Topological Structures for Spatial Databases in Two and Three Dimensions

T. H. Merrett

School of Computer Science
McGill University,
3480 University St., Montreal, Que. H3A 2A7, Canada,
Email: `tim@cs.mcgill.ca`

Abstract

In two dimensions, faces are cycles of edges and vertices are cycles of edges. In three dimensions, faces are cycles of edges and edges are cycles of faces. These insights provide the basis for topologies of components of any number of dimensions in two- and three- (and more-) dimensional spaces, as might be used for mapping, computer-aided design, etc. We give data structures and basic operations for two- and three-dimensional topologies.

Keywords: spatial data, topological model, 2D, 3D

1 Introduction

Although computer representations for spatial data have been available for over three decades, it does no harm, and may be beneficial, to think the subject through from first principles again from time to time. The multitudinous approaches that have been worked out (Samet 1990, Laurini & Thompson 1992) each have strengths and weaknesses, and alternatives which may combine the strengths of several should be investigated.

Spatial data cannot fully be represented merely as sets of vectors of coordinates. If computer memory, primary and secondary, had infinite capacity, and hence no round-off error, point data could be stored as such coordinate tuples, and coincidence of coordinates for two or more points would suffice to indicate coincidence of the points. Even so, useful information would be lost, because points are merely the zero-dimensional denizens of space. The one-dimensional denizens (edges), two-dimensional (faces), and so on, depending on the overall dimensionality of the space being considered, have relationships among themselves as well as with the points (vertices), and these should also be captured.

To represent these relationships requires a topological model as well as the metric data. This paper discusses a topological model. In order to get to essentials, it is preferable to follow ideas which are applicable to any number of dimensions, even though we will limit ourselves to the practical cases of topologies of ordinary spaces of only two or three dimensions. This should be enough to provide a basis for important applications such as map making, geographical information systems, and architectural and other three-dimensional computer-aided design systems.

Some of these applications, such as geographical maps, require very large amounts of data, and so we will consider data structures which are scalable to the point of being able to represent more data than can fit into RAM, the primary memory of computers. Even for smaller amounts of data, as might be needed to specify a building or an engineering CAD-CAM system, the abstractions required by processing on secondary storage provide helpful simplifications. For this reason, we need the experiences of databases rather than of conventional data structures.

The starting point for this paper is the "quad-edge" model of Guibas and Stolfi (Guibas & Stolfi 1985) for the topology of two dimensions. We will describe this in the next section and in section 3, but in terms suited to secondary storage rather than RAM. The quad-edge model captures the fact that both vertices and faces are cycles of edges in two dimensions. The resulting representation is sufficient for 2D structures on any surface, including the topological equivalents of spheres, toruses, etc., and it even offers structures on one-sided surfaces such as the Klein bottle. We will not go so far, but the ability to deal with surfaces with holes, at least, enables the quad-edge representation to cope with tunnels and bridges when it is modelling geospatial data.

Guibas and Stolfi's approach is important enough to be worth pushing to higher dimensions. The next section adapts their 2D representation to secondary storage and then presents the 3D extension. Section 3 discusses operations on the 2D structure, and section 4 introduces three-dimensional operations. We mention Guibas and Stolfi's very elegant approach at the end of section 3, but since it does not seem to extend to 3D, we mainly develop a more practical implementation. In section 5 we go on to consider disconnected topologies and holes, and in section 6 we relate the new data structures to some of the classical approaches.

2 Data Structures in 2D and 3D

Figures 1 and 2 give the topological properties of entities in, respectively, two and three dimensions. These entities are vertices (v), edges (e), faces (f) and "3-topoi" (^3t). The latter term is intended for volumetric cells, and is taken, in the same way as the word "polytope", from the Greek $\tau \acute{o} \pi o \sigma$ ("place"). The intention of this term, and of including both the two- and three-dimensional topologies in a single discussion, is to emphasize the extensibility of the ideas to any number of dimensions in fairly obvious ways.

The "# components" lines in each figure give the usual binomial coefficients. These represent the number of different basic lower-dimensional denizens of the respective 2D and 3D spaces. Thus, 2-space has two basic edges, which if normalized and orthogonal could form the conventional coordinate axes for metric data (except that we are not in this paper considering metrics but the more abstract topologies).

	v		e		f
# components	1		2		1
hasa$_1$		$\overset{\longleftarrow}{2}$		$\overset{\longleftarrow}{c}$	
hasa$_2$		$\overset{\longrightarrow}{c}$		$\overset{\longrightarrow}{2}$	
Euler	$v - e + f = 2$				

Figure 1: Two-dimensional Topology

	v		e		f		${}^3t$
# components	1		3		3		1
hasa$_1$		$\overset{\longleftarrow}{2}$		$\overset{\longleftarrow}{c}$		$\overset{\longleftarrow}{b}$	
hasa$_2$		$\overset{\longrightarrow}{b}$		$\overset{\longrightarrow}{c}$		$\overset{\longrightarrow}{2}$	
Euler	$v - e + f -{}^3 t = 0$						

Figure 2: Three-dimensional Topology

Similarly, 3-space has not only three basic edges, but also three basic faces: these might be visualized as the three planes formed by each possible pair of orthogonal basis edges (except, again, we are being more general than orthogonality, which is a metric property). The 1s for vertices and for faces (in 2-space) or for 3-topoi (in 3-space) indicate denizens which cannot be resolved into components.

