ENCE 688R Civil Information Systems

Engineering Software Development in Java

Lecture Notes for ENCE 688R,
Civil Information Systems

Spring Semester, 2013

Mark Austin,
Department of Civil and Enviromental Engineering,
University of Maryland,
College Park,
Maryland 20742, U.S.A.

Contents

0

Chapter 14

Modeling Three-Dimensional Solids

14.1   Solid Modeling Formalisms

Solid modeling is the process of building and analyzing geometrically complete representations of points, lines, polygons, and three-dimensional solid objects. In a functional sense, a model is said to be geometrically complete if it is possible to ...

> **... answer questions about the geometry of an object (e.g., what is the area of a polygon), or perhaps the relationship among various objects (e.g., compute the intersection of a plane and three-dimensional solid), with algorithmic operations that act on the underlying data representation, and without taxing the inferential capabilities and pattern recognition skills of the human user.**

For this to be possible, objects must be

> **... closed, orientable, non-self-intersecting, bounding, and connected.**

These techniques are based on the idea that for any physical object, its boundaries or skin divide three dimensional Euclidian space in two regions, one interior and another one exterior to it. A solid model must then: (1) have a boundary that limits and encloses the inside of the solid; (2) the boundary has to be in contact with the interior: no dangling faces or edges; (3) and is finite in size and described by a limited amount of information [1].

Solid Modeling in Civil Engineering

Within the Civil Engineering domain, solid modeling has already been demonstrated to be an appropriate base for interactive design of reinforced concrete beam structures [2, 12]. Further work is needed to understand how ideas from solid modeling techniques might be used in the modeling of large precast (building and bridge) elements, possibly containing holes/portals, and building/bridge assemblies.

Figure 14.1 shows, for example, a bridge structure represented as an assembly of precast solid components. Integral properties of interest include volume, surface area, centroid, and moments and products of inertia for solid volumes. The centroid and inertia properties act as input to the computation of principal moments of inertia and principal axes (and their orientation).
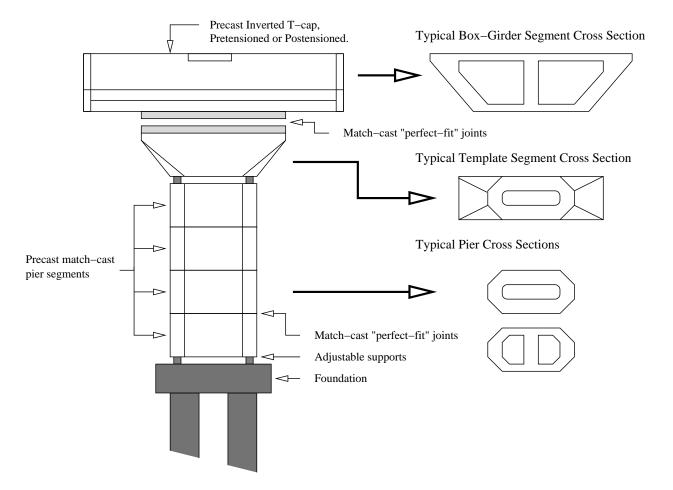
Precast Inverted T−cap,
Pretensioned or Postensioned.

Typical Box−Girder Segment Cross Section

Match−cast "perfect−fit" joints

Typical Template Segment Cross Section

Typical Pier Cross Sections

Precast match−cast
pier segments

Match−cast "perfect−fit" joints

Adjustable supports

Foundation

Figure 14.1. Elements of a precast bridge pier (Adapted from Billington [5]).

14.2   State-of-the-Art Solid Modeling

As illustrated in Figure 14.2, there are two basic approaches to state-of-the-art solid modeling: constructive solid geometry (CSG) and boundary representation (B-Rep) models.
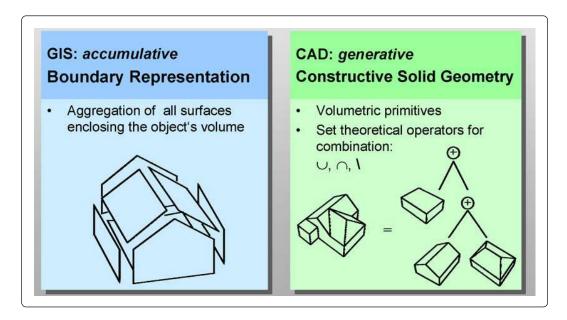

Figure 14.2. Boundary and constructive solid geometry (CSG) approaches to geometric modeling.

The former assumes that physical objects can be generated from elementary volumetric primitives (e.g., blocks, cylinders, tetrahedrons, spheres) plus a set of theoretical operators (e.g., union, intersection, difference) for their combination.

Boundary representation (B-Rep) models approach solid modeling through the aggregation of all orientable surface faces enclosing an object's volume. Surface faces that are connected to each other by their edges, which in turn, are defined by vertices. One complicating factor with B-Rep models is that not all collections of faces define a valid physical model. This problem can be addressed through restrictions on face collections; for instance, each edge must belong to exactly two loops, and the ordering of edges must be consistent throughout the model. The vertices are the only real geometric data of the model since they define, directly or indirectly, the other elements (faces, edges, ...). Mantyla and Sulonen [11] point out, however, that in order to speed up various operations on the model, boundary modelers frequently store other explicit data on the connections between faces, loops, edges and vertices.

