ME 290-R Spring 2013 McMains
Homework # 1
Due Friday Feb 15 5pm (slide under my office door; I will be at an all-day faculty retreat)

1) Obtain access to SolidWorks, for example in the CAD labs in 2105/2107 Etcheverry (currently only open 8am-5pm). If you've never used SolidWorks before, look under the Help menu, SolidWorks tutorials, and complete the first tutorial ("Lesson 1 - Parts" and the tutorial on Revolves and Sweeps.

2)

a) Download the SolidWorks part FlipSideToCutDemo.SLDPRT from the class website. Open the part and look at how it was made – expand the three features in the FeatureManager design tree on the left side of the screen by clicking on their "+" icons, drag your mouse slowly down across the features and their sketches to see the sketches used for each get hightlighted, right-click on the features and select the "Edit Feature" icon (to left of popup icon menu) to see how they were defined, etc. Once you understand how the part was built, try making this modification: Right click on the 2nd feature, "Cut-Extrude2," select the "Edit Feature" icon, and check the box next to "Flip side to cut." Accept the change by hitting Enter or clicking the green check button. What happens and why? (Note that Cut-Extrude3 was also defined in the same manner but has different qualitative behavior – why is it different?)

b) Based on your (admittedly perhaps limited) experience as a SolidWorks user, what guesses can you reasonably make about the solid modeling representation scheme(s) it might use? How do you think its architecture compares to the advanced architecture model proposed in Requicha's figure 17? How much of the domain of r-sets does it appear to cover? Come up with at least one example of a solid that you would think could be made using the extrude, rotate, and/or sweep operations you have learned in the tutorials but that nonetheless cannot be built using SolidWorks. Why do you think SolidWorks might not support the geometry you have described?

3) In this problem, you will compare three b-reps: the simple triangulated b-rep scheme described in Requicha section 2.7, the winged edge data structure as summarized in the textbook by Foley, van Dam, et al. section 12.5.2 (see attached scan), and the radial edge structure described in Weiler. Assume a right-handed coordinate system throughout (i.e. faces are oriented counter-clockwise when viewed from the exterior of the solid).

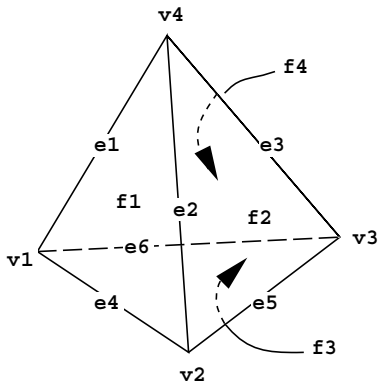


Figure 1: *A tetrahedron with ID labels for its vertices, edges, and faces.*

a) Draw diagrams showing in detail how the solid tetrahedron pictured in Figure 1 would be represented using each of the three data structures. Use the ID labels indicated in the figure to identify the faces, edges, and vertices of the tetrahedron. For ID labels for the face-uses, edge-uses, and vertex-uses, please use the referenced entity's ID number with a subscript, e.g. call edge e4's two edge-uses $eu4_1$ and $eu4_2$. Show and label all pointers between entities (if drawing where each pointer points clutters the diagram, just give the ID label of where each labeled pointer points). Show all linked lists required. If the specification in the reading is ambiguous, state your assumptions.

The attached full-page diagram for the Requicha representation has been partially completed for you as an example of what I'm looking for. Complete the missing information in this diagram.

Make a similar diagram for the winged-edge structure, using the same names for pointers as given in the text. You will have pointers for the n-vertex, the p-vertex, the n-face, the p-face, the n-face-CW-edge, the n-face-CCW-edge, the p-face-CW-edge, and the p-face-CCW-edge. Please keep them in this same order, for each edge in your diagram.