The "hasa$_1$" lines include the left-pointing arrows above, and show how denizens relate to denizens of the immediately lower dimension. Thus, any edge has exactly two (2) vertices, its endpoints, in either two or three dimensions. Similarly, in both cases, any face has a cycle (c) of edges around its boundary. We will follow the convention that cycles are taken counterclockwise (just as, in a discussion of metrics, positive angles are measured counterclockwise). In three dimensions, a 3-topos is bounded by a ball (b) of faces, which have their own topological relationships among themselves, depending on shared edges, that we will need to be able to infer from the data structure we aim to build.

The "hasa$_2$" lines are *dual* to the "hasa$_1$" lines, in the usual topological sense. In d dimensions, $(d-k)$-dimensional entities are dual to k-dimensional entities: faces to vertices in 2D, 3-topoi to vertices and faces to edges in 3D. The "hasa$_2$" entries reverse the orders of the "hasa$_1$" entries. Any face in 3D separates exactly two (2) 3-topoi. Any edge in 2D separates exactly two (2) faces, but any edge in 3D is surrounded by a cycle (c) of faces (in the 2D special case, the cycle has two elements). Any vertex in 2D terminates a cycle (c) of edges, but vertices in 3D are emballed (b) by edges, which relate to each other through shared faces. The duality of these statements with those in the previous paragraph can be seen by substituting the names of dual entities and suitably modifying the names of the relationships.

Finally, we give, for each case, the Euler relationships (for connected topologies and embedding spaces without holes). In two dimensions, the surface containing the entities to be described, must be topologically equivalent to a sphere for the given Euler relationship to hold. This includes an infinite planar surface if all directions are imagined to attain the same "point at infinity". Thus, a triangle in 2D has 3 vertices, 3 edges and 2 faces (internal and external): $3-3+2 = 2$. (A circle must have a vertex, which both starts and ends its bounding edge.) There are analogous considerations in three dimensions. The Euler relationships provide a check on our thinking about spatial entities, but we do not explicitly use them in the paper and so do not extend them to embedding spaces with holes.

Everything else in figures 1 and 2 is independent of the nature of the embedding space.

These figures guide us to the data structures we must build to capture two- and three-dimensional topologies in full generality. The first is the 2D structure proposed by Guibas and Stolfi (Guibas & Stolfi 1985). Here is the "quad-edge" representation of the square shown in figure 3 in a form suited for secondary storage, i.e., relational.

QE(*vf*	*seqE*	*edge*	*dirE*)
1	1	a	0
1	2	d	2
2	1	b	0
2	2	a	2
3	1	c	0
3	2	b	2
4	1	d	0
4	2	c	2
F	1	a	3
F	2	b	3
F	3	c	3
F	4	d	3
B	1	d	1
B	2	c	1
B	3	b	1
B	4	a	1

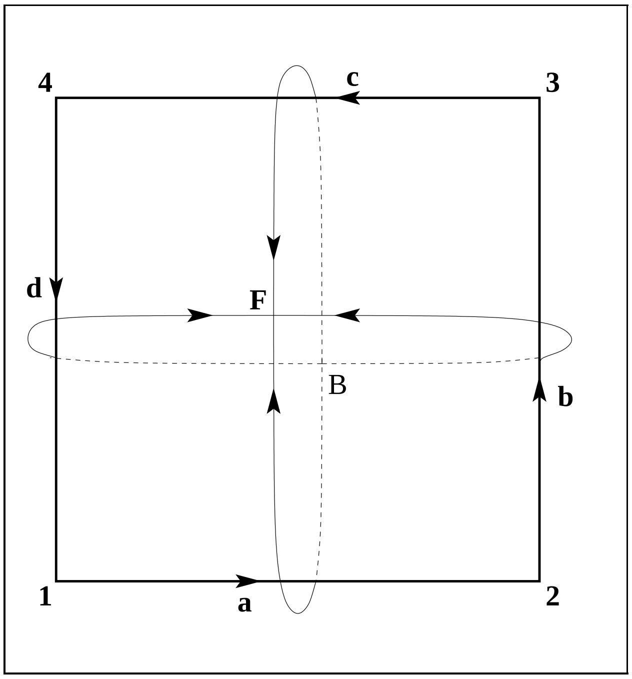

Figure 3: Two-dimensional Square

Since figure 1 shows that vertices and faces are dual and that both are cycles of edges, it makes sense to combine vertices and faces into the one attribute, *vf*, and for each vertex and face to show the sequence, *seqE*, of edges, *edge* that make up the cycle. (We will understand the sequence numbers to be cyclic, so that 1 follows 4 in each case.) Each edge clearly must appear four times in the data structure, once for each endpoint vertex and once for each face it separates: this follows from the two 2 entries in figure 1, hence the name, "quad-edge". Guibas and Stolfi thus supposed that each edge can have four "directions" associated with it, and they assigned 0 as the vertex-direction of an edge leaving the vertex, 2 as the vertex-direction of an edge arriving at the vertex, 1 as the face-direction of the edge the face is to the right of, and 3 as the face-direction of the edge the face is to the left of. This is a useful labelling, because it allows the representation to encode directions, and because any tuple with an even *dirE* then describes a vertex in *vf*, while any tuple with an odd *dirE* describes a face. Note that the odd directions are at right angles to the even, and that the sequence, 0, 1, 2, 3, of directions forms a counterclockwise complete revolution. Figure 3 shows the face-directions as thin lines.

We will describe operations on this structure in section 3.

Do not be distracted by the apparent three-dimensional nature of figure 3. The "back" face, `B`, shown behind the "front" face, `F`, is presented there as if the square were drawn on a sphere. It could equally well be the face outside the square, on an infinite plane. This just underlines that, topologically, the exterior face has the same footing as the interior.

