Protecting Critical Facets in Layered Manufacturing: Implementation and Experimental Results*

Jörg Schwerdt† Michiel Smid‡ Ravi Janardan§ Eric Johnson§

October 24, 2002

Abstract

In Layered Manufacturing, a three-dimensional polyhedral object is built by slicing its (virtual) CAD model, and manufacturing the slices successively. During this process, support structures are used to prop up overhangs. An important process-planning step in Layered Manufacturing is choosing a suitable build direction, as it affects, among other things, the location of support structures on the part, which in turn impacts process speed and part finish. We describe a robust, exact, and efficient implementation of an algorithm that computes a description of a subset of all build directions for which a prescribed facet is not in contact with supports. We also present test results on models obtained from industry, and on collections of random triangles.

Keywords: Computational geometry, layered manufacturing, implementation, spherical geometry, union of convex polygons.

1 Introduction

Layered Manufacturing (LM) is an emerging technology that is gaining importance in the manufacturing industry; see e.g. the book by Jacobs [2]. This technology makes it possible to rapidly build three-dimensional objects directly from their computer representations on a desktop-sized machine connected to a workstation. A specific process of LM, that is widely in use, is StereoLithography. The input to this process is the triangulated boundary of a polyhedral CAD model. This model is first sliced by horizontal planes into layers. Then, the object is built layer by layer in the following way. The StereoLithography apparatus consists of a vat of photocurable liquid resin, a platform, and a laser; see Figure 1. Initially, the platform is below the surface of the resin at a depth equal to the layer thickness. The laser traces out the contour of the first slice on the surface and then hatches the interior, which hardens to a depth equal to the layer thickness. In this way, the first layer is created; it rests on the platform. Then, the platform is lowered by the layer thickness and the just-vacated region is re-coated with resin. The subsequent layers are then built in the same way.

*This work was funded in part by a joint research grant by DAAD and by NSF. This work was done while JS and MS were at the Fakultät für Informatik, University of Magdeburg, Germany. Part of this work was done while JS and MS visited the University of Minnesota in Minneapolis, and while RJ visited the University of Magdeburg.

†Algorithmic Solutions Software GmbH, Saarbrücken, Germany. E-mail: joerg.schwerdt@algorithmic-solutions.com.

‡School of Computer Science, Carleton University, Ottawa, Ontario, Canada K1S 5B6. E-mail: michiel@scs.carleton.ca.

§Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455, U.S.A. E-mail: {janardan,johnson}@cs.umn.edu. Research also supported in part by NSF grant CCR-9712226.

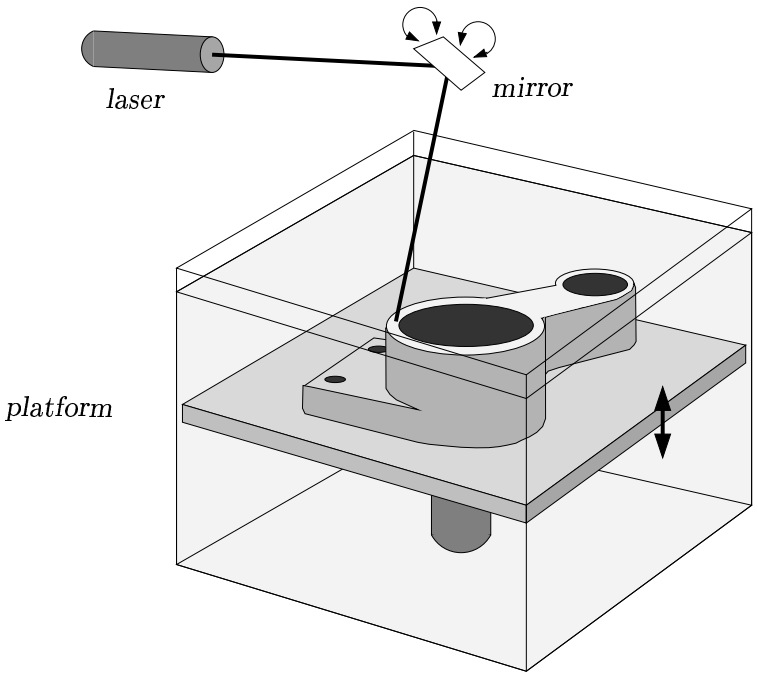

Figure 1: *The StereoLithograpy Apparatus.*

It may happen that the current layer overhangs the previous one. Since this leads to instabilities during the process, so-called *support structures* are generated to prop up the portions of the current layer that overhang the previous layer, see Figure 2. These support structures are computed before the process starts. They are also sliced into layers, and built simultaneously with the object. After the object has been built, the supports are removed. Finally, the object is postprocessed in order to remove residual traces of the supports.

An important issue in this process is choosing an orientation of the model so that it can be built in the vertical direction. Equivalently, we can keep the model fixed, and choose a direction in which the model is built layer by layer. This direction is called the *build direction*. It affects the number of layers, the surface finish, the quantity of support structures used, and their location on the object being built—all of which impact the speed, accuracy, and cost of the process.

1.1 Our results

Let $\mathcal{P}$ be the three-dimensional polyhedron, possibly with holes, that we want to build using LM. Throughout, we assume that the facets are triangles. (This is the standard STL format used in industry.) The number of facets of $\mathcal{P}$ is denoted by n. We consider the following problem:

Problem 1 *Given a facet F of $\mathcal{P}$, compute a description of all build directions for which F is not in contact with supports (i.e., protected from supports).*

This is an important problem because support removal from a facet can affect surface quality and accuracy adversely, thereby impacting the functional properties of critical facets, such as, for instance, facets on gear teeth. This problem, which we define more precisely in Section 2, arose from discussions with engineers at Stratasys, Inc.—a Minnesota-based company specializing in LM.

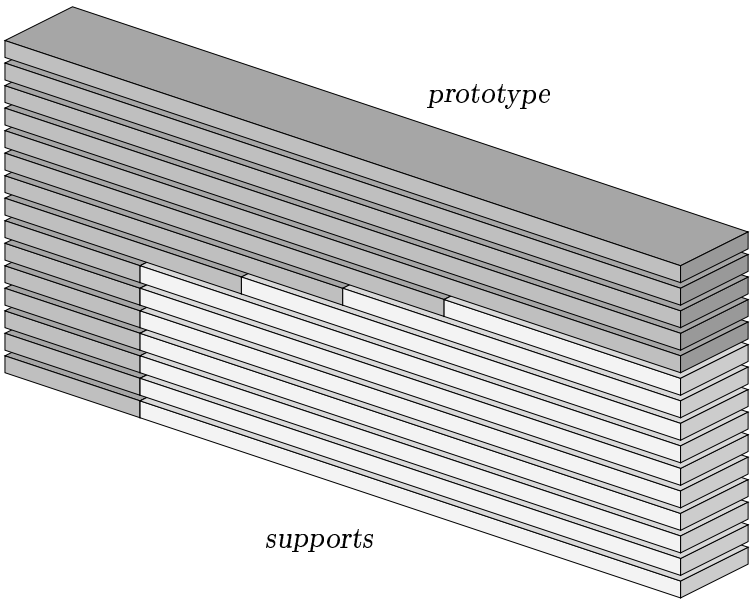

Figure 2: *Illustrating support structures. The model is in dark gray, whereas the supports are in light-gray.*

Later, we will see that a solution to Problem 1 consists of a collection of simple polygons, possibly with holes, on the unit-sphere, all of whose edges are great arcs. In [10], we give an algorithm that solves Problem 1 in $O(n^2)$ time, which can be shown to be worst-case optimal, because the output may consist of $\Omega(n^2)$ connected components. The disadvantage of this algorithm is that its running time is quadratic on *any* input.