In general terms, no single solid modeling technique is superior for all applications [1]. Evidence of this observation can be found in the large number of solid modeling data structures (and variants therein) that have been proposed over the past four decades – well known examples include Baumgarts winged-edge data structure [3, 4] Weilers radial-edge data structure [13], the Partial Entitiy data structure [9], as well as various techniques for modeling polyhedral surfaces [6, 8]. Before choosing a particular technique for a solid modeler, the pros and cons of both have to be weighted and compared

with the desired features of the modeler.

When it comes to evaluating integral properties of solids, different approaches are available depending on the type of representation in use. For CSG models, it is done by decomposing the model into small cells and computing the contribution of each cell to the total result. The decomposition is made by a conversion algorithm and the accuracy of results depend on the arrangement of cells selected [10]. Better results for integral properties can be achieved when using B-Rep models. Because of the complete definition of the faces of the model, only the contribution of each face has to be evaluated, rather than evaluating a much larger number of more or less small cells. Furthermore, since no approximations are made, there is a great improvement in the accuracy of results, which can be considered to be exact except for round off errors.

14.3   Boundary Representations and Adjacency Relations

Boundary models represent solids by ...

> **... a set of boundary surface faces, which in turn, are bounded by sets of edges.**

The information in the boundary represenation is stored as a hierarchical data structure. It links points together to form edges, and then groups edges to form the bounding faces of the solid. The entire organization of the data structure serves to maintain the topological connectivity and coherence of these points; the only true geometric data is the point coordinates.

An important concept in boundary representation models is that of adjacency relations. Adjacency relations are the conceptual links among the different levels of the model hierarchy, and between neighboring elements at the same level. For example, each edge is adjacent to two faces upward in the hierarchy, to two vertices downward in the hierarchy, and to some number of other edges at the same level. Nine adjacency relationships for boundary models have been identified; they are:

> **...   vertex-vertex, vertex-edge, vertex-face, edge-vertex, edge-edge, edge-face, face-vertex, face-edge, and face-face.**

Figure 14.3 shows, for example, schematics of vertex-vertex and vertex-edge relationships.

Weiler [13] states ...

> **... if a topological representation contains enough information to recreate all nine of these adjacency relations without error or ambiguity, it can be considered a sufficient adjacency topology representation.**

In other words, a data representation must provide for the determination of all of the adjacency relations in order to be geometrically complete. Answers to this problem are complicated by the fact that solids can be defined through a variety of topological formalisms which, in turn, means that data structure implementations need not be unique. Which one is best often boils down to issues of performance and ability of the data structure to efficently answer adjacency queries such as:
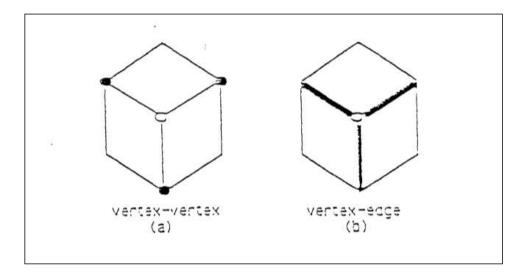
Figure 14.3. Vertex-vertex and vertex-edge adjacency relationships.

- What edges are connected to a vertex?

- What vertices are connected to a face?

A good implementation should also have compact storage requirements, and only require a constant amount of data for each element.

14.4   Euler Operators

Euler operators were first introduced by Baumgart for generating complete polyhedral boundary representation of solids [3]. These operators also guaranty that Eulers law is satisfied when manipulating the topological properties of models. The Euler-Poincare law states that for all solids the following relationship holds:

$$F - E + V - L = 2(B - G) \tag{14.1}$$

where the letters stand for faces, edges, vertices, faces inner loops, bodies and through holes or "genus" respectively. Equation 14.1 applies to manifolds; that is, each edge is shared by exactly two faces, each edge is connected by exactly two vertices, and for the three-dimensional case, at least three edges join at each vertex. The range of applicability includes solids with curved faces or edges, and planar graphs and polygon meshes in general [1, 7]. However, all polyhedral faces must be bounded by a single loop of connected vertices. And faces cannot be rings or otherwise have holes in them. Equation 14.1 does not apply to non-manifolds; that is, a solid where an edge is connected to more than two faces.

Example. A Simple Cube. Figure 14.4 shows the elements of cube: 6 faces, 8 vertices, 12 edges, zero inner loops, one body, zero holes.
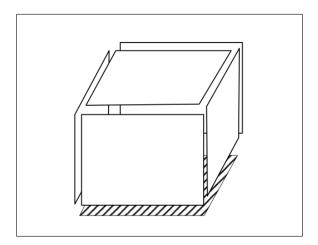

Figure 14.4. Faces, edges, and vertices in a cube.

Checking Euler's formula: faces - edges + vertices - loops = 6 - 12 + 8 = 2(1 - 0) = 2.