A truly three-dimensional entity can have more than two faces per edge. As figure 2 shows, edges are cycles of faces as well as faces being cycles of edges. We illustrate this in figure 4, which shows a square "sandwich" with a diagonal cut, C, which we also consider to be a face.

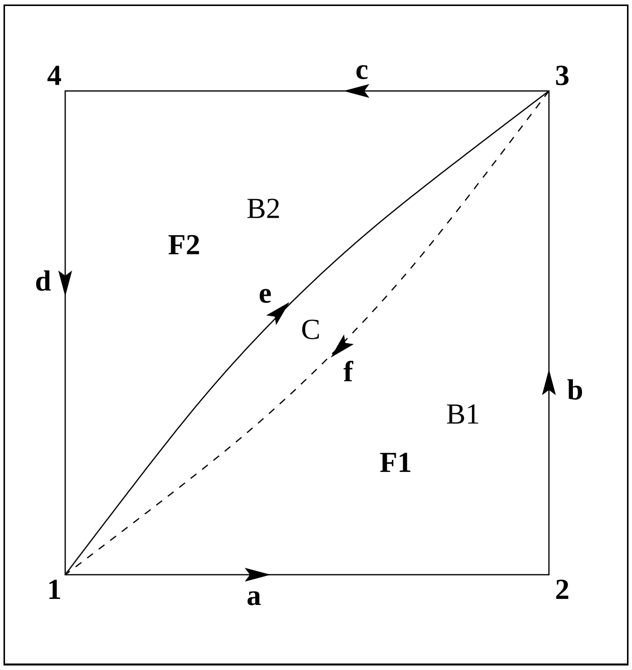

Figure 4: Three-dimensional Square With Cut Face

The data structure dictated by figure 2 takes three relations. The first two, *VE* and *TF*, assign directions to edges, and, since faces are duals of edges, to faces in an analogous way.

VE(*vertex*	*edge*	*dirE*)
1	a	0
1	d	2
1	e	0
1	f	2
2	a	2
2	b	0
3	b	2
3	c	0
3	e	2
3	f	0
4	c	2
4	d	0

TF(*topos*	*face*	*dirF*)
IN1	F1	0
IN1	B1	0
IN1	C	2
IN2	F2	0
IN2	B2	0
IN2	C	0
OUT	F1	2
OUT	F2	2
OUT	B1	2
OUT	B2	2

Here, the edge directions from vertex to vertex are as shown in figure 4 with *dirE* set to `0` for its source vertex and `2` for the destination. The topoi are not shown, to keep the diagram uncluttered, so we now describe them. As relation *TF* indicates, there are two interior topoi: `IN1` bounded by faces `F1` in front, *B1* behind and the "cut", `C`; and `IN2` bounded by faces `F2` in front, *B2* behind and `C`. Since each face separates exactly two topoi, the directions of faces are defined from topos to topos, with the sources and sinks in this example chosen arbitrarily to be from interior topoi to `OUT` through the front and back faces, and from `IN2` to `IN1` through face `C`. Thus each face has *dirF* `0` for its source topos, and `2` for the sink topos.

The final relation, *FE*, shows both faces as cycles of edges and edges as cycles of faces.

FE(*face*	*dirF*	*seqF*	*seqE*	*edge*	*dirE*)
F1	3	2	1	a	3
F1	3	2	2	b	3
F1	1	1	3	e	1
F2	3	2	1	c	3
F2	3	2	2	d	3
F2	3	3	3	e	3
B1	1	1	1	f	1
B1	1	1	2	b	1
B1	1	1	3	a	1
B2	3	3	1	f	3
B2	1	1	2	d	1
B2	1	1	3	c	1
C	1	2	1	e	1
C	1	2	2	f	1

This shows sequences for both *edge* and *face*: because they are cycles, it does not matter where the sequences start, but we have supposed that *seqF* starts on an outside face and passes through the interior, ending on another outside face, and that *seqE* follows edges in alphabetical order. (The way the tuples are ordered in the above display of *FE*, it is easier to see the cycles of edges, but tuple order does not matter in a relation, and the display is equivalent to one rearranged to bring out the cycles of faces.)

Edges have the directions already shown relative to their endpoints. Their directions relative to the faces they bound (the odd-numbered directions in *dirE*) can no longer be determined by (thin) lines crossing from one face to another because of the possibility that more than two faces meet at an edge. But since we assigned a direction to each face, from the topos on one side to the topos on the other, then both the directions of edges relative to faces, *dirE*, and the directions of faces relative to edges, *dirF*, can be given by invoking a right-hand, or counterclockwise, rule. We will say that if the direction of the face opposes the right-hand direction around the edge, then the edge-direction of the face, *dirF*, is 1 otherwise it is 3. Similarly, if the direction of the edge opposes the right-hand direction of the face, then the face-direction of the edge, *dirE*, is 1 otherwise it is 3. The "right-hand direction" is the direction the fingers of the right hand point when its thumb points along the direction of the face or the edge as given in relations *VE* and *TF*, respectively.

We can refine the topologies relating "balls" of edges around vertices and of faces around topoi by joining *FE* with *VE* and *TF*, respectively. Thus, *FE* **join** *VE* reveals, for vertex 1, for instance, the connections $(a,e),(a,f),(d,e)$ and (d,f) between pairs of edges that share faces, and similarly for the other vertices. (Note that (e,f) connect through both vertices 1 and 3.) Correspondingly, *FE* **join** *TF* gives the pairs of faces (F1,B1), (F1,B1) and (F1,B1) that share edges for topos IN1, and so on.