In this paper, we describe an implementation of a simplified version of this algorithm, whose running time is acceptable for practical applications. Our implementation is written in C++ and uses LEDA [5]. In particular, we use LEDA's exact rational arithmetic to solve geometric predicates exactly. Hence, our program solves the problem exactly, and is robust, in the sense that it can handle degenerate polyhedra (e.g., several neighboring facets can be co-planar). The most interesting and challenging part of the algorithm involves computing the boundary of the union of a collection of convex spherical polygons, whose edges are great arcs.

We have tested our program on polyhedral models and on collections of random triangles. We also discuss several simple heuristics, and show that these significantly improve the running time of our implementation. (In fact, in one case, these heuristics improved the running time from more than eleven weeks to just 175 seconds.)

This paper can be considered as the practical counterpart of the theoretical paper [10]. The latter paper mainly concentrates on the non-trivial correctness proof of our algorithm. The emphasis of the current paper is on the implementation, the development of the heuristics, and the extensive testing of our program. We believe that the heuristics may be of independent interest as they could help in speeding up the implementation of other geometric algorithms on the sphere.

1.2 Related work

This paper continues our work on the implementation of algorithms for geometric problems that arise in Layered Manufacturing. Most of these problems lead to algorithms for geometric objects on the unit-sphere, rather than in a Euclidean space. In [11], we discussed our implementation of an algorithm that computes the width of a polyhedron. (See also [9].) Computing the width leads to the problem of computing the intersections of great arcs on the unit-sphere, and to the point location problem in planar graphs on the unit-sphere. See also [3, 4, 8].

In the geometry chapter of the LEDA-book by Mehlhorn and Näher [5], algorithms are given that perform Boolean operations on polygons in the Euclidean plane $\mathbb{R}^2$. They use a sweep algorithm that computes the planar graph obtained by overlaying the polygons. Then, they use this overlay to perform the Boolean operation on the two polygons. Our algorithm is different: we compute the boundary of the union of spherical polygons incrementally, adding one polygon at a time. Since in our application, many of the intersection points will not be on the boundary of the union, this leads to an algorithm that is faster in practice.

We are not aware of any previous implementations that compute the boundary of the union of spherical polygons.

From the algorithmic side, there is related work by Nurmi and Sack [7]. They consider the following problem: Given a convex polyhedron A and a set of convex polyhedral obstacles, compute all directions of translations that move A arbitrarily far away such that no collision occurs between A and any of the obstacles. If we take for A a facet F of a polyhedron $\mathcal{P}$, and for the obstacles the other facets of $\mathcal{P}$, then we basically get Problem 1. Our algorithm for solving Problem 1 is similar to that of Nurmi and Sack. They only considered the problem from the theoretical point of view and did not give an implementation.

2 Geometric preliminaries

The *unit-sphere*, i.e., the boundary of the three-dimensional ball centered at the origin and having radius one, is denoted by $\mathbb{S}^2$. We consider *directions* as points—or unit-vectors—on $\mathbb{S}^2$. For any point $x \in \mathbb{R}^3$, and any direction $\mathbf{d} \in \mathbb{S}^2$, we denote by $r_{x\mathbf{d}}$, the ray emanating from x having direction $\mathbf{d}$.

Let F be a facet of $\mathcal{P}$, $\mathbf{n}_F$ the outer normal of F, and $\mathbf{d} \in \mathbb{S}^2$ a direction. If $\mathbf{n}_F \cdot \mathbf{d} > 0$, then we say that F is a *front facet w.r.t.* $\mathbf{d}$. Similarly, if $\mathbf{n}_F \cdot \mathbf{d} < 0$, then we say that F is a *back facet w.r.t.* $\mathbf{d}$. Finally, if $\mathbf{n}_F \cdot \mathbf{d} = 0$, then we say that $\mathbf{d}$ is *parallel* to F.

In this paper, we will not consider directions $\mathbf{d}$ that are parallel to facet F. (For a discussion of how these directions can be handled, we refer the reader to [10].)

Let $\mathbf{d}$ be a direction that is not parallel to F, and let x be a point on F. We say that point x is *in contact with supports for build direction* $\mathbf{d}$, if one of the following two conditions holds.

1. F is a back facet w.r.t. $\mathbf{d}$.
2. F is a front facet w.r.t. $\mathbf{d}$, and the ray $r_{x\mathbf{d}}$ intersects the boundary of $\mathcal{P}$ in a point that is not on facet F.

We say that facet F is *in contact with supports for build direction* $\mathbf{d}$, if there is a point in the interior of F that is in contact with supports for build direction $\mathbf{d}$.

In Figure 3, we illustrate the two-dimensional variant of the notion of being in contact with supports, for a planar simple polygon. Note that no interior point of the vertical edges $(5, 6)$ and

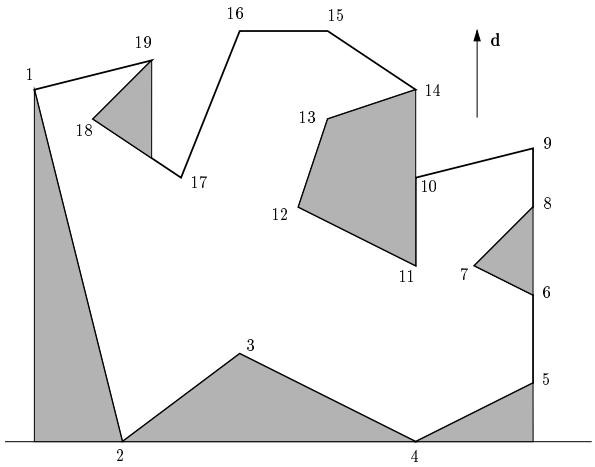

Figure 3: *The shaded regions are the supports for the vertical build direction* $\mathbf{d}$.

$(8,9)$ is in contact with supports. On the other hand, all points of the vertical edge $(10,11)$ are in contact with supports. For build direction $\mathbf{d}$, the edges $(5,6)$, $(8,9)$, $(9,10)$, $(14,15)$, $(15,16)$, $(16,17)$, and $(19,1)$ are not in contact with supports, whereas the remaining edges are in contact with supports.

Let V be a subset of $\mathbb{S}^2$. We say that V is *spherically convex*, if for all points $\mathbf{d}$ and $\mathbf{d}'$ in V, the shortest great arc joining $\mathbf{d}$ and $\mathbf{d}'$ is completely contained in V. We call this shortest great arc a *geodesic*. (If $\mathbf{d}$ and $\mathbf{d}'$ are not antipodal, then this geodesic is unique. Otherwise, every great arc joining $\mathbf{d}$ and $\mathbf{d}'$ must be contained in V. In the latter case, the set V is the entire unit-sphere.)

The *spherical convex hull* of a finite set D of points on $\mathbb{S}^2$ is defined as the smallest spherically convex set that contains all points of D. We say that the set D is *hemispherical*, if there is a three-dimensional plane H through the origin, such that all elements of D are strictly on one side of H. If D is hemispherical, then the spherical convex hull of D is not the entire unit-sphere. In this case, the spherical convex hull is a *spherical polygon*, i.e., its vertices are points of D, and each of its edges is a great arc.

