Computing the width of a three-dimensional point set: an experimental study

Jörg Schwerdt, Michiel Smid
University of Magdeburg, Magdeburg, Germany
and
Jayanth Majhi
Mentor Graphics Corporation, Wilsonville, OR, U.S.A.
and
Ravi Janardan
University of Minnesota, Minneapolis, MN, U.S.A.

We describe a robust, exact, and efficient implementation of an algorithm that computes the width of a three-dimensional point set. The algorithm is based on efficient solutions to problems that are at the heart of computational geometry: three-dimensional convex hulls, point location in planar graphs, and computing intersections between line segments. The latter two problems have to be solved for planar graphs and segments on the unit sphere, rather than in the two-dimensional plane. The implementation is based on LEDA, and the geometric objects are represented using exact rational arithmetic.

Categories and Subject Descriptors: F.2.2 [**Nonnumerical Algorithms and Problems**]: Geometrical problems and computations; J.6 [**Computer-Aided Engineering**]: Computer-aided manufacturing (CAM)

General Terms: Computational Geometry, Layered Manufacturing

Additional Key Words and Phrases: Implementation, spherical geometry

---

This work was funded in part by a joint research grant by DAAD and by NSF. Research of JM and RJ also supported in part by NSF grant CCR-9712226.
Address: Jörg Schwerdt, Michiel Smid. Department of Computer Science, University of Magdeburg, D-39106 Magdeburg, Germany. E-mail: {schwerdt,michiel}@isg.cs.uni-magdeburg.de. Jayanth Majhi. Mentor Graphics Corporation, 8005 S.W. Boeckman Road, Wilsonville, OR 97070, U.S.A. E-mail: jayanth_majhi@mentorg.com. Ravi Janardan. Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455, U.S.A. E-mail: janardan@cs.umn.edu.

1. INTRODUCTION

In this paper, we study a geometric problem motivated by applications in an emerging technology called Layered Manufacturing. This technology makes it possible to rapidly build physical models of computer designed three-dimensional models. A specific process of Layered Manufacturing which is in wide use is StereoLithography (see the book by Jacobs [Jacobs 1992]). The input to the process is a surface triangulation of the CAD model in a format called STL.[1] The triangulated model is sliced by horizontal planes into layers, and then built layer by layer in the positive z-direction, as follows. The StereoLithography apparatus consists of a vat of photocurable liquid resin, a platform, and a laser. 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, which rests on the platform. Next, the platform is lowered by the layer thickness and the just-vacated region is re-coated with resin. The next layers are then built in the same way.

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. These support structures are computed before the process starts. They are also sliced into layers, and built simultaneously with the actual 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 amount of support structures used, and the parts of the object that are in contact with supports.

In our recent papers [Majhi et al. 1999a; Majhi et al. 1999b; Majhi et al. 1998], we have considered the problem of computing a build direction which optimizes design criteria (or combinations of criteria), including number of layers, volume of support structures, area of contact between supports and part, and surface finish. We have also studied, in [Schwerdt et al. 1999; Schwerdt et al. 2000], the problem of choosing a build direction which protects prescribed facets from being in contact with supports.

In this paper, we discuss an implementation and experimental results of an algorithm that computes all build directions that minimize the number of layers. This problem turns out to be equivalent to computing the width of a polyhedron [Agarwal and Sharir 1996; Chazelle et al. 1993; Houle and Toussaint 1988], and it leads to problems that are at the heart of computational geometry: three-dimensional convex hulls, point location in planar graphs, and computing intersections between line segments. The latter two problems, however, have to be solved for planar graphs and segments (more precisely, great arcs) on the *unit sphere* rather than in the two-dimensional plane.

[1] Thus the input to the process is polyhedral even if the original model is composed of arbitrary freeform surfaces.

1.1 The width problem

Throughout this paper, $\mathcal{P}$ denotes a polyhedron, possibly with holes, and n denotes the number of its facets. The *unit sphere*, i.e., the boundary of the three-dimensional ball centered at the origin and having radius one, is denoted by $\mathbb{S}^2$. The *upper hemisphere* is defined as

$$\mathbb{S}^2_+ := \mathbb{S}^2 \cap \{(x, y, z) \in \mathbb{R}^3 : z \geq 0\}.$$

Similarly, we define the *lower hemisphere* as

$$\mathbb{S}^2_- := \mathbb{S}^2 \cap \{(x, y, z) \in \mathbb{R}^3 : z \leq 0\}.$$

Finally, the *equator* is the intersection of $\mathbb{S}^2$ with the plane $z = 0$. We consider *directions* as points—or unit vectors—on $\mathbb{S}^2$.

