Quad-Edge Data Structure and Library

Computer Graphics 2, 15-463
Paul Heckbert
Carnegie Mellon University

Revision 1: 14 Feb. 2001.

Overview

The Quad-Edge data structure is useful for describing the topology and geometry of polyhedra. We will use it when implementing subdivision surfaces (a recent, elegant way to define curved surfaces) because it is elegant and it can answer adjacency queries efficiently. In this document we describe the data structure and a C++ implementation of it.

This document has the following sections:

Motivation for the Quad-Edge Data Structure

We'd like to be able to create a variety of polyhedra. Some simple and highly summetric polyhedra are the five Platonic Solids (tetrahedron, octahedron, cube, dodecahedron, icosahedron), or more generally the Archimedean Solids (cuboctahedron, soccerball, ...), or even more generally the **Uniform Polyhedra**. Follow that link **now** to see a very nice collection of pictures of polyhedra. Click on the link "visual index" there, then click on a polyhedron, then click on its picture and after a second you'll see the object spin (animated GIF)! See also the very cool collection of links on Polyhedra and Polytopes that is part of David Eppstein's nice web collection, The Geometry Junkyard (from which I got most of the preceding links). Some of my own creations are shown on his zonohedron page.

All of these can be created by doing truncations (cutting off each vertex), stellations (building a prism on each face), or other simple operations on the Platonic solids. It would be interesting to write code to generate such polyhedra, possibly even to animate transformations (3-D morphs) between them, but what data structure should we use for this?

First Try: list of polygons. The first, and simplest data structure we might think of is a simple list of polygons, each one storing (redundantly) all of its vertex coordinates. That is, in C++:

```
struct Vert {double x, y, z;}; // vertex position
Vert tri[NFACES][3];  // array of triangles, each with 3 vertices
```

With this data structure, the vertices of face f would be at the xyz points `tri[f][i]` for i=0,1,2. The above scheme is for triangulated models, where each face (polygon) has three sides, but it could obviously be generalized for models with n-sided faces. With this data structure, it would be very clumsy to try to do an operation like vertex truncation, where you need to find all the vertices adjacent (connected by an edge) to a given vertex. To do that you'd need to search through the face list for other vertices with equal coordinates -- inefficient, and inelegant.

Second Try: vertex and face lists. A better alternative would be to store the vertices separately, and make the faces be pointers to the vertices:

```
Vert vert[NVERTS];  // array of vertices
struct Tri {Vert *p, *q, *r;}; // triangle holds 3 vertex pointers
Tri tri[NFACES];  // array of triangular faces
```

Again, this is the code for triangular faces only. This second method reduces redundancy, but finding the vertices adjacent to a given vertex would still be costly (O(NFACES)), as you'd have to search the entire face list. The above two data structures record the geometric information (vertex positions) just fine, but they are lacking in topological information that records connectedness (adjacencies) between vertices, edges, and faces. The first data structure stored no topological information, the second stored only pointers from faces to vertices.

We can do better. To do so we'll need to store even more topological information, so that we can find the vertices/edges/faces immediately adjacent to a given vertex/edge/face in constant time.

The Quad-Edge Data Structure

A particularly elegant data structure for polyhedra is the quad-edge data structure, invented by Guibas and Stolfi. It can't represent all collections of polygons; it is limited to manifolds (surfaces where the neighborhood of each point is topologically equivalent to a disk; edges are always shared by two faces).