Low-Level Operators. The low-level Euler operators are denoted using descriptive mnemonic names. Each letter of the name stands for particular operation or node:

```
M: make
K: kill
S: split
J: join
V: vertex
E: edge
F: face
S: solid
H: hole
R: ring
```

Note that the equivalent to ring in the half edge data structure is loop. There are ten Euler operators, five of which are the inverse of the other five. A short description of the operators and a definition of the respective C functions of some operators as used in the GWB is given ahead:

1. **MVFS** (make vertex face solid). This is the first operator used when creating a model. The result is the initialization of the data structure for a solid with one face with one vertex, an empty loop and no edges. This is called a skeletal primitive.

2. **KVFS** (kill vertex face solid). This is the inverse of the previous operator.

3. **MEV** (make edge vertex). Creates a new vertex which, along with a previous one, defines a new edge.

4. **KEV** (kill edge vertex): undoes the operation made by a MEV.

5. **MEF** (make edge face). Creates a new edge defined by two existing vertices and hence defines a new face. But note that one face is actually two faces with opposite orientations glued together

6. **KEF** (kill edge face). inverse of MEF.

7. **KEMR** (kill edge make ring). Splits one loop in two loops of a same face by erasing an edge. This way, a hole is created inside the face.

8. **MEKR** (make edge make ring). Inverse of KEMR. Connects two loops in one. These last six operators are called local manipulators since they are applied on the local topological properties of boundary models.

9. **MEKR** (make edge make ring) and **KFMRH** (kill face make ring hole). While KEMR creates a hole in one face, this operator "extends" the hole to other face and therefore to the whole solid. However, KFMRH is also capable of creating one shell if the two argument faces belong to two different shells.

10. **MFKRH** (make face kill ring hole): This is the inverse of KFMRH. These last two operators are called global manipulators because they modify the global topological properties of the models.

Now let's see how sequences of operators can be designed to create three dimensional solids.
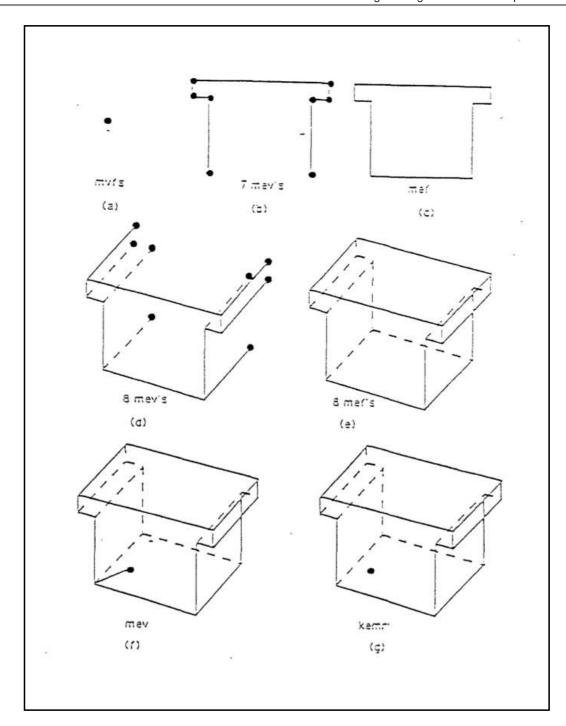
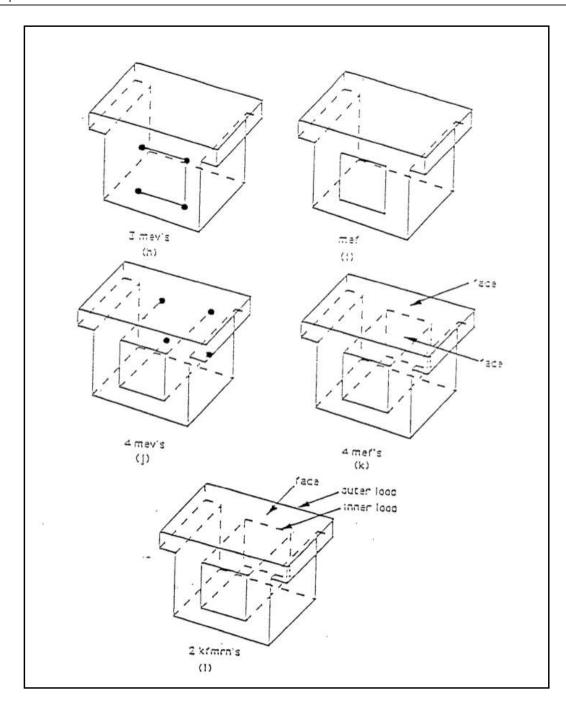**Figure 14.5.** Using Euler operations for the step-by-step assembly of a TBeam.

Figure 14.6. Using Euler operations for the step-by-step assembly of a TBeam.

14.5 Half-Edge Data Structure

Figure 14.7 is a schematic of the key relationships among the six component types – solid, face, loop, edge, halfedge and vertex – making up the half edge data structure.
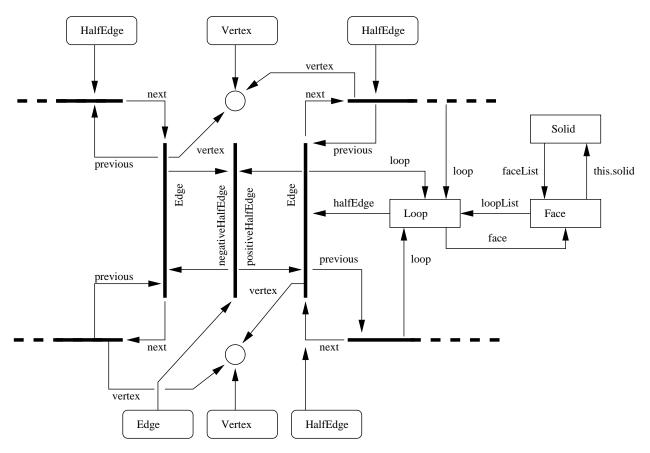

Figure 14.7. Schematic of relationships in the halfedge data structure. Note: Some relationships have been omitted for clarity.

