Vertex-Based (Lath) Representations for Three-Dimensional Objects and Meshes

Hanan Samet

hjs@cs.umd.edu www.cs.umd.edu/~hjs

Department of Computer Science

Center for Automation Research

Institute for Advanced Computer Studies

University of Maryland

College Park, MD 20742, USA

Vertex-based Data Structures

1. Edge-based winged-edge family are uniform-size
   - one record per edge
2. Want uniform-size vertex-based or face-based representations
   - vertex is simplest topological entity while edge and face are more complex
     - cannot have one record per vertex
     - each edge is always associated with just two faces (assuming a two-manifold) and with just two vertices
     - variable number of edges and faces associated with each vertex
   - cannot have one record per face
     - variable number of edges and vertices associated with each face
3. Object can be described by set of all possible edge-face, edge-vertex pairs, or face-vertex pairs
   - each pair is termed a *lath* (Joy, Legakis, and MacCracken)
   - one vertex is associated with each lath
   - more than one lath can be associated with a particular vertex
   - a single edge is associated with each lath
   - a single face is associated with each lath

Lath Data Structures for Manifold Objects

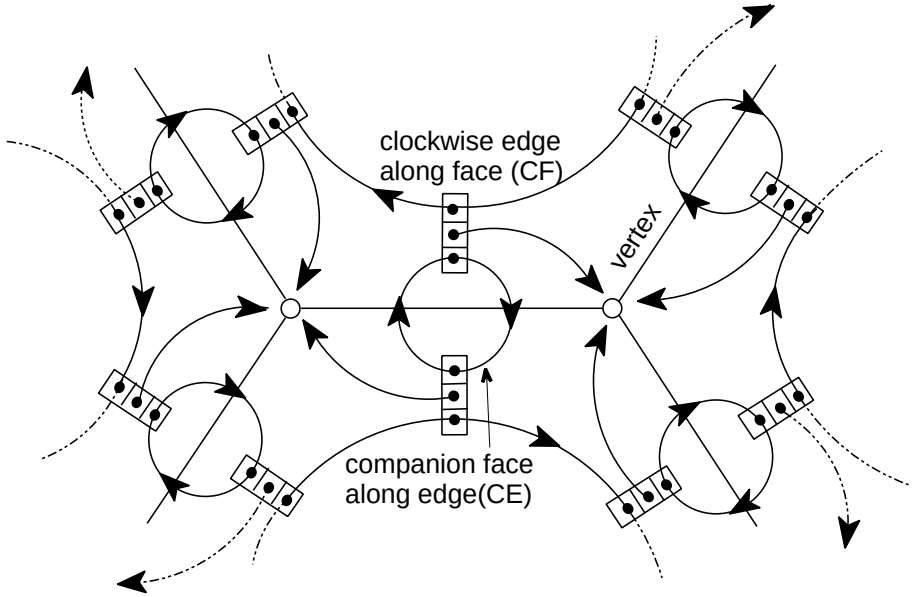
1. Encodes lath-lath relation rather than edge-edge, edge-face, etc.
2. Must be able to make transitions between instances of lath data structure
3. Given specific lath instance (x, y) of relation $(a, b) - (a, b)$, we need to be able to transition to
   - lath corresponding to the next object of type b for object x, and to the
   - lath corresponding to the next object of type a for object y
4. Three items of information
   - associated vertex
   - transitions to next (c=clockwise) or prior (cc=counterclockwise) objects
   - if edge object, then just one transition to companion