For the radial edge data structure, it gets very repetitive to show all the details, so it is okay to only show f1 and its associated faces-uses, e1 and its associated edge-uses, and v1 and its associated vertex-uses (just for radial-edge; for winged-edge show all faces, edges, vertices). Put each list in numerical order. Be sure to refer to the Pascal declarations in the Weiler paper for complete details; the figures in the paper are somewhat simplified. See me during office hours if you have any questions.

b) Assuming each coordinate and each pointer takes 4 bytes to store, what are the storage requirements for the tetrahedron in each of the three schemes?

c) For each of the three representation schemes, give an efficient algorithm (using pseudo-code) that takes a pointer to a vertex of a polyhedron as input, and outputs all faces adjacent to it. For the winged-edge scheme, you may assume that the vertex is on a 2-manifold solid. For the Requicha scheme, you may assume you have a pointer to the top level of the hierarchy for the entire model. An example of pseudo-code for a related sample HW problem is included at the end of the assignment.

d) For each of your pseudo-code algorithms, how many pointers do you need to follow, worst case, to find the faces adjacent to one of the vertices of the tetrahedron?

the pairs of adjacent vertices in the polygon vertex lists. Edges may instead be represented explicitly as pairs of vertices, with each face now defined as a list of indices into the list of edges. These representations were discussed in more detail in Section 11.1.1.

12.5.2 The Winged-Edge Representation

Simple representations make certain computations quite expensive. For example, discovering the two faces shared by an edge (e.g., to help prove that a representation encodes a valid solid) requires searching the edge lists of all the faces. More complex b-reps have been designed to decrease the cost of these computations. One of the most popular is the *winged-edge* data structure developed by Baumgart [BAUM72; BAUM75]. As shown in Fig. 12.16, each edge in the winged-edge data structure is represented by pointers to its two vertices, to the two faces sharing the edge, and to four of the additional edges emanating from its vertices. Each vertex has a backward pointer to one of the edges emanating from it, whereas each face points to one of its edges. Note that we traverse the vertices of an edge in opposite directions when following the vertices of each of its two faces in clockwise order. Labeling the edge's vertices n and p, we refer to the face to its right when traversing the edge from n to p as its p face, and the face to its right when traversing the edge from p to n as its n face. For edge $E1$ in Fig. 12.16, if n is $V1$ and p is $V2$, then $F1$ is $E1$'s p face, and $F2$ is its n face. The four edges to which each edge points can be classified as follows. The two edges that share the edge's n vertex are the next (clockwise) edge of the n face, and the previous (counterclockwise) edge of the p face, $E3$ and $E2$, respectively. The two edges that share the edge's p vertex are the next (clockwise) edge of the p face, and the previous (counterclockwise) edge of the n face, $E4$ and $E5$, respectively. These four edges are the "wings" from which the winged-edge data structure gets its name.

Note that the data structure described here handles only faces that have no holes. This limitation can be removed by representing each face as a set of edge loops—a clockwise outer loop and zero or more counterclockwise inner loops for its holes—as described in

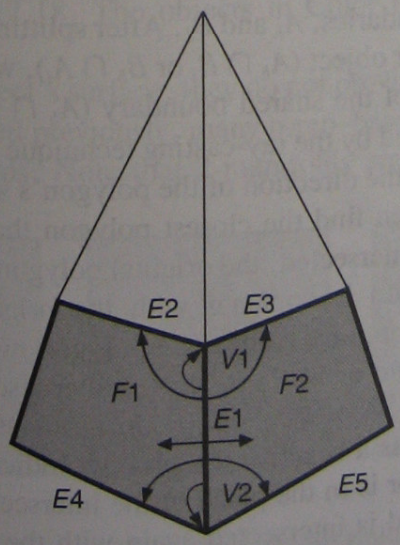


Fig. 12.16 Winged-edge data structure for $E1$. Each of $V1$, $V2$, $F1$, and $F2$ also have a backward pointer to one of their edges (not shown).

Section 19.1. Alternatively, a special auxiliary edge can be used to join each hole's boundary to the outer boundary. Each auxiliary edge is traversed twice, once in each direction, when a circuit of its face's edges is completed. Since an auxiliary edge has the same face on both of its sides, it can be easily identified because its two face pointers point to the same face.

