1990

Building Solid Models from Polygonal Data

George Vanecek

Report Number:
90-978

BUILDING SOLID MODELS FROM POLYGONAL DATA

George Vaněček, Jr.*

Computer Sciences Department
Purdue University
Technical Report CSD-TR-978
CAPO Report CER-90-20
May, 1990

* This work has been supported in part by NSF grant CCR-86-19817 to Purdue University.

Building Solid Models from Polygonal Data

George Vaněček, Jr.[1]
Department of Computer Science
Purdue University
West Lafayette, IN 47907

May 1, 1990

Abstract

Boundary representations (BRep) of three-dimensional solids are complex data structures that have to be created with topological and geometrical integrity. In many applications, the boundary of a solid may be specified as a list of polygons. In this paper, we consider the problem of constructing a BRep solid model from a list of polygons with the assumption that the points of each polygon lie in plane within some small tolerance, that two polygons do not penetrate each other, and that together the polygons enclose a volume. The algorithm works with any boundary representation based on directed edges and capable of representing nonmanifold edges and vertices, and accommodates a measure of numerical inaccuracy.

[1]This work has been supported in part by NSF Grant CCR-86-19817 to Purdue University.

1 Introduction

Boundary representations (BRep) of three-dimensional solids are complex data structures that have to be created with topological and geometrical integrity. Creating solids from instantiated parameterized primitives, and using Boolean set operations is one way of ensuring the integrity of the created data structures; e.g. [5]. In many applications, however, the boundary of a solid may have been specified as a list of polygons, where each polygon is given as a list of points in 3-space ordered counter-clockwise when viewed from outside the solid. For example, such polygon lists may be created by generating surfaces from contour data. Given the polygons, the construction of the solid model requires as an example, the creation of and matching up of unique vertices and edges, and simplifying the original boundary by removing adjacent coplanar faces and adjacent collinear edges. Topologically, this is a nontrivial task, both in terms of efficiency and correctness. The resulting boundary-based data structure may contain multiply connected faces, multiple shells, and nonmanifold edges and vertices [17].

In this paper, we consider the problem of constructing a minimal BRep from a list of polygons. The points of a polygon are assumed to lie in a plane within some small tolerance and two polygons that touch do so only at edges and vertices and with reasonable accuracy. Moreover, the polygons are not self-intersecting and together bound a volume. Finally, we assume that the points are specified with sufficient accuracy so that collinear edges and coplanar faces can be reasonably determined [12]. These assumptions are reasonable when the polygons are generated by methods that intend to yield solid models. As an example, the creation algorithm described here is used in a solid modeler called ProtoSolid [14] developed at University of Maryland and in Shilp [2], a geometric system, developed at Purdue University.

In ProtoSolid, the algorithm is used to construct the resulting solid of a Boolean set operation [13]. The Boolean set algorithm used in ProtoSolid fragments the faces of two solids such that no face of one solid penetrates the other solid. The faces of each solid are then classified as being either outside, or inside the other solid, or on the boundary of the other solid. This is referred to as boundary classification. The BRep data structure of the solid resulting form the set operation is created by copying the appropriate faces (i.e., polygons) from either of the two solids. To give an example of this process, the sphere of Figure2. was constructed by taking a unit cube, rotating it 45° about the x axis and intersecting. This is followed by 10 more rotations and intersections with the previous result, thus obtaining the sphere. Figure 1 shows the sphere before topological reduction. The sphere consisted of 9553 faces (input polygons), 20953 edges and 11432 vertices. After reduction, the sphere contains 3034 faces, 8236 edges, and 5204 vertices. The algorithm took 53 seconds for phase one and 11 seconds for phases two and three at the previous to last iteration, using a TI-Explorer II Lisp machine.

In Shilp, the algorithm converts medical data obtained from computed tomography, using a Symbolics 3620 Lisp machine. Triangulated boundaries are obtained using either the marching cubes approach [8] from 3d-grids, or from contour data [4, 3]. An example solid model of a left and a right lung is shown in Figure3.

Our algorithm will work for any boundary representation based on directed edges and capable of representing nonmanifold edges and vertices. This includes Vaněček's *fedge-based data structure* [15], Karasick's *star-edge data structure* [7, 6], and indirectly Weiler's radial-edge data structure [17]. In case that the input polygons form a manifold surface, Mäntylä's *half-edge data structure* is also included [9]. A boundary-based data structure can be viewed as a wire frame consisting of connected wire-edges, and vertices with pieces of canvas stretched across the wire edges, along with topological adjacency information [16]. Typically, the structure contains nodes of type *solid*, *shell*, *face*, *loop*, *edge*, *vertex* and *directed edge*, with each node pointing to some of its adjacent nodes. For access efficiency, the nodes of type solid usually reference all the vertex, edge, face, and shell nodes. As an example of such a data structure, Figures 4 and 5 show the necessary nodes and pointers to adjacent nodes representing the top face of a block. In the figures, FA, ED, DE, and VE represent the face, the edge, the directed edge and the vertex nodes.