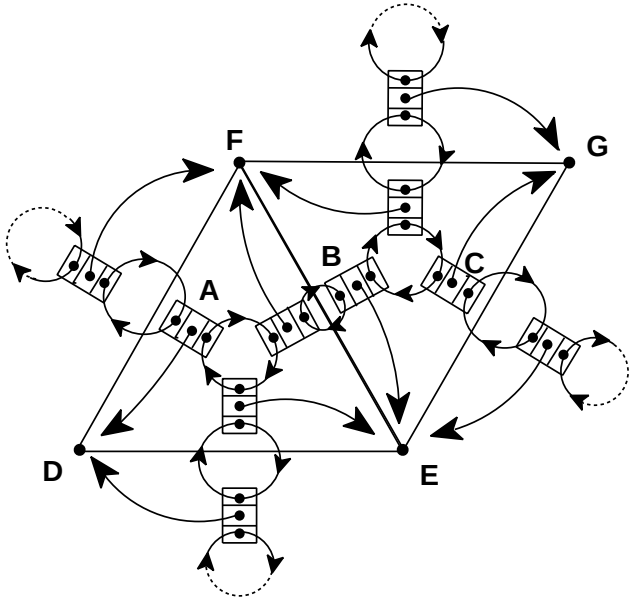
Nature of Lath Data Structure

1. Implicit data structure in contrast to winged-edge
2. Identity of faces adjacent to an edge as well as one of the vertices that comprises an edge are represented implicitly
3. Only vertex associated with each lath is represented explicitly
4. Need vertex-lath, face-lath, and edge-lath tables
   - lath analogs of vertex-edge and face-edge tables

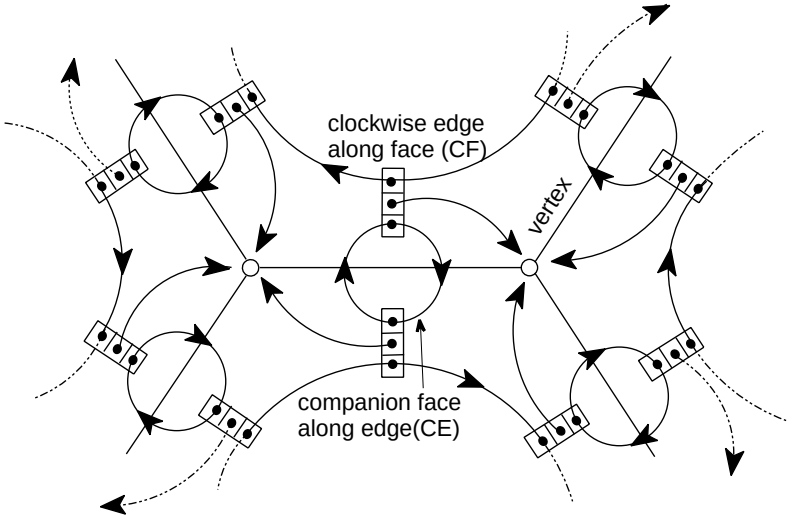
Split-Face Lath Data Structure: Edge-Face Pairs

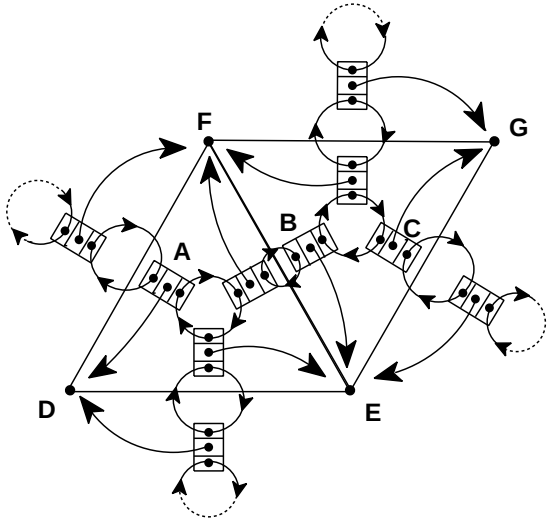
- Split an edge record into two: one per adjacent face
- Record structure $L = (e, f)$:
  1. Pointer to the vertex v associated with L
  2. Pointer to the lath record corresponding to the other face f adjacent to e as well as opposite vertex of edge e to the one associated with L (i.e., the next edge-face pair along e) — that is, $\mathsf{CE}(L)$
  3. Pointer to the lath record corresponding to the next edge that follows L in a clockwise traversal of face f (i.e., the next edge-face pair along f) — that is, $\mathsf{CF}(L)$





