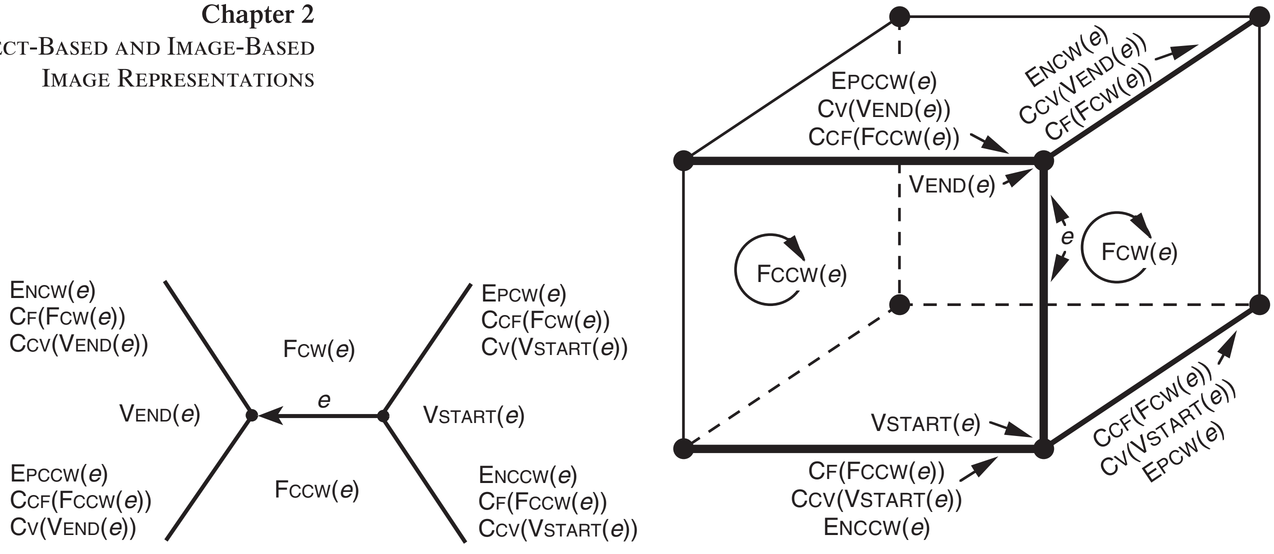


Figure 2.89
(a) Example of an edge $e$ and its four wings and (b) the physical interpretation of $e$ and its wings when $e$ is an edge of a parallelepiped represented by the winged-edge data structure.

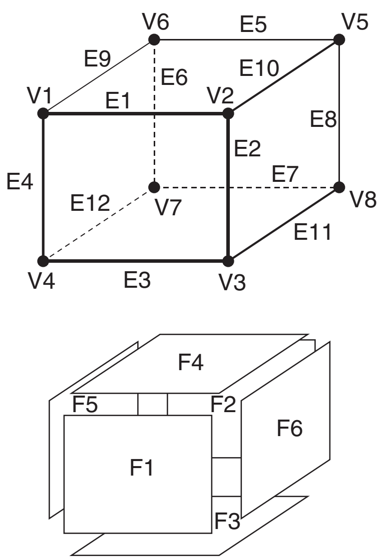


Figure 2.90
Sample parallelepiped with vertices (V1–V8), edges (E1–E2), and faces (F1–F6).

3. Edge-edge relation: the preceding edges (CCV(VSTART($e$)) and CCV(VEND($e$))) and the next edges (CV(VSTART($e$)) and CV(VEND($e$))) incident at the two vertices, thereby incorporating the vertex-edge relation, as well as the face-edge relation when using an alternative interpretation of the wings of the edge.

As an example of the winged-edge data structure, consider the parallelepiped in Figure 2.89(b), whose individual vertices, edges, and faces are labeled and oriented according to Figure 2.90. One possible implementation of a winged-edge representation for it is given by tables VERTEXEDGETABLE and FACEEDGETABLE in Figures 2.91 and 2.92, which correspond to the partial vertex-edge and face-edge relations, respectively, and the collection of edge records that make up the edge-edge relation given by Figure 2.93. Observe that VERTEXEDGETABLE and FACEEDGETABLE are really indexes (i.e., access structures) that enable an efficient response to queries on the basis of the value of a given vertex $v$ or face $f$, such as finding all of the edges incident at $v$ or the edges that $f$ comprises, respectively, in both the clockwise and counterclockwise orders. This information is accessed by the field EDGE as the tables may contain additional information, such as the actual $x$, $y$, and $z$ coordinate values of the vertex in the case of VERTEXEDGETABLE.

| VERTEX $v$ | X | Y | Z | EDGE |
|---|---|---|---|---|
| V1 | X1 | Y1 | Z1 | E1 |
| V2 | X2 | Y2 | Z2 | E2 |
| V3 | X3 | Y3 | Z3 | E3 |
| V4 | X4 | Y4 | Z4 | E4 |
| V5 | X5 | Y5 | Z5 | E5 |
| V6 | X6 | Y6 | Z6 | E6 |
| V7 | X7 | Y7 | Z7 | E7 |
| V8 | X8 | Y8 | Z8 | E8 |

Figure 2.91
VERTEXEDGETABLE[$v$].

| FACE $f$ | EDGE |
|---|---|
| F1 | E1 |
| F2 | E5 |
| F3 | E11 |
| F4 | E9 |
| F5 | E4 |
| F6 | E8 |

Figure 2.92
FACEEDGETABLE[$f$].