3 Protecting one facet from supports

Let F be a fixed facet of $\mathcal{P}$. We will describe the basic approach for solving Problem 1 for facet F. For the details, especially the non-trivial correctness proofs, the reader is referred to [10].

The idea is as follows. For each facet G, $G \neq F$, we define a set $C_{FG} \subseteq \mathbb{S}^2$ of directions, such that for each $\mathbf{d} \in C_{FG}$, facet F is in contact with supports for build direction $\mathbf{d}$ "because of" facet G. That is, there is a point x in the interior of F, such that the ray $r_{x\mathbf{d}}$ emanating from x and having direction $\mathbf{d}$ intersects facet G. Hence, for each direction in the complement of the union of all sets C_{FG}, $G \neq F$, facet F is not in contact with supports. Because of several special cases that can occur, we have to be careful when formalizing this idea. We will do this in Section 3.1.

For any facet G of $\mathcal{P}$, the set C_{FG} is formally defined as follows:

$$C_{FG} := \bigcup_{x \in F} \{\mathbf{d} \in \mathbb{S}^2 : (r_{x\mathbf{d}} \cap G) \setminus \{x\} \neq \emptyset\}.$$

3.1 When is facet F not in contact with supports?

We say that a facet G is *not below* facet F, if at least one vertex of G is strictly on the same side of the plane through F as the outer normal of F. We denote by $\mathcal{U}_F$ the union of the sets C_{FG}, where G ranges over all facets that are not below F.

Note that a back facet is completely in contact with supports. The following lemma states when a front facet F is not in contact with supports.

Lemma 1 ([10]) *Let* $\mathbf{d}$ *be a direction on* $\mathbb{S}^2$ *such that* F *is a front facet w.r.t.* $\mathbf{d}$*. Facet* F *is not in contact with supports for build direction* $\mathbf{d}$ *if and only if*

1. $\mathbf{d}$ *is not in the interior of* $\mathcal{U}_F$*, or*
2. $\mathbf{d}$ *is in the interior of* $\mathcal{U}_F$*, but not in the interior of* C_{FG}*, for any facet* G *that is not below* F*. (In this case,* $\mathbf{d}$ *is on the boundary of at least two such sets* C_{FG}*.)*

3.2 The algorithm

We denote by P_F the great circle on $\mathbb{S}^2$ consisting of all directions that are parallel to F. The part of the unit-sphere consisting of all directions that are on P_F or on the same side of P_F as the outer normal of F will be called the *F-hemisphere*.

Lemma 1 immediately implies an algorithm for solving Problem 1. We remark that—for simplicity—in our implementation, we did not consider directions $\mathbf{d}$ that satisfy the second condition of Lemma 1. Hence, our algorithm computes a subset of all directions for which facet F is not in contact with supports.

Step 1: For each facet G of $\mathcal{P}$ that is not below F, compute the part of the boundary of the set C_{FG} that is contained in the F-hemisphere. We will see below that this part of C_{FG} is a spherical polygon on the F-hemisphere, all of whose edges are great arcs.

Step 2: Compute and report the boundary of the union of the parts of the spherical polygons C_{FG} that were computed in Step 1.

It follows from Lemma 1 that for each direction that is in the F-hemisphere, and that is not in the reported union, facet F is not in contact with supports.

3.2.1 Characterizing the sets C_{FG}

At first sight, it is not clear what the sets C_{FG} look like. We will see that these sets can in fact be computed easily. It turns out that we have to distinguish two cases.

For any two distinct points s and t in $\mathbb{R}^3$, we denote by $\mathbf{d}_{st}$ the point on $\mathbb{S}^2$ having the same direction as the directed line segment from s to t. Let G be a facet of $\mathcal{P}$, such that G is not below F.

Case 1: F and G are disjoint, or intersect in a single point which is a vertex of both facets. Let

$$D_{FG} := \{\mathbf{d}_{st} \in \mathbb{S}^2 : s \text{ is a vertex of } F,\ t \text{ is a vertex of } G,\ s \neq t\}.$$

In [10], we show that C_{FG} is the spherical convex hull of the at most nine directions in D_{FG}.

Case 2: F and G share an edge. Note that F and G are not coplanar, because G is not below F. Also, for each vertex t of G, one of the following is true: t is in the plane through F, or t is on the same side of the plane through F as the outer normal of F. Consider the great circles P_F and P_G

consisting of all directions that are parallel to F and G, respectively. Then C_{FG} is the set of all directions that are

1. on or on the same side of P_F as the outer normal of facet F, and
2. on or on the same side of P_G as the inner normal of facet G.

Note that in this case, the set C_{FG} is not spherically convex, because it contains antipodal directions.

4 The implementation

In this section, we give some details about the implementation. The program is written in C++ and uses LEDA 3.8 and its rational arithmetic to solve geometric predicates exactly. The program takes as input a file in STL-format, and a non-negative integer, which indicates the number in the STL-file of the facet F that we want to protect from supports.

The STL-file contains the facets of the triangulated polyhedron, where each facet is specified by three vertices and an outer normal. The coordinates of these vertices are converted to rational points (`d3_rat_point`) from LEDA. Since the outer normals in the STL-file may not be exact, we recompute them by computing a cross product based on the three rational vertices defining a facet.

Recall that for each facet G, $G \neq F$, the set C_{FG} is a spherical polygon. In Case 1 of Section 3.2.1, each vertex of this polygon is a direction $\mathbf{d}_{st}$, for some vertex s of F, and some vertex t of G. If we want to compute this direction, then we have to use the expensive and inexact square root operation (to normalize the vector). In order to avoid this, we *represent* any direction $\mathbf{d}$ by a non-zero vector $\mathbf{d}'$ having the same direction as the ray from the origin through $\mathbf{d}$, and use $\mathbf{d}'$ in our program. (We have used this already in our previous work [11].) Hence, this vector does not necessarily have length one. As a result, we have to implement all geometric primitives—which actually operate on unit-vectors—using these vectors.

4.1 Computing the sets C_{FG}

Let G be a facet of $\mathcal{P}$, $G \neq F$. How do we compute the vertices of the set C_{FG} using our representation of directions? To answer this question, consider the set

$$S_{FG} := \{t - s \in \mathbb{R}^3 : s \text{ is a vertex of } F,\ t \text{ is a vertex of } G\}.$$

Let S be the set $S_{FG} \cup \{(0,0,0)\}$, and consider the convex hull $CH(S)$ of this three-dimensional point set S.

First assume that the origin is a vertex of $CH(S)$. Then we are in Case 1 of Section 3.2.1. Consider the set of all vertices of $CH(S)$ that are connected by an edge to the origin. This set is a representation of the set of all directions that form the vertices of the spherical convex hull of the set

$$D_{FG} = \{\mathbf{d}_{st} \in \mathbb{S}^2 : s \text{ is a vertex of } F,\ t \text{ is a vertex of } G,\ s \neq t\}.$$

Hence, the vertices of $CH(S)$ that are connected by an edge to the origin give us representations of the unit-vectors that are the vertices of the set C_{FG}.

The other case is when the origin is not a vertex of $CH(S)$. Then we are in Case 2 of Section 3.2.1. In this case, vectors that represent the vertices of the spherical polygon C_{FG} can easily be computed using the normal vectors of the facets F and G.