Non-primitive Split-Face Operations

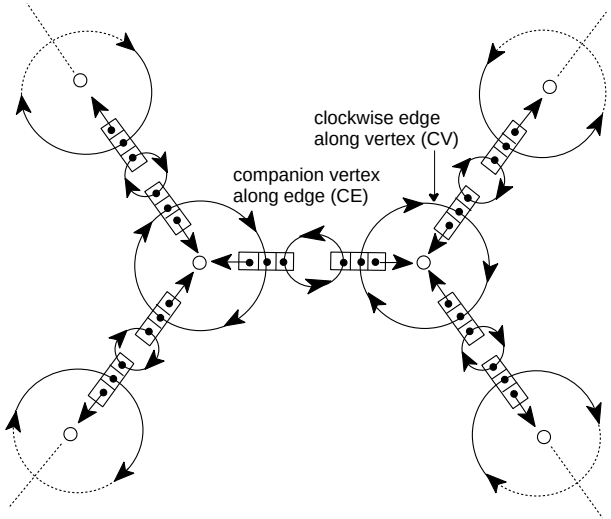


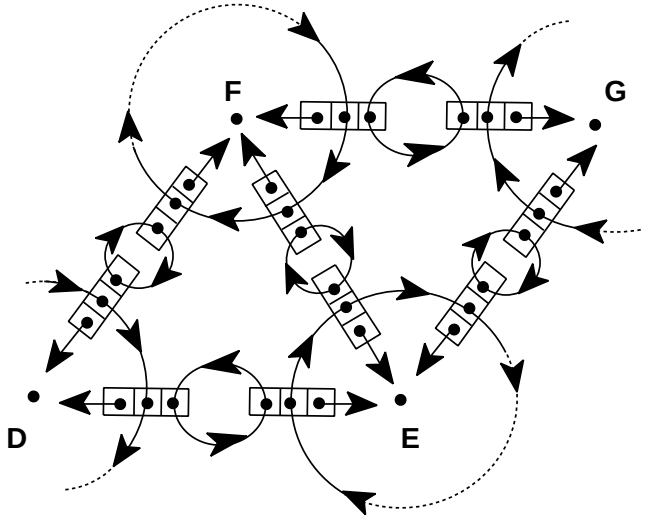


- $\mathsf{CCV}(L)$: $\mathsf{CF}(\mathsf{CE}(L))$
- $\mathsf{CV}(L)$: two possibilities (neither of which is great!)
  1. successively traverse laths representing face f using CF until obtaining a lath L' such that $\mathsf{CF}(L') = L$ and then apply $\mathsf{CE}(L')$ to obtain L'' so that $\mathsf{CV}(L) = L''$, OR
  2. successively traverse laths surrounding vertex associated with L using CCV until obtaining a lath L' such that $\mathsf{CCV}(L') = L$, which means that $\mathsf{CV}(L) = L'$.
- $\mathsf{CCF}(L)$: $\mathsf{CE}(\mathsf{CV}(L))$

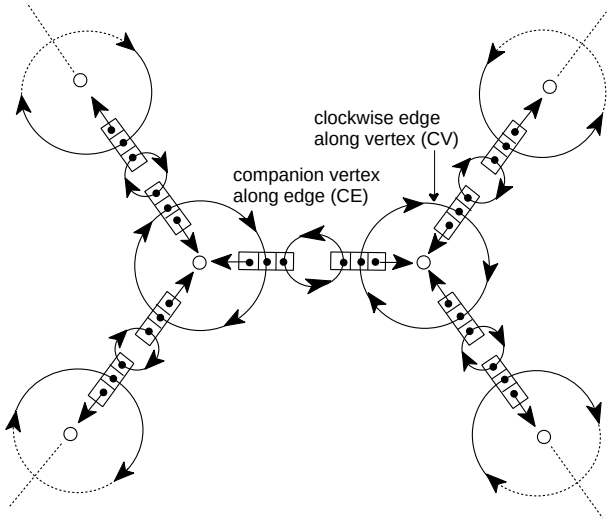
Split-Vertex Lath Data Structure: Edge-Vertex Pairs

