provision of engineering data sets and the University of North Carolina for provision of the iron protein data. Our thanks also to Wayne Paverd for his assistance in rendering the images.

References

[1] Dürst, M.J. (1988). Letters: "Additional reference to 'marching cubes'," *Computer Graphics* (ACM SIGGRAPH Quarterly), Vol. 22, No. 2, **pp.** 72–73.

[2] Gallagher, R.S., and Nagtegaal, J.C. (1989). "An Efficient 3-D Visualization Technique for Finite Element Models and Other Coarse Volumes," *Computer Graphics* (ACM SIGGRAPH Proceedings), Vol. 23, No. 3, **pp.** 185–194.

[3] Lorenson, W.E., and Clinc, H.E. (1987). "Marching cubes: A High Resolution 3-D Surface Construction Algorithm," *Computer Graphics* (ACM SIGGRAPH Proceedings), Vol. 21, No. 4, **pp.** 163–169.

[4] Ning, P., and Bloomenthal, J. (1993) "An Evaluation of Implicit Surface Tilers," *IEEE Computer Graphics and Applications* Vol. 13, No. 6, **pp.** 33-41,

[5] Wilhelms, J., and Van Gelder, A. (1990). "Topological considerations in isosurface generation," (Extended abstract) *Computer Graphics* (ACM SIGGRAPH Proceedings), Vol. 24, No. **5**, pp. 79–86.

[6] Wilhelms, J., and Van Gelder, A. (1992). "Octrees for faster isosurface generation," *ACM Transactions on Graphics*, Vol. 11, No. 3, **pp.** 201–227.