Data Structures for Multiresolution Representation of Unstructured Meshes

Kenneth I. Joy[1], Justin Legakis[2], and Ron MacCracken[3]

[1] Center for Image Processing and Integrated Computing
Computer Science Department
University of California,
Davis, CA 95616 USA
[2] NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
[3] Centric Software, Inc.
50 Las Colinas Lane
San Jose, CA 95119

Summary. A major impediment to the implementation of visualization algorithms on very-large unstructured scientific data sets is the suitable internal representation of the data. Not only must we represent the data elements themselves, but we must also represent the connectivity or topological relationships between the data. We present three data structures for unstructured meshes that are designed to fully represent the topological connectivity in the mesh, but also minimize the data storage requirements in representing the mesh. The key idea is to represent the topology of the mesh by the use of a single data item – the lath – which can be used to encapsulate the topological relationships within the mesh. We present and analyze algorithms that query the spatial relations and properties of these data structures, and analyze the data structures of the dual mesh induced by each.

1 Introduction

With the rapid increase in the power of workstations, the development of state-of-the-art imaging systems, and the increasing sophistication of computational simulations, enormous quantities of information relevant to a particular problem area are now being produced. The primary difficulty faced in many decision and planning situations is analyzing these massive data sets and extracting the relevant information that provides solutions to the problems at hand. These *scientific data sets* are usually multi-valued, meaning that multiple dependent variables – *e.g.*, velocity, pressure, temperature, salinity, sound speed, chemical or nuclear contamination, or even entire "matrices" (tensors) – are associated with each data point.

The topological structure underlying the data set may belong to various topological types: it may be *structured*, where the faces or cells are topologically equivalent to a square or a cube, respectively; or it may be *unstructured*, with a face arrangement consisting of triangular or quadrilateral element, cell arrangement consisting of hexahedral or tetrahedral elements, or combinations of various types [19]. The underlying data structure for structured data sets is completely defined by the indices of the data elements, *e.g.*, each node $\mathbf{p}_{i,j}$ in a mesh is connected with the nodes $\mathbf{p}_{i-1,j}$, $\mathbf{p}_{i+1,j}$, $\mathbf{p}_{i,j-1}$, and $\mathbf{p}_{i,j+1}$. This connectivity simplifies the data representation dramatically, and allows efficient algorithms for moving about the data. With the unstructured mesh however, no such elegant data structure exists, as there is no rigid connectivity rules at the vertices. In this paper, we discuss data structures that represent two-dimensional unstructured meshes.

The representation of these very large unstructured data sets impacts a number of problems in computer graphics and visualization:

- The unstructured meshes generated by subdivision algorithms [5,4,8,7] require a structure that can represent faces with an arbitrary number of edges. The subdivision steps in these algorithms require additional vertices to be inserted into the mesh, and a new connectivity structure to be generated.
- The polygonal representations of implicit surfaces [13] requires the storage of large, unstructured meshes and requires methods to easily traverse these structures.
- The mesh-simplification routines for very-large data sets [14,10,12,22,17] require a data structure that can be easily traversed and locally modified as data elements are collapsed.
- Techniques based on Delaunay triangulations [6,18,3] or Voronoi diagrams require a general data structure that can represent an unstructured mesh. The duality between the Delaunay Triangulation and the Voronoi diagram requires a mesh structure that can easily product its dual mesh. In these algorithms the grids are frequently generated or modified by inserting points and performing local mesh modifications.

To service this variety of applications, a simple data structure is necessary that defines the topology of the mesh, is compact, and facilitates the implementation of efficient algorithms that operate on the data.

We present a method of representing data structures on these unstructured meshes that is based upon the used of a single data type – the lath[1]. In a lath representation, this single data type will be used to represent each of the vertices, edges, and faces of the mesh – and give direct links to the data held at each vertex. Traversals, or accesses of the elements of the mesh, can be accomplished easily via a set basic set of traversal operators that take lath elements as input and output, and a set of queries that take lath elements and return sets of vertices, edges or faces in the neighborhood of the lath. These queries can be used to develop in-place algorithms on the mesh.

[1] Laths are the thin strips of material fastened together to form the network of elements forming a mesh or grid.

In Section 2 we discuss data structures related to the *winged-edge representation* that have been extensively used in non-manifold geometric modeling. These edge-based data structures are the ones that have inspired this work. In Section 3, we present three data structures for unstructured surface meshes: the *split-edge structure*, the *half-edge structure*, and the *corner structure*. We define five basic traversal operators on the elements of data structure, and analyze the differences between these operators in the three representations. We also examine the nine fundamental data-access operations that allow queries on the data structure and analyze the algorithms that implement them. Implementation of the three structures is straightforward, and is discussed in Section 4. Conclusions and future work are addressed in Section 5.

2 Related work

Boundary representations have become the fundamental representation technique in geometric modeling. In these methods, surfaces, edges and vertices of solid objects are represented explicitly, and the topological information about geometrical relationships between these basic elements are represented in a data structure. Various data structures have been presented in [1,2,20] that both efficiently store these relationships and allow modeling operations to be performed on the solids.

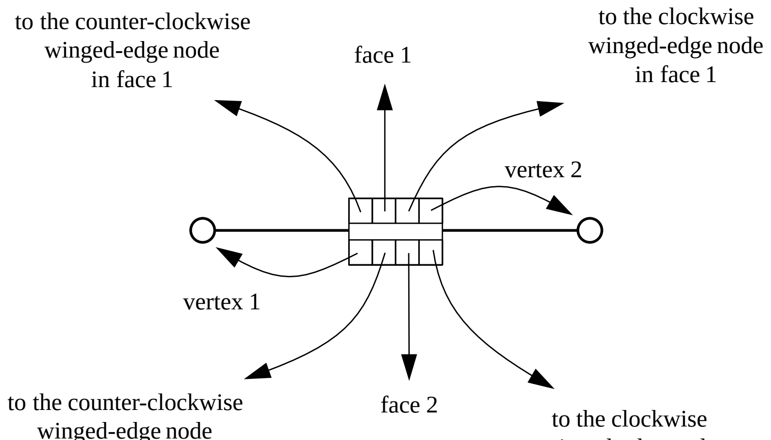
to the counter-clockwise winged-edge node in face 1
face 1
to the clockwise winged-edge node in face 1
vertex 2
vertex 1
to the counter-clockwise winged-edge node in face 2
face 2
to the clockwise winged-edge node in face 2

Fig. 1. The winged-edge node: This node has two groups of links that represent the uses of both "sides" of the edge. It contains a link to the vertex, as well as links to both clockwise and counter-clockwise edges and the data structures for the faces adjacent to the edge.

Ideally, the boundary models are manifolds: any neighborhood on the surface of the model is homeomorphic to an open disk in the plane. Unfortunately, the results of many operations commonly used in geometric modeling (specifically the Boolean set operations) produce non-manifold models – *e.g.*, structures containing a single wire edge in space, structures with two surfaces touching at a single point, or two distinct structures sharing a face. The data structures representing these non-manifold boundary models have received considerable study over the past two decades.

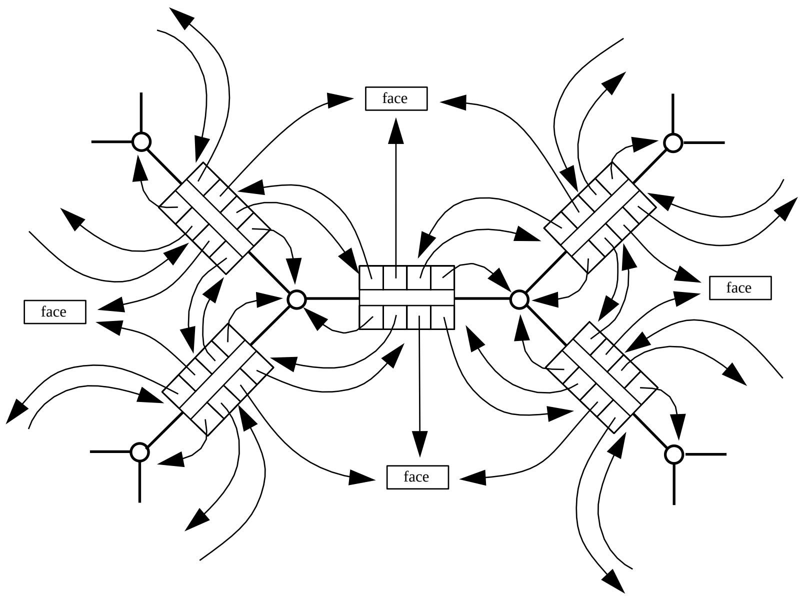

Fig. 2. The winged-edge node when used in a mesh.

Most edge-based data representation schemes are based upon the winged-edge representation of Baumgart [2], who generated a data structure for models that represent the bounding surface of an arbitrary polyhedral structure. Baumgart based his data structure on a single edge record which includes both directional information and information about the faces containing the edge. The winged-edge element, depicted in its most general form, is shown in Figure 1. Each winged-edge node contains links to the two vertices that bound the edge, the two faces that bound the edge, and the winged-edge nodes in the clockwise and counter-clockwise direction around the two adjacent faces, forming a doubly-linked list. The face link is used to access a face data structure, and the vertex link points to a separate vertex data structure. Figure 2 illustrates the use of this element in a mesh.

The primary problem with the classic winged-edge structure is the bundling of the two roles of the edges within one record, and most variations of edge-based data structure are based upon the splitting of the winged-edge element – decoupling the two uses of the edge into two (or more) separate records (see [1]). The first of these, *the split-edge representation*, is commonly credited to

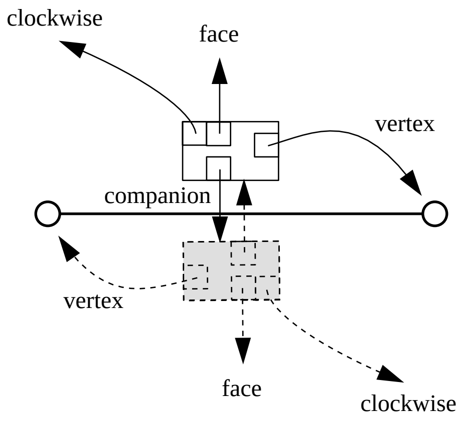

Fig. 3. The split-edge representation of an edge in a mesh.