In particular, EDGE(VERTEXEDGETABLE[$v$]) = $e$ contains a pointer to an edge record $e$ that is incident at vertex $v$, while EDGE(FACEEDGETABLE[$f$]) = $e$ contains a pointer to an edge record $e$ that is part of face $f$.

It is also important to note that, given a pointer to an edge record $e$, the edge-edge relation makes use of fields CCFFCW($e$), CVVSTART($e$), CFFCW($e$), CCVVEND($e$), CCFFCCW($e$), CVVEND($e$), CFFCCW($e$), and CCVVSTART($e$), instead of CCF(FCW($e$)), CV(VSTART($e$)), CF(FCW($e$)), CCV(VEND($e$)), CCF(FCCW($e$)), CV(VEND($e$)), CF(FCCW($e$)), and CCV(VSTART($e$)), respectively. This is done in order to indicate that the pointer to the appropriate edge record in the corresponding field in the relation is obtained by storing it there explicitly instead of obtaining it by dynamically computing the relevant functions each time the field is accessed (e.g., the field CCFFCW($e$) stores the value CCF(FCW($e$)) directly rather than obtaining it by applying the function CCF to the result of applying FCW to $e$ each time this field is accessed).

A crucial observation is that the orientations of the edges are not given either in VERTEXEDGETABLE or FACEEDGETABLE or in the edge-edge relation. The absence of the orientation is compensated for by the presence of the VSTART, VEND, FCW, and FCCW fields in the edge-edge relation. This means that, given a face $f$ (vertex $v$), the edge $e$ stored in the corresponding FACEEDGETABLE (VERTEXEDGETABLE) entry is not sufficient by itself to indicate the next or previous edges in $f$ (incident at $v$) without checking whether $f$ is the value of the clockwise FCW($e$) or the counterclockwise FCCW($e$) face field ($v$ is the value of the start VSTART($e$) or end VEND($e$) vertex field) of record $e$ in the edge-edge relation. Thus, the algorithms that use this orientationless representation must always check the contents of FCW($e$) and FCCW($e$) (VSTART($e$) and VEND($e$)) for the use of face $f$ (vertex $v$). The same is also true upon making a transition from one edge to another edge when the edge has not been obtained from VERTEXEDGETABLE or FACEEDGETABLE.

As an example of the use of these tables, consider procedure EXTRACTEDGESOFFACE given below, which extracts the edges of face $f$ in either clockwise or counterclockwise order. Let $e$ denote an edge in $f$, obtained from FACEEDGETABLE, and use the interpretation of $e$'s wings as adjacent edges along adjacent faces of $e$ (i.e., the winged-edge-face variant). For a clockwise ordering, if $f$ = FCW($e$), then the next edge is CFFCW($e$); otherwise, $f$ = FCCW($e$), and the next edge is CFFCCW($e$). For a counterclockwise ordering, if $f$ = FCW($e$), then the next edge is CCFFCW($e$); otherwise,

| EDGE $e$ | VSTART | VEND | FCW | FCCW | CCFFCW<br>EPCW<br>CVVSTART | CFFCW<br>ENCW<br>CCVVEND | CCFFCCW<br>EPCCW<br>CVVEND | CFFCCW<br>ENCCW<br>CCVVSTART |
|---|---|---|---|---|---|---|---|---|
| E1 | V1 | V2 | F1 | F4 | E4 | E2 | E10 | E9 |
| E2 | V2 | V3 | F1 | F6 | E1 | E3 | E11 | E10 |
| E3 | V3 | V4 | F1 | F3 | E2 | E4 | E12 | E11 |
| E4 | V4 | V1 | F1 | F5 | E3 | E1 | E9 | E12 |
| E5 | V5 | V6 | F2 | F4 | E8 | E6 | E9 | E10 |
| E6 | V6 | V7 | F2 | F5 | E5 | E7 | E12 | E9 |
| E7 | V7 | V8 | F2 | F3 | E6 | E8 | E11 | E12 |
| E8 | V8 | V5 | F2 | F6 | E7 | E5 | E10 | E11 |
| E9 | V1 | V6 | F4 | F5 | E1 | E5 | E6 | E4 |
| E10 | V5 | V2 | F4 | F6 | E5 | E1 | E2 | E8 |
| E11 | V3 | V8 | F3 | F6 | E3 | E7 | E8 | E2 |
| E12 | V7 | V4 | F3 | F5 | E7 | E3 | E4 | E6 |

Figure 2.93
Edge-edge relation.