Hence, the vertices of the set C_{FG} can be computed by using any program that computes the convex hull of a three-dimensional point set. In our program, we use LEDA's `D3_HULL` to compute the convex hull of the set S. Note that S contains at most ten points.

4.2 Computing intersections between great arcs

In order to compute the boundary of the union of the parts of the spherical polygons C_{FG} that are on the F-hemisphere, we need to compute intersections between edges of these parts. Recall that each edge is a great arc on the F-hemisphere, and that our program represents each endpoint of an edge by a vector that can have an arbitrary length. We claim that we can simulate all operations that are needed to compute the intersections between these great arcs by operations that only use our representations of directions.

Let us give two examples. Consider two great arcs $(\mathbf{a}, \mathbf{b})$ and $(\mathbf{c}, \mathbf{d})$ that are both on the F-hemisphere. Let a', b', c', and d' be the vectors that represent the unit-vectors $\mathbf{a}, \mathbf{b}, \mathbf{c}$, and $\mathbf{d}$, respectively.

Assume that we know that the two arcs $(\mathbf{a}, \mathbf{b})$ and $(\mathbf{c}, \mathbf{d})$ have a unique intersection point, which we denote by $\mathbf{e}$. We compute a representation of this intersection point, as follows.

Let H be the plane through the origin, a', and b'. This plane has the cross product $a' \times b'$ as its normal vector. (Note that the plane through the origin, and the unit-vectors $\mathbf{a}$ and $\mathbf{b}$ is equal to H. We do not, however, "know" these two unit-vectors.) Similarly, let H' be the plane through the origin, c', and d'. This plane has the cross product $c' \times d'$ as its normal vector. Let ℓ be the line of intersection of the planes H and H'. Note that this line contains the cross product of the normal vectors of H and H'; hence it can easily be computed. It is clear that the intersection point $\mathbf{e}$ is one of the two intersection points of the line ℓ with the unit-sphere. Hence, if we take the cross product of the normal vectors of the two planes H and H'—oriented appropriately—then we get a vector which represents the intersection point $\mathbf{e}$.

As a second example, assume we want to test if the direction $\mathbf{d}$ is in the interior of the great arc $(\mathbf{a}, \mathbf{b})$. This is done in the following way.

Let H be the plane through the origin, a', and b', and let n_H be its normal vector. Hence, $n_H = a' \times b'$. Note that, in general, n_H does not have length one. Let H_1 be the plane through the origin, a', and n_H, and let H_2 be the plane through the origin, b', and n_H. Then the unit-vector $\mathbf{d}$ is in the interior of the great arc $(\mathbf{a}, \mathbf{b})$ if and only if

1. $d' \in H$,
2. b' and d' are strictly on the same side of H_1, and
3. a' and d' are strictly on the same side of H_2.

These three conditions can be tested using the LEDA functions `H.side_of(x)` and `H.contains(x)`, where `H` is a `d3_rat_plane`, and `x` is a `d3_rat_point`.

4.3 Computing the union of the sets C_{FG}

Consider the sets C_{FG}, where G ranges over all facets of $\mathcal{P}$ that are not below F. For each such set C_{FG}, we compute the part of its boundary that is on the F-hemisphere. For simplicity, we denote this part again by C_{FG}. We want to compute the boundary of the union of these sets.

In [10], we give an algorithm that computes this boundary in $O(n^2)$ time. The basic approach of this algorithm is as follows. Recall that the edges of the sets C_{FG} are great arcs. The algorithm computes the arrangement of the great circles containing these great arcs. Then by traversing the arrangement, the boundary of the union of the sets C_{FG} is computed. Although this algorithm is worst-case optimal, the disadvantage is that its running time is $\Theta(n^2)$ on any input.

An alternative approach would be to use an algorithm that is similar to the Bentley-Ottmann sweep algorithm [1] for computing the intersections of line segments in the Euclidean plane. In our

case, we would sweep a half-circle over the unit-sphere, and compute the intersections of the great arcs that form the boundaries of the sets C_{FG}. From these intersections, we could then deduce the boundary of the union. This approach has as a disadvantage that *all* intersections are computed. We are, however, only interested in intersections that are on the boundary of the union. Since we expect the number of boundary-intersections to be much smaller than the total number of intersections (in fact, our experiments confirm this), we did not implement this sweep algorithm.

Our program computes the boundary of the union of the sets C_{FG} using an incremental algorithm. That is, it starts with an initial spherical polygon C_{FG}. Then, the other polygons C_{FG} are added one after another, and the boundary of the union of the polygons that have been added so far is maintained.

The current boundary of the union consists of a list L of spherical polygons. Each polygon is represented by an ordered list of vectors. The ordering of these vectors is such that the interior of the union is to the "left" of the edges. Note that a polygon of L may represent a hole in the union.

Let us briefly sketch how a polygon C_{FG} is added to the current boundary of the union. Recall that C_{FG} has at most nine edges. For each edge e of C_{FG}, we walk through all polygons of the list L, compute all edges of these polygons that intersect e, and insert the intersections into the polygons of L. (Here, we have to be careful with degenerate intersections. For example, two edges may overlap, or may intersect in their endpoints.) After we have done this for all edges of C_{FG}, we again walk along all polygons of L, and compute the list of polygons that form the boundary of the new union. This step had to be implemented very carefully, because of the many degenerate cases that may occur. For example, one vector may represent several vertices in one single polygon of L, see Figure 6.

Since the total number of vertices of the polygons in L can be $\Theta(n^2)$ in the worst case, the time to add one polygon C_{FG} to the current union can be quadratic in n. Therefore, the worst-case time of our incremental algorithm is $\Theta(n^3)$. Our experiments, however, show that on polyhedral models that are used in practice, the number of vertices on the boundary of the union is quite small and, as a result, the program is fast.

4.4 Some heuristics to speed up the implementation

The first version of our program was very slow. In this section, we give some heuristics that significantly increased the performance on real-world polyhedral models.

Heuristic 1: Let $\mathbf{a}, \mathbf{b}$ and $\mathbf{c}$ be three distinct directions on the F-hemisphere. We say that these directions are *collinear*, if they are contained in a great arc. When we add a polygon C_{FG} to the current boundary of the union, collinear points may arise. In this first heuristic, we delete the "middle" point $\mathbf{b}$ of each collinear triple, provided that $\mathbf{a}$ and $\mathbf{c}$ are not antipodal. (Clearly, such a "middle" point is redundant.)

Heuristic 2: The vectors that represent directions on $\mathbb{S}^2$ are stored as rational points (`d3_rat_point`) from LEDA. In this representation, a three-dimensional point with rational Cartesian coordinates (a, b, c) is stored using homogeneous coordinates (x, y, z, w) of arbitrary length integers, where $a = x/w$, $b = y/w$, $c = z/w$, and $w > 0$. When computing normal vectors, or the intersection of two great arcs, the fourth coordinate w may become a very large integer. However, since all Cartesian points $(\lambda a, \lambda b, \lambda c)$, $\lambda > 0$, represent the same direction on the unit-sphere, we can set the fourth homogeneous coordinate w to one. This is what we did in the second heuristic.