Eastman [9]. This representation is achieved by splitting each winged edge into two halves, one for each of the two adjacent faces (see Figure 3). Connectivity is maintained by adding an explicit pointer in each edge record that references the opposite "half". Figure 4 illustrates the use of the split-edge nodes in a mesh.

Other variations on the winged-edge data structure also attempt to give a unique representation to the multiple uses of each edge. The hybrid-edge representation of Kalay [15] and Mantyla [16][2] attempts to use the best characteristics of the winged-edge structure and the split-edge structure. It breaks the representation of an edge into a single edge node that represents the edge itself, and two directional nodes, called segments, that specify the directional uses of the node.

By examining Figure 4, a symmetric representation of the structure can be visualized by noting the seemingly dual role played by faces and vertices. In this case two vertices bound an edge, as well as two faces, and the edge-based structure can be constructed so that the representation of faces and vertices is remarkably similar. This symmetry was first observed by Woo [21].

In mesh-based applications used in multiresolution analysis and visualization, the meshes represent manifolds, and the data is held in the vertices of the mesh. Here the primary operations focus on movement about the mesh and queries that inquire about information in the neighborhood of a vertex

[2] We note that Mantyla called his representation a *half-edge* data structure – a term that we will use in Section 3. His representation is very close to the hybrid-edge representation of Kalay.

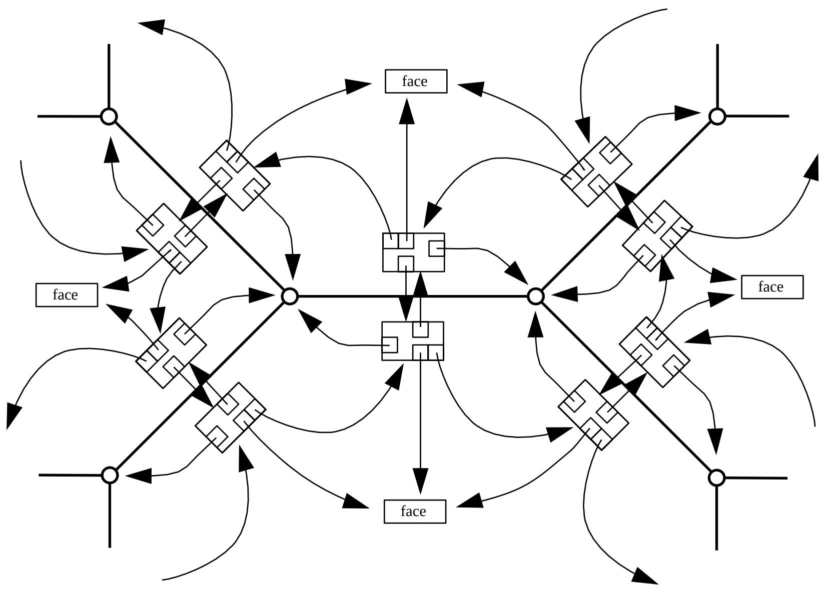

Fig. 4. The split-edge representation of an edge in a mesh.

or face. These operations are required for the in-place calculations necessary for subdivision algorithms, mesh-reduction algorithms, or mesh-enhancement methods. Thus, we define a *mesh* $\mathcal{M}$ to be a set of vertices $\{\mathbf{p}_0, \mathbf{p}_1, ..., \mathbf{p}_n\}$ and an associated simplicial complex which specifies the connectivity of the vertices. Each edge is defined by two vertices that are connected in the simplicial complex. Each face is defined by a minimal connected loop of vertices. We assume that the mesh is well-connected (*i.e.*, no vertex lies on an edge not containing that vertex), all faces are closed in a mesh, and no two adjoining faces of the mesh intersect.

3 Lath-based data structures

In our representations, a data structure storing a mesh will be based upon a single data type called a *lath*. Each lath will be identified with exactly one vertex, one edge and one face of the mesh and each face-edge pair, face-vertex pair, or edge-vertex pair in the mesh can be associated with a single lath. Thus, each edge of the mesh will have two laths associated with it: one for each face-edge pair (two faces per edge), or for each edge-vertex pair (two vertices per edge). There is no need for edge-based elements or face-based elements in the data structure, as each of these can be specified by specifying a single lath element. Lists of laths can represent lists of edges, lists of faces or lists of vertices.

We assume that vertices contain the geometric information and laths contain the topological, or connectivity, information in the mesh. The lath elements can be connected in various ways, and we present three examples of lath-based data structures in the following sections.

3.1 The split-edge representation

In the split-edge data structure the topology of the mesh is carried by a lath element that is similar to the split-edge element given in Section 2.[3] Here

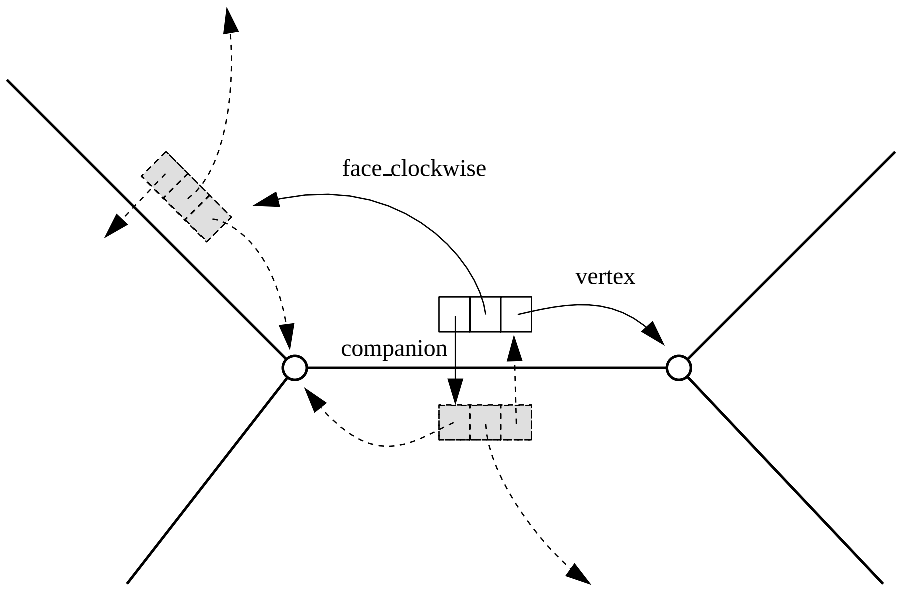

Fig. 5. The lath element for the split-edge representation of a mesh. Three links are explicit in the lath element: a link to the vertex information, a link to the lath that forms the edge companion of the element, and a link to the lath is the next lath in a clockwise traversal of laths in the same face.

a lath element L contains three separate links: First, a link to the vertex information associated with L; second, a link (the *companion* link) to a lath that represents the same edge as L, but the opposite vertex; and third, a link (the *face_clockwise* link) to the lath that follows L in a clockwise traversal of the face that L represents. The split-edge lath can be pictured as in Figure 5 and its use in a mesh is illustrated in Figure 6. Every edge-face pair, edge-vertex pair, and face-vertex pair in the mesh is associated with exactly one lath, and each lath is identified with exactly one vertex, one edge, and one face.

[3] Here we have eliminated the face links for clarity.

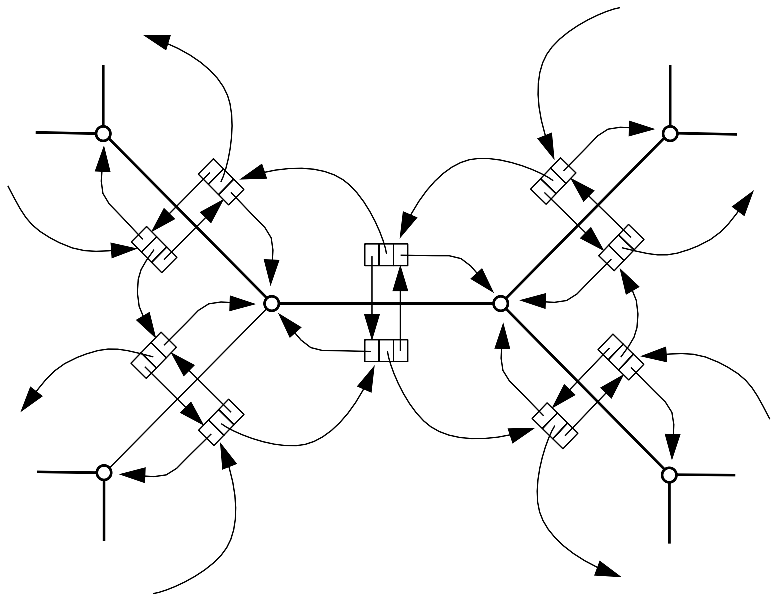

Fig. 6. The split-edge lath elements in a mesh. We note the basic loops induced by the split-edge laths: One in a clockwise direction around the faces of the mesh, and one in a counter-clockwise direction about the vertex.

From Figure 6, we can see that the lath structure sets up a contiguous structure in the mesh, and induces two basic loops around the mesh elements: one in a clockwise direction about the face of the mesh (Figure 7), and one in a counter-clockwise direction about the vertex (Figure 8).