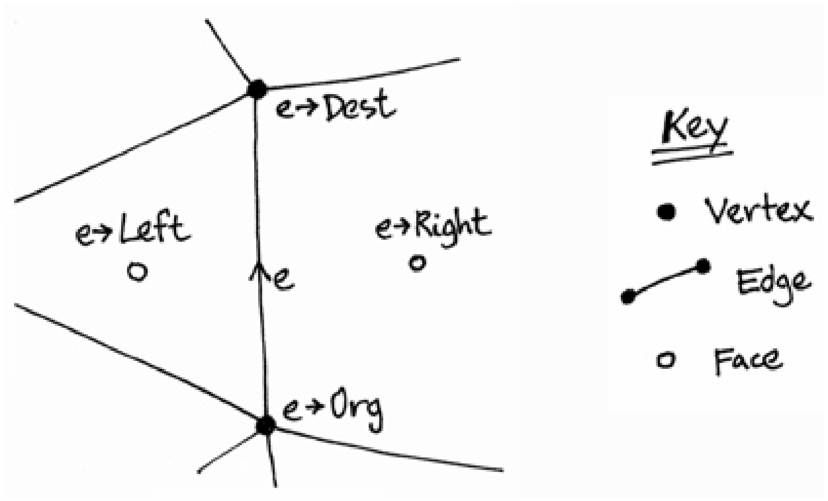
In the quad-edge data structure, there are classes for vertices, edges, and faces, but edges play the leading role. The edges store complete topological information; all of the topological information stored by the faces and vertices is redundant with information in the edges. Figuratively speaking, the edges form the skeleton, and the vertices and faces are optional decorations, hanging off of the edges. Vertices hold most of the geometric (shape) information.

We now describe our implementation of quad-edge. We emphasize the high level public routines that you are encouraged to use. The full details are in the code in the cell directory.

How to Use our Quad-Edge Library

Edge

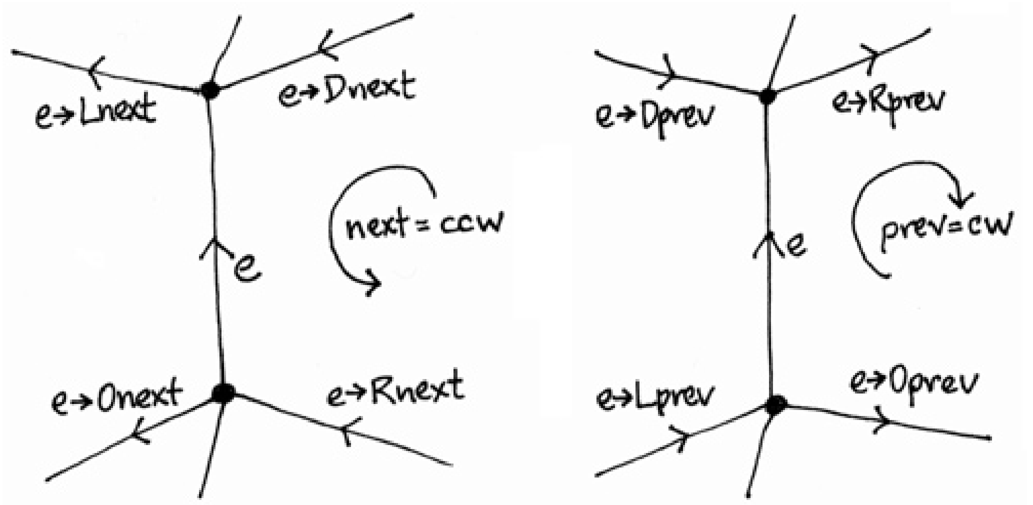
The class `Edge` represents a directed edge. Given `Edge *e`, you can find the immediately adjacent vertices, faces, and edges, and the "symmetric" edge that points in the opposite direction. These operators are all fast (just a few memory references). Because edges are directed and we always imagine ourselves viewing the object from the outside, we can speak of the origin and destination vertices and left and right faces of an edge. We summarize the interface below (see cell/edge.hh, for the full story).



- `Edge *e` - directed edge

- `Vertex *e->Org()` - vertex at the origin of e
- `Vertex *e->Dest()` - vertex at the destination of e

- `Face *e->Left()` - face on the left of e
- `Face *e->Right()` - face on the right of e



In the following functions, "next" means next in a counterclockwise (ccw) sense around a neighboring face or vertex.

- `Edge *e->Rnext()` - next edge around right face, with same right face
- `Edge *e->Lnext()` - next edge around left face, with same left face
- `Edge *e->Onext()` - next edge around origin, with same origin
- `Edge *e->Dnext()` - next edge around dest, with same dest

In the following functions, "prev" means next in a clockwise (cw) sense around a neighboring face or vertex.