Often, layer thickness in the Layered Manufacturing process is measured in thousandths of an inch. As a result, the number of layers needed to build a model can run into the thousands if the part is oriented along its longest dimension. If the layer thickness is fixed, then the number of layers for a given build direction $\mathbf{d}$ is proportional to the smallest distance between two parallel planes that are normal to $\mathbf{d}$, and which enclose $\mathcal{P}$. We call this smallest distance the *width* of $\mathcal{P}$ *in direction* $\mathbf{d}$, and denote it by $w(\mathbf{d})$. Note that $w(\mathbf{d}) = w(-\mathbf{d})$.

The *width* $W(\mathcal{P})$ of the polyhedron $\mathcal{P}$ is defined as the minimum distance between any two parallel planes that enclose $\mathcal{P}$, i.e.,

$$W(\mathcal{P}) = \min\{w(\mathbf{d}) : \mathbf{d} \in \mathbb{S}^2\}.$$

In this paper, we will consider the following problem: Given the three-dimensional polyhedron $\mathcal{P}$, compute all build directions $\mathbf{d}$ for which $w(\mathbf{d}) = W(\mathcal{P})$. Houle and Toussaint [Houle and Toussaint 1988] gave an algorithm which solves this problem. We have implemented a variant of their algorithm. Our implementation is written in C++ and uses LEDA 3.7 [Mehlhorn and Näher 1999]. In particular, we use efficient data structures from the LEDA library, such as planar maps and dictionaries. Also, we use LEDA's *rational arithmetic* to solve geometric predicates *exactly*. Hence, our implementation solves the problem exactly, and is *robust*, in the sense that it is correct even for a degenerate polyhedron (e.g., several neighboring facets can be co-planar). As far as we know, this is the first exact and robust implementation of an algorithm for computing the width of a three-dimensional polyhedron.

Why do we want to solve predicates exactly? Our implementation uses data structures such as balanced binary search trees, that are based on non-trivial ordering relations. The order of two objects is determined by one or more orientation tests of the form "on which side of a three-dimensional plane is a given point". Solving these tests exactly guarantees that our compare functions define "real" ordering relations, i.e., they are reflexive, anti-symmetric, and transitive. As a result, data structures whose correctness heavily depends on properties of an ordering relation can use these compare functions without having to worry about rounding errors.

The rest of this paper is organized as follows. In Section 2, we describe the algorithm. Section 3 discusses the implementation, especially the primitive operations where two objects on the unit sphere are compared. In Section 4, we present the results of our experiments on real-world polyhedral models obtained from Stratasys, Inc.—a Minnesota-based world leader in Layered Manufacturing. The largest

model we tested has about 200,000 facets; our program computed its width within ten minutes. We also present test results on randomly generated point sets of size up to 100,000. We conclude in Section 5 with directions for future work.

2. THE ALGORITHM

The asymptotically fastest known algorithm for computing the width of a three-dimensional point set is due to Agarwal and Sharir [Agarwal and Sharir 1996]; its expected running time is roughly $O(n^{1.5})$. Our implementation is based on the algorithm of Houle and Toussaint [Houle and Toussaint 1988], which has $O(n^2)$ running time in the worst case. The reason we implemented the latter algorithm is that (i) it is much simpler, (ii) in practice, the running time is much less than quadratic, as our experiments show (Tables 1–4), and (iii) it finds *all* directions that minimize the width. (Finding all optimal directions has applications when computing a build direction that minimizes a multi-criteria function, see [Majhi et al. 1998].)

To compute the width of the polyhedron $\mathcal{P}$, we do the following. First, we compute the convex hull $CH(\mathcal{P})$ of (the vertices of) $\mathcal{P}$. It is clear that the set of directions that minimize the width of $\mathcal{P}$ is equal to the set of directions that minimize the width of $CH(\mathcal{P})$.

Let V be a vertex and F a facet of $CH(\mathcal{P})$. We call (V, F) an *antipodal vertex-facet pair* (or VF-pair), if the two parallel planes containing V and F, respectively, enclose $CH(\mathcal{P})$. (Note that these two planes are unique.) We say that these parallel planes *support* $CH(\mathcal{P})$.