The algorithm for constructing the solid model has three phases. In the first phase, each input

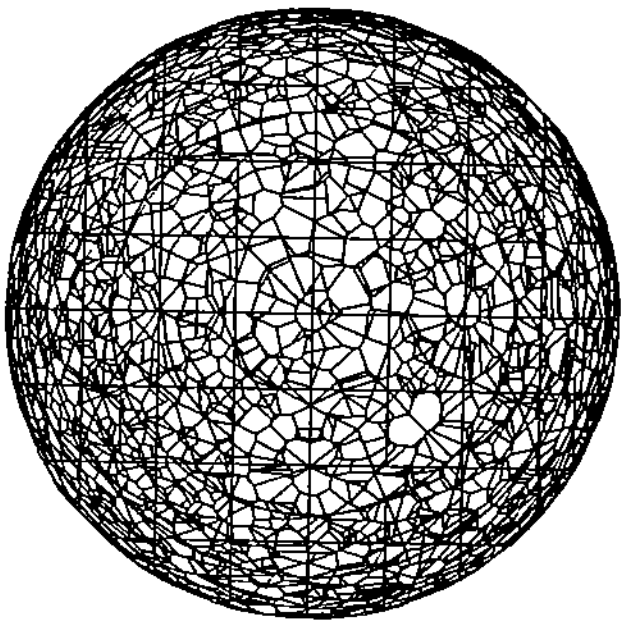


Figure 1: The BRep of a sphere before topological reduction consisting of 9553 faces, 20983 Edges, and 11432 vertices.

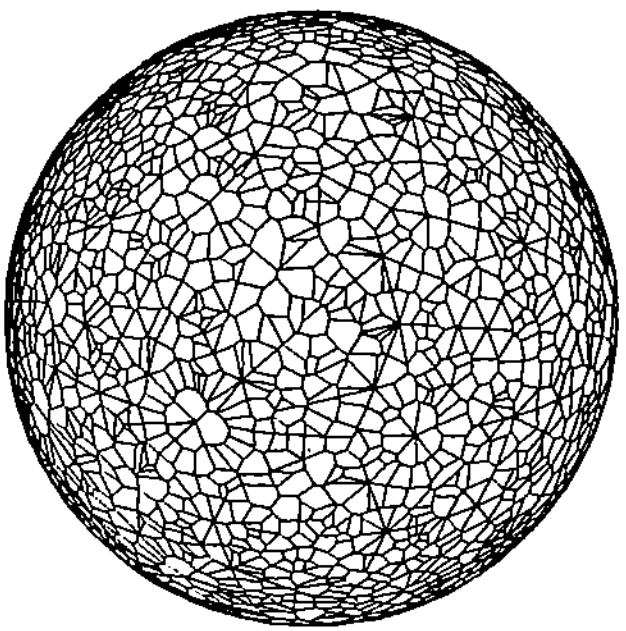


Figure 2: The minimal BRep after topological reduction of the sphere with 3034 faces, 8236 edges and 5204 vertices.

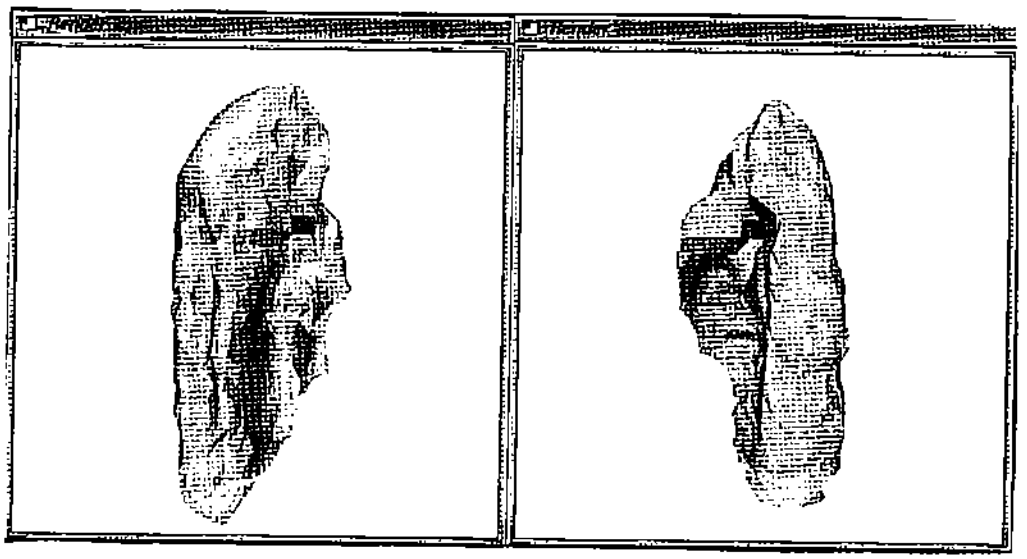


Figure 3: A solid model created from triangles generated from contours.

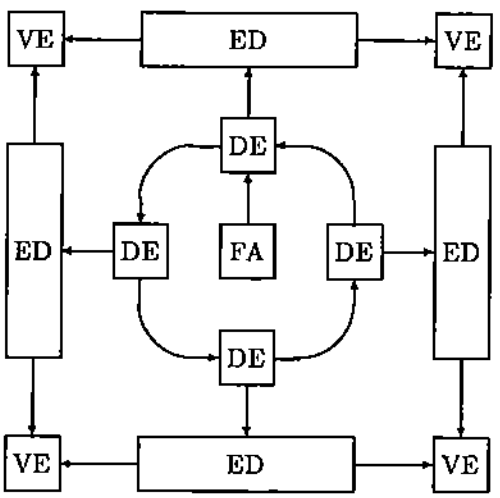


Figure 4: Diagram of a typical data structure showing the down pointers only, where FA, DE, ED and VE stand for face, directed edge, edge and vertex respectively. The directed-edge cycle is ordered counter-clockwise around the face.

polygon is simply inserted into the data structure as a new face by creating and properly linking the nodes necessary to represent the faces (i.e., face, directed edge, edge and vertex nodes). This construction generates a representation having adjacent coplanar faces and overlapping collinear edges. Thus the boundary can be viewed as being fragmented and containing gaps between adjacent faces.

In the second phase, the topological structure is reduced to obtain a minimal boundary representation, and its validity is established. The topological reduction consists of a sequence of local edge and vertex operations consisting of edge reductions that merge two adjacent and coplanar faces by removing the common edge, and vertex reductions that merge two adjacent collinear edges by removing the joining vertex. The result is a BRep with maximally connected faces that may have collinear and overlapping edges. Validity is established by merging all such edges.