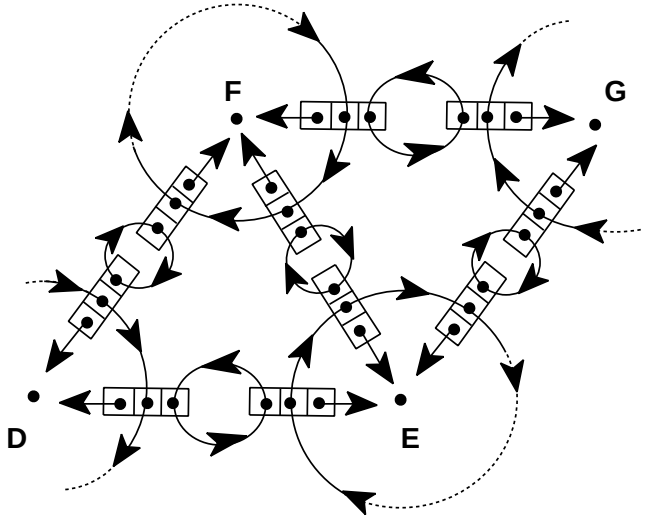
- Split an edge record into two: one per incident vertex
- Record structure $L = (e, v)$:
  1. Pointer to the vertex v associated with L.
  2. Pointer to the lath record that represents the same edge e as L but the opposite vertex of e (i.e., the next edge-vertex pair along e) — that is, $\mathsf{CE}(L)$
  3. Pointer to the lath record corresponding to the next edge that follows L in a clockwise traversal of vertex v (i.e., the next edge-vertex pair along v) — that is, $\mathsf{CV}(L)$





Non-primitive Split-Vertex Operations

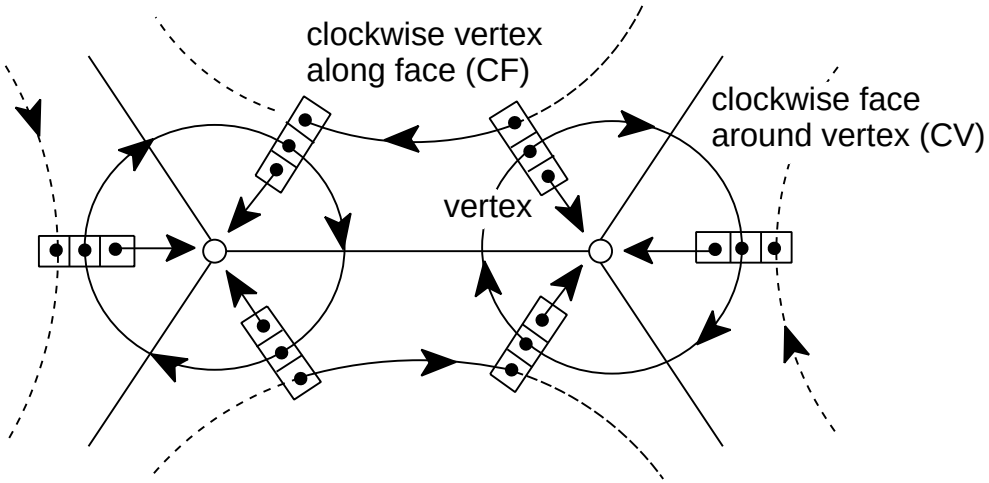


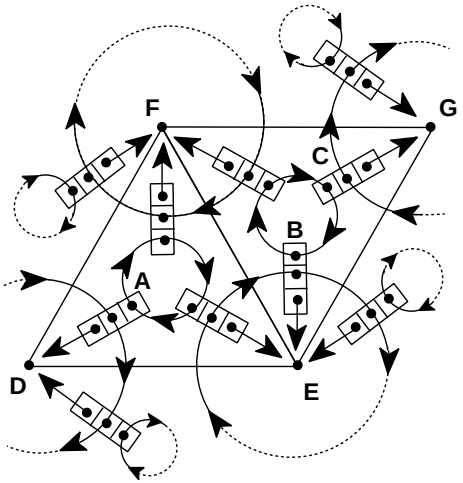


- $\mathsf{CCF}(L)$: $\mathsf{CE}(\mathsf{CV}(L))$
- $\mathsf{CF}(L)$: two possibilities (neither of which is great!)
  1. first apply $\mathsf{CE}(L)$ to obtain the lath L' of the same edge e but opposite vertex v', and then successively traverse laths surrounding v' using CV until obtaining a lath L'' such that $\mathsf{CV}(L'') = L'$ which means that $\mathsf{CF}(L) = L''$, or
  2. successively traverse laths representing face f in which L is a member using CCF until obtaining a lath L' such that $\mathsf{CCF}(L') = L$, which means that $\mathsf{CF}(L) = L'$
- $\mathsf{CCV}(L)$: $\mathsf{CF}(\mathsf{CE}(L))$