We can traverse this structure in several ways, and for uniformity define the following operators on laths: Given a lath L,

- $\mathsf{ec}(L)$ returns the edge companion of L – the lath element that represents the same edge as L, but the opposite vertex and face.
- $\mathsf{cf}(L)$ returns the lath that follows L in a clockwise traversal of the face that L represents.
- $\mathsf{ccf}(L)$ returns the lath that follows L in a counter-clockwise traversal of the face that L represents.
- $\mathsf{cv}(L)$ returns the lath that follows L in a clockwise traversal of laths about the vertex that L represents.
- $\mathsf{ccv}(L)$ returns the lath that follows L in a counter-clockwise traversal of laths about the vertex that L represents.

In this case, the operators ec and cf are embodied in the split-edge data structure. The function $\mathsf{ccv}(L)$ is quickly identified as returning the lath defined by $\mathsf{cf}(\mathsf{ec}(L))$. The function $\mathsf{ccf}(L)$ can be obtained in two ways:

(1) by traversing laths representing the face (via cf), until reaching the lath L' where $\mathsf{cf}(L') = L$, or

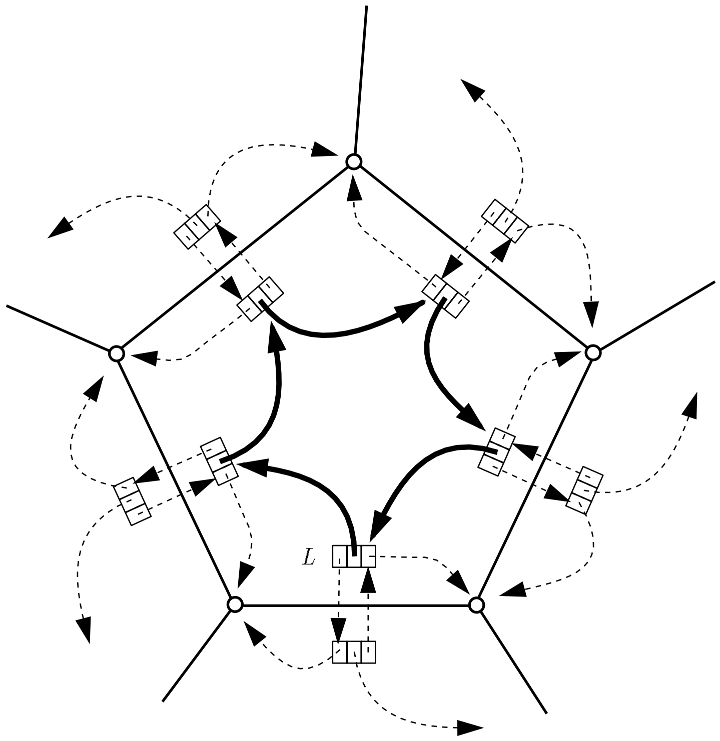

Fig. 7. A face loop in the split-edge structure. Starting with the lath L and following *face_clockwise* links, a sequence of laths is generated that all reference the same face.

(2) by traversing laths surrounding the vertex (using ccv) until we reach the lath L' with $\mathsf{ccv}(L') = L$, then $\mathsf{ccf}(L) = \mathsf{ec}(L')$.

In the first case, this algorithm is linear in the number of edges in the face represented by L. In the second case, this algorithm is linear in the number of edges radiating from the vertex represented by L. The operators depend only on the local complexity in the mesh, not the size of the entire mesh.

The function cv can also be defined in two ways:

(1) as $\mathsf{cv}(L) = \mathsf{ec}(\mathsf{ccf}(L))$,
(2) or by successively traversing laths surrounding the vertex (using ccv). The lath L' is returned by the algorithm, where L' has the property that $\mathsf{ccv}(L') = L$.

These operators allow us to move about the mesh by using the lath elements.

Boundary Considerations Boundaries are easily represented in this structure by storing `Null` values in the companion links (see Figure 9). This implies

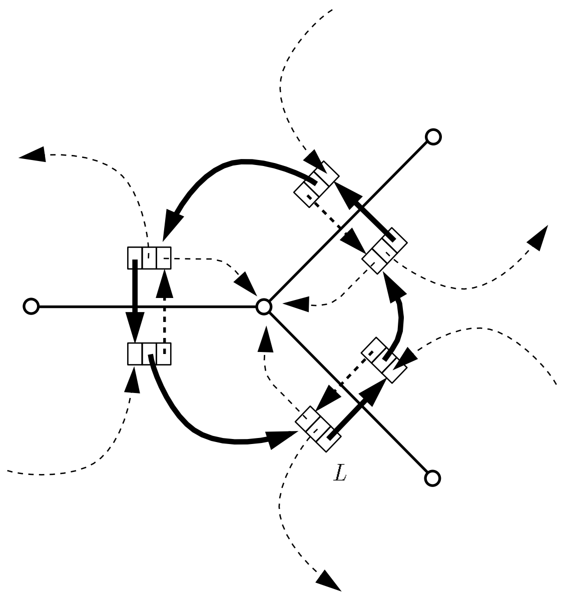

Fig. 8. A vertex loop in the split-edge structure. Starting with the lath L, and following *companion* links and *face_clockwise* links, a sequence of laths is generated that all reference the same vertex.

that the edge-companion function $\mathsf{ec}(L)$ may not return meaningful information, and alternate steps must be substituted in the traversal operators. Since cf is always defined in this structure, all operators can be implemented without the ec operation if necessary.

Data access primitives in the split-edge data structure Given a lath representing one of the three elements of the mesh (vertex, edge, or face), the data structure can be queried for all of the neighboring elements of a given type. We can identify 9 separate queries, each returning a list of laths that represent the desired elements in the mesh:

- $\mathcal{Q}_{\mathrm{VV}}$: Given a vertex V, find all vertices that share an edge with V.[4]
- $\mathcal{Q}_{\mathrm{VE}}$: Given a vertex V, find all edges that radiate from V.
- $\mathcal{Q}_{\mathrm{VF}}$: Given a vertex V, find all faces of the mesh that contain this vertex.
- $\mathcal{Q}_{\mathrm{EV}}$: Given an edge E, find its two vertices.
- $\mathcal{Q}_{\mathrm{EE}}$: Given an edge E, find all edges that share a vertex with E.
- $\mathcal{Q}_{\mathrm{EF}}$: Given an edge E, find all faces of the mesh that contain E.
- $\mathcal{Q}_{\mathrm{FV}}$: Given a face F, find all its vertices.
- $\mathcal{Q}_{\mathrm{FE}}$: Given a face F, find all its edges.

[4] The result of this query is commonly called the *one-neighborhood* of the vertex.

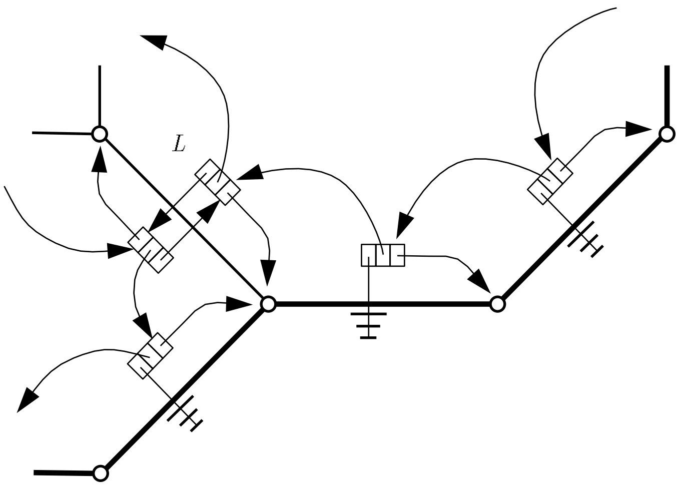

Fig. 9. The boundary of a mesh with an associated split-edge data structure. Here the companion links of the split-edge elements are `Null` on the boundary.

- $\mathcal{Q}_{\mathrm{FF}}$: Given a face F, find all faces of the mesh that share an edge or a vertex with F.[5]

The four basic queries are $\mathcal{Q}_{\mathrm{EF}}$, $\mathcal{Q}_{\mathrm{EV}}$, $\mathcal{Q}_{\mathrm{FE}}$, and $\mathcal{Q}_{\mathrm{VE}}$, as they either return information stored explicitly in laths, or they return information stored in the two basic loops introduced in the split-edge structure.

- $\mathcal{Q}_{\mathrm{EF}}$– Given a lath L representing an edge-face pair in the mesh, laths representing the two faces bounding the edge are given by L and $\mathsf{ec}(L)$. If the edge represented by L is on the boundary, then this query returns only L, since $\mathsf{ec}(L)$ does not exist.
- $\mathcal{Q}_{\mathrm{EV}}$– Given a lath L representing an edge of the mesh, the two laths that represent the vertices that bound the edge are given by L and $\mathsf{cf}(L)$.
- $\mathcal{Q}_{\mathrm{FE}}$– Given a lath L representing a face-edge pair in the mesh, laths representing the edges of the face are given by successively applying the cf operator – following the clockwise loop about the face.
- $\mathcal{Q}_{\mathrm{VE}}$– Given a lath L representing an edge-vertex pair, laths representing edges that radiate from the vertex are obtained by successively applying the ccv operator – following the counter-clockwise loop about the vertex.

The remaining queries are implemented in terms of these basic queries.

- $\mathcal{Q}_{\mathrm{FV}}$– Given a lath L representing a face, the laths representing the vertices of the face are the same as those of $\mathcal{Q}_{\mathrm{FE}}$.
- $\mathcal{Q}_{\mathrm{VV}}$– Given an lath L, laths representing the vertices of the edges that radiate from the vertex represented by L are given by applying $\mathsf{cf}(L)$ for each lath L in $\mathcal{Q}_{\mathrm{VE}}$.

[5] The result of this query is commonly called the *stencil* of the face (see [11]).