The third phase separates faces with inner holes into multi-connected faces and partitions the faces into shells.

The three phases of the algorithm are described in detail after the definition of some terms are given in Section 2. In Section 3 a brief overview of the boundary-based data structure is given, and its initial creation from the polygons is given. A data structure for locating vertices in sublinear time is also presented. Section 4 gives the topological reduction of phase two of the algorithm and informally proves its termination. Topological validity is presented in Section 5. The removal of bridge edges and the creation of multi-connected faces is described in Section 6 and the partitioning of faces into shells follows in Section 7.

2 Terminology

We define some needed terms. Unless stated explicitly, vertices and edges are assumed to be manifold. We also distinguish points from vertices, and polygons from faces. A point is just a zero-dimensional geometrical entity in three-space while a vertex is a topological entity with edge and face adjacencies. A polygon and a corresponding face are similarly defined.

1. A *manifold vertex* is a vertex incident on one corner, i.e., a vertex with one edge-face cycle. A *nonmanifold vertex* is incident on more than one corner.

2. An *isthmus vertex* has exactly two adjacent collinear edges. (See Figure 15).

3. A *wire edge* is an edge without any incident faces.

4. A *lamina edge* is an edge incident to only one face. Both the wire edge and the lamina edge can exist temporarily during the creation of a valid BRep.

5. A *manifold edge* is an edge with two incident faces. A *nonmanifold edge* has $2n$ incident faces, for $n \geq 1$. It can be thought of as a composite of n manifold edges.

6. *Parallel edges* are collinear and overlapping and usually consist of lamina edges. In a manifold representation such as the half-edge data structure, parallel edges occur in place of a single non-manifold edge.

The following terms are for manifold edges in reference to a given face. These are referred to as *pseudo* (or *artifact* edges by some authors).

1. One of the vertices of a *strut edge* is adjacent to no other edges, and it is a manifold vertex. Thus a strut edge has two directed edges of the same face. (See Figure 12).

2. An *isthmus edge* has two incident faces that are coplanar. (See Figure 13).

3. A *bridge edge* has two directed edges of the same face [18]. (See Figure 12). Usually a bridge edge connects an inner edge loop to another loop of a multi-connected face.

A *fragmented BRep* is a BRep containing adjacent coplanar faces, and/or adjacent collinear edges—in other words, it contains pseudo entities. A *minimal BRep* does not contain pseudo entities, and it is not fragmented. It is not possible to remove edges or vertices from a minimal BRep without altering the solid or invalidating the representation.

3 Creation Phase

The first phase of the algorithm is the creation phase in which the faces are created from the input polygons. To do this, two operations are needed. The first operation finds a vertex given a point, and the second operation finds an edge given two vertices.

- The vertex-finding operation requires checking a given point against all currently known vertices. A brute-force approach would maintain the vertices in a list and find the correct vertex with a linear search. A more efficient approach is to search for the vertex using a balanced search tree, for example, based on a total ordering of points in 3-space. The search tree will be called *vertex directory* and is described later.

- Given two vertices, the edge-finding operation, based purely on topological adjacency information, finds the edge between the two vertices, and if one does not exist, creates it. New edges are created when they are needed by an adjacent face inserted first. Inserting the other adjacent faces causes the existing edge to be returned. The edge-finding operation takes the vertex with fewer number of adjacent edges and finds the correct one.

Initially, an empty *solid* node is created and with the use of the vertex, and the edge-finding operations, each input polygon is processed as follows:

1. Given the points $(p_0, \ldots, p_{n-1})$ of a polygon find (and if not found then create) the vertices $(v_0, \ldots, v_m)$ using the vertex-finding operation (described later). Note that m could be less than $n-1$, as described below. If $m < 3$ then skip the remaining steps.

2. Create and properly insert a new face node in to the BRep data structure.

3. For each vertex pair $(v_0, v_1), \ldots, (v_{n-1}, v_0)$, find the edge node between the vertices using the edge-finding operation described above, and for each edge node create a directed edge node and properly link it to the edge node, to the face node, and to the next and previous directed edge nodes if they already exist.

The robustness of this phase depends on the ability to handle ϵ-edges and ϵ-faces and match up points that are, although different, very close together. The vertex-finding algorithm declares two points coincident when their Euclidean distance is less than ϵ, for some small $\epsilon > 0$. With this, an ϵ-edge is an edge with no length, and an ϵ-face has zero area. As such, the n points of a polygon may yield m vertices, for $m < n$, of distinct adjacent vertices. Since adjacent vertices are not coincident, ϵ-edges are not created. Furthermore, if $m < 3$ vertices are determined from the input polygon points, then no corresponding face is constructed.

To facilitate the vertex-finding algorithm, a vertex directory is maintained. As new points are added to the directory, the points that fall within ϵ of existing vertices in the directory are ignored and the existing vertices are returned. Although this schema introduces certain anomalies that must be recognized and handled consistently, it does solve the problem of walking points. That is, if points are not handled consistently on a global level, declaring approximate point equality on a local level can cause points far apart to be incorrectly inferred as equal. Figure 6 illustrates this problem; the distance between q and p is more than 2ϵ but due to the two points in between, q is assumed to be equal to p by transitivity.