A b-rep allows us to query which faces, edges, or vertices are adjacent to each face, edge, or vertex. These queries correspond to nine kinds of *adjacency relationships*. The winged-edge data structure makes it possible to determine in constant time which vertices or faces are associated with an edge. It takes longer to compute other adjacency relationships. One attractive property of the winged edge is that the data structures for the edges, faces, and vertices are each of a small, constant size. Only the number of instances of each data structure varies among objects. Weiler [WEIL85] and Woo [WOO85] discuss the space–time efficiency of the winged edge and a variety of alternative b-rep data structures.

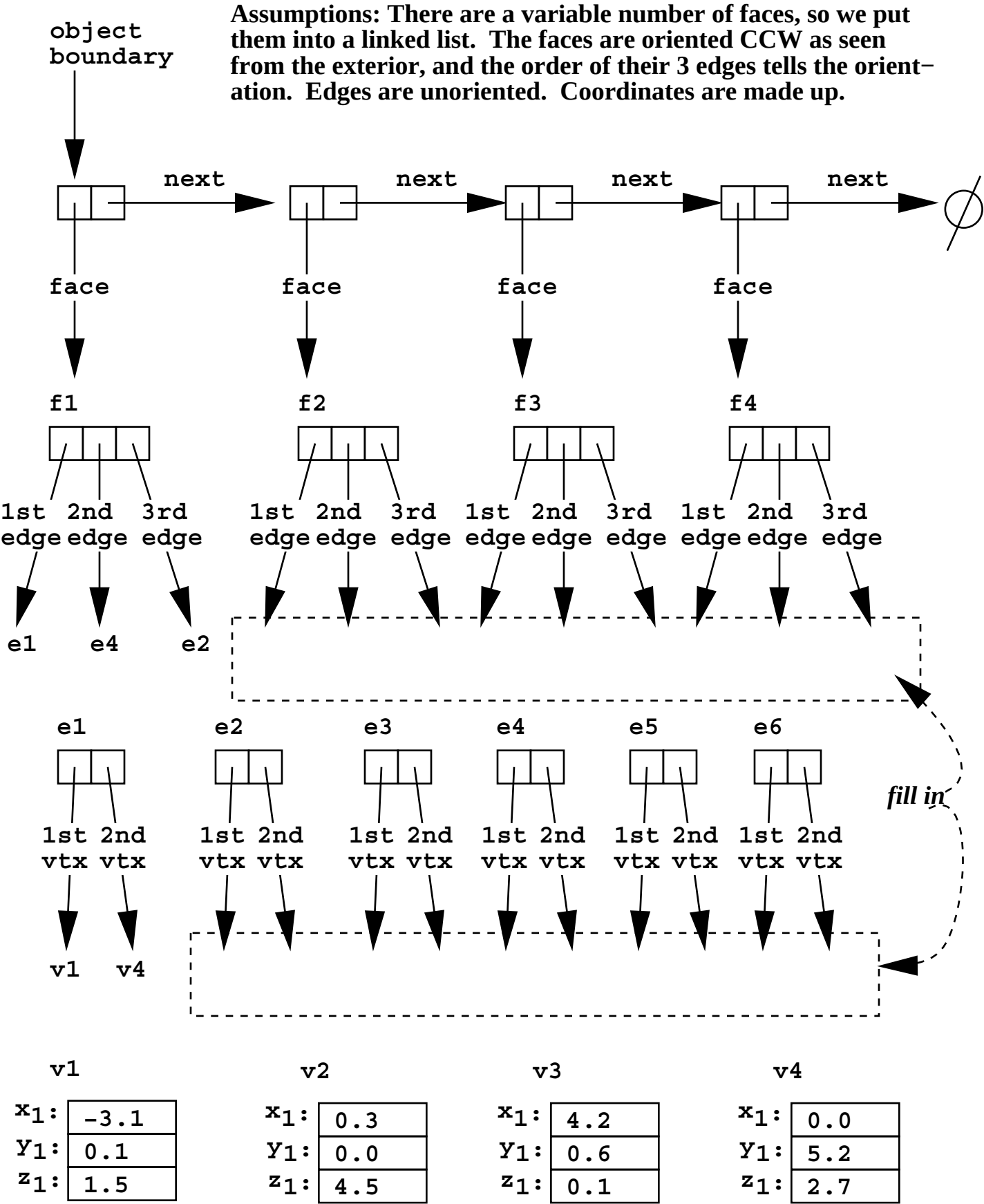
12.5.3 Boolean Set Operations

B-reps may be combined, using the regularized Boolean set operators, to create new b-reps [REQU85]. Sarraga [SARR83] and Miller [MILL87] discuss algorithms that determine the intersections between quadric surfaces. Algorithms for combining polyhedral objects are presented in [TURN84; REQU85; PUTN86; LAID86], and Thibault and Naylor [THIB87] describe a method based on the binary space-partitioning tree representation of solids discussed in Section 12.6.4.

