Some Efficient Procedures for Correcting Triangulated Models

I. Mäkelä*
A. Dolenc
Helsinki University of Technology
Institute of Industrial Automation

July 1993

Abstract

This paper describes methods for handling efficiently a large class of problems encountered when dealing with 3D models represented by a collection of triangles in STL format. In spite of its drawbacks, the STL format has become a *de facto* industrial standard for transferring models to manufacturing processes generally known as Rapid Prototyping Techniques (RPT) or Solid Freeform Fabrication (SF^2). As the accuracy and size of the workspace of such processes increases, so does the size of the models one wishes to manufacture. Therefore, the efficiency of application programs is an important consideration. Previous published work has focused on the problem of eliminating gaps in triangulated models. In addition to efficiency, this paper describes methods for dealing with other problems such as overlapping triangles and intersecting triangles. The algorithms have been implemented and tested in industry. The underlying data structures based on adaptive space subdivision also allow the development of other efficient tools such as slicing.

1 Introduction

Data transfer between CAD systems and RP processes is mainly based on data exchange formats capable of representing faceted models. The current *de facto* standard is the STL format [1] which allows one to represent triangulated models, *i.e.* each facet is a triangle.

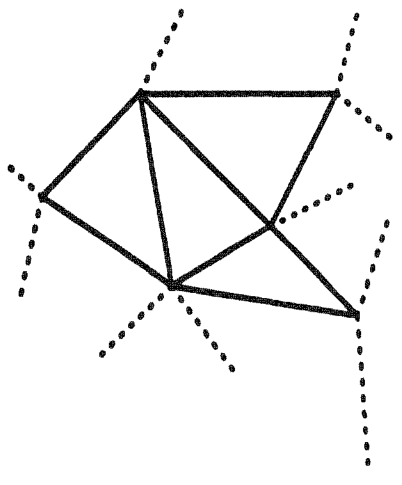
Figure 1: A correct triangulation.

In order for models to be correctly manufactured they must represent a collection of one or more non-intersecting solids. The manufacturer hopes to receive "well-behaved" STL-files such as the one outlined in Figure 1. In a correct STL-file, each triangle has exactly one neighbour along each edge and triangles are only allowed to intersect at common edges and vertices. Under these conditions, it is possible to distinguish precisely the inside from the outside of the model.

*Otakaari 1, SF-02150 Espoo, Finland. Tel:+358-0-4513372. Fax:+358-0-4513293. Email (Internet): ima@cs.hut.fi.

Unfortunately, quite often incorrect faceted models are used. The mistakes can be numerous (Figure 2). The models can contain **gaps** due to missing facets, facets may intersect at incorrect locations, the same edge may be shared by more than two facets, etc. Special cases of these errors may occur that require separate treatment, *e.g.* overlapping facets (coplanar facets whose intersection results in another facet). The reasons for such errors are related to the application that generated the faceted model, the application that generated the original 3D CAD model, and the user. Many STL interfaces in CAD systems fail to inform the user that the result is not correct and problems remain undetected until the manufacturer attempts to process the model.

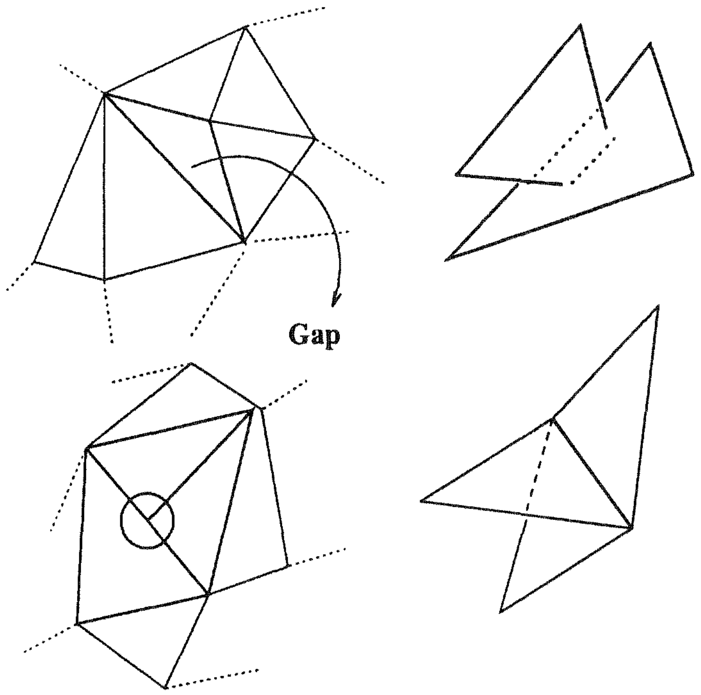

Figure 2: Incorrect triangulations.