For convenience, the nodes of the data structure can be organized into a vertical structure containing six layers:

I. Solid. As illustrated in Figure 14.8, solids are defined explicitly by collections of faces, edges and vertices. Solid is the root of the data structure. Inside the data structure, a structure `Solid` has references to lists of faces, edges and vertices. Lower level details (i.e., loops and halfedges) are accessed indirectly through the Face data structure.

II. Face. Face represents one planar face delimited by a list of loops, each representing one polygonal boundary of the face. Each face is defined by exactly one outer loop and zero or more inner loops, representing holes in the face. Each structure `Face` has references to the parent solid (i.e., this.solid), and to the lists of loops that define it. It also has references to the face equation and to its outer loop.
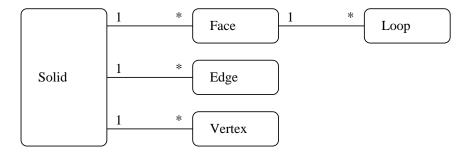
Figure 14.8. UML schematic of the solid data structure.

The face equation is a four dimensional vector: the first three coefficients store a unitary vector normal to the face and oriented to the inside of the solid; the fourth coefficient is the distance from the origin to the center of the face projected on the direction of the face vector. In order for the normal vectors to be consistently oriented to the inside of the solid, the vertices of all outer loops have to be arranged in the same direction, in this case clockwise, when faced from the outside.

III. Loop. A loop represents a polygonal boundary of its parent face. While the vertices of outer loops are arranged clockwise (as viewed from the outside of the solid), for inner loops the arrangement is counter clockwise. As we will soon see, this arrangement of vertices plays a central role in the design of algorithms to compute integral properties of objects with holes.

The structure `Loop` has references to its parent face, to the list of half edges that define the loop.

IV. Edge. Surprisingly, edges play a relatively minor role in the half edge data structure. As detailed below, each edge is associated with exactly two half edges. All other information (e.g., start and end vertices) is obtained indirectly through the half-edge connectivity relationships.

V. HalfEdge. The structure `HalfEdge` ties everything together and provides the connectedness information critical to the development of adjacency queries. Each half edge is connected to a vertex (from which it starts), a face on its left side (when looking along the half-edge from the origin), if there is one, and to exactly one edge. Each half edge also has references to predecessor and successor half edges. The latter enables development of loops within a face. Each edge is associated with exactly two half edges, each belonging to different, but adjacent, faces. One half edge will have a positive orientation; the other half edge will have a negative orientation.

VI. Vertex. A vertex contains the homogeneous coordinate representation of a point in a three dimensional space. The structure `Vertex` has pointers to a vertex identifier, to its parent half edge and to the next and previous vertex in the vertex list.

Implementation of the Half Edge Data Structure. In early implementations of the half edge data structure (see, for example, references [2, 12, 10]), the `Node`, `Edge`, and `Loop` data structures also contained references to next and previous items (i.e., they were explicitly coded as doubly linked lists). While that latter allows for systematic access to every element of the data structure, the resulting graph of data structures is overly complicated in our opinion.

To simplify the details of implementation, we assume that each face, edge and vertex object will have a unique numerical ID. As illustrated in Figure 14.7, only the `Halfedge` data structure will contain direct references to next and previous items in a Loop. Retrieval of adjacency information (e.g., finding the two face objects bounding an edge) is achieved through the halfEdges bounding an edge and, in turn, references to the parent loop and face.

14.6   Application 1. Assembly/Properties of a Rectangular Block

We begin with definition and engineering analysis of a rectangular block having edge lengths of 3 units, 3 units, and 5 units along the x, y, and z axes respectively. Three-dimensional views of the block are shown in Figures 14.9 and 14.10. Two steps are needed to define the block topology and geometry:

Step I. Create a Planar Lamina. First we create a rectangular lamina in the x-y plane having four verticies and four edges. The lamina has one surface composed of two faces (c.f., the two faces of a piece of paper). This task is completed with five function calls to the Euler operators, namely:

```
Solid sA = new Solid ( "Rectangular Block");
sA.mvfs( 1, 1,    1.0, 1.0, 0.0 );
sA.mev(  1, 1, 2, 4.0, 1.0, 0.0 );
sA.mev(  1, 2, 3, 4.0, 4.0, 0.0 );
sA.mev(  1, 3, 4, 1.0, 4.0, 0.0 );
sA.mef(  1, 4, 1, 2);
```

The operator `mfvs` and the three `mev`'s define the four vertices of face 1. The first three edges are defined by the `mev`'s, along with the end vertex of each edge. Face 1 is then completely represented by defining the last edge spanning vertices 1 and 4 previously created. The `mef` function also creates face 2 that, together with face 1, makes up a surface.