- $\mathcal{Q}_{\mathrm{VF}}$is identical with $\mathcal{Q}_{\mathrm{VE}}$, as each lath identified by $\mathcal{Q}_{\mathrm{VE}}$belongs to a unique face.
- $\mathcal{Q}_{\mathrm{EE}}$– Given a lath L representing an edge E, $\mathcal{Q}_{\mathrm{EE}}$produces a set of laths that correspond to the edges that radiate from the two vertices of E. This is implemented by taking the union of the results of $\mathcal{Q}_{\mathrm{VE}}$for both L and $\mathsf{cf}(L)$. To insure that two laths do not exist in the output for the edge represented by L, either L or $\mathsf{ec}(L)$ (which is generated by $\mathcal{Q}_{\mathrm{VE}}(\mathsf{cf}(L)))$ is removed before output.
- $\mathcal{Q}_{\mathrm{FF}}$– Given a lath L representing a face-edge pair, all such faces are obtained by applying $\mathcal{Q}_{\mathrm{VE}}$to each lath returned by $\mathcal{Q}_{\mathrm{FE}}$. Unfortunately, this produces multiple laths in the output for each face. To obtain a single lath for each face in the output, we use a marking strategy that operates as follows:
  - Mark each lath encountered in $\mathcal{Q}_{\mathrm{FE}}$.
  - For each lath L encountered by $\mathcal{Q}_{\mathrm{VE}}$, if L is unmarked, insert it in the output list. Then mark all laths corresponding to edges of the face that L represents – *i.e.*, all laths in $\mathcal{Q}_{\mathrm{FE}}(L)$.

  The output list contains laths returned by this query, one lath per face.

It is possible that $\mathcal{Q}_{\mathrm{VE}}$(and $\mathcal{Q}_{\mathrm{VF}}$, which is identical in this structure) fails on the boundary. For example, in Figure 9, $\mathcal{Q}_{\mathrm{VE}}$(L) returns two laths before encountering a `Null` link on the boundary. To retrieve all laths that represent edges that radiate from the vertex, the query must use `cv` to traverse around the vertex in the opposite direction.

3.2 The half-edge representation

In the half-edge data structure the topology of the mesh is carried by a lath element that is defined as in Figure 10. Here a lath element L contains three separate links: First, a link to the vertex information associated with L; second, a link (the *companion* link) to a lath that represents the same edge as L, but the opposite vertex; and third, a link (the *vertex_clockwise* link) to the lath that is the next lath in a clockwise vertex traversal of the vertex that L represents.

From Figure 11, we can see that the lath structure sets up a contiguous structure in the mesh and again induces two basic loops around the mesh elements: one in a counter-clockwise direction about the face (Figure 12) and one in a clockwise direction about a vertex (Figure 13). We associate a lath L with the face traversed by this counter-clockwise loop, and with the edge bounded by the vertices given by the vertex pointers of L and L's companion. In this way, every edge-face pair, edge-vertex pair and face-vertex pair in the mesh is associated with exactly one lath, and each lath is identified with exactly one vertex, one edge, and one face.

We can traverse this structure by setting up the same operators as in the split-edge representation. For a given lath L,

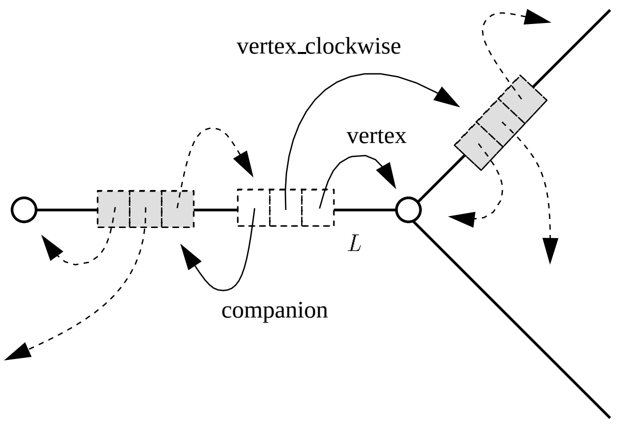

Fig. 10. The lath element for a half-edge representation of a mesh. Three links are explicit in the lath element: a link to the vertex information, a link to the lath that forms the edge companion of the element, and a link to the lath that is the next lath in a clockwise traversal about the vertex defined by L.

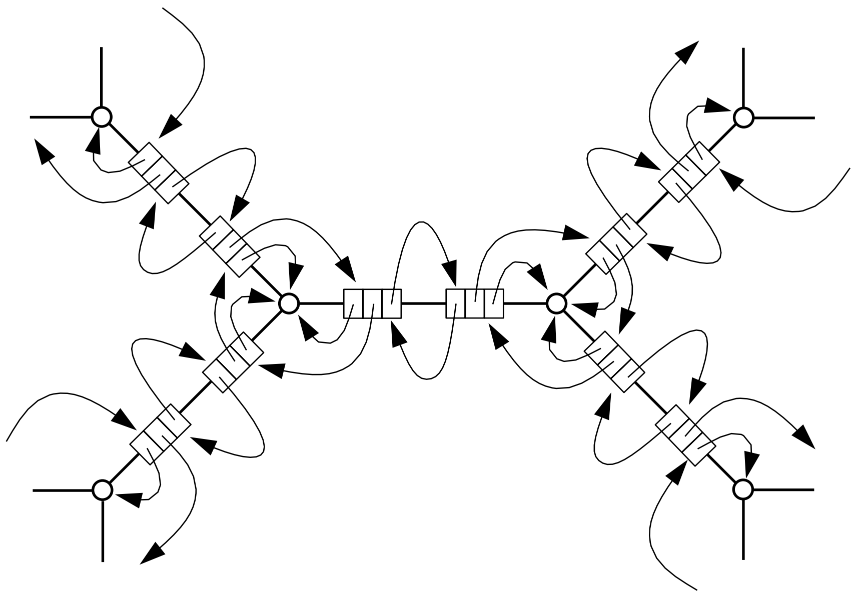

Fig. 11. The half-edge lath elements in a mesh. We note the basic loops induced by the split-edge laths: One in a counter-clockwise direction around the faces of the mesh, and one in a clockwise direction about the vertex.

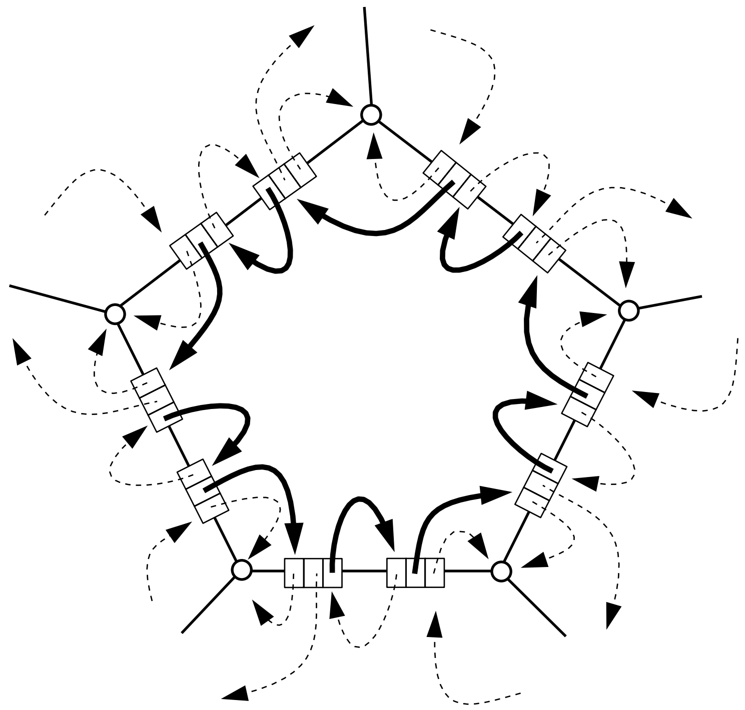
Fig. 12. A face loop in the half-edge structure. Starting with the lath L, and following *vertex_clockwise* links and *companion* links, a sequence of laths is generated that all reference the same face.

- $\mathsf{ec}(L)$ returns the edge companion of L – this information is contained in the links of the lath element, and returns the lath that represents the same edge, but opposite vertex and adjacent face that bound the edge.
- $\mathsf{cv}(L)$ returns the lath that follows L in a clockwise traversal of laths about the vertex that L represents. This information is contained in the lath through the *vertex_clockwise* link.
- $\mathsf{ccf}(L)$ returns the lath that follows L in a counter-clockwise traversal of the face that L represents. It is implemented as $\mathsf{ccf}(L) = \mathsf{ec}(\mathsf{cv}(L))$
- $\mathsf{cf}(L)$ returns the lath that follows L in a clockwise traversal of the face that L represents. This can be obtained in two ways: first by traversing (through ccf) the edges of the face in a counter-clockwise manner until reaching the lath L' that has the property $\mathsf{ccf}(L') = L$; or secondly, by successively applying cv to $\mathsf{ec}(L)$ until reaching L'.
- $\mathsf{ccv}(L)$ returns the lath that follows L in a counter-clockwise traversal of laths about the vertex that L represents. This can be obtained in two ways: First by traversing (via cv) the clockwise link about the vertex until obtaining a lath L' such that $\mathsf{cv}(L') = L$; or alternatively, as $\mathsf{cf}(\mathsf{ec}(L))$.

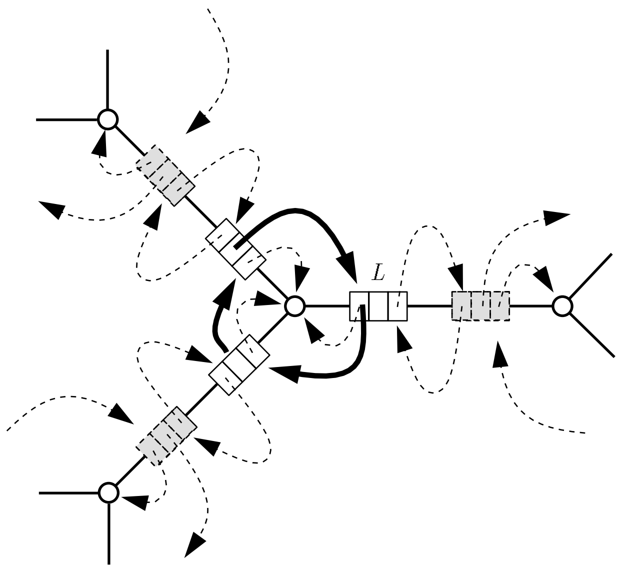

Fig. 13. A vertex loop in the half-edge structure. Starting with the lath L, and following *vertex_clockwise* links, a sequence of laths is generated that all reference the same vertex.

ec, ccf and cv can be implemented in constant time. cf and ccv are linear in the number of edges belonging to a face, or the number of edges radiating from a vertex, depending on the implementation.