The objective of this paper is to describe efficient algorithms for *(i)* verifying if the model is correct, *(ii)* detect the mistakes in the model, and *(iii)* automatically correct as many mistakes as possible. Additional requirements are portability and user friendliness, the latter implying that the parameters should be easy to understand and have a predictable effect on the output.

The paper is organized as follows. One of the key aspects are the data structures used so we begin with their description in Section 2. Next, in Section 3 we outline the algorithm used to process an STL-file. The algorithm for determining the **topology** of the model—*i.e.* the neighbours of a given triangles along its edges—is described in Section 4. Determining the correct orientation of the triangle normals—a requirement of the STL format—is described in Section 5. Detecting incorrect intersection and the handling of special cases of overlapping triangles is described in Section 6. Section 7 outlines a gap elimination algorithm for a restricted class of gaps. Related and future work is discussed in Section 8 and, finally, we state our conclusions in Section 9.

2 Data structures

Efficient handling of large geometrical data sets requires special data structures. We use an adaptive space subdivision scheme that reduces the amount of browsing involved when searching for objects nearby a given object, and adopted a method that is a variant of *quadtrees* [9]. The system is called *the CELL space subdivision system* (other data structures are also suitable for this application [9, 10]).

A bounding box, or *space*, surrounding the objects must be known in advance. Associated to each object is a bounding box of dimension n. The space is subdivided into smaller regions, or *cells*, until either each cell contains at most N objects or the depth of the subdivision reaches a maximum D. The subdivision is done by splitting the cell using iso-oriented hyperplanes of dimension $n - 1$[1] and always at the midpoint. The direction alternates at each level of subdivision.

[1]The hyperplanes will correspond to lines in two dimensions and planes in three dimensions.

Figure 3 illustrates a space subdivision with six objects, $n = 2$ and $N = 3$. A suitable data structure for representing the space subdivision is a tree. In the example shown, white nodes are the internal nodes and the black nodes are the leaf nodes, or simply *leaves*. The information stored in the nodes is very simple. The internal nodes contain only pointers to the subtrees; it is not necessary to store bounding boxes or splitting directions because this information can be derived as the tree is searched. The leaves contain simply an array of pointers to the objects.

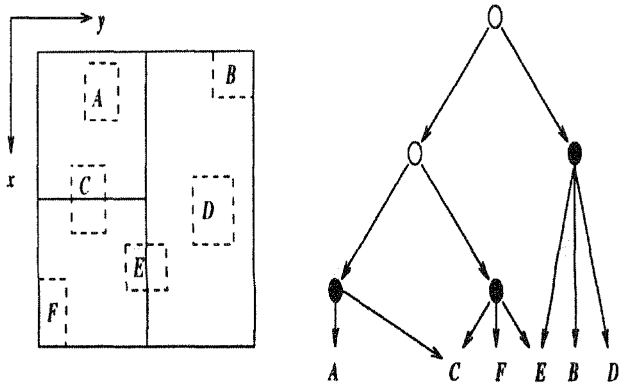

Figure 3: An example space subdivision and tree representation.

Finally, objects contain a bounding box, a query number and a reference to user data. Suppose a query Q, a rectangle, intersects the bounding box of object E. In order to avoid E being reported twice, the queries are numbered and objects that are reported by a given query are assigned the same number. If an object already has the same number as the current query then it is ignored. The query numbers of all objects would have to be reset only when the query counter overflows, an unlikely event if the platform uses 32-bit integer arithmetic.

When inserting a new object, it may happen that it is impossible to subdivide a cell such that only N objects remain, *e.g.* if $N + 1$ triangles share the same point. The default action is to subdivide until a maximum depth D is reached. Alternatively, the application programmer can supply a boolean function *Subdivide*? that will browse the objects and determine if it is worth attempting the subdivision.

Searching such a data structure is efficient because the number of possible neighbour candidates is usually small. We only have to check nearby objects, instead of going through the entire collection. Furthermore, in our applications the data structure is semi-static. After initial object insertion, objects added to the data structure do not change significantly the original bounding box of the space, and very few deletions are done. Thus, no reorganization of the data structure is needed during the processing.