Step II. Translational Sweep. The rectangular block is created by moving face 1 through a translational sweep, i.e.,

```
Face face = sA.findFace(1);
sA.sweep( face, 0.0, 0.0, 5.0 );
```

The function `sweep()` sweeps face 1 away from face 2 a distance of 5.0 units in the z direction. Face 1 is retrieved by method `findFace()`.

Systematic Assembly of the Block Topology and Geometry. At the conclusion of Step I the block topology and geometry, and details of half-edge and edge-vertex connectivity are as follows:

```
List Solid: "Rectangular Block"
Face    count = 2
Edge    count = 4
Vertex count = 4
Max face    ID = 2
Max vertex ID = 4

Faces:
FaceId(1):
<OUTER LOOP> :v1+he(1,4) -> v4+he(4,3) -> v3+he(3,2) -> v2+he(2,1) ->
Face Equation = Vector(0.0, 0.0, -1.0)
FaceId(2):
<OUTER LOOP> :v4-he(1,4) -> v1-he(2,1) -> v2-he(3,2) -> v3-he(4,3) ->
Face Equation = Vector(0.0, 0.0, 1.0)
```
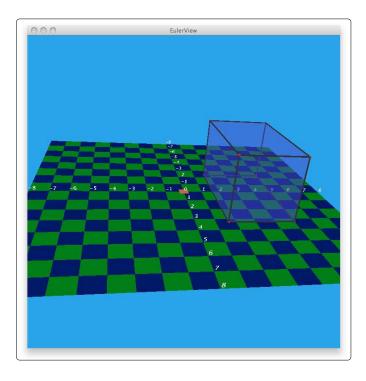
Figure 14.9. Three-dimensional view of the rectangular block example. Viewpoint is looking along the z axis, with negative z values on the horizon.
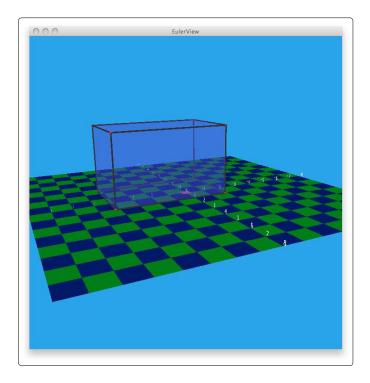

Figure 14.10. Three-dimensional view of the rectangular block example. Viewpoint is looking along the x axis, with negative x values on the horizon.
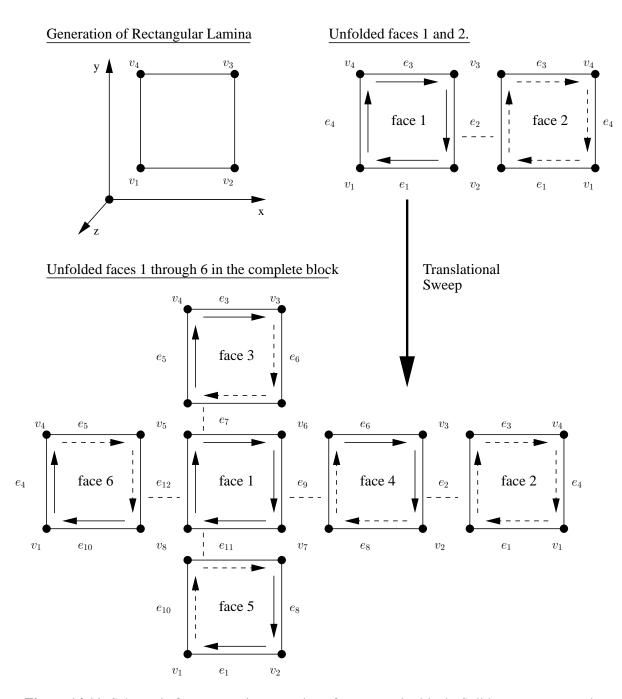
Generation of Rectangular Lamina

Unfolded faces 1 and 2.

Unfolded faces 1 through 6 in the complete block

Translational
Sweep

Figure 14.11. Schematic for systematic generation of a rectangular block. Solid oneway arrows denote a positive half edge. Negative half edges are drawn as dashed oneway arrows.

```
Edges:
Edge():+ve he -> vertex 2        -ve he -> vertex 1
Edge():+ve he -> vertex 3        -ve he -> vertex 2
Edge():+ve he -> vertex 4        -ve he -> vertex 3
Edge():+ve he -> vertex 1        -ve he -> vertex 4

Verticies:
VertexId 1: Vertex(1) has coordinate (1.0, 1.0, 0.0)
VertexId 2: Vertex(2) has coordinate (4.0, 1.0, 0.0)
VertexId 3: Vertex(3) has coordinate (4.0, 4.0, 0.0)
VertexId 4: Vertex(4) has coordinate (1.0, 4.0, 0.0)

Volume("Rectangular Block") = 0.0
SurfaceArea("Rectangular Block") = 18.0
```

Points to note:

1. Faces 1 and 2 are each defined by a single outer loop. By convention, the outside perimeter of each face is defined by a loop of half edges, oriented so that a systematic traversal will occur in a clockwise direction (as viewed from outside the solid). Inward pointing face normals are computed by applying the right-hand rule to the exterior loop of half edges.