- `Edge *e->Rprev()` - prev edge around right face, with same right face
- `Edge *e->Lprev()` - prev edge around left face, with same left face
- `Edge *e->Oprev()` - prev edge around origin, with same origin
- `Edge *e->Dprev()` - prev edge around dest, with same dest

It is recommended that you take a moment to verify for yourself in the figure how Lnext and Rnext use rotation about a face, while Onext and Dnext use rotation about a vertex.

The following member function returns a unique integer ID for each edge.

- `unsigned int e->getID()` - id# of this edge

When debugging, you might want to print id numbers.

Using `Lnext`, we could loop around the edges of the face on the left of edge `estart` in ccw order:

```
void leftFromEdge(Edge *estart) {
    Edge *e = estart;
    do {
```

```
 <do something with edge e>
 e = e->Lnext();
    } while (e!=estart);
}
```

Note that we need "do ... while" as opposed to "for" or "while" because (a) we don't know beforehand how many edges the face has (quad-edge is not limited to triangulated surfaces), (b) following the `Lnext`'s yields a cycle, and (c) we want to visit each edge exactly once. If `Lprev` were used instead of `Lnext` we'd visit the left face's edges in cw order.

The number of edges of a face (the face *degree* or valence) is 3 or greater for "real" polyhedra, but sometimes during construction of data structures, it is useful to have faces with 1 or 2 edges, which would correspond geometrically to loops or degenerate sliver polygons.

Similarly, the edges around the origin vertex of edge `estart` can be visited in ccw order like so:

```
void orgFromEdge(Edge *estart) {
    Edge *e = estart;
    do {
 <do something with edge e>
 e = e->Onext();
    } while (e!=estart);
}
```

The degree of a vertex is 3 or greater for "real" polyhedra, but as with faces, during construction we sometimes build vertices of degree 1 or 2.

Since visiting the edges around a face (or edges around a vertex) is quite common, we have set up some iterator classes to simplify your code. Using the iterator, an alternative to `leftFromEdge` is:

```
void edgesOfFace(Face *face) {
    // visit edges of face in ccw order; edges have face on the left
    FaceEdgeIterator faceEdges(face);
    Edge *edge;
    while ((edge = faceEdges.next()) != 0)
 <do something with edge e>
}
```

Note that this functions a bit differently from `leftFromEdge`; its input is a face pointer, not an edge pointer, so you don't have control over which of the face's edges will get visited first, but that usually doesn't matter, anyway. Once an iterator has gone through its list, it is "spent". If you want to go through the list again, you must construct a new iterator.

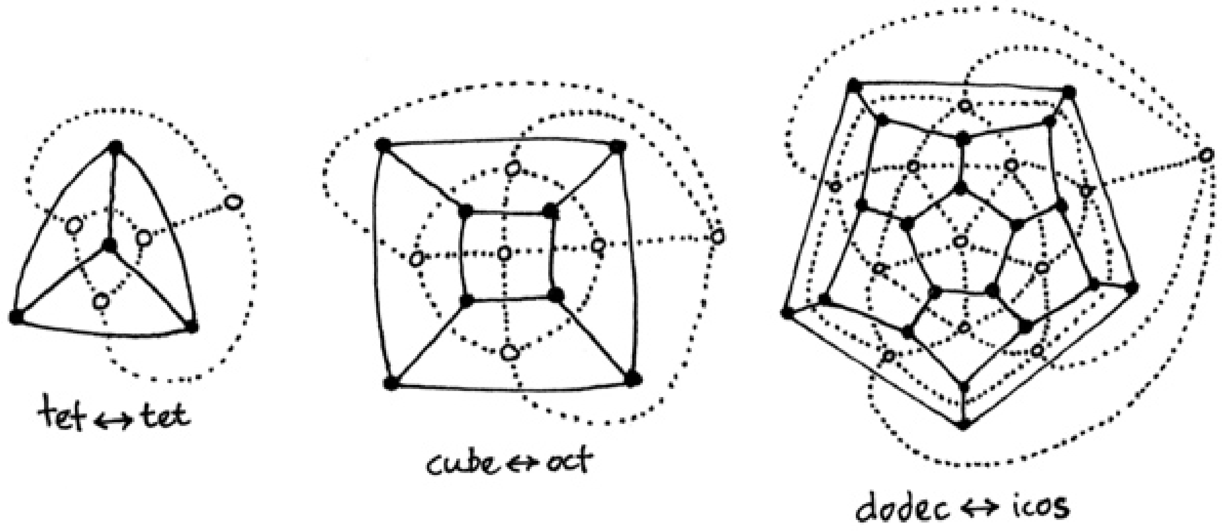
Using an iterator, the alternative to `orgFromEdge` is:

```
void edgesOfVertex(Vertex *vert) {
    // visit edges of vertex in ccw order; edges have vert as origin
    VertexEdgeIterator vertEdges(vert);
    Edge *edge;
```

```
        while ((edge = vertEdges.next()) != 0)
      <do something with edge e>
    }
```