Boundary Considerations Boundaries are represented in the half-edge structure by storing `Null` values in the *vertex_clockwise* links (see Figure 14). In this case, the cv operator may not be defined, and steps must be taken to use alternate operators to move about the mesh in the area of the boundary. However, since there are always two laths that exist on each edge, the ccf operator and the ec operator are always defined.

Data access primitives in the half-edge data structure Given a lath representing one of the three elements of the mesh (vertex, edge, or face), the half-edge data structure can be queried for all of the neighboring elements of a given type.

The four basic queries are $\mathcal{Q}_{\mathrm{EF}}$, $\mathcal{Q}_{\mathrm{EV}}$, $\mathcal{Q}_{\mathrm{FE}}$, and $\mathcal{Q}_{\mathrm{VE}}$, as they either return information stored explicitly in laths, or they return information stored in the two basic loops introduced by the data structure. The queries are basically the same here, except that we use ccf to traverse the edges of the face, and cv to traverse the edges around the vertex. The boundaries are handled differently as the ec and ccf operators are always defined, but the cv operator may not be.

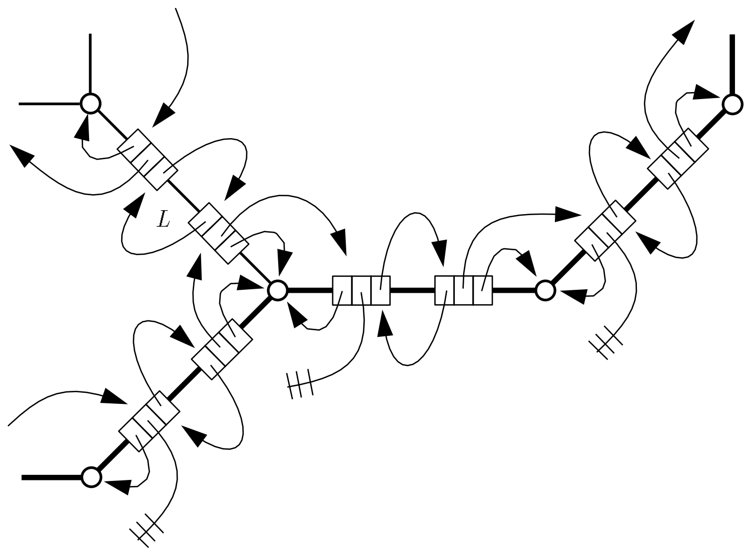

Fig. 14. The boundary of a mesh with an associated half-edge data structure. Here the *vertex_clockwise* links of the half-edge elements are `Null` on the boundary.

- $\mathcal{Q}_{\mathrm{EF}}$– Given a lath L representing an edge in the mesh, laths representing the two faces bounding the edge are given by L and $\mathsf{ec}(L)$. In this case, there are always two laths that represent an edge.
- $\mathcal{Q}_{\mathrm{EV}}$– Given a lath L representing an edge of the mesh, the two laths that represent the vertices that bound the edge are given by L and $\mathsf{ec}(L)$.
- $\mathcal{Q}_{\mathrm{FE}}$– Given a lath L representing a face-edge pair in the mesh, laths representing the edges of the face are found by successively using the ccf operator – following the counter-clockwise loop about the face.
- $\mathcal{Q}_{\mathrm{VE}}$– Given a lath L representing an edge-vertex pair, the laths representing edges that radiate from the vertex are obtained by successively using the cv operator – following the clockwise loop about the vertex.

The remaining queries are implemented in terms of these basic queries.

- $\mathcal{Q}_{\mathrm{FV}}$– Given a lath L representing a face, the laths representing the vertices of the face are the same as those of $\mathcal{Q}_{\mathrm{FE}}$.
- $\mathcal{Q}_{\mathrm{VV}}$– Given an lath L, we can obtain the required laths by taking $\mathsf{ec}(L)$ for each lath L in $\mathcal{Q}_{\mathrm{VE}}$.
- $\mathcal{Q}_{\mathrm{VF}}$is identical with $\mathcal{Q}_{\mathrm{VE}}$, as each lath identified by $\mathcal{Q}_{\mathrm{VE}}$belongs to a unique face.
- $\mathcal{Q}_{\mathrm{EE}}$– Given a lath L representing an edge E, $\mathcal{Q}_{\mathrm{EE}}$produces a set of laths that correspond to the edges that radiate from the two vertices of E. This is implemented by taking the union of the results of $\mathcal{Q}_{\mathrm{VE}}$for both L and its companion $\mathsf{ec}(L)$. Again, we insert only one of L and $\mathsf{ec}(L)$ in order to have only one lath per edge in the output.

- $\mathcal{Q}_{\mathrm{FF}}$is defined exactly as in the split-edge representation with the above modifications to $\mathcal{Q}_{\mathrm{EV}}$and $\mathcal{Q}_{\mathrm{FE}}$.

It is possible that $\mathcal{Q}_{\mathrm{VE}}$(and $\mathcal{Q}_{\mathrm{VF}}$, which is identical in this structure) encounters `Null` links on the boundary. For example, in Figure 9, $\mathcal{Q}_{\mathrm{VE}}(L)$ returns two laths before encountering a `Null` link on the boundary. To retrieve all laths that represent edges that radiate from the vertex, the query must use `ccv` to traverse around the vertex in the opposite direction until another `Null` link is encountered. Unlike the split-edge structure, one always encounters laths that represent an edge-vertex pair.

3.3 Duality

The dual of a mesh is constructed by swapping the roles of the faces and vertices in the mesh. If, in each lath of the data structure, we replace the vertex link by a link to a similar face structure, we obtain a data structure for the dual mesh. If the mesh is represented by a half-edge structure (see Figure 15), the counter-clockwise vertex loop is transformed into a clockwise loop on the face of each element of the dual, and the counter-clockwise face loop of the half-edge structure is transformed into a clockwise loop about each vertex in the dual. That is, the half-edge structure induces a split-edge data structure on the dual mesh. Conversely, if the original mesh is represented with a split edge structure, a half-edge data structure is induced on the dual mesh (see Figure 16).

3.4 The Corner Data Structure

In the corner data structure the topology of the mesh is carried by a lath element that is defined as in Figure 17. Here an element L contains three separate links: First, a link to the vertex information associated with L; second, a link (the *face_clockwise* link) to a lath that represents the next lath in a clockwise traversal of laths of the face L represents; and third, a link (the *vertex_clockwise link*) to the lath that is the next lath in a clockwise traversal of the vertex that L represents. Here the lath elements are easily identified with each vertex-face pair. The edge identified with each lath L is that edge bounded by the vertex of L and the vertex of the lath identified by the *face_clockwise* link.

From Figure 18, we can see that the lath structure sets up a contiguous structure in the mesh. Again, there are two basic loops that can be identified, one which traverses the clockwise laths about a face (Figure 19), and one which traverses the clockwise laths about a vertex (Figure 20). We can traverse this structure by setting up the same operators as in the split-edge and half-edge representations. Given a lath L,

- $\mathsf{cf}(L)$ returns the lath that follows L in a clockwise traversal of the face that L represents. This information is given directly in the *face_clockwise* link of L.

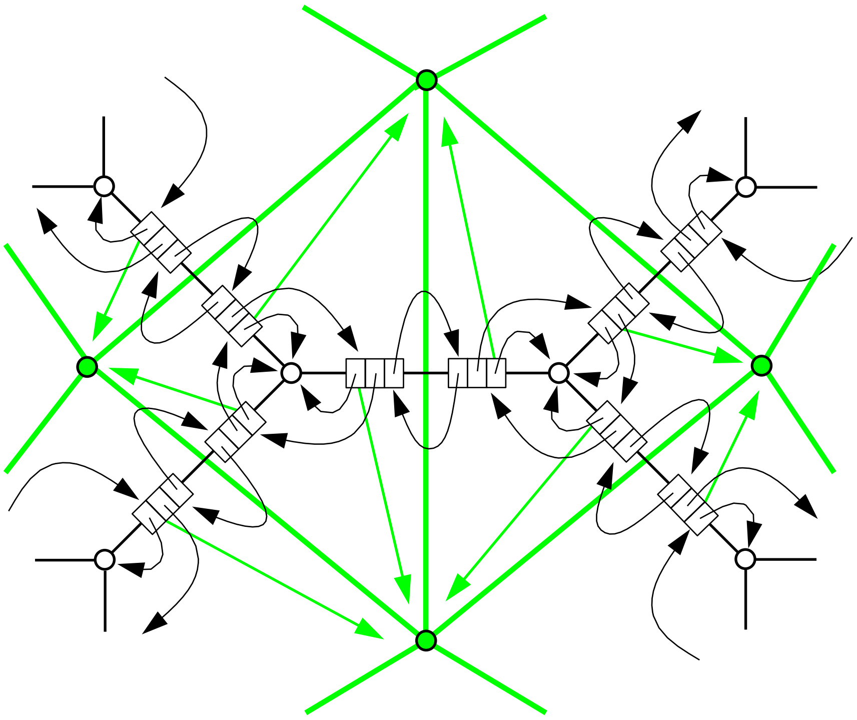

Fig. 15. Representing the dual of a mesh with a half-edge data structure. The original mesh and data structure are shown in black with the dual mesh shown in gray. The gray vertex links are the links to the face nodes in the dual structure. Note the counter-clockwise loops about the face, and the clockwise loops about the vertex in the dual mesh.