To avoid the walking point problem, the vertex directory is restricted to contain only unique points that are no closer than ϵ apart. Inserting a point into the directory causes either a new vertex node to be created, or an existing vertex to absorb the inserted point, if the inserted point

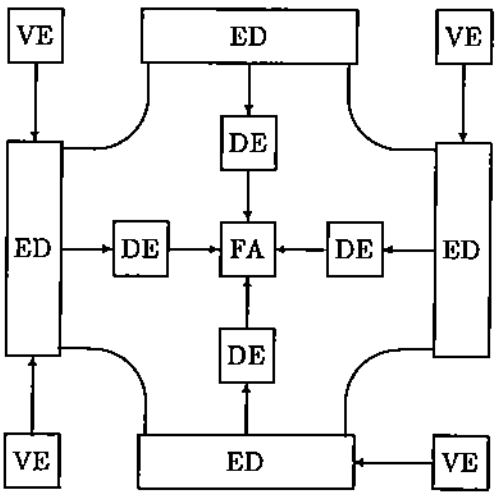


Figure 5: Diagram of the same data structure showing the up pointers only.

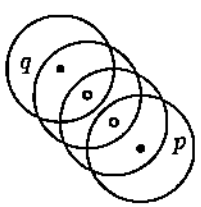


Figure 6: The problem of walking points.

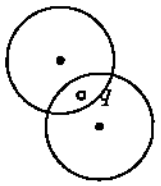


Figure 7: Ambiguity in point insertion

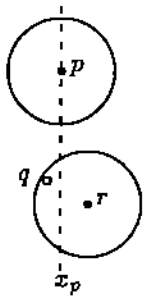


Figure 8: $q \prec p \prec r$; the need for searching both subtrees of p.

is approximately equal to it. Milenkovic calls this *data adaptation* [10]. When a new point is absorbed into an existing vertex, that information is returned to the face-creation algorithm which verifies the shifting of the given point.

However, it is not always possible to insert a new point without ambiguity, as illustrated in Figure 7. Although no two points can be approximately equal, two points can have overlapping ϵ-neighborhoods. In 3-space, a point can be approximately equal to at most eight vertices already in the directory. If a point to be inserted is approximately equal to more than one vertex, a conflict arises as to which of the vertices already in the directory the new point ought to belong. From the directory itself this cannot be determined. Topological assistance is necessary to resolve the conflict consistently, and so, the vertex-finding function returns all the vertices that are within ϵ of the point. The face-creation algorithm picks the vertex that is closest to the plane containing the face and that avoids, at the same time, ϵ-edges.

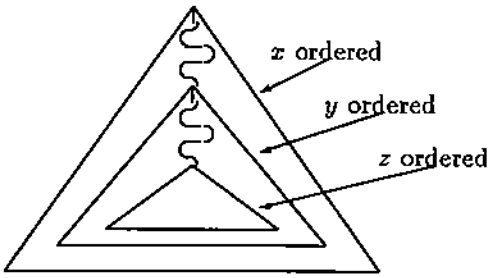
Imposing a total order on vertices allows for a more efficient implementation of a directory using a balanced binary search tree structure [1], than simple linear structures such as arrays, or linked lists. The ordering is based on the exact values of the points' coordinates. Given a vertex directory $\mathcal{D} = [v_1, \ldots v_n]$, the sequence of vertices is ordered by a precedence relation $\prec$, where $v_i \prec v_{i+1}$ for $i = 1, \ldots n-1$, with

$$\begin{aligned} p \prec q \quad \text{iff} \quad & (x_p < x_q) \text{ or} \\ & (x_p = x_q \text{ and } y_p < y_q) \text{ or} \\ & (x_p = x_q \text{ and } y_p = y_q \text{ and } z_p < z_q) \end{aligned} \tag{1}$$

It is easy to see that an inorder traversal of the directory generates the sequence $[v_1, \ldots, v_n]$ in ascending lexicographic order [11, p351].

Now consider locating all the vertices of $\mathcal{D}$ which are approximately equal to some query point q. Figure 8 illustrates a case in which $q \prec p \prec r$, and $q \approx r$, but $p \not\approx q$. That is, a search with point q is at a node in the tree corresponding to vertex p, and since $q \prec p$ the search should continue down the left subtree. However, the desired vertex r is in the right subtree of p.

This shows that the directory must be searched for all vertices with coordinates in the range $x_q - \epsilon \leq x \leq x_q + \epsilon$, $y_q - \epsilon \leq y \leq y_q + \epsilon$, and $z_q - \epsilon \leq z \leq z_q + \epsilon$. The usual search of a binary search tree which follows a single path from the root down to a leaf must be extended to follow



$$[p_1 \prec \ldots \underbrace{p_i \prec \ldots \overbrace{p_j \prec \ldots p_k}^{z \text{ ordered}} \prec \ldots p_l}_{y \text{ ordered}} \prec \ldots p_n]$$

Figure 9: Subtrees of D that are ordered by y and the z coordinates.

several paths. Therefore, for any vertex p in the tree for which the query point q has

$$x_p - \epsilon \leq x_q \leq x_p + \epsilon, \tag{2}$$

the search splits and continues downwards in both the left and the right subtrees. Specifically, the search continues down the left path if $x_q \leq x_p + \epsilon$, and the search continues down the right path if $x_q \geq x_p - \epsilon$.

The conditions of Eq. (2) are sufficient for finding all the vertices in the directory that are approximately equal to q. However, with only the conditions of Eq. (2), the search is based solely on the x coordinates. The search will take less than linear time only as long as all the vertices in the directory are spread out in space so that any two points are separated in x by at least ϵ—or at least there are very few of such points.

When a vertex directory is used in solid modeling, where the solids are represented with respect to a specific orientation, it is more likely that there are many points with the same x coordinate. In the worst case, a search of the directory could be linear under the conditions of Eq. (2). This is undesirable. Clearly searching through a sequence of points having the same x coordinates dictates that the y coordinates be referred to as well.