Similarly, two *non-parallel* edges e_0 and e_1 of $CH(\mathcal{P})$ are called an *antipodal edge-edge pair* (or EE-pair), if the two (unique) parallel planes containing e_0 and e_1, respectively, enclose $CH(\mathcal{P})$. Again, we say that these parallel planes *support* $CH(\mathcal{P})$.

In [Houle and Toussaint 1988], it is shown that any direction minimizing the width of $\mathcal{P}$ is perpendicular to the parallel planes associated with some VF- or EE-pair. Therefore, we compute all VF- and EE-pairs, and for each of them compute the distance between the corresponding supporting parallel planes. The smallest distance found is the width $W(\mathcal{P})$ of the polyhedron $\mathcal{P}$.

We now describe how the VF- and EE-pairs can be computed. The *dual graph* G of $CH(\mathcal{P})$ is the planar graph on the unit sphere $\mathbb{S}^2$ that is defined as follows. The vertices of G are the unit outer normals of the facets of $CH(\mathcal{P})$, and two vertices are connected by an edge in G, if the corresponding facets of $CH(\mathcal{P})$ share an edge. Note that edges of this dual graph are great arcs on $\mathbb{S}^2$. Moreover, edges (resp. faces) of G are in one-to-one correspondence with edges (resp. vertices) of $CH(\mathcal{P})$.

We transform G into a planar graph G' on $\mathbb{S}^2$, by cutting all edges that cross the equator, and "adding" the equator to it. Hence, G' contains all vertices of G, and all edges of G that do not cross the equator. Additionally, each edge e of G that crosses the equator is represented in G' by two edges that are obtained by cutting e at the equator. Moreover, by following edges of G', we can completely walk around the equator. Note that edges of G that are on the equator are also edges in G'. (Adding the equator is not really necessary—in fact, in our implementation, we do not even add it. Adding the equator makes the description of the algorithm cleaner, because all faces of the graphs that are defined below are bounded by "real" edges.)

Let G'_u be the subgraph of G' containing all vertices and edges that are in the upper hemisphere $\mathbb{S}^2_+$. Let G'_l be the subgraph of G' containing all vertices and edges that are in the lower hemisphere $\mathbb{S}^2_-$. (Hence, all edges and vertices of G' that are on the equator belong to both G'_u and G'_l.) Finally, let G'_l be the *inverse image* of G'_l, i.e., the graph obtained by mapping each vertex $\mathbf{v}$ in G'_l to the vertex $-\mathbf{v}$. Note that both graphs G'_u and G'_l are in the upper hemisphere $\mathbb{S}^2_+$.

2.1 Computing VF-pairs

Consider a vertex V and a facet F of $CH(\mathcal{P})$ that form a VF-pair. Let f_V be the face of G that corresponds to V, and let $\mathbf{d}_F$ be the vertex of G that corresponds to F. We distinguish two cases.

Case 1: $\mathbf{d}_F$ is on or above the equator. Then $\mathbf{d}_F$ is a vertex of G'_u. Let f'_V be the face of G'_l that is contained in f_V. (Face f_V is completely or partially contained in the lower hemisphere. If f_V was not cut when we transformed G into G', then $f'_V = f_V$. Otherwise, f'_V is that part of f_V that is in the lower hemisphere.) Let f'_V be the face of G'_l that corresponds to f'_V. Since the unique planes that support $CH(\mathcal{P})$ at V and F are parallel, vertex $\mathbf{d}_F$ of G'_u is contained in face f'_V of G'_l.

Case 2: $\mathbf{d}_F$ is strictly below the equator. Then $\mathbf{d}_F$ is a vertex of G'_l, and $-\mathbf{d}_F$ is a vertex of G'_l. Let f'_V be the face of G'_u that is contained in f_V. Since the unique planes that support $CH(\mathcal{P})$ at V and F are parallel, vertex $-\mathbf{d}_F$ of G'_l is contained in face f'_V of G'_u.

It follows that we can find all VF-pairs, by performing a point location query with each vertex of G'_u in the graph G'_l, and performing a point location query with each vertex of G'_l in the graph G'_u. Note that in these point location problems, all query points are known in advance.

We consider some special cases for Case 1. (The analogous special cases for Case 2 can be treated in a similar way.)