$f = \textsc{Fccw}(e)$, and the next edge is $\textsc{CcfFccw}(e)$. This process terminates when we encounter the initial value of $e$ again. For example, extracting the edges of face F1 in Figure 2.90 in clockwise order yields E1, E2, E3, and E4. The execution time of EXTRACTEDGESOFFACE is proportional to the number of edges in $f$ as each edge is obtained in $O(1)$ time. This is a direct consequence of the use of FACEEDGETABLE, without which we would have had to find the first edge by a brute-force (i.e., a sequential) search of the edge-edge relation. Similarly, by making use of VERTEXEDGETABLE to obtain an edge incident at vertex $v$, we can extract the edges incident at $v$ in time proportional to the total number of edges that are incident at $v$ as each edge can be obtained in $O(1)$ time (see Exercise 2).

```
procedure EXTRACTEDGESOFFACE(f, CWFlag)
/* Extract the edges making up face f in clockwise (counterclockwise) order
   if flag CWFlag is true (false). */
value face f
value Boolean CWFlag
pointer edge e, FirstEdge
e ← FirstEdge ← EDGE(FACEEDGETABLE[f])
do
   output e
   if CWFlag then
      e ← if FCW(e) = f then CFFCW(e)
            else CFFCCW(e)
            endif
   else e ← if FCW(e) = f then CCFFCW(e)
             else CCFFCCW(e)
             endif
   endif
   until e = FirstEdge
enddo
```

The above interpretations are not the only ones that are possible. Another interpretation, among many others, which finds much use, interprets the four wings in terms of the next edges at each of the faces that are adjacent to $e$ and the next edges incident at each of the two vertices that make up $e$. In this case, we have combined the interpretations of the wings $\textsc{Cf}(\textsc{Fcw}(e))$ and $\textsc{Cf}(\textsc{Fccw}(e))$ as used in the winged-edge-face data structure with the interpretations of the wings $\textsc{Cv}(\textsc{Vstart}(e))$ and $\textsc{Cv}(\textsc{Vend}(e))$ as used in the winged-edge-vertex data structure. The result is known as the *quad-edge data structure* [767]. It keeps track of both the edges that make up the faces in the clockwise direction and the edges that are incident at the vertices in the clockwise direction.

The quad-edge data structure is of particular interest because it automatically encodes the dual graph, which is formed by assigning a vertex to each face in the original graph and an arc to each edge between two faces of the original graph. In other words, we just need to interpret the cycles through the edges around the vertices in the original graph as faces in the dual graph and the cycles through the edges that the faces comprise in the original graph as vertices in the dual graph. In addition, the exterior face, if one exists, in the original graph is also assigned a vertex in the dual graph, which is connected to every face in the original graph that has a boundary edge. This makes the quad-edge data structure particularly attractive in applications where finding and working with the dual mesh is necessary or useful. For example, this is the case when the mesh corresponds to a Voronoi diagram whose dual is the Delaunay triangulation (DT). We discuss this further in Section 2.2.1.4. Another advantage of the quad-edge data structure over the winged-edge-face and winged-edge-vertex data structures is that, in its most general formulation, the quad-edge data structure permits making a distinction between the two sides of a surface, thereby allowing the same vertex to serve as the two endpoints of an edge, as well as allowing dangling edges, and so on.

From the above, we see that the winged-edge-face, winged-edge-vertex, and quad-edge data structures are identical in terms of the information that they store for each

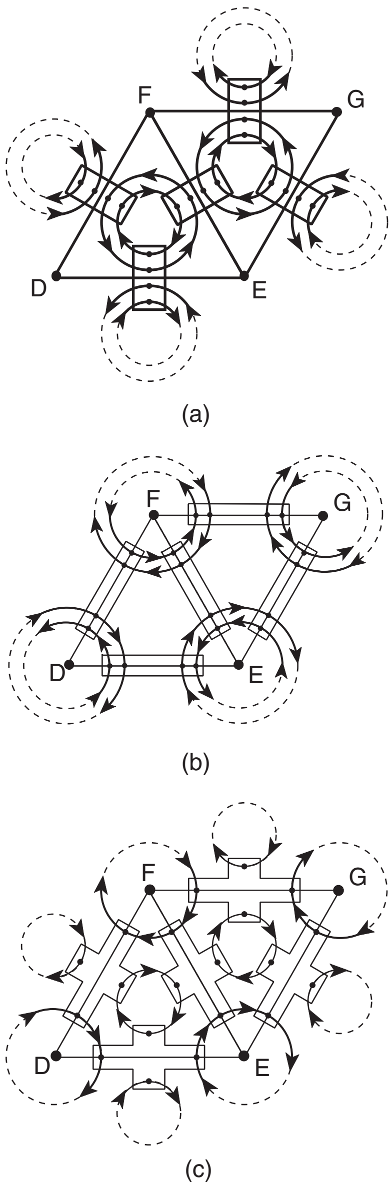


Figure 2.94
The physical interpretation of the (a) winged-edge-face, (b) winged-edge-vertex, and (c) quad-edge data structures for a pair of adjacent faces of a simple object. Assume an implementation that links the next and preceding edges in clockwise order for faces in (a), for vertices in (b), and next edges in clockwise order for both faces and vertices in (c).