Because the directory is a sequence of points that is ordered in ascending lexicographic order, it has a special property. Given the ordered sequence of points $p_1 \ldots p_n$, as shown in Figure 9, there are subsequences $p_i, \ldots p_l$ with $1 \leq i \leq l \leq n$, for which the x coordinates alone or both the x and the y coordinates do not change. In other words, D is x-ordered with y-ordered subsequences.

Whenever the search enters a y-ordered subtree (as shown in Figure 10), the search proceeds down the left branch of some node p if $y_p \leq y_q + \epsilon$, and similarly proceeds down the right branch if $y_p \geq y_q - \epsilon$. Essentially this becomes a two-dimensional search in the plane $X = x_q$. Finally, within the two-dimensional search, a one-dimensional search occurs on the z coordinate whenever a subtree is entered in which all the points contains the same x coordinates and the same y coordinates.

To analyze the cost required to search the directory, let $C(n,k)$ be the cost of locating k' vertices, for $1 \leq k' \leq n$, in the ϵ-neighborhood of a given point q. Here k is the total number of nodes in $\mathcal{D}$ that require the search to proceed in both the left and the right subtrees (i.e., $k' \leq k \leq n$). If $k = 0$, that is, there is no node in $\mathcal{D}$ whose x coordinate is within ϵ distance of the query point, then a single path from the root down to a leaf is traversed at a cost of $C(n,0) = \lfloor \log_2 n \rfloor$.

The maximum cost occurs when all of the k nodes appear at the top of the tree. See Figure 11.

$$C(n,k) \quad \leq \quad k + (k+1)\lfloor \log_2 n - \log_2 k \rfloor, \; 0 < k \leq n$$

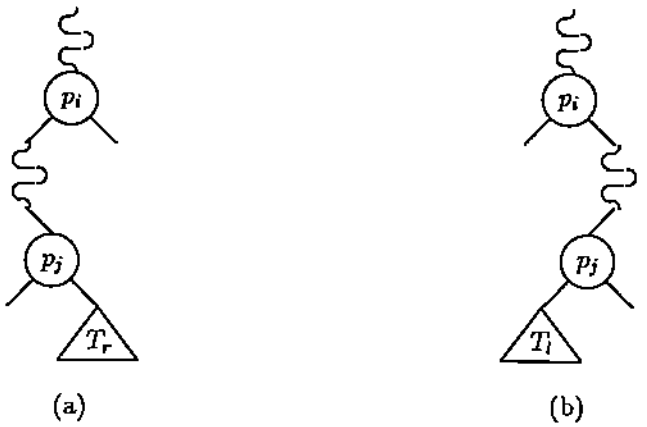


Figure 10: When $x(p_i) = x(p_j)$, the tree T_l (or T_r) is y-ordered.

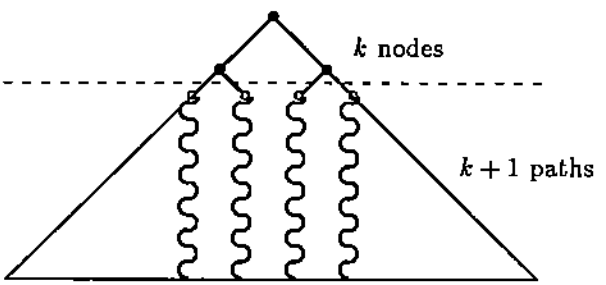


Figure 11: Proof of $C(n, k)$.

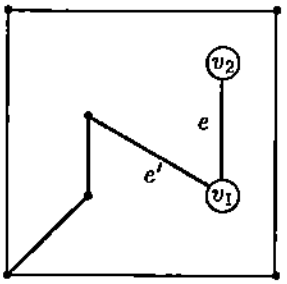


Figure 12: A face with eight edges and a chain of bridge edges that ends with a strut edge, e. Note that removal of e makes e' a strut edge.

$$\begin{aligned} &= k + (k+1)\left\lfloor \log_2 \frac{n}{k} \right\rfloor \\ &= O\left(k + k \log \frac{n}{k}\right). \end{aligned}$$

The minimum cost occurs when all of the k nodes appear at the bottom of the tree,

$$C(n,k) \geq k + \left\lfloor \log_2 \frac{n}{k} \right\rfloor, \ 0 < k \leq n.$$

Thus, the exact value of $C(n,k)$ depends on where the k nodes appear in the tree. The closer the k nodes lie to the root, the higher the cost. Although k can be as high as n, it would be so only if

$$(\forall q \in \mathcal{D}) \ |x_p - x_q| \leq \epsilon,$$

and this is extremely rare. Because the vertices of a solid are frequently distributed throughout $\mathbf{E}^3$, k' is empirically found to be less than 2, and k is small. However, from a theoretical perspective, the cost is bound by

$$\lfloor \log_2 n \rfloor \leq C(n,k) \leq n,$$

and the total retrieval or insertion cost is $O(k + k \log_2 \frac{n}{k})$.

4 Topological Reduction

The topological reduction phase creates maximally connected faces and removes all pseudo entities. Three operations are needed to remove pseudo vertices and edges. The operations preserve topological consistency locally. They are:

1. Isthmus edges that join adjacent coplanar faces are removed and the two faces are merged.
2. In a face, a chain of bridge edges that ends with a strut edge and contains only manifold vertices is removed (refer to Figure 12).
3. Adjacent collinear edges are merged into a single edge by removing one of the edges and the joining isthmus vertex.