Interface

The interface consists of four functions. *CellInit* creates the data structure according to the set of parameters already mentioned earlier, namely the bounding box, maximum cell occupancy, and maximum depth. *CellInsert* and *CellDelete* are used to update the data structure. Finally, *CellQuery* is a function that takes as arguments a bounding box and a query number and it returns a list of all objects that intersect the given bounding box.

Analysis

In general, the size of a CELL data structure is bounded by $O(2^D + n)$, where n is the number of objects. In our applications, we set D equal to $\lfloor \log_2(\lfloor n/N \rfloor) \rfloor + 1$, so the size is bounded by $O(n)$ and the height of the tree by $O(\log n)$. Regarding the computation time for updates and queries, so far we have not made any attempt to analyse the complexity of the worst and average cases. However, experiments indicate that the performance is excellent.

Application-dependent data structures

Triangle records are referenced with an application data pointer from the CELL objects. The triangle record has pointers to the vertices and to the edge-neighbors. A field is reserved for a normal vector and a reference counter for memory management. The vertices are indicated by a set of pointers, $\{v_0, v_1, v_2\}$. Edge i is defined by the vertices $v_{i \bmod 3} v_{i+1 \bmod 3}$, and associated to each edge is a pointer n_i to the corresponding neighbouring triangle.

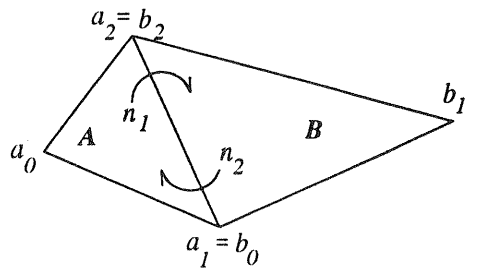

Figure 4: Two neighbouring triangles A and B.

The data associated to each vertex consist of three floating-point values defining the xyz-coordinates, a reference counter and a vertex id.

3 Outline of the algorithm

The CELL data structure is very flexible in the sense that when one is equipped with such a data structure there are various solutions to the problems laying ahead. We have chosen those that favour minimizing memory resources at the expense of computation time.

Before the CELL structures can be initialized, it is necessary to read the input file to gather the required information, *i.e.* the bounding box enclosing the model and the number of triangles. Next, two CELL structures are created, namely $TCell$ will contain objects associated to the input triangles, and $ECell$ will contain the edges with no neighbours and is used to construct the gaps. The user may supply the maximum occupancy for each of these CELL structures to override the internal defaults. Based on the maximum occupancy, the depth of the associated trees is bounded by a suitable constant to obtain a balance between the main resources, namely memory and computation time.

The first stage evaluates the **topology** of the model, detects gaps, exact duplicates, and degenerated triangles. $TCell$ is used to collect vertices that are equal within a tolerance ϵ. Once the vertices have been merged and values have been assigned to the pointers v_i then we proceed to detect the **edge-neighbours** and assign values to the pointers n_i. The triangle is now ready to be inserted into $TCell$. After all triangles have been inserted, it is possible to determine the **existence** of gaps but not their actual description.

The direction of the normals is now evaluated because this information is needed for correcting the gaps and other errors that might be detected subsequently. The next stage consist of verifying the model. The description of the gaps is evaluated and errors such as improper intersections and overlapping triangles are also detected. Some of these errors can be corrected whereas others can only be reported. Once errors have been corrected, the normals are oriented again because if gaps where present it is possible that the first orientation is incorrect. In fact, if the gaps have not been all eliminated, it is not possible to guarantee that the second attempt is successful either. Finally, the triangles and a description of the errors are written to files in the required format.

4 Creating the topology

Creating the topology for a model consists of collecting for each triangle t neighbourhood information about t. More specifically, it consists of detecting the triangles that share common vertices and edges with t. Naturally, the first step is to begin with the vertices.

4.1 Detecting vertex-neighbours

Let us assume that all triangles inserted so far into $TCell$ have their corresponding v_i pointers already initialized to correct values. (In the sequel, the Euclidean coordinates of an unprocessed triangle are denoted V_i as opposed to the associated pointer which is denoted v_i.)