2. Each edge is associated with one positive half edge and one negative half edge. The top right-hand schematic of Figure 14.11 shows faces 1 and 2 unfolded, together with the vertices and edges. The positive and negative half edges are drawn as solid and dashed oneway arrows respectively.

3. The notation:

    ```
    <OUTER LOOP> :v1+he(1,4) -> v4+he(4,3) ....
    ```

    indicates that the outer loop for face 1 begins with an edge anchored to vertex v_1 and positive and negative half-edges beginning at vertices v_1 and v_4 respectively. This is followed by an edge anchored to vertex v_4 with positive and negative half-edges beginning at vertices v_4 and v_3, and so forth. The perimeter of face 2 is also traversed in a clockwise direction, but in this case all of the participating half edges are negative half edges, as indicated by the notation $-he(\ldots)$.

4. The rectangular lamina has one surface aggregated from two faces and zero thickness. This is refected in our computation of lamina surface area and volume.

Step II expands the model from 2 faces to 6 faces, from 4 edges to 12 edges, and from 4 vertices to 8 vertices:

```
List Solid: "Rectangular Block"
Face    count = 6
Edge    count = 12
Vertex count = 8
Max face   ID = 6
Max vertex ID = 8
```

```
Faces:
FaceId(1):
<OUTER LOOP> :v8+he(8,5) -> v5+he(5,6) -> v6+he(6,7) -> v7+he(7,8) ->
Face Equation = Vector(0.0, 0.0, -1.0)
FaceId(2):
<OUTER LOOP> :v4-he(1,4) -> v1-he(2,1) -> v2-he(3,2) -> v3-he(4,3) ->
Face Equation = Vector(0.0, 0.0, 1.0)
FaceId(3):
<OUTER LOOP> :v6-he(5,6) -> v5+he(5,4) -> v4+he(4,3) -> v3-he(6,3) ->
Face Equation = Vector(0.0, -1.0, 0.0)
FaceId(4):
<OUTER LOOP> :v7-he(6,7) -> v6+he(6,3) -> v3+he(3,2) -> v2-he(7,2) ->
Face Equation = Vector(-1.0, 0.0, 0.0)
FaceId(5):
<OUTER LOOP> :v8-he(7,8) -> v7+he(7,2) -> v2+he(2,1) -> v1-he(8,1) ->
Face Equation = Vector(0.0, 1.0, 0.0)
FaceId(6):
<OUTER LOOP> :v5-he(8,5) -> v8+he(8,1) -> v1+he(1,4) -> v4-he(5,4) ->
Face Equation = Vector(1.0, 0.0, 0.0)

Edges:
Edge():+ve he -> vertex 2        -ve he -> vertex 1
Edge():+ve he -> vertex 3        -ve he -> vertex 2
Edge():+ve he -> vertex 4        -ve he -> vertex 3
Edge():+ve he -> vertex 1        -ve he -> vertex 4
Edge():+ve he -> vertex 5        -ve he -> vertex 4
Edge():+ve he -> vertex 6        -ve he -> vertex 3
Edge():+ve he -> vertex 5        -ve he -> vertex 6
Edge():+ve he -> vertex 7        -ve he -> vertex 2
Edge():+ve he -> vertex 6        -ve he -> vertex 7
Edge():+ve he -> vertex 8        -ve he -> vertex 1
Edge():+ve he -> vertex 7        -ve he -> vertex 8
Edge():+ve he -> vertex 8        -ve he -> vertex 5

Verticies:
VertexId 1: Vertex(1) has coordinate (1.0, 1.0, 0.0)
VertexId 2: Vertex(2) has coordinate (4.0, 1.0, 0.0)
VertexId 3: Vertex(3) has coordinate (4.0, 4.0, 0.0)
VertexId 4: Vertex(4) has coordinate (1.0, 4.0, 0.0)
VertexId 5: Vertex(5) has coordinate (1.0, 4.0, 5.0)
VertexId 6: Vertex(6) has coordinate (4.0, 4.0, 5.0)
VertexId 7: Vertex(7) has coordinate (4.0, 1.0, 5.0)
VertexId 8: Vertex(8) has coordinate (1.0, 1.0, 5.0)

Volume("Rectangular Block") = 45.0
SurfaceArea("Rectangular Block") = 78.0
```

Engineering Properties. Recall that the product of inertia of a solid is zero if one of the coordinate axis involved is perpendicular to a plane of symmetry of the body. By inspection, it can be seen that the prism has two planes of symmetry, and therefore the three products of inertia are zero. As defined, the principal axes are parallel to the coordinate axes (see Appendix A.8).

14.7   Application 2: Properties of a Rectangular Block containing a Hole

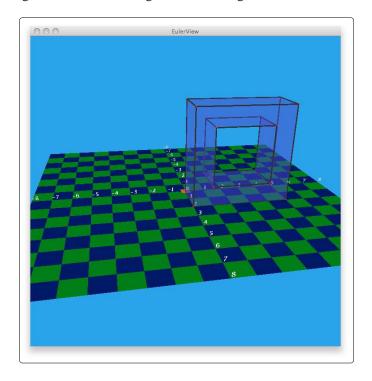We now define a rectangular block containing a hole. See Figure 14.12.