Consider first the removal of isthmus edges. The removal of an isthmus edge from the data structure is shown in Figure 13. Referring to the labels of that figure, e is the isthmus edge node, v and w are the vertices, f_i is a face node, and g_{ij} is the jth directed edge node of face i. Before removing the edge, the directed edges of face f_2 change their face references to face f_1. The edge node and both its directed edge nodes g_{12} and g_{22} are then removed. The two directed edge cycles—now broken—are spliced by connecting the directed edge node g_{11} to g_{23}, and by

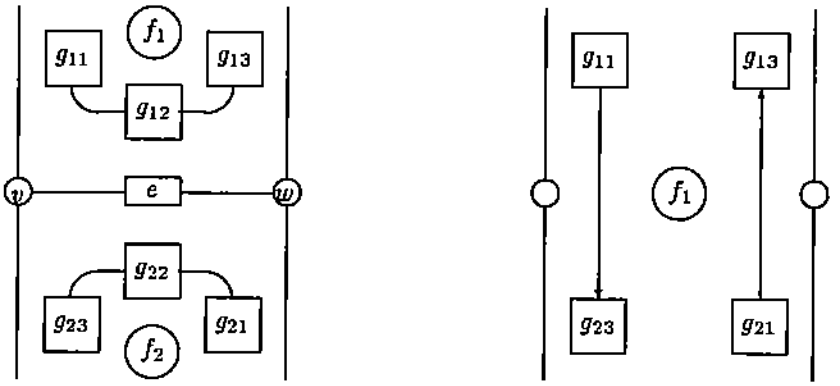


Figure 13: Before and after removal of isthmus edge e and its directed edges g_{12} and g_{22}.

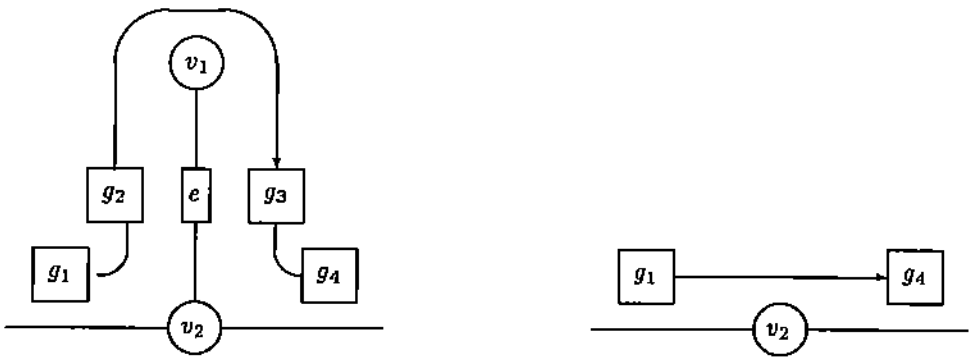


Figure 14: Before and after the removal of strut edge e and its directed edges g_2 and g_3.

connecting g_{21} to g_{13}. After this, the face node f_2 which was arbitrarily chosen is removed from the list of faces.

When an isthmus edge is removed, an adjacent edge can become a strut edge, and these also have to be removed. Removing individual strut edges is a simple process. However, the problem is that edges become strut edges as the result of removing both the isthmus edges and other strut edges. Thus, although initially there may be no strut edges at all, in the end many strut edges may have to be removed. Consider the chain of bridge edges shown in Figure 12. When the strut edge e is removed, the bridge edge e' becomes a strut edge. This implies that a single pass over the initial edges may not remove all the isthmus and the strut edges. By simply considering all the original edges one at a time, many of the newly created strut edges would be missed. Since however, strut edges are created only by removing an adjacent edge, the single pass over the initial edges can be modified to recursively check and remove adjacent edges. This method is analogous to *rehashing in situ* [11].

When a strut edge is detected, it is removed. If it has only one adjacent edge, then that edge is checked. If the adjacent edge has become a strut edge, the process continues with that edge. Otherwise, the process continues with the next edge on the list of the remaining edges.

The recursive removal of isthmus and strut edges may create isthmus vertices that have to subsequently be removed. Since an isthmus vertex has exactly two adjacent collinear edges, the two edges share the same faces and must therefore have the same number of directed edges. The removal of an isthmus vertex v is shown in Figure 15. One of the incident edges, say e_2—by convention—and all its directed edges are removed. The other edge, e_1, is extended towards

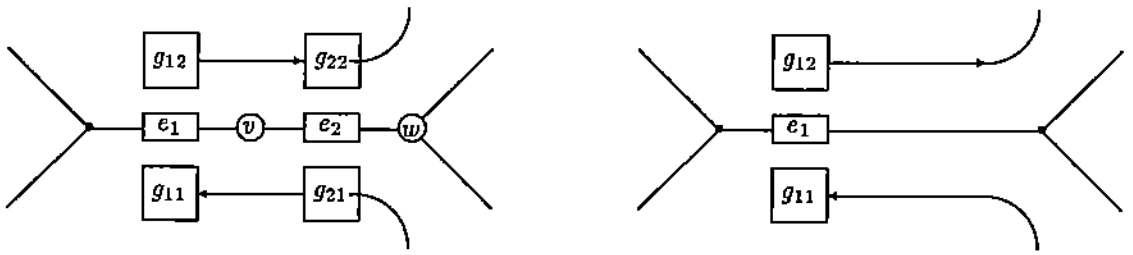


Figure 15: Isthmus vertex v before and after removal; forces edge e_2 and its directed edges to be removed as well.

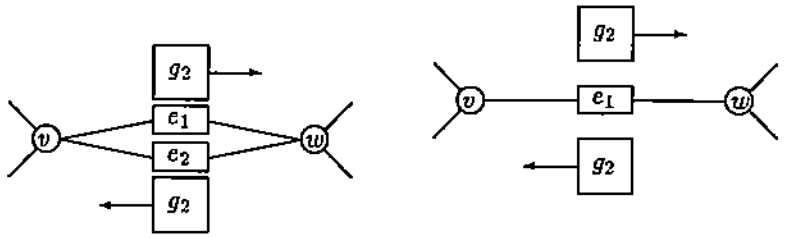


Figure 16: Parallel edges e_1 and e_2 having two vertices in common are merged by removing edge e_2.

vertex w.