These examples conclude our discussion of the two-and three-dimensional data structures to capture the general topology. It can be formalized at the expense of intuitive clarity. In keeping with the name "quad-edge" for the 2D structure, we can call the three relations of the 3D structure "face-edge", reflecting the cycles that are central to the structure. Data structures alone are of small use, so we must now consider operations on them.

3 Operations in 2D

Guibas and Stolfi (Guibas & Stolfi 1985) assert that a single, simple, symmetric and self-inverting operator is sufficient for two-dimensional processing. We illustrate and implement it now on relation *QE* from section 2.

Guibas and Stolfi's single operator, $splice(p,q)$, works on edge-direction pairs, $p{=}(edge_p, dirE_p)$ and $q{=}(edge_q, dirE_q)$, and consists of two swaps (although that they are swaps will not be apparent from our initial implementation).

$$\begin{array}{l} splice((e,d),(e',d'))\ \mathbf{is} \\ \{\ swapAfter((e,d),(e',d')); \\ \ \ swapBefore((e,(d{-}1)\bmod 4),(e',(d'{-}1)\bmod 4)) \\ \} \end{array}$$

Here the subtraction mod 4 takes direct advantage of the way we indexed the four edge-directions in section 2.

As an example, we open the square of figure 3 at vertex 2, using the *QE* representation: *splice*((a,2),(b,0)) = {*swapAfter*((a,2),(b,0)); *swapBefore*((a,1),(b,3))}. The first swap will operate on the vertex part of *QE*, and will generate a new vertex whose name we will suppose is also supplied as a parameter, *VF*, along with the parameters of *splice*(). The second operates on the face part and will have the necessary effect of fusing the two faces F and B.

In our implementation (not Guibas and Stolfi's),

> *swapAfter*(*VF*)(p,q) **is**
> **if** p,q are in different cycles
> (i.e. have different *vf* values)
> **then** resequence *seqE* for p,q
> so that each is last in its respective cycle;
> merge the cycles by adding the highest
> *seqE* value for one cycle to all *seqE*
> values for the other and by changing
> *vf* ← *VF* for both
> **else** resequence *seqE* for q
> so that it is last in the cycle;
> split the cycle into two by subtracting the
> *seqE* value for p from all larger *seqE*
> values and by changing *vf* ← *VF*
> for these tuples.

The algorithm for *swapBefore*() is identical except that "last" becomes "first" in both the **then** and **else** clauses, and the *seqE* value subtracted (in the **else** clause) becomes that of the predecessor of p.

Here is the effect on *QE* of *splice*(5)(F)((a,2),(b,0)) (where the 5 is the *vf* parameter for *swapAfter*() and the F is for *swapBefore*().

QE(*vf*	*seqE*	*edge*	*dirE*)
1	1	a	0
1	2	d	2
2→5	1→2→1	b	0
2	2→1	a	2
3	1	c	0
3	2	b	2
4	1	d	0
4	2	c	2
F	1→4	a	3
F	2→1	b	3
F	3→2	c	3
F	4→3	d	3
B→F	1→2→6	d	1
B→F	2→3→7	c	1
B→F	3→4→8	b	1
B→F	4→1→5	a	1

We have coded this algorithm in a database programming language with a suitably flexible relational algebra (don't try it in SQL!), but providing the background to explain even the brief result is beyond the scope of this paper.

To reverse the process and close up the square again, we repeat the *splice*() operation, changing only the two name parameters:

splice(2)(B)((a,2),(b,0)).

The fused vertices are now named 2 again, and the face B reappears. Apart from this naming, the operation is its own inverse.

As a further exploration, try converting the square to a pyramid (remember, a pyramid is a two-dimensional topological structure if it is taken to be a partitioning of the sphere), using, in addition to *QE* from section 2, the following supplies

Supplies(*vf*	*seqE*	*edge*	*dirE*)
1′	1	e	0
2′	1	f	0
3′	1	g	0
4′	1	h	0
5	1	e	2
5	2	f	2
5	3	g	2
5	4	h	2
F′	1	h	1
F′	2	h	3
F′	3	g	1
F′	4	g	3
F′	5	f	1
F′	6	f	3
F′	7	e	1
F′	8	e	3

and the four operations

splice(1)(F1)((a,0),(e,0))
splice(2)(F2)((b,0),(f,0))
splice(3)(F3)((c,0),(g,0))
splice(4)(F4)((d,0),(h,0))

You can then go on to build, for example, a tetrahedron from the pyramid, using four splices, and a triangular prism from the tetrahedron, using eight splices.

We promised a mention of Guibas and Stolfi's more elegant but less practical implementation, which also makes clear that the two components of the *splice*() operator are indeed swaps. This requres us to replace the "enumerated sequence" representation of cycles in *QE* by an "element pair" representation. For the square of figure 3 we show this for both faces but for only vertex 2 among the vertices. Note that faces and vertices are not explicitly named in the enumerated sequence representation, and that the cycles alone suffice.

ElementPair

($edge_p$	$dirE_p$	$edge_q$	$dirE_q$)
:	:	:	:
a	2	b	0
b	0	a	2
:	:	:	:
a	3	b	3
b	3	c	3
c	3	d	3
d	3	a	3
d	1	c	1
c	1	b	1
b	1	a	1
a	1	d	1

SwapAfter((a,2),(b,0)) operates on the ($edge_p$,$dirE_p$) columns and swaps (a,2) with (b,0) there: clearly this produces two cycles of one element each, namely the two new, free endpoints replacing vertex 2.