Figure 14.12. Rectangular block containing a hole.

Since this volume only has one plane of symmetry (i.e., the $X - Y$ plane), the POIs I_{xz} and I_{yz} must be zero and one of the principal axis must be perpendicular to the XY plane.

Source Code

Here is the source code for creating and printing an empty floorslab solid containing a hole.

```
// Test Problem 2: Create and print an empty "floorslab" solid containing hole ...

double dr = 4.0; double dx = 20.0; double dy = 10.0;

// Define the floorslab exterior ....

Solid sB = new Solid ( "Floorslab" );
sB.mvfs( 1, 1,    0.0, 0.0, 0.0 );
sB.mev(  1, 1, 2,  dx, 0.0, 0.0 );
sB.mev(  1, 2, 3,  dx,  dy, 0.0 );
sB.mev(  1, 3, 4, 0.0,  dy, 0.0 );
sB.mef(  1, 4, 1, 2);

// Create a hole in the slab ...

sB.mev(  1, 1, 5,     dr,    dr,  0.0);
```

```
        sB.mev(  1, 5, 6,        dr, dy-dr,   0.0);
        sB.mev(  1, 6, 7, dx-2*dr, dy-dr,   0.0);
        sB.mev(  1, 7, 8, dx-2*dr,    dr,   0.0);
        sB.mef(  5, 8, 1, 3);

        // Kill edge make ring .....

        sB.kemr( 1, 5, 1 );

        // Kill face make ring hole .....

        sB.kfmrh( 2, 3 );

        // Sweep face into a solid .....

        Face faces = sB.findFace(1);
        sB.sweep( faces, 0.0, 0.0, 1.0 );

        System.out.println("");
        System.out.println("Flat floor slab exterior with offset hole");
        System.out.println("=========================================");

        System.out.println( sB );
        System.out.println( "Volume(\"" + sB.solidName + "\") = " + sB.getVolume() );
        System.out.println( "SurfaceArea(\"" + sB.solidName + "\") = " + sB.getSurfaceArea() );
```

Output

```
Flat floor slab exterior with offset hole
=========================================

List Solid: "Floorslab"
Face   count = 10
Edge   count = 24
Vertex count = 16
Max face   ID = 10
Max vertex ID = 16

Faces:
FaceId(1):
<OUTER LOOP> :v12+he(12,9) -> v9+he(9,10) -> v10+he(10,11) -> v11+he(11,12) ->
<INNER LOOP> :v16+he(16,13) -> v13+he(13,14) -> v14+he(14,15) -> v15+he(15,16) ->
Face Equation = Vector(0.0, 0.0, -1.0)
FaceId(2):
<OUTER LOOP> :v4-he(1,4) -> v1-he(2,1) -> v2-he(3,2) -> v3-he(4,3) ->
<INNER LOOP> :v8-he(5,8) -> v5-he(6,5) -> v6-he(7,6) -> v7-he(8,7) ->
Face Equation = Vector(0.0, 0.0, 1.0)
FaceId(3):
<OUTER LOOP> :v10-he(9,10) -> v9+he(9,1) -> v1+he(1,4) -> v4-he(10,4) ->
Face Equation = Vector(1.0, 0.0, 0.0)
FaceId(4):
<OUTER LOOP> :v11-he(10,11) -> v10+he(10,4) -> v4+he(4,3) -> v3-he(11,3) ->
Face Equation = Vector(0.0, -1.0, 0.0)
FaceId(5):
<OUTER LOOP> :v12-he(11,12) -> v11+he(11,3) -> v3+he(3,2) -> v2-he(12,2) ->
```

```
Face Equation = Vector(-1.0, 0.0, 0.0)
FaceId(6):
<OUTER LOOP> :v9-he(12,9) -> v12+he(12,2) -> v2+he(2,1) -> v1-he(9,1) ->
Face Equation = Vector(0.0, 1.0, 0.0)
FaceId(7):
<OUTER LOOP> :v14-he(13,14) -> v13+he(13,5) -> v5+he(5,8) -> v8-he(14,8) ->
Face Equation = Vector(0.0, -1.0, 0.0)
FaceId(8):
<OUTER LOOP> :v15-he(14,15) -> v14+he(14,8) -> v8+he(8,7) -> v7-he(15,7) ->
Face Equation = Vector(1.0, 0.0, 0.0)
FaceId(9):
<OUTER LOOP> :v16-he(15,16) -> v15+he(15,7) -> v7+he(7,6) -> v6-he(16,6) ->
Face Equation = Vector(0.0, 1.0, 0.0)
FaceId(10):
<OUTER LOOP> :v13-he(16,13) -> v16+he(16,6) -> v6+he(6,5) -> v5-he(13,5) ->
Face Equation = Vector(-1.0, 0.0, 0.0)

Edges:
Edge():+ve he -> vertex 2        -ve he -> vertex 1
Edge():+ve he -> vertex 3        -ve he -> vertex 2
Edge():+ve he -> vertex 4        -ve he -> vertex 3
Edge():+ve he -> vertex 1        -ve he -> vertex 4
Edge():+ve he -> vertex 6        -ve he -> vertex 5
Edge():+ve he -> vertex 7        -ve he -> vertex 6
Edge():+ve he -> vertex 8        -ve he -> vertex 7
Edge():+ve he -> vertex 5        -ve he -> vertex 8
Edge():+ve he -> vertex 9        -ve he -> vertex 1
Edge():+ve he -> vertex 10        -ve he -> vertex 4
Edge():+ve he -> vertex 9        -ve he -> vertex 10
Edge():+ve he -> vertex 11        -ve he -> vertex 3
Edge():+ve he -> vertex 10        -ve he -> vertex 11
Edge():+ve he -> vertex 12        -ve he -> vertex 2
Edge():+ve he -> vertex 11        -ve he -> vertex 12
Edge():+ve he -> vertex 12        -ve he -> vertex 9
Edge():+ve he -> vertex 13        -ve he -> vertex 5
Edge():+ve he -> vertex 14        -ve he -> vertex 8
Edge():+ve he -> vertex 13        -ve he -> vertex 14
Edge():+ve he -> vertex 15        -ve he -> vertex 7
Edge():+ve he -> vertex 14        -ve he -> vertex 15
Edge():+ve he -> vertex 16        -ve he -> vertex 6
Edge():+ve he -> vertex 15        -ve he -> vertex 16
Edge():+ve he -> vertex 16        -ve he -> vertex 13

Verticies:
VertexId 1: Vertex(1) has coordinate (0.0, 0.0, 0.0)
VertexId 2: Vertex(2) has coordinate (20.0, 0.0, 0.0)
VertexId 3: Vertex(3) has coordinate (20.0, 10.0, 0.0)
VertexId 4: Vertex(4) has coordinate (0.0, 10.0, 0.0)
VertexId 5: Vertex(5) has coordinate (4.0, 4.0, 0.0)
VertexId 6: Vertex(6) has coordinate (4.0, 6.0, 0.0)
VertexId 7: Vertex(7) has coordinate (12.0, 6.0, 0.0)
VertexId 8: Vertex(8) has coordinate (12.0, 4.0, 0.0)
VertexId 9: Vertex(9) has coordinate (0.0, 0.0, 1.0)
VertexId 10: Vertex(10) has coordinate (0.0, 10.0, 1.0)
VertexId 11: Vertex(11) has coordinate (20.0, 10.0, 1.0)
```

```
VertexId 12: Vertex(12) has coordinate (20.0, 0.0, 1.0)
VertexId 13: Vertex(13) has coordinate (4.0, 4.0, 1.0)
VertexId 14: Vertex(14) has coordinate (12.0, 4.0, 1.0)
VertexId 15: Vertex(15) has coordinate (12.0, 6.0, 1.0)
VertexId 16: Vertex(16) has coordinate (4.0, 6.0, 1.0)

Volume("Floorslab") = 184.00000000000006
SurfaceArea("Floorslab") = 448.0
```