First assume that $\mathbf{d}_F$ is strictly above the equator, and is in the interior of an edge of G'_l bounding f'_V. Let g' be the other face of G'_l that has this edge on its boundary, and let g be the face of G'_l that corresponds to g'. Let W be the vertex of $CH(\mathcal{P})$ that corresponds to g. Then the distance between V and the plane through F is the same as the distance between W and the plane through F. Therefore, when locating vertex $\mathbf{d}_F$ in G'_l, it does not matter if we get V or W as answer.

Next assume that $\mathbf{d}_F$ is on the equator, and is in the interior of an edge bounding f'_V. Since G'_l is considered as a graph in the upper hemisphere, the face of G'_l containing $\mathbf{d}_F$ is uniquely defined.

Finally, consider the case when $\mathbf{d}_F$ coincides with a vertex $\mathbf{d}_V$ of f'_V. Let $g \neq f'_V$ be an arbitrary face of G'_l having $\mathbf{d}_V$ as a vertex on its boundary. Let W be the vertex of $CH(\mathcal{P})$ that corresponds to g. Then the distance between V and the plane through F is the same as the distance between W and the plane though F. Therefore, when locating vertex $\mathbf{d}_F$ in G'_l, it does not matter if we get V or W as answer.

2.2 Computing EE-pairs

Consider two edges e_0 and e_1 of $CH(\mathcal{P})$ that form an EE-pair. Recall that these edges are not parallel. Let g_0 and g_1 be the edges of G that correspond to e_0 and e_1, respectively. Then g_0 and g_1 are not both on the equator, and they can have at most one point in common.

Assume w.l.o.g. that g_0 is (completely or partially) contained in the upper hemisphere. Then g_1 is (again, completely or partially) contained in the lower hemisphere. Let g_0' be the part of g_0 that is contained in $\mathbb{S}^2_+$. Then g_0' is an edge of G_u'. Let g_1^0 be the part of g_1 that is contained in $\mathbb{S}^2_-$, and let g_1' be its inverse image. Then g_1' is an edge of G_l'. Since the unique planes that support $CH(\mathcal{P})$ at e_0 and e_1 are parallel, the edges g_0' and g_1' intersect.

Again, we consider some special cases. Assume that one endpoint, say $\mathbf{d}_0$, of g_0' coincides with one endpoint, say $\mathbf{d}_1$, of g_1'. Note that $\mathbf{d}_0 = \mathbf{d}_1$ is a vertex in both G_u' and G_l'. If it is also a vertex in G, then $\mathbf{d}_0$ and $\mathbf{d}_1$ correspond to two facets F_0 and F_1 of $CH(\mathcal{P})$. In this case, the distance between the planes containing F_0 and F_1 is equal to the distance between the parallel planes through e_0 and e_1. Let V be any vertex of F_1. Then we have found the direction $\mathbf{d}_0$ already because of the VF-pair (V, F_0). Hence, we do not have to worry about intersections of this type. So assume that $\mathbf{d}_0$ is not a vertex of G. Then it is on the equator. In this case, we have to find the intersection between g_0' and g_1'.

Next consider the case when one endpoint, say $\mathbf{d}_0$ of g_0', is in the interior of g_1'. Assume that $\mathbf{d}_0$ is a vertex of G. Let F be the facet of $CH(\mathcal{P})$ that corresponds to $\mathbf{d}_0$. Also, let V be one of the endpoints of edge e_1. Then the distance between V and the plane through F is equal to the distance between the parallel planes through e_0 and e_1. Hence, we have found this distance already because of the VF-pair (V, F). So assume that $\mathbf{d}_0$ is not a vertex of G. Then $\mathbf{d}_0$ is on the equator, and g_1' is also on the equator. In this case the edge e_1 is orthogonal to the plane $z = 0$, and we have to find the intersection between g_0' and g_1'.

Hence, we can find all necessary EE-pairs, by computing all edge pairs (g_0', g_1') such that (i) g_0' is an edge of G_u', (ii) g_1' is an edge of G_l', and (iii) g_0' and g_1' intersect in their interiors or the common point of g_0' and g_1' is on the equator.

2.3 The running time of the algorithm

Houle and Toussaint show in [Houle and Toussaint 1988] that the entire algorithm can be implemented such that the worst-case running time is bounded by $O(n^2)$. Moreover, they show that the number of directions minimizing the width can be as large as $\Theta(n^2)$.

In order to get a better understanding of the performance of our implementation, we express the running time as a function of (i) the number n of facets of $\mathcal{P}$, (ii) the number h of facets of $CH(\mathcal{P})$, and (iii) the number k of intersections between edges of G_u' and edges of G_l'.