SwapBefore((a,1),(b,3)) operates on the ($edge_p$,$dirE_p$) columns and swaps (a,1) with (b,3), making a single sycle of eight elements, namely the fusion of faces F and B. (Try drawing the new thin lines that result in the opened square!)

The impracticality of this appealing approach is that it does not easily allow the user to specify the new name(s) needed (for example, for a new vertex). (For full treatment, not only is the name needed, but also the metric data, such as the coordinates for vertex 5.) It also does not extend to three dimensions.

4 Operations in 3D

In the two-dimensional, *quad-edge*, representation, vertices and faces are each cycles of edges, so two complementary *swap*() operations suffice as the basic update. In three dimensions, the *face-edge* representation holds mutual cycles in *FE* and the other, noncyclic links in *VE* and *TF*. Updates consist of *swap*()s for insertion into and removal from cycles, and further operations, not self-invertible, to deal with the noncyclic links.

In two dimensions, the dual of an operation is its own inverse, as in the case, discussed in section 3, of opening and closing a square. This example may be summarized as

> **Operation** split vertex, fuse faces;
> **Dual/inverse** split face, fuse vertices.

In three dimensions, we may have four distinct possibilities, as for example

> **Operation** add face, split topos;
> **Inverse** remove face, fuse topoi;
> **Dual** add edge, split vertex;
> **Inverse of dual** remove edge, fuse vertices.

We will here only illustrate the 3D update operations for an instance of this particular family, leaving formalization for another publication. We start with the **Inverse** case, removing face C from the "sandwich" of figure 4.

The update is

> In *FE*:
> *swapAfter*(e′)((F1,1),(C,1))
> *swapAfter*(f′)((B1,1),(C,1))
> In *TF*:
> Rename IN2 to IN1
> Everywhere:
> Delete tuples containing C

Here, we are removing face C from after face F1 in the cycle around edge e, and from after face B1 in the cycle around edge f. *SwapAfter*() is as specified in section 3, adapted for cycles of faces around an edge. (We could have used *swapBefore*() instead, discarding edges e and f and keeping edges named e′ and f′.) Topoi IN1 and IN2 can be found associated with C in *FT* and we can suppose that the user has specified IN1 as the name of the fused topos.

The result, before deleting C, is

FEnoC(*face*	*dirF*	*seqF*	*seqE*	*edge*	*dirE*)
F1	3	2	1	a	3
F1	3	2	2	b	3
F1	1	1	3	e	1
F2	3	2	1	c	3
F2	3	2	2	d	3
F2	3	1	3	e	3
B1	1	2	1	f	1
B1	1	1	2	b	1
B1	1	1	3	a	1
B2	3	1	1	f	3
B2	1	1	2	d	1
B2	1	1	3	c	1
C	1	1	1	e′	1
C	1	1	2	f′	1

and

VE(*vertex*	*edge*	*dirE*)
1	a	0
1	d	2
1	e	0
1	f	2
2	a	2
2	b	0
3	b	2
3	c	0
3	e	2
3	f	0
4	c	2
4	d	0

TFnoC(*topos*	*face*	*dirF*)
IN1	F1	0
IN1	B1	0
IN1	C	2
IN1	F2	0
IN1	B2	0
IN1	C	0
OUT	F1	2
OUT	F2	2
OUT	B1	2
OUT	B2	2

Then all C tuples are removed to give the final result.

The inverse of this, namely the operation that inserts face C and splits topos IN1 into IN1 and IN2, starts with the above final result and provides, for *FE*, the additional tuples

(*face*	*dirF*	*seqF*	*seqE*	*edge*	*dirE*)
C	1	1	1	e′	1
C	1	1	2	f′	1

and, for *TF*

(*topos*	*face*	*dirF*)
IN1	C	2
IN2	C	0

The update is

In *FE*:
  swapAfter(e)((F1,1),(C,1))
  swapAfter(f)((B1,1),(C,1))
Using *TF* and *FE*, find faces of topos IN1 linked to F1 or B1 via any edge except e or f and then, in *TF*, change the topos for any *other* face to IN2

SwapAfter() effectively undoes its previous invocations on the same faces. The operation on *TF* finds all faces that will bound the new topos and renames them (actually, it works through the complement); the inputs e, f and IN2 are supplied to the swaps, or can be found in the new *TF* tuples for C.

The dual operation splits a vertex by adding an edge. Given additional tuples for *FE*

(*face*	*dirF*	*seqF*	*seqE*	*edge*	*dirE*)
F1′	1	1	1	g	1
F2′	3	2	1	g	3

and, for *VE*

(*vertex*	*edge*	*dirE*)
5	g	2
1	g	0

the operation to split vertex 1, by inserting a new edge g, to a new vertex 5, from which edge e continues to vertex 3 as before is

In *FE*:
  swapAfter(F1)((e,1),(g,1))
  swapAfter(F2)((d,3),(g,3))
Using *VE* and *FE*, find edges of vertex 1 linked to e or d via any face except F1 or F2 and then, in *VE*, change the vertex for any *other* edge to 5

Comparison with the previous insertion, of face C, shows the duality.

The inverse of the dual is the dual of the inverse, i.e., of the first update discussed in this section.

There are further families of examples, whose implementation is now evident. (They do not exhaust the possibilites in 3D or the applications of our approach.) For instance, there are families (operation, inverse, dual, dual inverse) based on

1. add edge, split face;
2. add vertex, split edge;

and, more closely related to the 2D updates

3. remove face from edge (split edge, fuse topoi).

(In items (1) and (2), the inverse operations must check that the cardinalities are 2: 2-element cycles of edges (i.e., 2D) or faces in (1); 2-element collections around vertices or topoi in (2). Operations in (3) require only single swaps and simple renaming.)