Heuristic 3: This heuristic is also based on the representation of rational points. When computing normal vectors, or the intersection of two great arcs, Cartesian coordinates of points have to be

multiplied. When two rational numbers, say p/q and r/s, are multiplied, LEDA stores the result as $(pr)/(qs)$. That is, the resulting rational is not simplified. By running our program, we noticed that this leads to extremely large integers.

In this third heuristic, we simplify the homogeneous coordinates $(x, y, z, w = 1)$, as follows. We compute the greatest common divisor g of x, y, and z. Then, we divide x, y, and z by g. (These operations are supported by LEDA.) Note that the homogeneous points $(x, y, z, 1)$ and $(x/g, y/g, z/g, 1)$ represent the same *direction* on $\mathbb{S}^2$. Therefore, we use the representation $(x/g, y/g, z/g, 1)$, whose coordinates are, in general, smaller than the ones in the original representation $(x, y, z, 1)$.

Performing this simplify-operation each time when a polygon C_{FG} is added did not improve the running time of the program. Our initial experiments yielded the best speed-up by simplifying homogeneous points $(x, y, z, 1)$ when (i) x, y, and z are all greater than or equal to 10^{24}, and (ii) each time when 20 polygons C_{FG} have been added.

Heuristic 4: Recall that we compute the boundary of the sets C_{FG} in an incremental way. Clearly, if the current union covers the entire F-hemisphere, we can stop the computation, because the remaining sets C_{FG} do not change the union. In early versions of our program, we did not check this condition.

In the fourth heuristic, each time a polygon C_{FG} has been added, we check if the F-hemisphere is covered by the current union. If so, the program stops. Otherwise, the next polygon C_{FG} is added.

Heuristic 5: The running time of the program depends on the order in which the polygons C_{FG} are added. In all previous versions of the program, these polygons are added in the order in which the facets G appear in the input file. In this final heuristic, the polygons C_{FG} are added in random order.

5 Experimental results

As mentioned already, the program is written in C++ using LEDA 3.8 [5]. We did our experiments on a SUN Ultra (300 MHz, 512 MByte RAM). In these experiments, we ran the program on real-world polyhedral models from Stratasys, and on collections of random triangles.

5.1 Experiments on polyhedral models

The models that we tested are: (i) `rd_yelo.stl`, a long rod, with grooves cut along its length. The two ends of the rod are decagons; (ii) `cover-5.stl`, a rectangular object, with three vertical sides. The front has a rectangular cut-out. The object resembles a drawer for a filing cabinet; (iii) `tod21.stl`, a bracket, consisting of a hollow quarter-cylinder, with two flanges at the ends, and a through-hole drilled in one of the flanges; and (iv) `mj.stl`, a curved part with a base and a protrusion, shaped like a pistol; see Figure 4.

Each model is given as an STL-file. Such a file contains the facets of the triangulated polyhedron, where each facet is specified by three vertices and an outer normal, given to seven decimal digits of precision. For each model and for different facets F that we wish to protect in the model, we ran six versions of the program, and measured the running times after reading the input. In the zeroth version, none of the heuristics mentioned in Section 4.4 is used. For each i, $1 \leq i \leq 5$, version i of the program uses heuristics 1 through i.

It is clear that the actual performance heavily depends on the model and the facet F. We indeed observed that the running times vary heavily for different facets F. Table 1 gives our test results

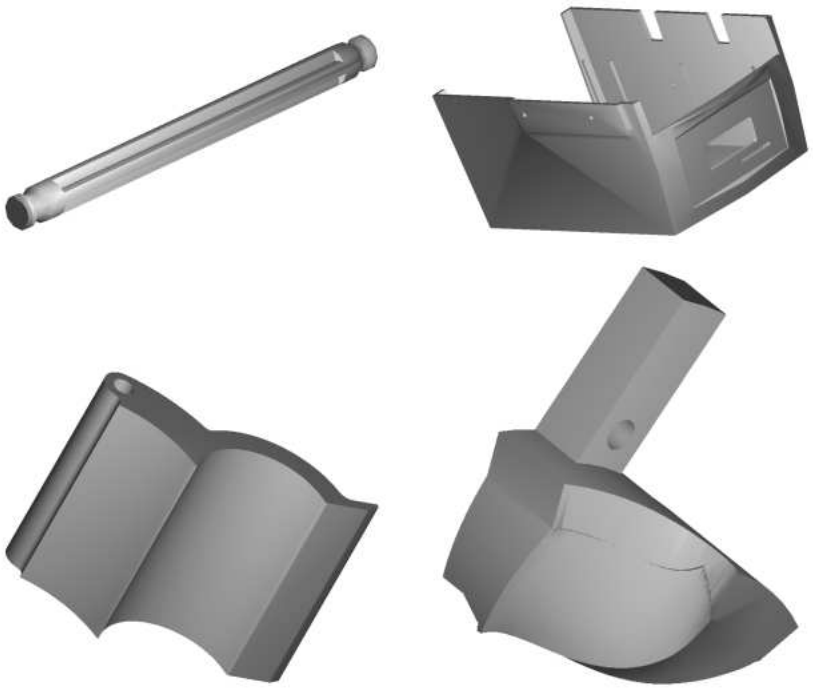

Figure 4: *The models* `rd_yelo.stl` *(top left),* `cover-5.stl` *(top right),* `tod21.stl` *(bottom left), and* `mj.stl` *(bottom right).*

---

for some "extreme" facets. The interpretation of this table is as follows.

1. $\#F$: the number of the facet F for which we ran the program.
2. $\#C_{FG}$: the number of facets G that are not below F. Hence, we compute the boundary of the union of $\#C_{FG}$ spherical polygons.
3. $\#int$: the total number of pairs of edges of all sets C_{FG} that intersect. These numbers were computed using a brute-force algorithm.
4. For each i, $0 \leq i \leq 5$, the column labeled i gives the running time, in seconds, of our program, when it uses heuristics $1, \ldots, i$. Moreover, T denotes the total number of intersections among the sets C_{FG} that are computed during the incremental construction, whereas M denotes the maximum number of vertices on the boundary of the union $\mathcal{U}_F$ during its incremental construction, for these heuristics. Note that T is the same for $1 \leq i \leq 3$. Also, M is the same for $1 \leq i \leq 3$.
5. For `rd_yelo.stl`, `cover-5.stl`, and `tod21.stl`, we ran all six versions of the program for all facets F. For `mj.stl`, we ran all six versions for the first 100 facets F.
6. For each model, and each i, $0 \leq i < 5$, the facet F was chosen such that heuristics $1, \ldots, i+1$ had maximum speed-up compared to heuristics $1, \ldots, i$; these values are given in boldface. For example, for `rd_yelo.stl` and $\#F = 94$, heuristic 1 had a speed-up of $591/35 \approx 17$ compared to the version that uses no heuristics. For each of the first 100 facets of `mj.stl`, we ran the version using heuristics 1–5 twice, and did not get any speed-up compared to heuristics 1–4.

As can be seen from Table 1, the heuristics can dramatically reduce the running time. For example, on model `mj.stl` and $\#F = 65$, the program without any heuristic had not terminated after eleven weeks (indicated by $**$), and the programs with heuristic 1, and heuristics 1 and 2, had not terminated after two weeks (indicated by $*$). On the other hand, using heuristics 1–3, the program took only 175 seconds. Without heuristic 3, we observed that the homogeneous coordinates of the vectors were integers having up to 500 decimal digits. Since the program multiplies these coordinates, this explains the extremely high running time. With heuristic 3, i.e., when simplifying the homogeneous coordinates, the integers were significantly smaller.