For each vertex $V_i \in t$ we apply the following procedure. The bounding box $BB_i \doteq BoundingBox(V_i)+\epsilon$ is evaluated, where ϵ is a small constant to account for rounding errors that probably occurred when the model was created. Next, the set of triangles $VNeighbourCandidates \doteq CellQuery(TCell, BB_i)$ is evaluated. The vertices V' of all triangles $t' \in VNeighbourCandidates$ are searched to find the closest one to $V_i \in t$. Once found, the associated pointer is assigned to v_i.

If a triangle collapses to a line or a point then it is deleted and is not inserted into $TCell$.

4.2 Detecting edge-neighbours

If a little more effort is done while finding the vertex-neighbours then enough information can be made available to easily detect the triangles that share a common edge with t. Let $V_{\min}$ be the closest vertex to V_i evaluated by the procedure outlined above. Since the entire set $VNeighbourCandidates$ is searched one can collect the triangles that already share the same pointer $v_{\min}$.

Let us say that these triangles are placed in a set denoted by VN_i. For each $k = 0, 1, 2$ we compute $I \doteq NV_k \cap NV_{k+1 \bmod 3}$. If $|I| = 0$ then the neighbour of t along the edge k either does not exist or has not yet been inserted in $TCell$. In this case, no further action is taken. If $|I| > 1$ then a possible error condition has been detected, namely an edge is being shared by more than two triangles. The edge is tagged accordingly and the tag will be cleared if another edge from another triangle matches it. Edges shared by $2n$ triangles with $n > 1$ are reported has **warnings** whereas edges shared by $2n + 1$ triangles with $n \geq 0$ are reported has **errors**. We call the latter **odd-edges**. Finally, if $|I| = 1$ then the neighbour of t along edge k has been found. The pointer n_k is set equal to $v_{\min}$ and a global counter, $EdgeHit$, is incremented by one.

After all triangles have been inserted into $TCell$, the value of $EdgeHit$ must equal $3N$, where N is the number of triangles. If $EdgeHit < 3N$ then the entire set of triangles is searched for the **odd-edges**. These are placed in $ECell$ for subsequent use during the gap elimination stage. Before a triangle is inserted, though, we verify if it is an exact duplicate, *i.e.* if all pointers n_i have the same value. In this case, the triangle is discarded because it is likely that it is the result of duplicate or coincident surfaces[2].

5 Orienting the triangles

An initial triangle t_0 is chosen arbitrarily and a queue of triangles U is initialized. A ray is cast in the direction of the normal vector $\vec{N}_0$ of t_0. The direction of $\vec{N}_0$ is determined by the number of intersections between the ray and the model. The vertex pointers v_i are reordered according to the specifications of the STL format which states that as the fingers of the rigth-hand follow the vertices the thumb must be pointing towards the outside of the model. $TCell$ is used to minimize the number of ray-triangle intersection tests. Only the triangles that are located in cells that are hit by the ray need to be taken into consideration.

Once a triangle t has been oriented, all its edge-neighbours that have not yet been processed are placed in the queue U, if not already there. The next triangle to be processed is the one at the head of the queue. When the queue is empty we verify if all triangles in the model have been processed. If not then the next unprocessed triangle t'_0 is again chosen arbitrarily and the same procedure is applied once more.

At this point, we would like to introduce the concept of a **component**. Loosely speaking, a triangle t belongs to the same component as t' if one can be reached by the other by "walking" along edge-neighbours. (For precision purposes we include the following definition using concepts from Graph Theory. Let $G = (V, E)$ be a graph where each vertex $v \in V$ corresponds to a triangle and an edge $e = vw$ with $v, w \in V$ is an edge of G if the triangle associated to v is an edge-neighbour of the triangle associated to w. Because the property is reflexive, G is an un-directed graph. In a sense, the graph G is the "dual" of the triangulated model. A *path* in G is any connected set of edges. Two vertices are said to be in the same component C of G if a finite path can be found between them. Two triangles t_v and t_w are in the same component if the associated vertices $v, w \in V$ are in the same component of G. Finally, notice that the algorithm above corresponds to a breadth-first search—BFS, for short—on the graph G.)

[2]In some cases, discarding **both** is a better alternative.

Clearly, a "by-product" of the above algorithm is the set of components of the model. This information will be useful to detect more efficiently errors such as improper intersections.

It may happen that the orientation of a triangle is not consistent with all its neighbours. This happens when the model contains a non-orientable component such as a Moebius strip. The user is notified of such errors but no attempt is made to correct them.