One approach [LAID86] is to inspect the polygons of both objects, splitting them if necessary to ensure that the intersection of a vertex, edge, or face of one object with any vertex, edge, or face of another, is a vertex, edge, or face of both. The polygons of each object are then classified relative to the other object to determine whether they lie inside, outside, or on its boundary. Referring back to Table 12.1, we note that since this is a b-rep, we are concerned with only the last six rows, each of which represents some part of one or both of the original object boundaries, A_b and B_b. After splitting, each polygon of one object is either wholly inside the other object ($A_b \cap B_i$ or $B_b \cap A_i$), wholly outside the other object ($A_b - B$ or $B_b - A$), or part of the shared boundary ($A_b \cap B_b$ *same* or $A_b \cap B_b$ *diff*).

A polygon may be classified by the ray-casting technique discussed in Section 15.10.1. Here, we construct a vector in the direction of the polygon's surface normal from a point in the polygon's interior, and then find the closest polygon that intersects the vector in the other object. If no polygon is intersected, the original polygon is outside the other object. If the closest intersecting polygon is coplanar with the original polygon, then this is a boundary–boundary intersection, and comparing polygon normals indicates what kind of intersection it is ($A_b \cap B_b$ *same* or $A_b \cap B_b$ *diff*). Otherwise, the dot product of the two polygons' normals is inspected. A positive dot product indicates that the original polygon is inside the other object, whereas a negative dot product indicates that it is outside. A zero dot product occurs if the vector is in the plane of the intersected polygon; in this case, the vector is perturbed slightly and is intersected again with the other object's polygons.

Vertex-adjacency information can be used to avoid the overhead of classifying each polygon in this way. If a polygon is adjacent to (i.e., shares vertices with) a classified


Assumptions: There are a variable number of faces, so we put them into a linked list. The faces are oriented CCW as seen from the exterior, and the order of their 3 edges tells the orient−ation. Edges are unoriented. Coordinates are made up.
object boundary
next
next
next
next
face
face
face
face
f1
f2
f3
f4
1st edge
2nd edge
3rd edge
e1
e4
e2
e1
e2
e3
e4
e5
e6
1st vtx
2nd vtx
v1
v4
fill in
v1
x1: -3.1
y1: 0.1
z1: 1.5
v2
x1: 0.3
y1: 0.0
z1: 4.5
v3
x1: 4.2
y1: 0.6
z1: 0.1
v4
x1: 0.0
y1: 5.2
z1: 2.7

what the answer to the following sample HW question might look like:

Q: write pseudocode that takes a ptr to an edge & outputs all adjacent faces for the Requicha b-rep.

A: Function FindFacesFromEdge {
output: a list of adjacent faces (pointers to them, actually)
input: "InputEdgePtr" as specified in problem, & assume also given as input "ObjectBoundaryPtr" which points to the first face — you can assume this for the Requicha problem in the HW too.

FacePtr = ObjectBoundaryPtr;
do { — assignment operator (use your favorite programming language syntax)
if {FacePtr → face → 1st edge == InputEdgePtr
— follow ptr. w/ this name; comparing if equal to (use your favorite programming language syntax)
or
else if FacePtr → face → 2nd edge == InputEdgePtr
or
else if FacePtr → face → 3rd edge == InputEdgePtr}
add FacePtr → face to output;
FacePtr = FacePtr → next
} while FacePtr ≠ ∅;
output the output ☺
}

Society for Industrial and Applied Mathematics
3600 University City Science Center • Philadelphia, PA 19104-2688
215-382-9800 • Fax 215-386-7999 • siam@siam.org • www.siam.org