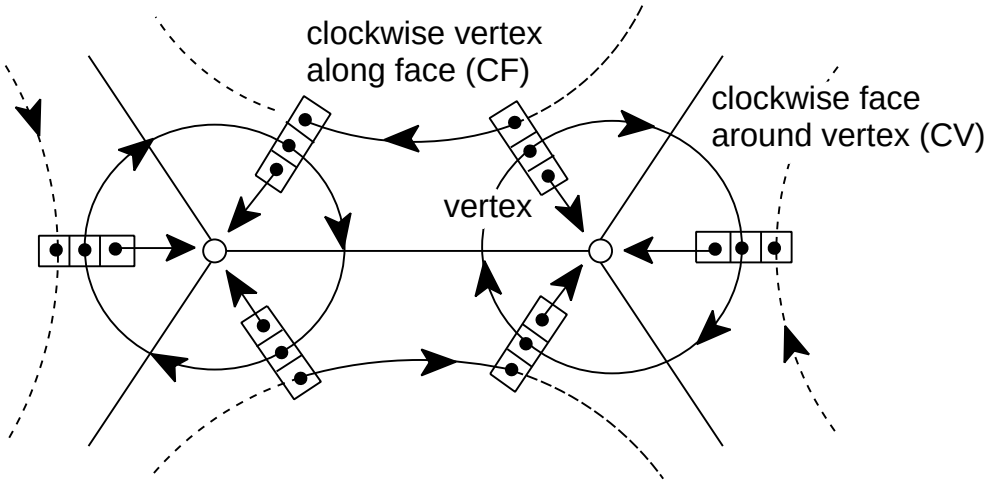
Corner Lath Data Structure: Face-Vertex Pairs

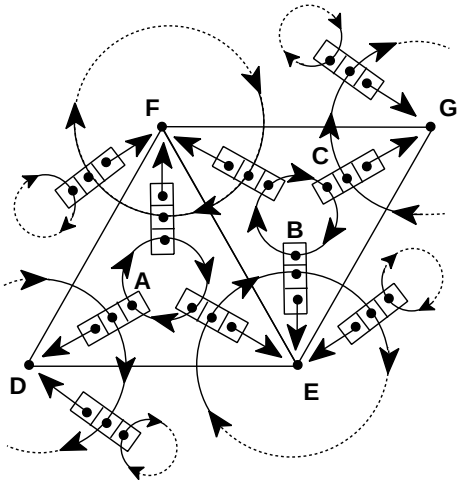
- Denotes corner of a face
- Record structure $L = (f, v)$:
  1. Pointer to the vertex v associated with L
  2. Pointer to the lath record corresponding to the next vertex that follows L in a clockwise traversal of face f (i.e., the next face-vertex pair along f) — that is, $\text{CF}(L)$
  3. Pointer to the lath record corresponding to the next face that follows L in a clockwise traversal of vertex v (i.e., the next face-vertex pair along v) — that is, $\text{CV}(L)$