It is instructive to visualize each of the operations in these families: try it!

5 Loops, Shells and Holes

Our discussion so far has been restricted to simple topologies. For example, faces are bounded by only one cycle of edges each, and we have not explicitly considered distinct solids. These are both examples of disconnected topologies. Our three-dimensional data structure copes with disconnected solids, e.g., two separate tetrahedra, ore even one tetrahedron inside another, without modification. Guibas and Stolfi's original two-dimensional structure, the element-pair representation at the end of section 2, can handle disconnected face boundaries without alteration, but our enumerated-sequence representations in both 2D and 3D must be augmented to distinguish the different cycles from each other that can now border a single face.

This is easy. We just add a *cycle* attribute, and include it an any treatment of faces, where *face* has hitherto been processed on its own. For instance, here is the representation of the three-dimensional flying saucer, with three "front" and one "back" (B) faces, shown in figure 5. (Note that each edge starts and ends on a single vertex.)

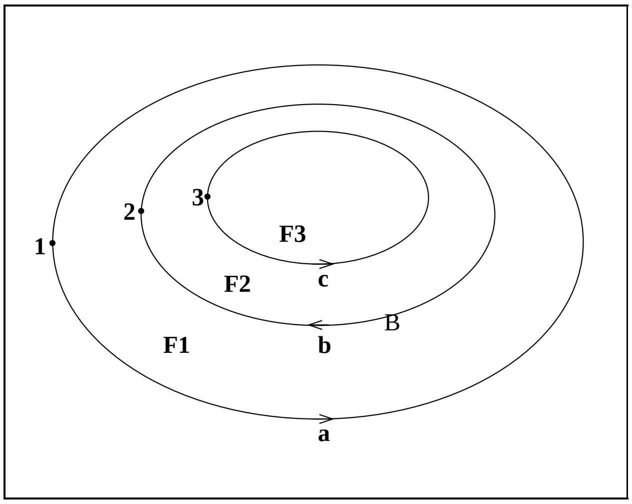

Figure 5: Three-dimensional Flying Saucer

VE(*vertex*	*edge*	*dirE*)
1	a	0
1	a	2
2	b	0
2	b	2
3	c	0
3	c	2

TF(*topos*	*face*	*dirF*)
IN	F1	2
IN	F2	2
IN	F3	2
IN	B	2
OUT	F1	2
OUT	F2	2
OUT	F3	2
OUT	B	2

FE(*cycle*	*face*	*dirF*	*seqF*	*seqE*	*edge*	*dirE*)
1	F1	3	2	1	a	3
2	F1	3	2	1	b	3
1	F2	1	2	1	b	1
2	F2	1	1	1	c	1
1	F3	3	2	1	c	1
1	B	1	1	1	a	3

In this minimal example, we use the *cycle* attribute to distinguish the two (one-element) cycles bounding face F1 and similarly for face F2. If these faces had been bounded by a couple of squares or triangles, the cycles would have had four or three elements each, but there would still be two cycles per face. A face with two islands in it would have three distinct bounding cycles, and so on.

A type of operation we have not so far considered includes converting the flying saucer to a doughnut with three faces, separated by edges a, b and c. Visually, we would do this by pushing face F3 down until it is completely flush with the central part of the back face, B, and then deleting F3 and the part of B it is in contact with: edge c now bounds faces F2 and B.

In terms of the data structure, the new doughnut has *VE* unchanged, *TF* unchanged except for the deletion of tuples pertaining to F3, and *FE* as follows.

FE(*cycle*	*face*	*dirF*	*seqF*	*seqE*	*edge*	*dirE*)
1	F1	3	2	1	a	3
2	F1	3	2	1	b	3
1	F2	1	2	1	b	1
2	F2	1	1	1	c	1
2	B	3	2	1	c	1
1	B	1	1	1	a	3

The change to *FE* is trivial: only the penultimate tuple shown has changed, by replacing F3 with B, and ensuring that face B now has two distinct cyles bounding it. We do not even need a swap operation.

6 Antecedents

One of the early topological models was DIME (Cooke & Maxfield 1967), the Dual Independent Map Encoding developed for the 1970s U.S. Censuses. It was meant for urban street networks, and identified for each street (edge) both the terminating intersections (vertices) and the blocks (faces) to the left and to the right. We can see that this presaged, for a particular application, parts of the present general approach.

In three dimensions, octrees, constructive solid geometry, and the boundary model have engaged the major efforts at representation and data structures. Since our work is closest to boundary models, we do not discuss the others.

The boundary model (b-rep) (Samet 1990, Mäntylä 1987) holds both metric and topological information about the boundaries of solids. Topologically, it follows Poincaré in extending the Euler equation (section 2) to topologies with any number of disconnected components in n-dimensional embedding spaces (Kline 1990). The model uses vertices, edges and faces, and adds two further topological elements which are aggregates: *loops* are cycles of edges, and allow faces to be bounded by more than one cycle; and *shells* are independent connected components. It includes the 5×5 possible pairwise relationships among these five topological entities. The model uses a specialization of the Euler-Poincaré equation in six variables (vertex, edge, face, shell, genus and hole, where *genus* gives the topology of the embedding space and *hole* describes holes in the solids). One equation in six variables describes a five-dimensional hyperplane in an abstract six-dimensional space, and so we need five "basis vectors" to described all possible solids represented by b-rep. These translate into five "Euler operators" (Baumgart 1974, Floriani 2003) used to create and alter solids in b-rep.