It is possible to avoid one ray casting operation by choosing a suitable triangle as the first one. Let t_0 to be a triangle that is not parallel to the xy-plane and that touches the bounding box of the model. If the model is correct then such a triangle must exist. Let $V \in t_0$ be a vertex that touches the bounding box. Then the xy-projection of $\vec{V} + \vec{N}_0$ must be outside the projection of the bounding box onto the xy-plane. Experiments indicate that, in our case, this method did not improve significantly overall execution times.

We draw the attention of the reader to the fact that if the model is incorrect then it is not possible to guarantee that the direction of the normals is correct. However, one can state that the normals of the triangles associated to the same component are oriented consistently. This property is sufficient for correcting some errors in the model.

6 Checking the model

Errors are categorized into three classes. **Gaps** are detected during topology re-construction and their handling is described in the following Section. **Improper intersections** may exist even if no errors where detected in previous stages of the algorithm. **Overlapping triangles** are a special case of intersections and are dealt with separately. One type of overlapping triangles, namely duplicate triangles, was already dealt with during topology re-construction. Two triangles t and t' intersect improperly if their intersection is a line segment that does not correspond to a common edge. On the other hand, they overlap if their intersection is a facet.

These errors are expensive to detect and report even when using the CELL data structure. The first stage treats each component C of the model separately. The second stage handles pairs of intersecting components.

The first stage is as follows. For each triangle $t \in M$ the set

$$ICandidates \doteq CellQuery(TCell, BoundingBox(t))$$

is evaluated. The triangle t is checked for errors against those triangles in $ICandidates$ that are in the same component as t. In the case of improper intersections, one can also exclude the neighbours of t.

The second stage uses a CELL structure called $CCell$. The objects in $CCell$ are associated to the components of the model. For each component C the set $CCandidates \doteq CellQuery(CCell, BoundingBox(C))$ is evaluated. Next, for each $C' \in CCandidates, C \neq C'$ the following procedure is applied. The bounding box $I = BoundingBox(C) \cap BoundingBox(C')$ is evaluated. We now take from $TCell$ the triangles enclosed by this bounding box, *i.e.* the set $T = CellQuery(TCell, I)$ is evaluated. Finally, the triangles in this set that belong to the component C must be checked against those of C', and vice-versa.

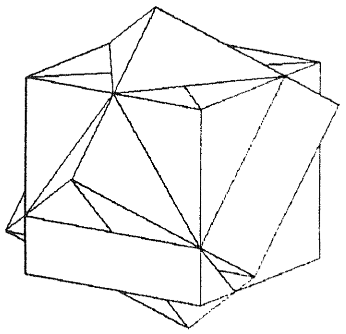
Figure 5: An incorrect model with improper intersections.

Consider the model shown in Figure 5. Each box forms one component. Although each box in isolation is correct, together they form an incorrect model because the result is ambiguous. Additional information from the user would be needed to correct the model and, therefore, improper intersections are reported but no attempt is made to correct them.

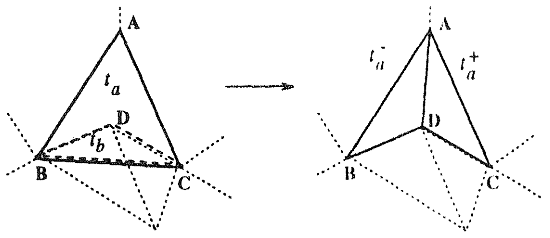

Figure 6: Special case of overlap removal.

Figure 6 illustrates a special case of overlapping triangles that is corrected. One of the overlapping triangles, $t_b = BCD$, is removed whereas another one, $t_a = ABC$, is split in two. This situation can occur for various reasons, *e.g.* it can be caused by errors in the triangulation of parametric surface models [5].

7 Filling gaps

A gap is a closed polyline composed of odd-edges. It is easy to find the gaps using the $ECell$ mentioned in previous Sections. However, filling the gaps, or gap elimination, poses many difficulties. One problem is choosing a set of "user-friendly" parameters. Another problem—which has no hope of being fully automated—is filling the gaps *such that the original intentions of the designer are preserved.* It cannot be fully automatic because given a model with more than one gap, it is not possible to decide algorithmically if triangles should be added such that each gap is filled separately or such that the gaps are joined. Finally, regardless of which choice is taken, we must find a method that adds triangles such that the result is a valid model.

The method we describe here can be applied to one individual gap. Triangles are added to the gap until it is closed or user-supplied tolerances prevent triangles from being added.