- $\mathsf{cv}(L)$ returns the lath that lath follows L in a clockwise traversal of laths about the vertex that L represents. This information is given directly in the lath through the *vertex_clockwise* link of L.
- $\mathsf{ec}(L)$ returns the edge companion of L – In the corner representation, this information is not given directly in the lath links, but can be calculated by $\mathsf{cv}(\mathsf{cf}(L))$. If L represents a boundary edge of the mesh, the edge companion does not exist.
- $\mathsf{ccf}(L)$ returns the lath that follows L in a counter-clockwise traversal of the face that L represents. It is implemented in two ways: as $\mathsf{ccf}(L) = \mathsf{cv}(\mathsf{ec}(L))$, or by traversing the face clockwise (via cf) until reaching the lath L' where $\mathsf{cf}(L') = L$.
- $\mathsf{ccv}(L)$ returns the lath that follows L in a counter-clockwise traversal of laths about the vertex that L represents. This can also be obtained in two ways: as $\mathsf{ccv}(L) = \mathsf{ec}(\mathsf{cf}(L))$, or by traversing about the vertex clockwise (via cv) until reaching the lath L' where $\mathsf{cv}(L') = L$.

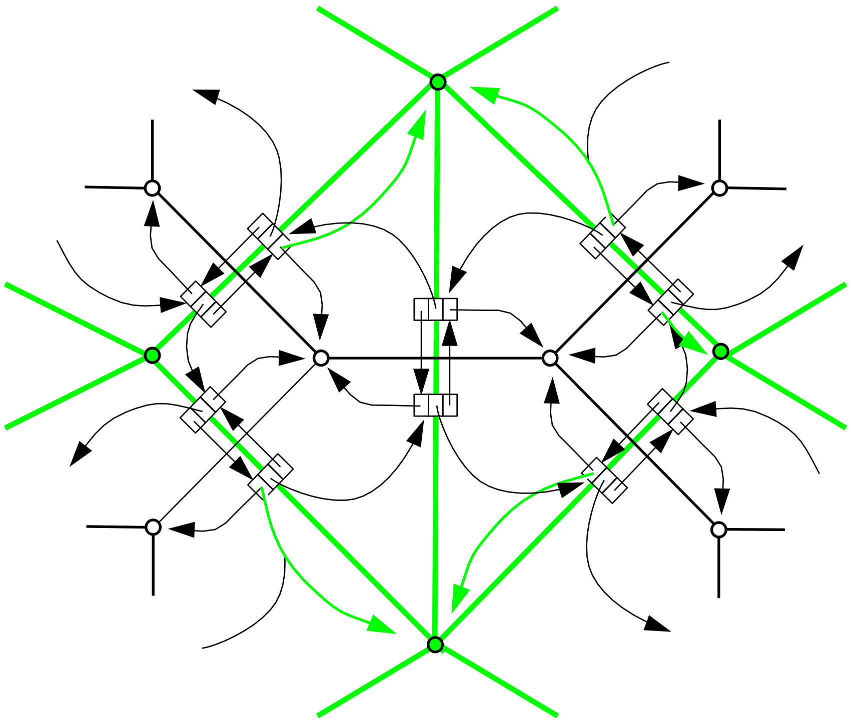
Fig. 16. Representing the dual of a mesh with a split-edge data structure. The original mesh and data structure are shown in black with the dual mesh shown in gray. The gray vertex links are the links to the face nodes in the dual structure. Note the counter-clockwise loops about the face and the clockwise loops about the vertex in the dual mesh.

Unlike the split-edge and half-edge operations, these operators can all be implemented in constant time except about the boundaries (see section 3.4 below). Along the boundary, cv is not available and in this case, ccf and ccv are linear in the number of edges in a face, or the number of edges radiating from a vertex, respectively.

Boundary Considerations Boundaries are represented in the corner structure by storing `Null` values in the *vertex_clockwise* links (see Figure 21). In this case, the cv operator may not be defined, and steps must be taken to use alternate operators to move about the mesh in the area of the boundary. However, the cf operator is always defined.

Data access primitives in the corner data structure The four basic queries are again $\mathcal{Q}_{\mathrm{EF}}$, $\mathcal{Q}_{\mathrm{EV}}$, $\mathcal{Q}_{\mathrm{FE}}$, and $\mathcal{Q}_{\mathrm{VE}}$, as they either return information stored explicitly in laths, or they return information stored in the two

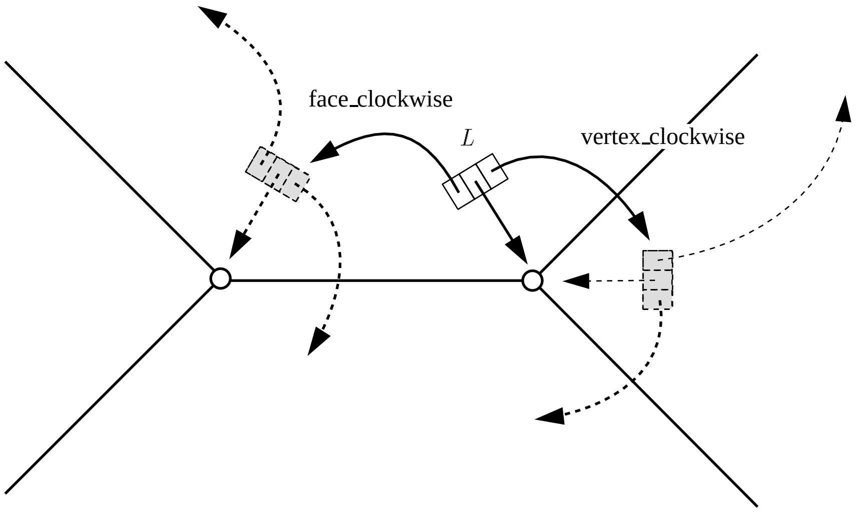

Fig. 17. The corner data element.

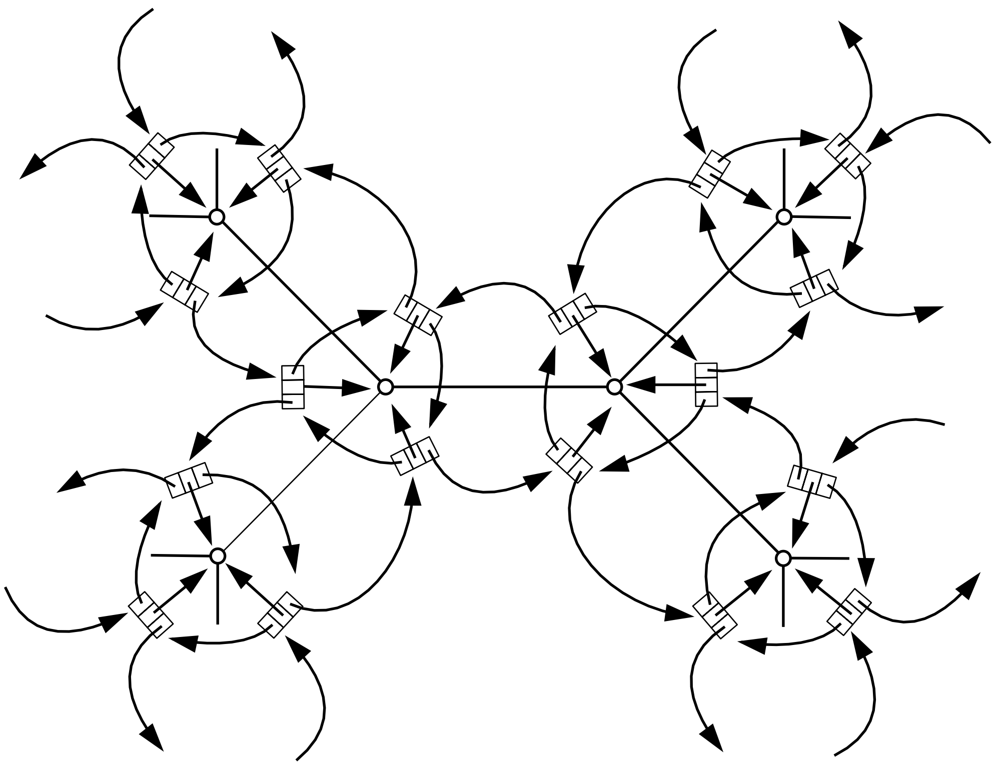

Fig. 18. The corner data element in a mesh.