Non-primitive Corner Operations



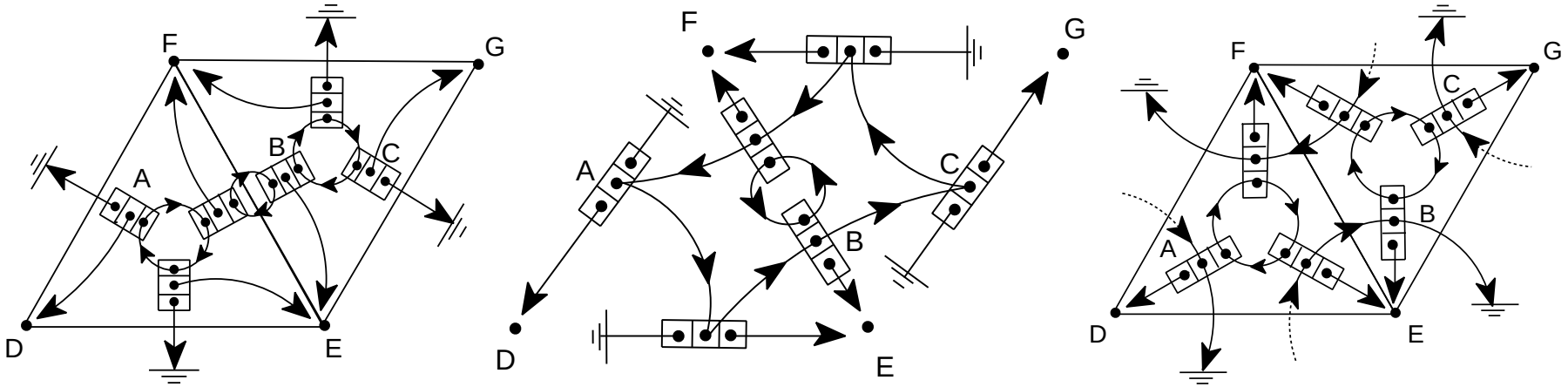


- CE(L): CV(CF(L))
- CCF(L): CE(CV(L))
- CCV(L): CF(CE(L))

Lath Data Structures for Meshes with Boundaries

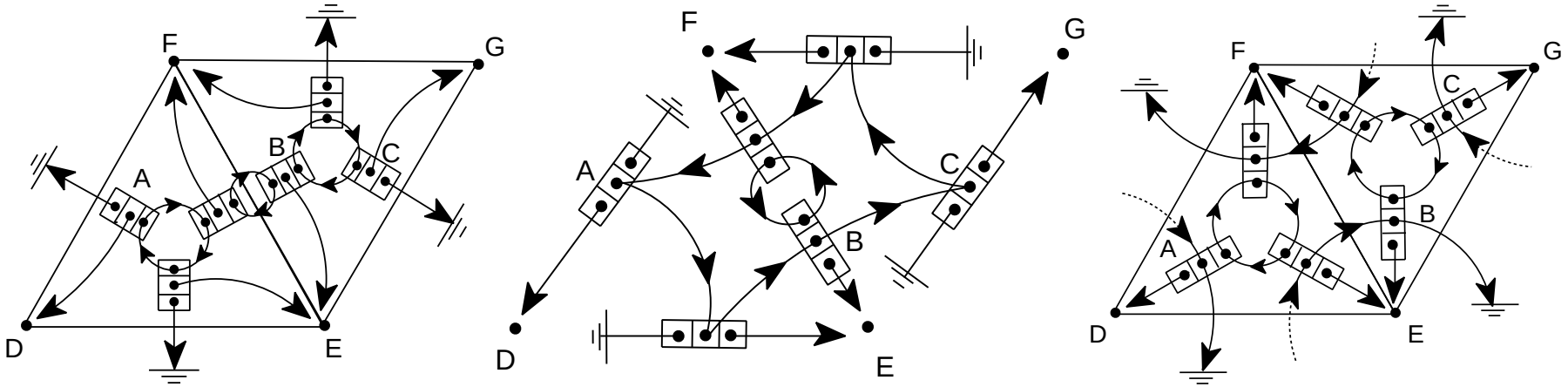
- Unstructured two-dimensional meshes
- There exists a face which is not part of the object (termed the *boundary face*
- Two possible methods
  1. add a flag to each lath record indicating whether the associated vertex and edge combination is part of the boundary of a face in the mesh
  2. overload of the primitive operations (transitions to L') on the lath L
     - to use the value `NULL` to indicate that L is part of the boundary (i.e., the vertex and edge combination associated with L corresponds to a boundary edge whose associated face f is not the boundary face and hence f is in the mesh)
     - while the face f' associated with the next lath L' is the boundary face and hence f' is not in the mesh