For the four models that we tested, most of the improvement in performance is caused by heuristics 1 and 3.

Heuristic 5, i.e., adding the polygons in random order, decreased the performance in many cases. (In fact, for `mj.stl`, we got no improvement at all, for any facet F.) This is caused by the fact that STL-files store facets in a structured order, e.g. neighboring facets of the polyhedron are likely to occur consecutively in the STL-file. Our experiments show that our program takes advantage of this structured order.

Recall that $\#int$ denotes the total number of intersections among the sets C_{FG}, whereas T denotes the number of intersections that are computed during the construction of the union $\mathcal{U}_F$. We expected $\#int$ to be substantially larger than T. Table 1 shows that this is indeed the case for versions one through five. This implies that our program is likely to be much faster than the two alternative approaches that were mentioned in the beginning of Section 4.3

In Table 1, the number T of intersections computed by the program can be larger than $\#int$, the total number of intersections among the sets C_{FG}. (For example, for `mj.stl`, $\#F = 30$, and version 0, T is about four times as large as $\#int$.) This can be explained as follows. Let G_1 and G_2 be two facets that have an edge in common. It is likely that the polygons C_{FG_1} and C_{FG_2} also share an edge, say e. Assume that C_{FG_1} is added before C_{FG_2}, but in between some other polygons C_{FG} are added that split edge e of C_{FG_1} into smaller subedges. When polygon C_{FG_2} is added, edge e of C_{FG_2} intersects each of the subedges of e of C_{FG_1}, yielding a large value of T.

`rd_yelo.stl`; 396 facets			no heuristics			with heuristics										
$\#F$	$\#C_{FG}$	$\#int$	0	T	M	1	2	3	T	M	4	T	M	5	T	M
94	283	18578	**591**	16659	106	**35**	36	20	1280	23	9	548	23	100	5368	71
320	230	14763	213	7542	72	**208**	**172**	51	3892	30	51	3826	30	54	4217	84
87	276	17959	28647	14098	91	28248	**27013**	**30**	2127	23	28	2037	23	120	9447	88
67	283	18431	526	14792	104	34	35	**19**	1187	19	**6**	437	19	120	8406	111
374	188	12699	572	18621	142	56	54	33	2107	23	**29**	1957	23	**7**	334	29
`cover-5.stl`; 906 facets			no heuristics			with heuristics										
$\#F$	$\#C_{FG}$	$\#int$	0	T	M	1	2	3	T	M	4	T	M	5	T	M
875	211	34070	**1286**	32226	304	**24**	23	15	1084	9	15	1084	9	20	1432	14
152	500	29431	1220	97825	345	**51**	**39**	31	2417	21	15	1124	21	170	20440	141
465	832	102572	40481	122187	208	85153	**80670**	**171**	23702	66	174	23702	66	104	13775	118
170	179	26104	865	24095	262	18	18	**11**	551	5	**1**	44	5	18	1131	26
746	893	98665	10562	101840	165	9023	8604	271	34773	84	**246**	33816	84	**27**	2501	71
`tod21.stl`; 1128 facets			no heuristics			with heuristics										
$\#F$	$\#C_{FG}$	$\#int$	0	T	M	1	2	3	T	M	4	T	M	5	T	M
44	856	209036	**2843**	98774	270	**58**	61	35	3238	31	9	802	31	426	66631	196
191	474	57202	877	39537	150	**151**	**136**	83	13541	120	82	13211	120	425	63882	209
425	940	373099	2608	68710	138	1080	**1052**	**109**	14536	73	110	14200	73	330	59568	116
8	505	64187	82	5281	19	23	25	**14**	1532	7	**2**	41	7	258	45944	195
225	506	63785	523	31388	162	157	162	100	17841	142	**97**	17478	142	**21**	1919	77
`mj.stl`; 2832 facets			no heuristics			with heuristics										
$\#F$	$\#C_{FG}$	$\#int$	0	T	M	1	2	3	T	M	4	T	M	5	T	M
30	2251	193424	**24987**	764798	594	**211**	213	84	6961	12	18	310	12	1240	159043	404
60	112	5639	215	5870	95	**38**	**35**	12	592	11	10	469	11	52	3331	46
65	2184	234283	**	**	**	*	*	**175**	11612	15	179	10415	15	901	114208	209
4	478	19636	2989	76648	364	57	58	**27**	1441	7	**3**	22	7	22	1258	25

Table 1: *Performance of our implementation on real-world models. An explanation of this table is given in Section 5.1.*

n	$\#C_{FG}$	$\max \lvert\mathcal{U}_F\rvert$	max	average	variance
100	60	30	21	6	34
200	125	38	49	16	198
300	180	42	84	26	559
400	253	48	121	40	1144
500	305	52	179	54	2293

Table 2: *Performance of our implementation on random triangles, using heuristics 1–4. n denotes the number of triangles. For each n, we ran the program for 1000 random inputs. $\#C_{FG}$ denotes the average number of facets G that are not below F. For each of the 1000 runs, we measured the maximum number of vertices on the boundary of the union $\mathcal{U}_F$ during its incremental construction;* $\max \lvert\mathcal{U}_F\rvert$ *denotes the average maximum number of these numbers.* max, average, *and* variance *denote the maximum and average time in seconds, respectively, and the variance. For each n, the minimum time was less than 0.1 seconds; in this case all facets G are below F.*

5.2 Experiments on random triangles

We also tested our implementation on collections of random triangles. We did the following two experiments.

First, we generated a set S of random triangles, where each vertex was drawn from a uniform distribution in the cube $[-1000; 1000]^3$, using LEDA's generator `random_d3_rat_points_in_cube`. Then, we generated a random triangle $F = (a, b, c)$, whose vertices were drawn from the uniform distribution in the cube $[1100; 3100] \times [-1000; 1000]^2$. The outer normal of F was set to the cross product $(b - a) \times (c - a)$. Then we ran our program for this triangle F, with the triangles G taken from the set S. We ran the version of the program that uses heuristics 1–4 given in Section 4.4. Note that the triangles of S do not—in general—form a polyhedron. These triangles will in general even be intersecting. The triangle F, however, does not intersect any of the triangles in S. On this input, our program computes a description of all directions in which F can be moved without colliding with any of the triangles of S.

For each value of $n \in \{100, 200, \ldots, 500\}$, we generated 1000 sets S of n triangles. (For each set S, we generated one triangle F.) We measured the time of our program after S and F were generated. Table 2 shows the results of this experiment.

As can be seen from Tables 1 and 2, the running time of our program heavily depends on the number of facets G that are not below facet F. This is not a surprise, because this number is equal to the number of spherical polygons whose union we have to compute. In order to get a better understanding of the performance of the part of the program that computes the union of spherical polygons, we did the following experiment.

In the same way as above, we generated a set S of random triangles in the cube $[-1000; 1000]^3$. Then we generated a random point a close to the sphere centered at the origin and having radius 2000, using LEDA's point generator `random_d3_rat_points_on_sphere`. We computed a plane H through point a, having as normal the vector from a to the origin. Finally, we computed two random points b' and c' in the cube $[-1000; 1000]^3$, and their orthogonal projections b and c onto the plane H. We took for F the triangle with vertices a, b, and c. Note that in this case, all triangles of S are not below F. We ran our program for triangle F, with the triangles G taken from the set S. Again, we ran the version of the program that uses heuristics 1–4 given in Section 4.4.