Bibliography

[1] Arnand V.B. *Computer Graphics and Geometric Modeling for Engineers*. John Wiley and Sons, New York, 1993.

[2] Austin M. A., Preston J. Solid Modeling of RC Beams : Data Structures and Algorithms. *Journal of Computing in Civil Engineering, ASCE*, 6(4):389–403, October 1992.

[3] Baumgart B.G. Geometric Modeling for Computer Vision. Technical Report Stanford Artificial Intelligence Laboratory, Memo No. AIM-249, Stanford University, October 1974.

[4] Baumgart B.G. In *AFIPS Joint Computer Conferences Archive, Proceedings of the May 19-22, 1975, National Computer Conference and Exposition*, Anaheim, California, 1975.

[5] Billington S.L., Barnes R.W., and Breen J.E. A Precast Segmental Substructure System for Standard Bridges. *PCI Journal*, 44(4):56–73, 1999.

[6] Chouadria R., and Veron P. Identifying and Re-Meshing Contact Interfaces in a Polyhedral Assembly for Digital Mock-Up. *Engineering with Computers*, 22(1):47–58, 2006.

[7] Ericson C. *Real-Time Collision Detection*. Morgan Kaufmann Publishers, 2005.

[8] Kettner L. Designing a data structure for polyhedral surfaces. In *SCG '98: Proceedings of the Fourteenth Annual Symposium on Computational Geometry*, pages 146–154, New York, NY, USA, 1998. ACM.

[9] Lee S.H., Lee K. Partial Entity Structure: A Compact Non-Manifold Boundary Representation based on Partial Topological Entities. In *6th ACM Symposium on Solid Modeling and Applications*, Ann Arbor, MI, 2001.

[10] Mantyla M. *An Introduction to Solid Modeling*. Computer Science Press, Rockville, MD, 1988.

[11] Mantyla M., Sulonen R. GWB: A Solid Modeler with Euler Operators. *IEEE Computer Graphics and Applications*, 2(7):17–31, September 1982.

[12] Preston J., Austin M.A. Solid Modeling of RC Beams: Computational Environment. *Journal of Computing in Civil Engineering, ASCE*, 6(4):404–416, October 1992.

[13] Weiler K. In *The Radial Edge Data Structure. A Topological Representation for Non-Manifold Geometric Boundary Modeling. In J.L. Encarnascao, M.J. Wozny, H.W. McLaughlin, editors, Geometric Modeling for CAD Applications*, pages 3–36, Amsterdam, 1988.

Index