Implementation of Meshes with Boundaries



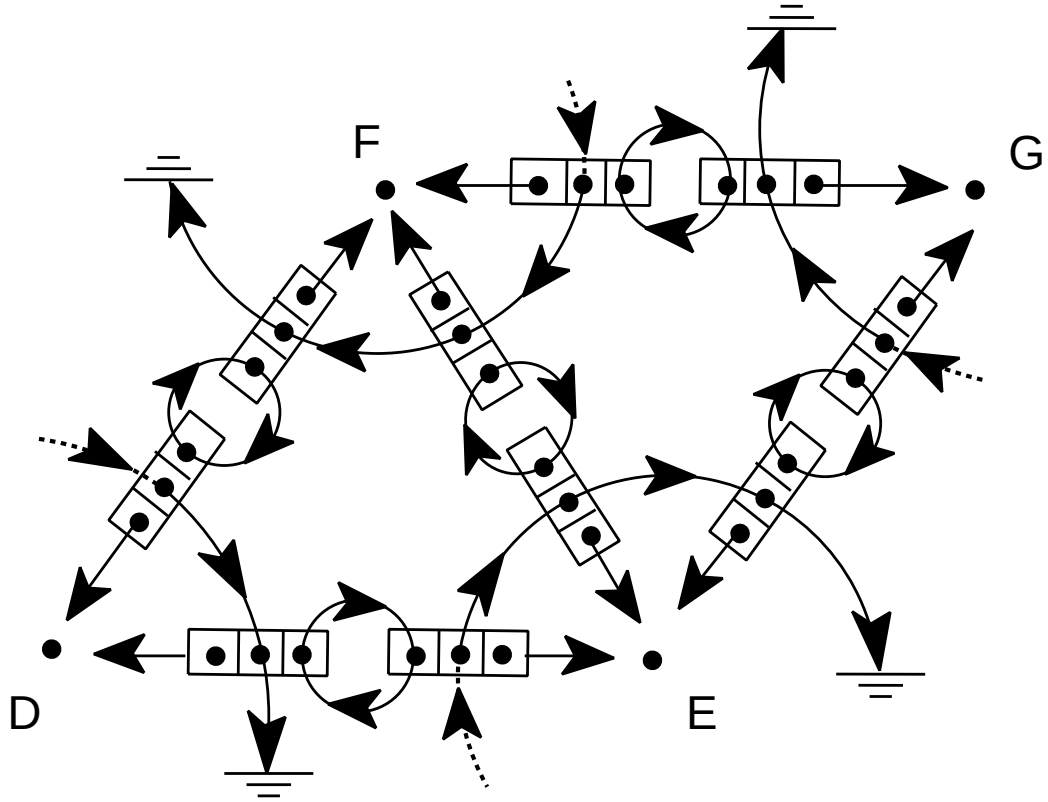
- Distinguish boundary edges from nonboundary edges by associating just one lath with them instead of two — that is, $\mathrm{CE}(L)$ is absent
  1. split-face: $\mathrm{CE}(L)$=NULL
  2. split-vertex: $\mathrm{CE}(L)$=NULL
     - $\mathrm{CV}(L'')=L$ instead of $\mathrm{CV}(L'')=L'$ due to $L'=\mathrm{CE}(L)$=NULL
  3. corner: $\mathrm{CV}(L)$=NULL as $\mathrm{CE}(L)=\mathrm{CV}(\mathrm{CF}(L))$
- Must use definitions of nonprimitive transitions that do not pass through missing laths

Problems Caused by Absence of Companion Lath



- Cannot return laths that correspond to an edge of the boundary face (i.e., a companion lath of a boundary edge) as a valid value of a transition
- Occurs when companion lath is to be returned as value of $\mathrm{CV}(L)$ which is NULL, for a lath L associated with vertex v regardless of whether L corresponds to a boundary edge
- Ex: laths A and B when want laths of all edges incident at vertices D and E, respectively
  - could use lath corresponding to $\mathrm{CCF}(L)$ (i.e., lath C for next lath after B incident at vertex E) but its associated vertex is different (i.e., G)

Alternative Implementation of Split-Vertex for Boundary Mesh



- Retain companion lath (Joy, Legakis, and MacCracken)
- Less modifications to transitions -
  - No need to set CE(L) to NULL
- But now need a different algorithm to trace boundary of the mesh than one used for split-face and corner lath data structures

Summary

- Corner lath data structure is best
  - self-dual
  - similar to quad-edge data structure
  - all primitive and non-primitive transitions in constant time as long as no boundaries
- Can also form a face-based lath representation
  - difference from vertex-based is that each lath contains a pointer to the face associated with the lath instead of the vertex associated with it
  - same lath types
  - cumbersome to identify vertices that make up a face as need to retrieve all faces incident at each vertex