For each value of $n \in \{100, 200, \ldots, 500\}$, we generated 500 sets S of n triangles. (For each set S, we generated one triangle F.) We measured the time of our program after S and F were

n	$\max \lvert\mathcal{U}_F\rvert$	min	max	average	variance
100	80	14	31	21	9
200	106	35	76	54	58
300	123	62	134	94	168
400	138	88	208	141	407
500	148	121	289	189	696

Table 3: *Performance of our implementation on random triangles that are all not below triangle F, using heuristics 1–4. n denotes the number of triangles. For each n, we ran the program for 500 random inputs.* $\max \lvert\mathcal{U}_F\rvert$ *denotes the average maximum number of vertices on the boundary of the union $\mathcal{U}_F$ during its incremental construction;* min, max, average, *and* variance *denote the minimum, maximum, and average time in seconds, respectively, and the variance.*

---

generated. In this case, we always computed the union of exactly n spherical polygons C_{FG}. That is, unlike in the previous experiment, there is no dependence on the number of facets G that are not below F. Table 3 shows the results of this experiment. As can be seen from this table, if we double n, the running time increases by a factor of about 2.6. This suggests a running time that is proportional to $n^{\log 2.6} \approx n^{1.38}$.

6 Verifying the output

The most complicated part of the program is the one that computes the boundary of the union of the sets C_{FG}. The many degenerate cases that can occur caused great problems during the design and debugging phases.

In [6], Mehlhorn *et al.* argue that *program checkers* should be used when implementing geometric algorithms. Unfortunately, we have not been able to design a simple and efficient algorithm that checks whether the output of a spherical polygon-union program is correct.

Instead, we wrote two programs that can be used to graphically verify the part of the program that computes the boundary of the union of spherical polygons. These verification programs take as input a collection of spherical polygons that are on the F-hemisphere. The output consists of a display of the boundary of the union of these polygons.

The first verification program displays the projection of the input polygons in one window, whereas the projection of the boundary of the union is displayed in another window. This program draws a vector v as a point which is obtained as follows. First, we rotate the scene such that the F-hemisphere coincides with the upper hemisphere $\mathbb{S}^2 \cap \{(x, y, z) \in \mathbb{R}^3 : z \geq 0\}$. Then we intersect the ray through the rotated vector v with the plane $z = 1$, and draw the intersection point. If the rotated vector v is in the plane $z = 0$, then it is drawn on the bounding square of the window. See Figures 5 and 6 for some examples.

A disadvantage of this program is that it displays the projection of the boundary of the union of the polygons. Therefore, we also wrote a second verification program—using OpenGL—that gives a three-dimensional view of the output. It displays the boundary of the union of the polygons on an "invisible" sphere. The scene can be rotated, and there is a zoom facility. See Figures 7 and 8 for a three-dimensional view of the union of the polygons of Figures 5 and 6.

We used both these graphical verifiers during the debugging phase of the implementation. They were of tremendous help for finding incorrect handlings of degenerate cases in early versions of the program.

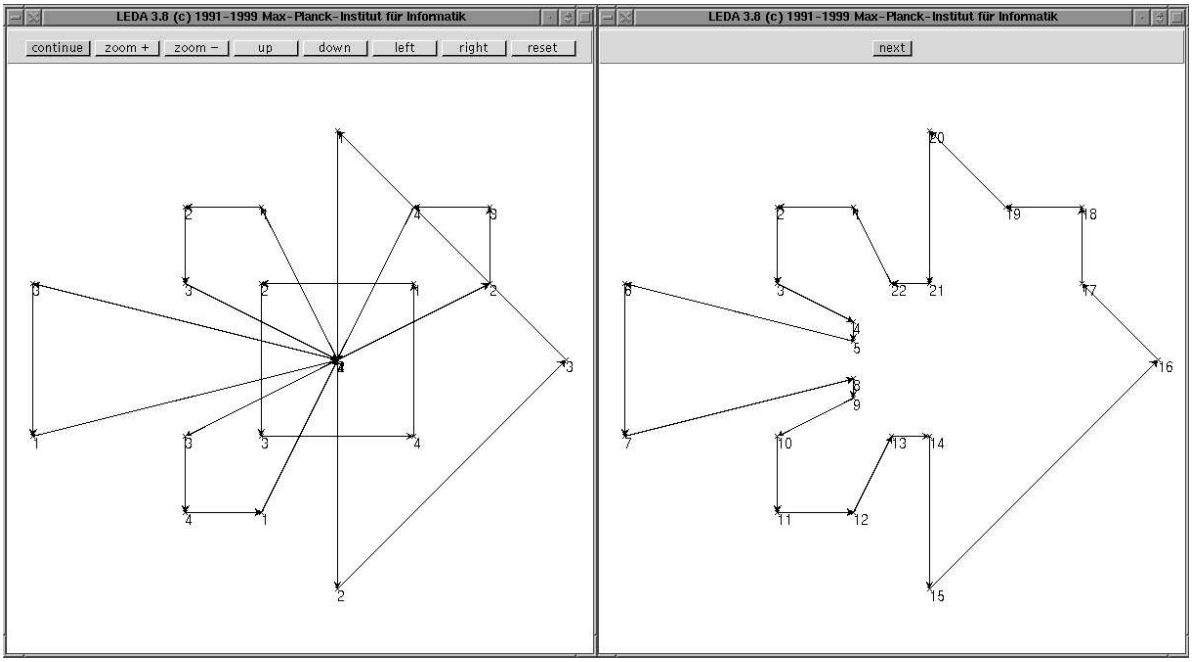

Figure 5: *Illustrating the graphical verifier. The left window shows six spherical polygons. Their union is shown in the right window.*

Finally, we wrote a verification tool that generates random non-zero vectors. Let v be such a vector. Clearly, v is contained in the union of the polygons C_{FG}, if and only if v is contained in at least one of these polygons. Both these conditions can easily be checked. We did a very large number of these tests, and did not find any inconsistencies. We are aware that this does not imply that our program is correct. These tests, however, greatly increased our confidence in the correctness.

7 Concluding remarks and further work

We have given a robust, exact and efficient implementation of an algorithm that solves an important problem in layered manufacturing. The most interesting contribution is the part of the program that computes the boundary of the union of spherical polygons. Representing the vertices of these polygons by vectors that can have an arbitrary length simplified the implementation and avoided numerical problems. We also showed that simple heuristics dramatically improve the performance of the program.

We have designed three tools that can be used to verify the part of the program that computes the boundary of the union of spherical polygons. Clearly, these verifiers do not prove the correctness of the output of our program. We leave open the problem of designing an efficient algorithm that checks (in the sense of [6]) whether the output of a spherical polygon-union program is correct.