Consider a closed polyline P with n vertices, $v_0, v_1, \ldots, v_{n-1}$, where $v_0 = v_{n-1}$. The distances d_{i-} and d_{i+} attached to a vertex v_i are the distance between the vertex v_i and vertices v_{i-2} and v_{i+2}, $d_{i-} = v_i - v_{i-2}$ and $d_{i+} = v_i - v_{i+2}$, respectively. Vertices are sorted in ascending order according to the smallest distances d_{i-} and $d_{i+}, i = 0, \ldots, n-1$. Let us assume that the distance d_{i+} was the smallest. The triangle $t_{i+} = (v_i, v_{i+1}, v_{i+2})$ is added if the normal direction of the new triangle does not differ too much from the normals of those triangles which contribute to the edges $v_i v_{i+1}$ and $v_{i+1} v_{i+2}$. When a triangle is added to the model, one vertex is removed from P and all distances are re-evaluated for the remaining points in the polygon. In the above example, if the triangle t_{i+} was added then the point v_{i+1} is removed. The procedure is applied until the polyline P "shrinks" to a polyline with only five vertices or tolerances prevent further triangles from being added. The latter occurs when the smallest distance is greater than a user-supplied tolerance t_g. If P has only five vertices then it is triangulated using two triangles such that the area of the result is minimized.

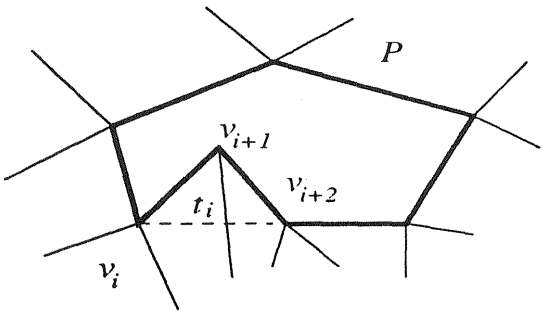

Figure 7: An erroneous fill triangle t_i.

The procedure uses an additional parameter t_a. This parameter is an upper bound on the angle between the normals of the original triangles and the fill triangles. Its main purpose is to avoid "cusps", *i.e.* to create a smooth blend to fill the gap. It minimizes the chances of adding a triangle that will overlap or intersect neighbouring triangles. Consider the situation in Figure 7. Without normal checking, the algorithm might suggest to add the triangle t_i which is clearly an error.

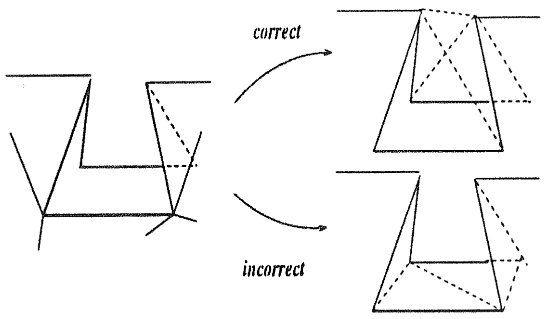

Figure 8: The effect of using normal information.

Another example is shown in Figure 8. If the normals are taken into account then better results are usually obtained. The default value for t_a is 90 degrees.

8 Related and future work

Rock [8] described a method for generating topological information from an STL-file. The fundamental difference is the use of AVL-trees for vertex matching. Apparently though, this data structure is not appropriate for range queries [9] (*e.g.* $CellQuery$ is an implementation of n-dimensional range queries). "Back-pointers" from merged vertices to the associated triangles are used to detect edge-to-edge relationships. We chose not to use such pointers in order to economize memory resources. A method for orienting the triangles is not described.

Barequet [2] describes the best method (so far) for eliminating gaps in faceted models. One of the ideas exposed is the use of global information to verify if adding a given set of facets would result in a valid solid. If the set fails this test then another one is taken until either a successful set is found or such a set cannot be computed such that tolerances are respected. The method used for finding an alternative set of matches is efficient. It is certainly an idea that could improve the algorithm described in this paper but significant changes would be necessary. The method described in this paper for filling gaps utilizes only local information.

Bøhn [3] categorizes gaps into five classes. Unlike Barequet, the gaps are apparently oriented prior to gap elimination. Gaps can be merged or connected if they share a common vertex and are located in different components (shells). In our opinion, this is not a general criterion. The method for adding triangles to fill the gap does not take into account the shape of the neighbouring triangles. The algorithm does not use any user-supplied tolerances although it is not difficult to include them in the algorithm described.