Duality

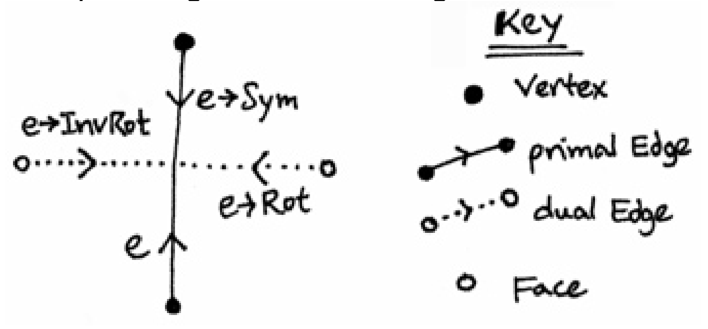
You may have noticed how similarly vertices and faces are treated in the above. That is no accident. Guibas and Stolfi designed the quad-edge data



structure with that duality in mind.

The *dual* of a polyhedron is the polyhedron resulting from rotating edges 90 degrees, replacing vertices with faces, and faces with vertices. The new vertex locations can be taken to be the centroids of the old faces. For example, the dual of a cube is an octahedron, and vice versa; and dodecahedra and icosahedra are duals of each other, also. A tetrahedron is dual with a rotated copy of itself.

The quad-edge data structure gets its name because the duality is built in at a low level by storing quadruples of directed edges together:



- `Edge *e` - directed edge
- `Edge *e->Sym()` - edge pointing opposite to e
- `Edge *e->Rot()` - dual edge pointing to the left of e
- `Edge *e->InvRot()` - dual edge pointing to the right of e

You will probably need `Sym`, but perhaps not `Rot` or `InvRot`. Internally, our implementation stores only four essential pieces of information with each edge (origin vertex, left face, Onext, and quadedge index) and the rest of the adjacency operators (`Dest`, `Right`, `Lnext`, `Rprev`, `Dnext`, ...) are derived using `Sym` and `Rot`. Duals are also extremely useful for Voronoi diagrams and Delaunay triangulation, but that's another course (computational geometry).

Vertex

The information stored at a vertex consists of one piece of topological information (a pointer to one of the outgoing edges of the vertex), plus geometric information (the (x,y,z) position), and optional attribute information (color, normals, etc). The public interface you should use is summarized below (see cell/vertex.hh for the full code).

```
class Vertex {
    Vec3 pos;   // (x,y,z) position of this vertex
    const void *data;  // data pointer for this vertex
    Edge *getEdge();  // an outgoing edge of this vertex
    unsigned int getID(); // id# of this vertex (unique for this Cell)
};
```

Here, `Vec3` is essentially an array of three doubles, but with *many* additional operators and functions that will come in quite handy. The `Vec3` class comes from the Simple Vector Library (SVL) of former CMU grad student Andrew Willmott. Take the time to skim his documentation; it will pay off.

There is a data pointer for each vertex, for extensibility. You can store arbitary (4 byte) information there, or pointers to additional memory (e.g. for colors, normals, or texture coordinates).