The removal of an isthmus vertex cannot cause adjacent vertices (which are not already isthmus vertices) to become isthmus vertices. Therefore, a single pass over the vertices suffices to remove all the isthmus vertices.

With the reduction operations just described, the topological reduction phase has two steps:

1. Remove all the isthmus edges in the data structure along with any adjacent strut edges (performed recursively). All the isthmus, and struct edges are removed in a single pass over all the edges.

2. Remove all the isthmus vertices in the data structure.

5 Merging Parallel Edges

We define three classes of parallel edges according to which one of the following three conditions hold.

1. The two edges e_1 and e_2 have vertices in common. Merging edges e_1 and e_2 is shown in Figure 16. Edge e_2 is removed, and all its directed edges are added to the list of directed edges of e_1.

2. The two edges e_1 and e_2 overlap but share only one vertex. Without loss of generality—we assume that e_1 is shorter than e_2. (See Figure 17). So we split edge e_2 into two edges at vertex r and absorb edge e_1 into the other equal sized edge according to rule one stated above.

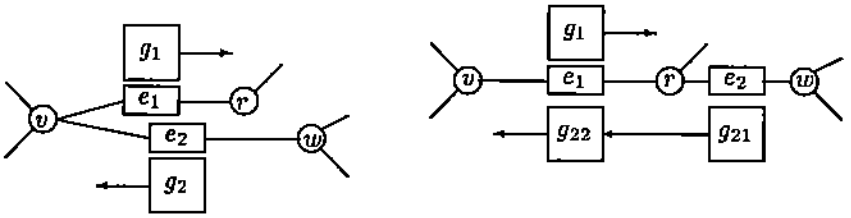


Figure 17: Parallel edges e_1 and e_2 having a common vertex are merged.

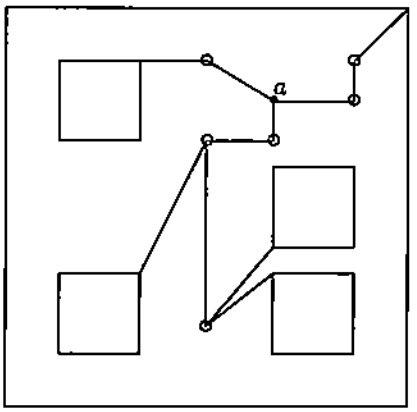


Figure 18: Three trees of bridge edges used in a multi-connected face with five holes, one of which is a singular nonmanifold vertex, a.

3. The two edges e_1 and e_2 overlap but do not share either vertices. We assume that either one or both vertices of e_1 are in the interior of e_2; that is •——•——•, or •——•——•.

Removing parallel edges is analogous to the removal of strut edges. Only the edges of the first two classes stated above are checked for, and merged. If edges of the second class are merged, vertex r (Figure 17) is checked and the process is repeated until no more adjacent parallel edges are found. Edges of the third class are not handled explicitly. Either such pairs of edges are manifold edges or they are lamina edges. If they are lamina edges, they have to be a part of a sequence of such edges terminated at both ends by edges with a common vertex. Thus, a recursive procedure similar to the one for the removal of edges merges all the lamina edges.

6 Removing Bridge Edges

At this stage of the algorithm, maximally connected faces and validity have been established. However, faces with inner holes are represented as a single directed-edge cycle with bridge edges connecting the various edge loops of a multi-connected face. For representations such as the fedge-based data structure which use bridge edges to represent multi-connected faces, the algorithm is now done. But for representations such as the star-edge data structure which do not use bridge edges but instead uses edge loops, the algorithm needs to remove the bridge edges, and return a list of pointers to each of the remaining directed-edge cycles. The removal of bridge edges is complicated by the fact that bridge-edges may form trees as shown in Figure 6.

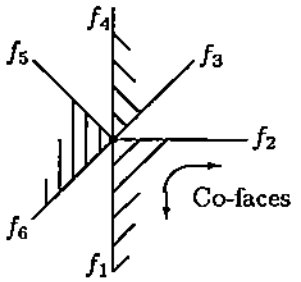


Figure 19: Cross section of a nonmanifold edge with three pairs of co-faces: (f_1, f_2), (f_3, f_4), and (f_5, f_6).

The removal proceeds as follows. The directed-edge cycle around the face is traversed. When a bridge edge is encountered, it is removed, the directed-edge cycle is split into two cycles and both are processed recursively. When a directed edge is encountered that has been already visited, the cycle containing that directed edge is added to the loop structure of the face. Since it is not known beforehand which faces contain bridge edges, all faces are processed requiring time linear in the number of directed-edges.

7 Partitioning Faces into Shells

Data structures that contain an explicit representation of shells, such as the star-edge data structure, additionally require the partitioning of faces into shells. Therefore, we have to identify which faces belong to which shell.

For manifolds, shells cannot touch and this leads to a simple mark-and-collect algorithm based on face-edge and edge-face adjacency relationships. However, for nonmanifolds where two or more shells may touch at vertices or along edges, the faces can be partitioned into two or more components, and the face adjacency must be considered for co-faces only, as explained below.

Recall that the faces adjacent to a nonmanifold edge can be ordered cyclically about the face, and that consecutive face pairs enclose volume of the solid [5].

So far, the faces around a nonmanifold edge have not yet been so paired and ordered in the data structure. Given a face f, its partner in the pairing is the *co-face* of f. See also Figure 19.

From the normal vector of each face and the edge vector we compute the in-vector of each face, where the in-vector of a face points into the face and away from the edge. Choosing one in-vector as a reference vector yields angles for the faces with which the faces around the edge can be sorted, and co-faces identified.