The problem of generating a triangulation to fill a gap can be related to the more general problem of generating a faceted model from a set of 3D points. In the case of gap elimination, some of the edges are already given. In this more general setting, O'Rourke [7] describes a method for generating a polyhedra of minimal area given a set of 3D points.

The algorithms described in this paper have been implemented by one of the authors and tested with numerous models from industry. It incorporates all the features described in this paper. It is written using the C language and is highly portable, being available on several different platforms. Several output formats are supported, namely binary and ASCII STL, Personal Visualizer WAVEFRONT (OBJ format), and IGES. Diagnostics (description of the gaps, non-manifold edges, improper intersections, etc) are reported in VDA-FS format. Full details can be obtained elsewhere [6]. The implementation is available as commercial product.

The CELL data structure has been used to implement other algorithms such gap elimination in parametric surface models [4] and slicing.

We are exploring the possibility of using secondary storage to handle models with $O(10^4)$ triangles. The algorithms described in this paper could be easily generalized to facets but a significant portion of the implementation would need re-writing. Besides, our feeling is that the limits of a non-interactive tool have been reached and that the greatest benefits are to be gained by implementing appropriate interactive tools.

9 Conclusions

In this paper we have presented an efficient algorithm for handling polyhedra models represented in STL format. An algorithm for filling individual gaps that takes into account the shape of the neighbouring triangles was presented. In addition, we have explained how to generate the topology of the model, orient the facets, and detect all errors that can occur in the description of a model in addition to gaps. Special cases of these errors are automatically corrected. The efficiency of the algorithms is based on the usage of good data structures, namely binary trees associated to spatial subdivision. The algorithm has been successfully tested in industry and proven useful.

We would expect productivity to improve if a better data exchange format replaced the STL-format. Not that the problems discussed here would simply vanish; rather they would be minimized and occur less frequently. The "quality" of a model represented in a neutral file format is directly related to the capabilities of the format, the tool used to convert the model to the given format, and the user. Due to user expectations regarding RPT, it will always be necessary to verify and correct models regardless of the format chosen for data transfer.

10 Acknowledgements

This work was done within the INSTANTCAM project, an European Consortium of partners from both industry and research centers. Although not all partners have been directly involved in software development, they have all contributed in some way, either with suggestions, support, patience, or testing of the tools. The partners are, in alphabetical order, BIBA (G), Black&Decker GmbH (G), Danfoss A/S (DK), the Danish Technological Institute (DK), E&D Design (SF), Helsinki University of Technology (SF), Instituto Superior Técnico (P), NTH-SINTEF Production Engineering (N), Raufoss A/S (N), Oy Saab-Valmet Ab (SF), and Wilhelm Karmann GmbH (G). At HUT, we received the financial support of TEKES.

References

[1] 3D Systems, Inc. *Stereolithography Interface Specification*, July 1988.

[2] G. Barequet and M. Sharir. Filling Gaps in the Boundary of a Polyhedron. Under preparation, 1993.

[3] J. H. Bøhn and M. J. Wozny. Automatic CAD-model Repair: Shell-Closure. In *Proceedings of Solid Freeform Fabrication Symposium*, pages 86–94, Austin, Texas USA, 1992.

[4] A. Dolenc. Rapid recipes for parametric surface models. Submitted to Computer-Aided Design, 1993.

[5] A. Dolenc and I. Mäkelä. Optimized Triangulation of Parametric Surfaces. Technical Report TKO-B74, Helsinki University of Technology, 1991. To be published in Mathematics of Surfaces IV.

[6] Helsinki University of Technology, Otakaari 1, SF-02150 Espoo, Finland. *TR2STL User Guide, Opus 1.9d*, January 1993.

[7] J. O'Rourke. Polyhedra of Minimal Area as 3D Object Models. In A. Drinan, editor, *Proceedings of the 7th Joint Conference on Artificial Intelligence (IJCAI-81)*, volume II, pages 664–666, August 1981.

[8] S. J. Rock and M. J. Wozny. Generating Topological Information from a Bucket of Facets. In *Proceedings of Solid Freeform Fabrication Symposium*, pages 1–15, Austin, Texas USA, 1992.

[9] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley Publishing Company, Inc., 1989.

[10] M. Tamminen and R. Sulonen. The EXCELL Method for Efficient Geometric Access to Data. In *ACM/IEEE 19th Design Automation Conference*, pages 345–351, June 1982.