Face

Each face stores one piece of topological information, a pointer to one of the ccw-oriented edges of the face, plus optional attribute information (color, etc). The public interface you should use is summarized below (see cell/face.hh code).

```
class Face {
    Edge *getEdge();  // a ccw-oriented edge of this face
    const void *data;  // data pointer for this vertex
    unsigned int getID(); // id# of this face (unique for this Cell)
};
```

The data pointer here is just like that for vertices.

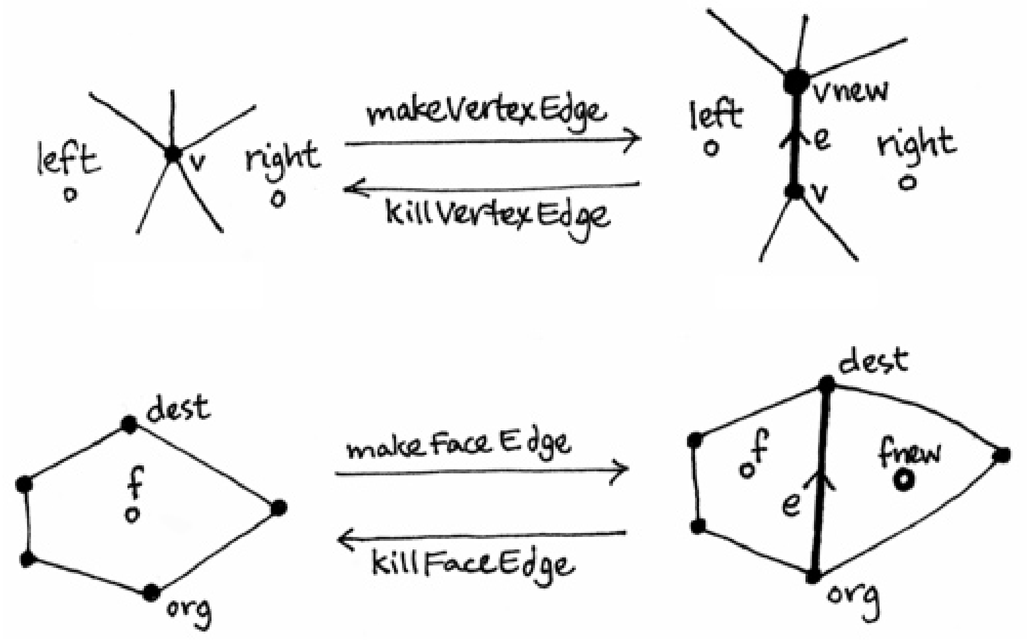
Cell, and Euler Operators

A `Cell` is a single polyhedron, which includes sets of vertices, edges, and faces. The routines you will need most are the following four:

```
class Cell {
        Edge *makeVertexEdge(Vertex *v, Face *left, Face *right);
        Edge *makeFaceEdge(Face *f, Vertex *org, Vertex *dest);
        void killVertexEdge(Edge *e);
        void killFaceEdge(Edge *e);
    };
```

which are called Euler operators, since they maintain Euler's formula V-E+F=2 interrelating the number of vertices, edges, and faces of a polyhedron of genus 0 (topologically equivalent to a sphere). If the topology is a valid polyhedron before the call, it will be valid after the call, as well. Note that these routines update the topology, but they use the default constructors for `Vertex` and `Face`, so the positions of new vertices are (0,0,0) -- you'll have to set them yourself.

Also, these routines (and the rest of the library) do not enforce linearity of edges or planarity of faces. It is permissible to have vertices and faces of degree 1 or 2, for example.



Given `Cell *c`, the calls do the following:

- `c->makeVertexEdge(v, left, right)` splits vertex `v`, creating a new edge and vertex that both lie between faces `left` and `right`. The new edge has `left` on its left and `right` on its right, `v` at its origin and the new vertex at its destination. The new edge is returned; the new vertex is easily found by taking `Dest()` of the return value. The new vertex and edge are stored in the sets associated with cell `c`. If `left` and `right` are not adjacent to `v` then an error message will be printed and core dumped.
- `c->makeFaceEdge(f, org, dest)` is the dual of this. It splits face `f`, creating a new edge and face that both lie between vertices `org` and `dest`. The new edge has `org` as its origin and `dest` as its destination, `f` as its left face and the new face as its right face. The new edge is returned; the new face is easily found by taking `Right()` of the return value. The new face and edge are stored in the sets associated with cell `c`. If `org` and `dest` are not adjacent to `f` then an error message will be printed and core dumped.