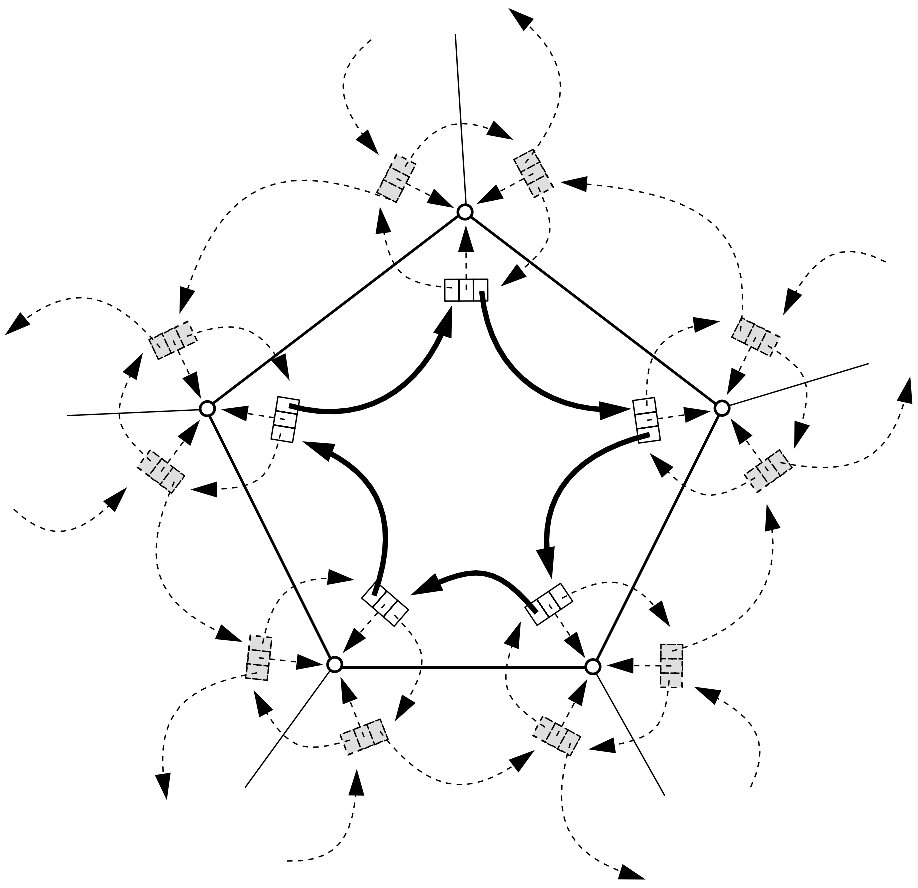

Fig. 19. A face loop in the corner representation. Starting with the lath L, and following *clockwise* links, a sequence of laths is generated that all reference the same face.

basic loops introduced by the data structure. The queries are basically the same here, except that we use cf to traverse the edges of the face, and cv to traverse the edges around the vertex. The boundaries are handled differently as only the cf operator is guaranteed to be defined, So, for a given lath L, we implement the queries as follows:

- $\mathcal{Q}_{\mathrm{EF}}$– Given a lath L representing an edge in the mesh, laths representing the two faces bounding the edge are given by L and ec(L). In the case the lath L represents a boundary edge, only L is returned, as ec(L) does not exist.
- $\mathcal{Q}_{\mathrm{EV}}$– Given a lath L representing an edge of the mesh, the two laths that represent the vertices that bound the edge are given by L and cf(L).

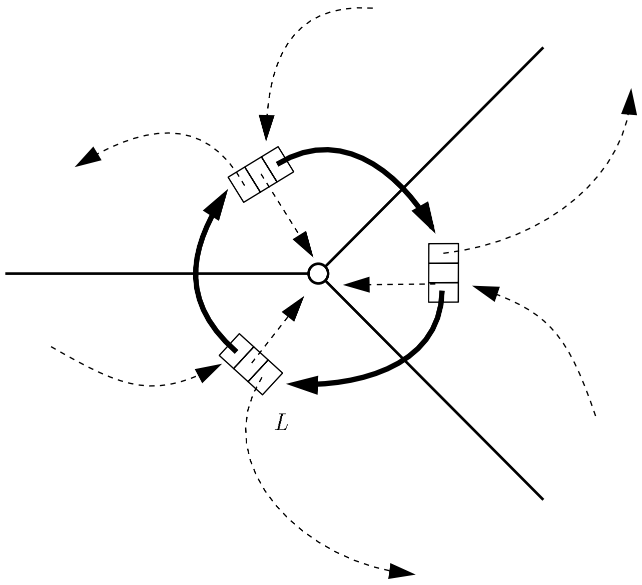

Fig. 20. A vertex loop in the corner representation. Starting with the lath L, and following *vertex_clockwise* links, a sequence of laths is generated that all reference the same vertex.

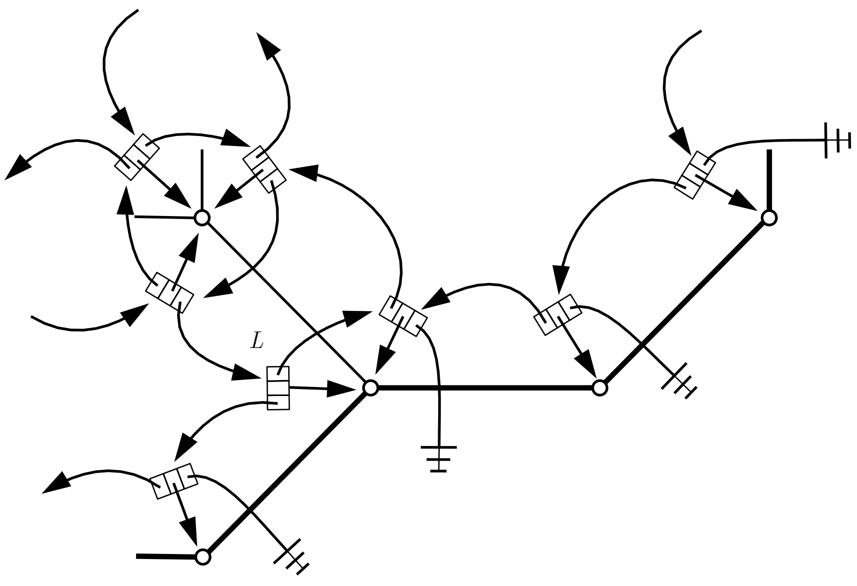

Fig. 21. The boundary of a mesh with an associated corner data structure. Here the *vertex_clockwise* links of the corner elements are `Null` on the boundary.

- $\mathcal{Q}_{\mathrm{FE}}$– Given a lath L representing a face-edge pair in the mesh, laths representing the edges of the face are given by successively using the cf operator – following the clockwise loop about the face.
- $\mathcal{Q}_{\mathrm{VE}}$– Given a lath L representing an edge-vertex pair, the laths representing edges that radiate from the vertex are obtained by successively using the cv operator – following the clockwise loop about the vertex.

The remaining queries are implemented in terms of these basic queries.

- $\mathcal{Q}_{\mathrm{FV}}$– Given a lath L representing a face, the laths representing the vertices of the face are the same as those of $\mathcal{Q}_{\mathrm{FE}}$.
- $\mathcal{Q}_{\mathrm{VV}}$– Given an lath L, we can obtain the required laths by taking cf(L) for each lath L in $\mathcal{Q}_{\mathrm{EV}}$.
- $\mathcal{Q}_{\mathrm{VF}}$is identical with $\mathcal{Q}_{\mathrm{VE}}$, as each lath identified by $\mathcal{Q}_{\mathrm{VE}}$belongs to a unique face.
- $\mathcal{Q}_{\mathrm{EE}}$– Given a lath L representing an edge E, $\mathcal{Q}_{\mathrm{EE}}$produces a set of laths that correspond to the edges that radiate from the two vertices of E. This is implemented by taking the union of the results of $\mathcal{Q}_{\mathrm{VE}}$for both L and cf(L). If ec(L) exists, we remove it from the list so that the original edge is only referenced by only one lath.
- $\mathcal{Q}_{\mathrm{FF}}$– Given a lath L representing a face-edge pair, all such faces are obtained by applying $\mathcal{Q}_{\mathrm{EV}}$to each lath in $\mathcal{Q}_{\mathrm{FE}}$, however this produces multiple laths for each face in the output. To obtain a unique lath for each face in the output, we use a marking strategy that marks each lath output by $\mathcal{Q}_{\mathrm{EV}}$. We only output those laths that are unmarked during the traversal.

It is possible that $\mathcal{Q}_{\mathrm{VE}}$(and $\mathcal{Q}_{\mathrm{VF}}$, which is identical in this structure) fails on the boundary. For example, given a lath L as in Figure 21, $\mathcal{Q}_{\mathrm{VE}}$(L) returns two laths before encountering a `Null` link on the boundary. To retrieve all laths that represent edges that radiate from the vertex, the query must use cv to traverse around the vertex in the opposite direction.

Similar to the split-edge representation, if we have n edges radiating from the vertex represented by L, only $n-1$ can be obtained through cv and ccv. In this case we can obtain a lath that represents the nth edge by using cf(L), but care must be taken with it's use as this lath does not point to the same vertex as L.

Duality If we form the dual of a mesh represented by a corner data structure (see Figure 22), we see that the dual mesh induced by the original mesh is also a corner data structure. The clockwise loop about a vertex transforms to a clockwise loop about a face in the dual mesh, and the clockwise loop about a face in the original mesh transforms to a clockwise loop about a vertex in the dual mesh. This makes the corner data structure quite useful for applications where finding and working with the dual mesh is necessary. Only one set of operators and queries need be implemented, and each works for both the original mesh and its dual.

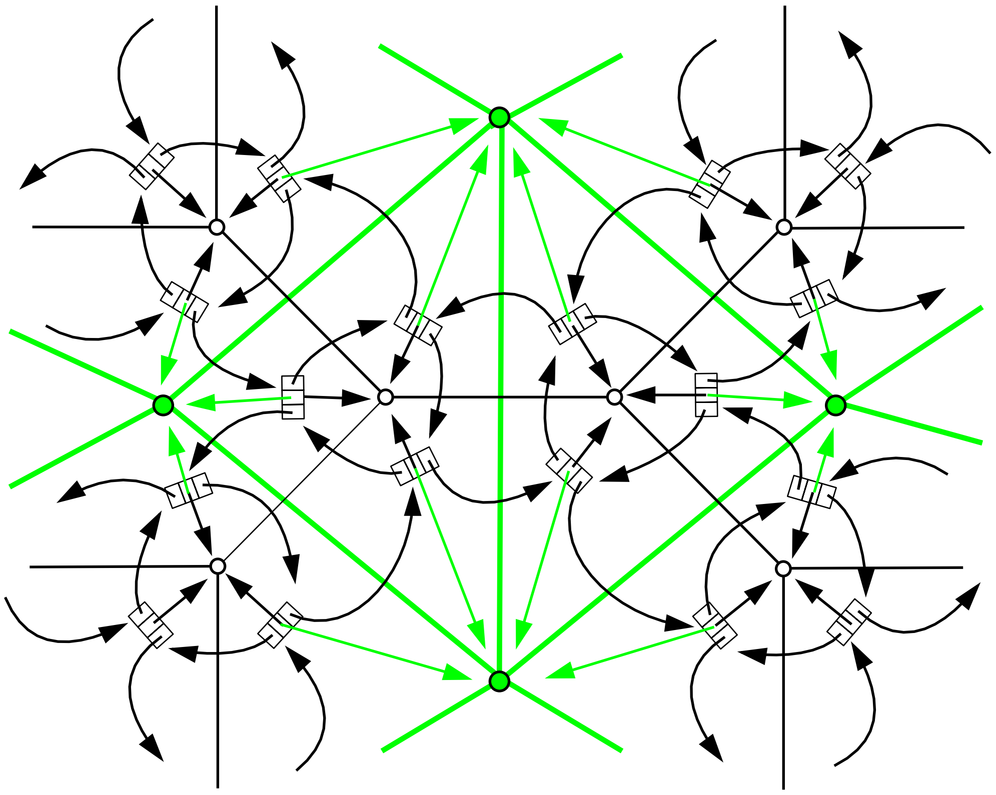

Fig. 22. Representing the dual of a mesh with a corner data structure. The original mesh and data structure are shown in black with the dual mesh shown in gray. The gray vertex links are the links to the face nodes in the dual structure.