Apart from the initialization operator, our above discussion already includes these Euler operators. Three are in section 4: namely the "add edge, split vertex" dual of the "add face, split topos" operator worked out in that section; the "remove face, fuse topoi" inverse of the same; and the "add edge, split face" operator listed among the families at the end of the section. The fourth is the hole-punching operator of section 5 that converted flying saucer to torus in the example given there.

Thus, our face-edge representation is at least as powerful as b-rep. In fact it is more powerful, because b-rep does not support free faces or wire-frame structures (Floriani 2003). Here is the face-edge representation of the free face shown as a "sail" on the three-sided "football" on the left side of figure 6. (Note the similarity to the free edges in 2D after we opened the square in section 3.)

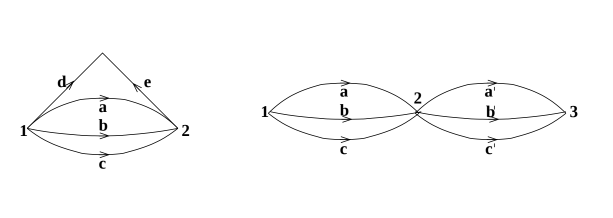

Figure 6: (l) Football with sail (r) Linked footballs

VE(*vertex*	*edge*	*dirE*)
1	a	0
1	b	0
1	c	0
2	a	2
2	b	2
2	c	2
1	d	0
2	e	0
3	d	2
3	e	2

TF(*topos*	*face*	*dirF*)
IN	ab	0
IN	bc	0
IN	ca	0
OUT	ab	2
OUT	bc	2
OUT	ca	2
OUT	ade	2
OUT	ade	0

FE(*cycle*	*face*	*dirF*	*seqF*	*seqE*	*edge*	*dirE*)
1	ab	1	1	1	a	1
1	ab	3	2	2	b	3
1	bc	1	1	1	b	1
1	bc	3	1	2	c	3
1	ca	1	2	1	c	1
1	ca	3	2	2	a	3
1	ade	3	3	1	a	3
1	ade	3	1	2	e	3
1	ade	1	1	3	d	1

Weiler (Weiler 1986*b*) gives a detailed discussion of three-dimensional entities which cannot be mapped to a two-dimensional manifold (which quad-edge, in our two-dimensional discussion, does). The right-hand part of figure 6 and both parts of figure 7 show examples adapted from that paper.

Here is the face-edge data structure for the two footballs, connected at a single vertex, shown on the right of figure 6. The first six tuples of each of *VE*, *TF* and *FE* are identical to the first six tuples of each for the single football just discussed, so we do not repeat them.

VE(*vertex*	*edge*	*dirE*)
⋮	⋮	⋮
2	a′	0
2	b′	0
2	c′	0
3	a′	2
3	b′	2
3	c′	2

TF(*topos*	*face*	*dirF*)
⋮	⋮	⋮
IN′	ab′	0
IN′	bc′	0
IN′	ca′	0
OUT	ab′	2
OUT	bc′	2
OUT	ca′	2

FE(*cycle*	*face*	*dirF*	*seqF*	*seqE*	*edge*	*dirE*)
⋮	⋮	⋮	⋮	⋮	⋮	⋮
1	ab′	1	1	1	a′	1
1	ab′	3	2	2	b′	3
1	bc′	1	1	1	b′	1
1	bc′	3	1	2	c′	3
1	ca′	1	2	1	c′	1
1	ca′	3	2	2	a′	3

The left side of figure 7 shows a similar construct, two cylinders with teardrop cross-sections, joined on the edge defined by the apexes of the tear.

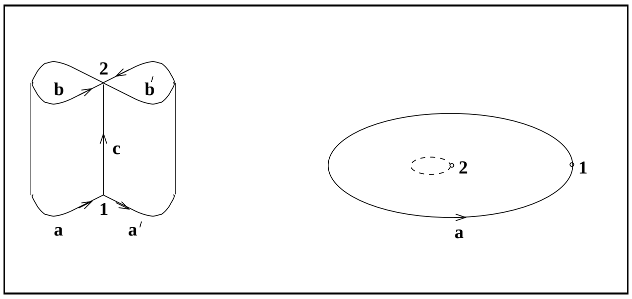

Figure 7: (l) Common edge (r) Vertex on face

VE(*vertex*	*edge*	*dirE*)
1	a	2
1	c	0
1	a′	0
2	b	2
2	c	2
2	b′	0

TF(*topos*	*face*	*dirF*)
IN	a	0
IN	b	0
IN	abc	0
IN′	a′	0
IN′	b′	0
IN′	abc′	0
OUT	a	2
OUT	b	2
OUT	abc	2
OUT	a′	2
OUT	a′	2
OUT	abc′	2

FE(*cycle*	*face*	*dirF*	*seqF*	*seqE*	*edge*	*dirE*)
1	a	1	1	1	a	1
1	b	3	2	1	b	3
1	abc	3	2	3	a	3
1	abc	1	1	2	c	1
1	abc	1	1	1	b	1
1	abc	3	2	4	c	3
1	a′	1	1	1	a′	1
1	b′	3	2	1	b′	3
1	abc′	3	2	3	a′	3
1	abc′	1	3	2	c′	1
1	abc′	1	1	1	b′	1
1	abc′	3	4	4	c′	3

Finally, from the right-hand side of figure 7, we represent a face with a vertex constrained to lie in the face. This requires either a fourth relation in the data structure, linking *face* with *vertex*, or a trick within the existing face-edge representation. The trick is indicated by the dashed edge in the figure, which has vertex 2 at both ends: we make a fake, single-edged face to be an inner boundary for the face including the vertex, and use the metric data (not covered in this paper) to ensure that it has vanishing area and circumference. In the data structure below, this face is called `f` (for "fake") and its bounding edge (dashed) is called `d`. (Note that there is only one topos in this two-dimensional structure, namely the surrounding space, `U`.)