With co-faces identified, the mark-and-collect algorithm is the same as for manifolds. That is, from a face all adjacent co-faces are recursively collected into a shell structure. This, therefore, takes $O(n \log n)$ time, for n faces in the solid.

8 Conclusion

In this paper we presented an algorithm for creating solid models given the polygons that bound the solid.

The vertex directory is a dynamic data structure allowing both insertions and deletions. It is also an improvement over linear structures and yields sublinear time operations. Although the worst-case behavior of inserting and searching for the n vertices in the directory is $O(n^2)$, in practice, the average behavior is closer to $O(n \log n)$. Lower worst-case behaviour could be attained by relinquishing the dynamic property and creating a static directory allowing lookups only (e.g.,

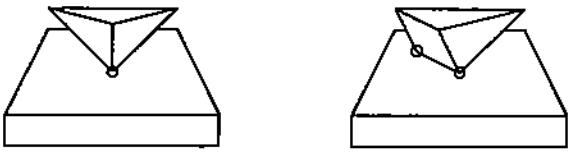


Figure 20: Unresolved vertex-on-face, (a), and edge-on-face, (b), contacts.

the *multidimensional binary search trees* also known as k-D trees). A static data structure cannot be used because insertions may be ambiguous and require topological assistance, and because the creation algorithm is frequently a component in a larger system requiring intermixed insertions and deletions (e.g., ProtoSolid).

The shortcommings of this algorithm is that nonmanifold vertices and edges lying in the interior of a face will not be incorporated into the face as singular holes. See Figure 8. To do so would require another step that would check every vertex against the interior of every face. Depending on the types of operations being performed this may or may not be necessary. A similar shortcomming is that parallel nonmanifold edges for which one edge is contained entirely in the interior of the other edge will not be merged. One way of solving this is to modify the vertex-finding function of Section 3 to crack an edge when a new vertex falls on it. However, without some kind of an efficient edge-directory suitable for finding the appropriate edge in sublinear time, this approach is too time consuming and is better left to a post processing step. In such a step, every vertex is checked for the containment in every edge.

After completion, the algorithm reports errors if after the third phase any lamina edges remain. Lamina edges indicate that the boundary is not closed. Reporting lamina edges is beneficial for the cases in which the input data contains missing polygons, as for example, when only half of a femur is scanned resulting in a missing patch at the last cross section. Such missing polygons must be inserted by hand to complete the solid model, as is done, for example, in Shilp.

9 Acknowledgements

I would like to acknowledge Christoph Hoffmann and Chanderjit Bajaj for their continued support at Purdue, Malcom Fields for implementing the contour data triangulation algorithm, and Bill Bouma for implementing the marching-cubes algorithm.

References

[1] G. M. Adel'son-Vel'skii and E. M. Landis. An algorithm for the organization of information. *Dokl. Acad. Nauk SSSR Math*, 146(2):263–266, 1962.

[2] V. Anupam, C. Bajaj, and S. Klinkner. *A Tutorial and User Guide to SHILP.* Department of Computer Science, Purdue University, West Lafayette, IN, February 1990.

[3] H. N. Christiansen and T. W. Sederberg. Conversion of complex contour line definitions into polygonal element mosaics. *Computer Graphics*, 12(3):187–192, 1978.

[4] H. Fuchs, Z. M. Kedem, and S. P. Uselton. Optimal surface reconstruction from planar contours. *Graphics and Image Processing*, 20(10):693–702, October 1977.

[5] C. H. Hoffmann. *Geometric and Solid Modeling: An Introduction.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1989.

[6] C. M. Hoffmann, J. E. Hopcroft, and M. S. Karasick. Robust set operations on polyhedral solids,. Technical Report 87-875, Department of Computer Science, Cornell University, 1987.

[7] M. Karasick. *On the Representation and Manipulation of Rigid Solids.* PhD thesis, McGill University, 1988.

[8] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *ACM Computer Graphics SIGGRAPH '87*, 21(4):163–168, July 1987.

[9] M. Mäntylä. *AN Introduction to Solid Modeling.* Helsinki University of Technology, 1984.

[10] V. J. Milenkovic. *Verifiable Implementations of Geometric Algorithms Using Finite Precision Arithmetic.* PhD thesis, Carnegie Mellon University, July 1988.

[11] T. A. Standish. *Data Structure Techniques.* Addison–Wesley Publishing Company, Reading, Mass., 1980.

[12] K. Sugihara and M. Iri. A solid modeling system free from topological inconsistency. Research Memorandum RMI 89–03, University of Tokyo, March 1989.

[13] G. Vaněček Jr. Obtaining boundaries with respect: A simple approach to performing set operations on polyhedra. CAPO Report CER-89-25, Purdue University, Department of Computer Science, West Lafayette, IN 47907, November 1989.

[14] G. Vaněček Jr. Protosolid: An inside look. CAPO Report CER-89-26, Purdue University, Department of Computer Science, West Lafayette, IN 47907, November 1989.

[15] G. Vaněček Jr. *Set Operations on Polyhedra using Decomposition Methods.* PhD thesis, University of Maryland, College Park, Maryland, June 1989.

[16] K. Weiler. Edge–based data structures for solid modeling in curved–surface environments. *IEEE Computer Graphics & Applications*, pages 21–40, January 1985.

[17] K. Weiler. *Topological Structures for Geometrical Modeling.* PhD thesis, Rensselaer Polytechnic Institute, Troy, N.Y., August 1986. Technical Report TR-86032.

[18] F. Yamaguchi and T. Tokieda. Bridge edge and triangulation approach in solid modeling. In Tosiyasu L. Kunii, editor, *Frontiers in Computer Graphics '84*, pages 44–65. Springer-Verlag, 1985.