4 Implementation

These data structures are straightforward to implement, as most of the operations are simple and only need to be modified on the boundaries. We have used them in a variety of applications and have found them to be most useful when "in-place" operations on large mesh structures are required. The corner structure is the most efficient for traversals as all basic operations can be implemented in constant time away from the boundary. It is also the most useful when dealing with the dual mesh, as it induces another corner data structure on the dual. The half-edge structure is most useful when frequent boundary operations are necessary. Here, the data structure is slightly larger, as not only are two laths kept for each interior edge of the structure, but two laths are kept for each edge on the boundary. This implies that only one of the basic operations is not available on the boundary and this this simplifies the operations around the boundary.

We should remark that there are other ways to connect laths. For example, we could specify a lath element that has a counter-clockwise face link and a counter-clockwise vertex link. Structurally this would be the same as the corner structure, just reversing the two loops. By using the five links (*edge_companion*, *face_clockwise*, *face_counter_clockwise*, *vertex_clockwise*, *vertex_counter_clockwise*), and taking two at a time, there are 10 possibilities of lath connection strategies available. As can be seen in Table 1, there is only one useful representation that we did not cover completely in this article (labeled corner∗ in the table). In this representation, where each lath has a *face_clockwise* link and a *vertex_counter_clockwise* link. The structure is very similar to the corner structure, however it does not have constant-time implementation of the basic operations, it has similar problems with the boundaries, and it induces a dual structure that has laths with *face_counter_clockwise* and *vertex_clockwise* links. We have not found an occasion where such a structure would be useful.

Lath Links	Data Structure Type	Induced Dual Structure
edge_companion *face_clockwise*	Split-Edge	Half-Edge
edge_companion *face_counter_clockwise*	Split-Edge (reverse orientation)	Half-Edge (reverse orientation)
edge_companion *vertex_clockwise*	Half-Edge	Split-Edge
edge_companion *vertex_counter_clockwise*	Half-Edge (reverse orientation)	Split-Edge (reverse orientation)
face_clockwise *face_counter_clockwise*	Not a useful structure	
face_clockwise *vertex_clockwise*	Corner	Corner
face_clockwise *vertex_counter_clockwise*	Corner∗	Corner∗ (reverse orientation)
face_counter_clockwise *vertex_clockwise*	Corner∗ (reverse orientation)	Corner∗
face_counter_clockwise *vertex_counter_clockwise*	Corner (reverse orientation)	Corner (reverse orientation)
vertex_clockwise *vertex_counter_clockwise*	Not a useful structure	

Table 1. Possible lath-based data structures.

5 Conclusion

We have presented three data structures for unstructured meshes all based upon a single data type – the lath. We have analyzed three connection strategies for the lath elements, creating a split-edge, half-edge and corner data structure, respectively. Each of these structures has several common operations that allow movement about the data, and nine basic queries that return vertices, edges and faces in the neighborhood of a given lath. The corner data structure is new and appears to be the most promising for future use in a variety of algorithms that support multiresolution representations of data.

This paper represents a "minimalist" approach to these data structures. Certainly we can add additional links for faces, or from vertices to laths, or from faces to laths. This complicates the data structure and utilizes much more storage per element, but can make for efficient operations and queries.

Future work should clearly focus on lath-based structures for three-dimensional meshes, and eventually, multi-dimensional meshes. Each lath in these structures should represent an edge-face-cell, vertex-edge-cell, or vertex-face-cell triple, and each lath should be identified with exactly one vertex, edge, face and cell in the mesh.

6 Acknowledgments

This work was supported by the National Science Foundation through the Large Scientific and Software Data Set Visualization (LSSDSV) program under contract ACI 9982251, and through the National Partnership for Advanced Computational Infrastructure (NPACI); the Office of Naval Research under contract N00014-97-1-0222; the Army Research Office under contract ARO 36598-MA-RIP; the NASA Ames Research Center through an NRA award under contract NAG2-1216; the Lawrence Livermore National Laboratory under ASCI ASAP Level-2 Memorandum Agreement B347878 and under Memorandum Agreement B503159; the Lawrence Berkeley National Laboratory; the Los Alamos National Laboratory; and the North Atlantic Treaty Organization (NATO) under contract CRG.971628. We also acknowledge the support of ALSTOM Schilling Robotics and SGI. We thank the members of the Visualization and Graphics Research Group at the Center for Image Processing and Integrated Computing (CIPIC) at the University of California, Davis.

References

1. Ala, S. R. Design methodology of boundary data structures. *Internat. J. Comput. Geom. Appl. 1*, 3 (1991), 207–226.
2. Baumgart, B. G. Geometric modeling for computer vision. AIM-249, STA -CS-74-463, CS Dept, Stanford U., Oct. 1974.

3. Bertram, M. *Multiresolution Modeling for Scientific Visualization.* PhD thesis, University of California, Davis, 2000.
4. Bertram, M., Duchaineau, M. A., Hamann, B., and Joy, K. I. Bicubic subdivision-surface wavelets for large-scale isosurface representation and visualization. *Proceedings of IEEE Visualization 2000* (Oct. 2000), 389–396.
5. Catmull, E., and Clark, J. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-Aided Design 10* (Sept. 1978), 350–355.
6. De Floriani, L., and Puppo, E. Hierarchical triangulation for multiresolution surface description geometric design. *ACM Transactions on Graphics 14*, 4 (Oct. 1995), 363–411.
7. Doo, D. A subdivision algorithm for smoothing down irregularly shaped polyhedrons. In *Proced. Int'l Conf. Ineractive Techniques in Computer Aided Design* (1978), pp. 157–165. Bologna, Italy, IEEE Computer Soc.
8. Doo, D., and Sabin, M. Behaviour of recursive division surfaces near extraordinary points. *Computer-Aided Design 10* (Sept. 1978), 356–360.
9. Eastman, C. M. Introduction to Computer Aided Design, Course Notes, Carnegie-Mellon University, Pittsburg, PA, 1982.
10. Garland, M., and Heckbert, P. Surface simplification using quadric error metrics. *Proceedings of SIGGRAPH'97* (1997), 209–215.
11. Gieng, T., Joy, K. I., Hamann, B., Schussman, G., and Trotts, I. Smooth hierarchical surface triangulations. In *Proceedings of Visualization '97* (Oct. 1997), H. Hagen and R. Yagel, Eds., IEEE Computer Society, pp. 379–386.
12. Gieng, T. S., Hamann, B., Joy, K. I., Schussman, G. L., and Trotts, I. J. Constructing hierarchies for triangle meshes. *IEEE Transactions on Visualization and Computer Graphics 4*, 2 (Apr. 1998), 145–161.
13. Hall, M., and Warren, J. Adaptive polygonalization of implicitly defined surfaces. *IEEE Computer Graphics and Applications 10*, 6 (Nov. 1990), 33–42.
14. Hoppe, H. Progressive meshes. In *SIGGRAPH 96 Conference Proceedings* (Aug. 1996), H. Rushmeier, Ed., Annual Conference Series, ACM SIGGRAPH, Addison Wesley, pp. 99–108. held in New Orleans, Louisiana, 04-09 August 1996.
15. Kalay, Y. E. *Modeling Objects and Environments.* John Wiley and Sons, 1989.
16. Mantyla, M. *An Introduction to Solid Modeling.* Computer Science Press, Rockville, Md, 1988.
17. Renze, K. J., and Oliver, J. H. Generalized surface and volume decimation for unstructured tessellated domains. In *VRAIS '96 (IEEE Virtual Reality Annual Intl. Symp.)* (Mar. 1996), pp. 24–32.
18. Schussman, S., Bertram, M., Hamann, B., and Joy, K. I. Hierarchical data representations based on planar voronoi diagrams. In *Proceedings of the Joint Eurographics and IEEE TVCG Conference on Visualization* (2000), R. van Liere, I. Hermann, and W. Ribarsky, Eds., pp. 63–72.
19. Thompson, J. F., and Hamann, B. A survey of grid generation techniques and systems with emphasis on recent developments. *Surveys on Mathematics for Industry 6* (1997), 289–310.
20. Weiler, K. J. *Topological structures for geometric modeling.* Ph.d. thesis, Rensselaer Polytechnic Institute, Aug. 1986.

21. Woo, T. C., and Wolter, J. D. A constant expected time, linear storage data structure for representing three-dimensional objects. *IEEE Transactions on Systems, Man and Cybernetics SMC-14* (May 1984), 510–515.
22. Xia, J. C., and Varshney, A. Dynamic view-dependent simplification for polygonal models. In *IEEE Visualization '96* (Oct. 1996), IEEE. ISBN 0-89791-864-9.