- `c->killVertexEdge(e)` is the inverse of `makeVertexEdge`. It removes edge `e` and vertex `e->Dest()`. Thus, `c->killVertexEdge(c->makeVertexEdge(v, left, right))` is a no-op.
- `c->killFaceEdge(e)` is the inverse of `makeFaceEdge`. It removes edge `e` and face `e->Right()`. Thus, `c->killFaceEdge(c->makeFaceEdge(f, org, dest))` is a no-op.

It is possible to build up a quad-edge data structure without using these routines, by using only the lower level routines, but it is many times more difficult and error-prone, so we discourage it.

For debugging or display purposes, you'll often want to loop over all the vertices, edges, or faces of a cell.

To loop over the vertices of `Cell *c`:

```
CellVertexIterator cellVerts(c);
Vertex *v;
while ((v = cellVerts.next()) != 0)
    <do something with vertex v>
```

To loop over the faces of `Cell *c`:

```
CellFaceIterator cellFaces(c);
Face *f;
while ((f = cellFaces.next()) != 0)
    <do something with face f>
```

Thus, to draw all the faces with OpenGL, using random colors:

```
#include <stdlib.h>
double frand() {return (double)rand()/RAND_MAX;}
CellFaceIterator cellFaces(c);
Face *f;
while ((f = cellFaces.next()) != 0) {
    glColor3f(frand(), frand(), frand()};
    glBegin(GL_POLYGON);
    FaceEdgeIterator faceEdges(f);
    Edge *edge;
    while ((edge = faceEdges.next()) != 0)
 glVertex3dv(&edge->Org()->pos[0]);
    glEnd();
}
```

In the above, `edge->Org()` is the origin of the current edge of the face, and the `&...->pos[0]` takes the address of its x coordinate. We could equally well use `Dest`. Stepping through the (undirected) edges of a cell is more complex, as we have things set up. Note that there are twice as many directed edges as undirected edges. The above code visits all directed edges once, so it visits each undirected edge twice. But for wireframe drawing and many other purposes you would want to operate on each undirected edge just once. A clever way to guarantee this, without marking edges or any additional arrays or lists, is to visit each undirected edge twice, but use the fact that the pointers to the two vertices are addresses, one of which is larger than the other:

```
CellFaceIterator cellFaces(c);
Face *f;
while ((f = cellFaces.next()) != 0) {
    // visit each face of cell c
    FaceEdgeIterator faceEdges(f);
    Edge *edge;
    while ((edge = faceEdges.next()) != 0) {
 // visit each edge of face f
 // if edge passes the following, its Sym will not,
 // and vice-versa
 if (edge->Org() < edge->Dest())
     <do something with edge>
    }
}
```

One could create a "CellEdgeIterator" using this approach, I suppose.

OBJ File I/O

So far we've seen how to modify polyhedra, but how would you create one in the first place? The easiest way is by reading a file. We have code to read and write polyhedral models in Wavefront's OBJ file format. Full documentation on the format is available from a 3-D format collection, and are here some complex models in OBJ format. The subset of the format that we read and write consists of lines with the syntax

```
# comment
v x y z
f v1 v2 ... vn
```

where the *i*th `v` line defines vertex *i*, with *i* starting at 1. *x*, *y*, and *z* are floating point numbers. Each `f` line defines an *n*-sided face, where the v_j are vertex indices. For example, the following defines a tetrahedron:

```
v -1 -1 -1
v 1 1 -1
v -1 1 1
v 1 -1 1
f 2 3 4
f 1 4 3
f 1 3 2
f 1 2 4
```

where vertex 1 (v1) is at (-1,-1,-1) and face 1 has vertices v2, v3, v4. Faces are counterclockwise when viewed from the outside, by convention. OBJ files for Platonic solids are in the model directory. These are the recommended test objects for this assignment.