Acknowledgements

We thank Larry Roscoe, Don Holzwarth, Jon Holt, Jeff Kotula, and Tom Studanski of Stratasys, Inc. for useful discussions. In particular, they proposed Problem 1 to us. We also thank them for

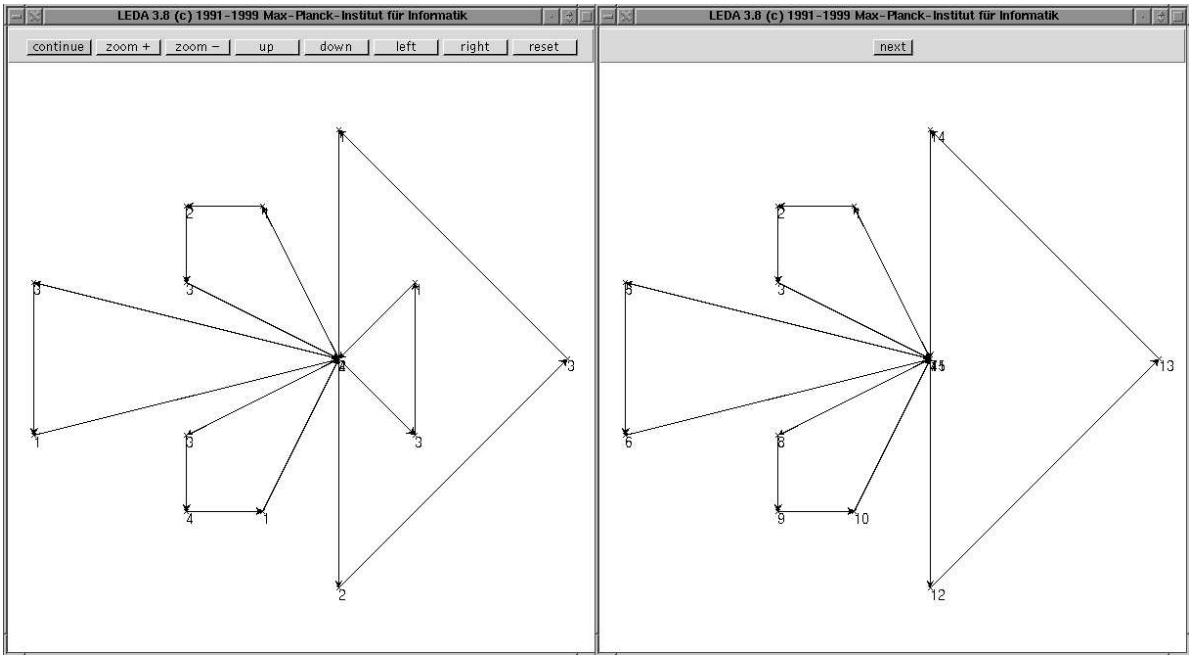

Figure 6: *Illustrating the graphical verifier. The left window shows five spherical polygons. Their union is shown in the right window. This union consists of one single polygon. The vertex in the center occurs four times in this polygon.*

---

allowing us to test our implementation on their polyhedral models.

References

[1] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.*, C-28:643–647, 1979.

[2] P. F. Jacobs. *Rapid Prototyping & Manufacturing: Fundamentals of StereoLithography.* McGraw-Hill, New York, 1992.

[3] J. Majhi, R. Janardan, M. Smid, and P. Gupta. On some geometric optimization problems in layered manufacturing. *Comput. Geom. Theory Appl.*, 12:219–239, 1999.

[4] J. Majhi, R. Janardan, M. Smid, and J. Schwerdt. Multi-criteria geometric optimization problems in layered manufacturing. *To appear in: International Journal of Mathematical Algorithms.*

[5] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing.* Cambridge University Press, Cambridge, U.K., 1999.

[6] K. Mehlhorn, S. Näher, M. Seel, R. Seidel, T. Schilz, S. Schirra, and C. Uhrig. Checking geometric programs or verification of geometric structures. *Comput. Geom. Theory Appl.*, 12:85–103, 1999.

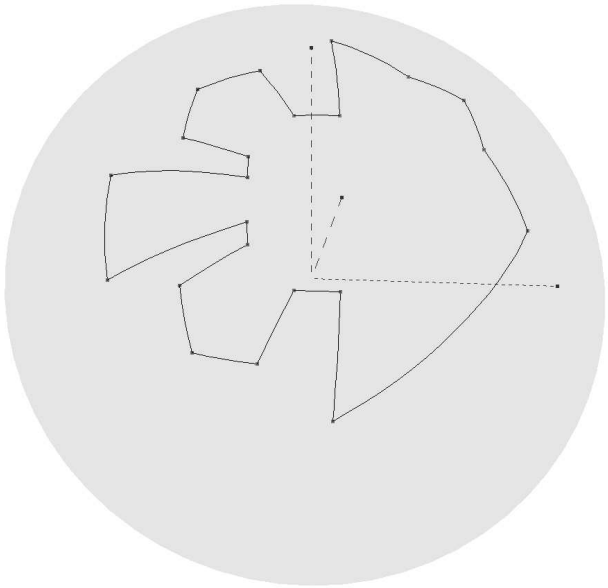

Figure 7: *Illustrating the union of the six spherical polygons of Figure 5, using our OpenGL-based graphical verifier.*

---

[7] O. Nurmi and J.-R. Sack. Separating a polyhedron by one translation from a set of obstacles. In *Proc. 14th Internat. Workshop Graph-Theoret. Concepts Comput. Sci. (WG '88)*, volume 344 of *Lecture Notes Comput. Sci.*, pages 202–212. Springer-Verlag, 1989.

[8] J. Schwerdt. *Entwurf von Optimierungsalgorithmen für geometrische Probleme im Bereich Rapid Prototyping und Manufacturing.* Ph.D. thesis, Department of Computer Science, University of Magdeburg, Magdeburg, Germany, 2001.

[9] J. Schwerdt and M. Smid. Computing the width of a three-dimensional point set: documentation. Report 4, Department of Computer Science, University of Magdeburg, Magdeburg, Germany, 1999.

[10] J. Schwerdt, M. Smid, R. Janardan, E. Johnson, and J. Majhi. Protecting critical facets in layered manufacturing. *Comput. Geom. Theory Appl.*, 16:187–210, 2000.

[11] J. Schwerdt, M. Smid, J. Majhi, and R. Janardan. Computing the width of a three-dimensional point set: an experimental study. *ACM Journal of Experimental Algorithmics*, 4, 1999. Article 8, `http://www.jea.acm.org/1999/SchwerdtWidth/`.

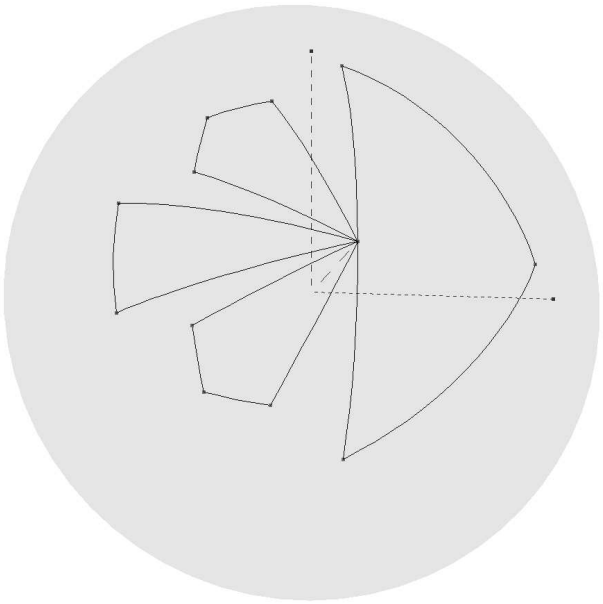

Figure 8: *Illustrating the union of the five spherical polygons of Figure 6, using our OpenGL-based graphical verifier.*