VE(*vertex*	*edge*	*dirE*)
1	a	0
1	a	2
2	b	0
2	b	2

TF(*topos*	*face*	*dirF*)
U	T	0
U	T	2
U	F	0
U	F	2

FE(*cycle*	*face*	*dirF*	*seqF*	*seqE*	*edge*	*dirE*)
1	T	3	1	1	a	3
2	T	1	1	1	b	1
1	f	3	2	2	b	3

We have not completed an analysis of Weiler's extensions (Weiler 1986*a*) to the Euler operators, which we hope to show can all be captured by the operators of section 4.

Nice support is provided for our approach by Tse and Gold, whose paper (Tse & Gold 2003) we did not obtain until after completing this work. They show that the Euler operators in the two-dimensional, quad-edge, case are much simpler to implement than with the traditional b-rep data structures. Our present work, which has stong parallels to theirs, is not restricted to components forming 2D manifolds. (Since they are interested in application to cadastres, for which hidden faces are not observable, the fact that they cannot accommodate edges that are cycles of more than two faces is not a problem for their application.) They are working with TINs, triangulated irregular networks, and go on to show how to construct them with their Euler operators.

7 Conclusions

We have given a preliminary discussion of topological representations for multidimensional data on secondary storage. We have covered two-dimensional data on open or closed surfaces, omitting discussion of one-sided surfaces. We have introduced a representation for three dimensions and a set of basic operations.

Completing three dimensions and pressing on to higher dimensions are interesting future possiblilities, but more significant would be a detailed study of a range of applications and existing systems in the light of the approach in this paper. We have commenced this for geospatial data, and have yet to cover engineering and architectural CAD. Such a study would ground the work of completing the 3D operations. The present paper does not consider applications because we believe that, for generality and depth, the initial approach should deal purely with the properties of space itself.

Accounting for metric aspects is the other major omission of this paper. For linear spaces (straight edges, planar faces, etc.), the Clifford, or geometric, algebra (Bayliss 1996, Snygg 1997), which handles angles in any number of dimensions, is a clear complement, supplemented by a coordinate system. (In three dimensions, the additions to the data structure are particularly compact, since points, edges and faces each have three components.) This will enable the calculations of areas, volumes, masses, etc., as well as to determine if a free edge lies above or below a face, and so on.

8 Acknowledgements

We are indebted to the Networks of Centres of Excellence program for support through the GEOIDE Project, GEODEM, and the Natural Sciences and Engineering Research Council of Canada for support under grant OGP0004365. We are grateful to Chris Gold for insisting on the importance of the quad-edge method for geospatial data and for bringing to our attention the work of Guibas and Stolfi.

References

Baumgart, B. (1974), Geometric modelling for computer vision, Technical Report CS-463, Stanford U., Palo Alto, CA. Ph.D. thesis.

Bayliss, W. E., ed. (1996), *Clifford (Geometric) Algebras: With Applications in Physics, Mathematics, and Engineering*, Birkhaeuser, Boston.

Cooke, D. & Maxfield, W. (1967), The development of a geographic base file and its uses for mapping, *in* 'Papers from the Fifth Annual Conference of the

Urban and Regional Information Systems Association', ., pp. 207–19.

Floriani, L. D. (2003), Solid modeling: part two (boundary schemes). Lecture notes, URL=www.cs.umd.edu/class/fall2003/cmsc828D/solidmodeling-828D-2.pdf.

Guibas, L. & Stolfi, J. (1985), 'Primitives for the manipulation of general subdivisions and the computation of voronoi diagrams', *ACM Transactions on Graphics* **4**, 74–123.

Kline, M. (1990), *Mathematical Thought from Ancient to Modern Times*, Oxford University Press, New York and Oxford. In three volumes.

Laurini, R. & Thompson, D. (1992), *Fundamentals of Spatial Information Systems*, Academic Press, London.

Mäntylä, M. (1987), *An Introduction to Solid Modeling*, Computer Science Press, Rockville, Maryland.

Samet, H. (1990), *The Design and Analysis of Spatial Data Structures*, Addison-Wesley Publishing Company Ltd., Reading, Mass. & Don Mills, Ont.

Snygg, J. (1997), *Clifford Algebra: A Computational Tool for Physicists*, Oxford University Press, New York.

Tse, R. O. C. & Gold, C. (2003), 'A proposed connectivity-based model for a 3-d cadastre', *Computers, Environment and Urban Systems* **27**(4), 427–45.

Weiler, K. (1986*a*), Boundary graph operators for non-manifold geometric modeling topology representations, *in* M. J. Wozny, H. V. McLaughlin & J. L. Encarnaćâo, eds, 'IFIP 5.2 Working Conf. on Geometric Modelling for CAD Applications', North-Holland, Rensslaerville, N.Y., pp. 37–66. IFIP WG 5.2 Working Conference, Rensselaerville, NY, 12–14 May 1986.

Weiler, K. (1986*b*), The radial edge structure: A topological representation for non-manifold geometric modeling, *in* M. J. Wozny, H. V. McLaughlin & J. L. Encarnaćâo, eds, 'IFIP 5.2 Working Conf. on Geometric Modelling for CAD Applications', North-Holland, Rensslaerville, N.Y., pp. 3–36. IFIP WG 5.2 Working Conference, Rensselaerville, NY, 12–14 May 1986.