To read an OBJ file, use the function `Cell *objReadCell(char *filename)`, and to write one, call the function `objWriteCell(Cell *cell, char *filename)`. The former returns NULL on error.

A Complete Quad-Edge Program

To put all the pieces together, we'll now show a program that reads in a cube file and then splits each edge in two (the first step in the construction of a cuboctahedron, perhaps). You might think that this will be a trivial application of the visit-all-edges code shown earlier, visiting each undirected edge once and calling `makeVertexEdge` on one of its vertices to split the edge. This would start to work, but note that the edge visiting algorithm operates sequentially, so as new edges are added to the cell, they will get traversed and split again! This is a problem!

To solve this, we can store a "splitme" bit for each edge in its `data` field (assuming `data` isn't being used for other purposes). Since `data` is a `void *`, it's necessary to cast when reading it. In our case we'll be storing an `int` in `data`.

```
Edge *split(Edge *e) {
    // split edge e, putting new vertex at midpoint

    // get Cell pointer from vertex (Edges don't have one)
    Cell *c = e->Org()->getCell();

    // split, creating new edge and vertex (sets topology only)
    Edge *enew = c->makeVertexEdge(e->Org(), e->Left(), e->Right());

    // At this point enew->Dest()==e->Org(),
    // and enew->Dest(), the new vertex, is between enew and e.
    // You might want to check the defn of makeVertexEdge to
    // convince yourself of this.

    // position new vertex at midpoint (note use of Vec3::operator+)
    enew->Dest()->pos = .5*(enew->Org()->pos + e->Dest()->pos);

    return enew; // return new edge
}

void splitAll(Cell *c) {
    {
 // first, set the splitme bits of all original edges
 CellFaceIterator cellFaces(c);
 Face *f;
 while ((f = cellFaces.next()) != 0) {
     // visit each face of cell c
     FaceEdgeIterator faceEdges(f);
     Edge *edge;
     while ((edge = faceEdges.next()) != 0) {
  // visit each edge of face f
  int splitme = edge->Org() < edge->Dest();
  // splitme = 0 or 1
	 // my Sym's bit will be the complement of mine
	 edge->data = splitme; // set bit
	    }
	}
	   }
	   {
	// go through again, splitting marked edges
	// need to construct a new iterator, hence the {}'s
	CellFaceIterator cellFaces(c);
	Face *f;

	while ((f = cellFaces.next()) != 0) {
	    // visit each face of cell c
	    FaceEdgeIterator faceEdges(f);
	    Edge *edge;
	    while ((edge = faceEdges.next()) != 0) {
	 // visit each edge of face f
	 // if its "splitme" bit set then split it
	 if ((int)edge->data) {
	     Edge *enew = split(edge);

	     // clear splitme bits on two sub-edges and
	     // their Syms to avoid recursive splitting
	     edge->data = 0;
	     edge->Sym()->data = 0;
	     enew->data = 0;
	     enew->Sym()->data = 0;
	 }
	    }
	}
	   }
	}

	void main() {
	    Cell *c;
	    c = objReadCell("cube.obj"); // read cube.obj
	    if (!c) exit(1);
	    splitAll(c);   // split all edges
	}
```

Acknowledgments: The quad-edge data structure was first described in the excellent, but rather abstract article [Leonidas Guibas and Jorge Stolfi, Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams, *ACM Transactions on Graphics*, 4(2), 1985, 75-123.]. CMU graduate student Andrew Bernard wrote this quad-edge library based on previous code from [Dani Lischinski, Incremental Delaunay Triangulation, *Graphics Gems IV*, Paul Heckbert, ed., Academic Press, 1994, 47-59. PS for article , C++ code.]. Michael Garland provided advice. I wrote the OBJ-to-quad-edge converter. Quad-edge is a variant of the earlier winged-edge data structure, which was described in the excellent, highly readable article [Bruce G. Baumgart, A polyhedron representation for computer vision, *Natl. Computer Conf. Proc.*, AFIPS, 1975, 589-596]. Our quad-edge library is partially based on Pat Hanrahan's winged-edge library from the New York Institute of Technology.

15-463, Computer Graphics 2
Paul Heckbert