We compute the convex hull of $\mathcal{P}$ using the LEDA function `D3_HULL`. According to the LEDA user manual [Mehlhorn et al.], this convex hull computation takes $O(n^2)$ time in the worst case, but $O(n \log n)$ time for most inputs.

The graphs G_u' and G_l' have total size $O(h)$. For each of the vertices in one of these graphs, we have to find the face of the other graph that contains the vertex.

We can do this using a sweep algorithm—adapted to the unit sphere—in $O(h \log h)$ time. Hence, given the convex hull of $\mathcal{P}$, the total time for computing all VF-pairs is bounded by $O(h \log h)$.

We compute the intersections between edges of G'_u with edges of G'_l in $O(h \log h + k \log h)$ time, using a variant of the implementation of Bartuschka et al. [Bartuschka et al. 1997] of the Bentley–Ottmann algorithm [Bentley and Ottmann 1979], adapted to great arcs on the unit sphere. This gives all EE-pairs.

The overall worst-case running time of our algorithm is thus bounded by

$$O(n^2 + k \log h) = O(n^2 \log n).$$

REMARK 1. We have reduced the problem of computing the width to problems on the upper hemisphere. Alternatively, we could have used *central projection* [Preparata and Shamos 1988] to map points and great arcs on $\mathbb{S}^2$ to points and line segments in the plane, respectively. The problem with this approach is that points on the equator are projected to points at infinity. As a result, we need compare functions that determine the order of different points at infinity. Stolfi's *oriented projective geometry* [Stolfi 1991] can be used for this, by computing the circular order of points on the "infinite circle". We chose the approach of solving the problems on the upper hemisphere, because all functions that are needed for this are available in LEDA.

3. THE IMPLEMENTATION

In this section, we give some details about the implementation. (The complete program has been documented using `noweb` [Ramsey 1994], and can be found in [Schwerdt and Smid 1999].)

As mentioned already, the program is written in C++ and uses LEDA 3.7 and its rational arithmetic to solve geometric predicates exactly. The program takes as input a set S of three-dimensional points (which are the vertices of the polyhedron $\mathcal{P}$). The coordinates of these points are represented using exact rational numbers (`d3_rat_point`) from LEDA.

First, we use LEDA's `D3_HULL`, which implements an incremental space sweep algorithm that computes the convex hull of the points of S. Given this convex hull, we compute an *implicit* representation of the graph G. Recall that each vertex $\mathbf{v}$ of G is a unit outer normal vector of a hull facet, and is a point on the unit sphere. This point can be computed from the cross product of three of the vertices of the hull facet. Instead of normalizing this point $\mathbf{v}$, we represent it as a non-zero vector having the same direction as the ray from the origin through $\mathbf{v}$. That is, this vector does not necessarily have length one. In this way, we avoid using expensive and inexact arithmetic operations such as square roots. Moreover, all our geometric primitives—which actually operate on unit vectors—can be implemented using these vectors.

The graph G is stored as a `planar_map` from LEDA. Given G, the graphs G'_u and G'_l can easily be computed. Again, the vertices of these two graphs are represented as vectors that do not necessarily have length one.

As explained before, we can now compute all width-minimizing directions, by (i) locating all vertices of G'_u in G'_l and vice versa, and (ii) computing intersections between edges of G'_u with edges of G'_l.

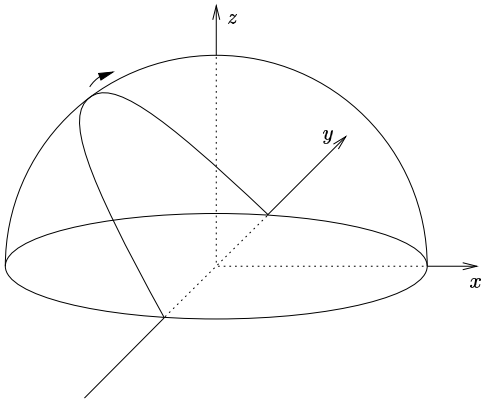

Fig. 1. *Sweeping on the upper hemisphere.*

We solve both (i) and (ii) using algorithms in the style of Bentley–Ottmann. Recall that this algorithm is based on the *plane sweep paradigm*. Because our objects are on the unit sphere, however, we have to adapt this algorithm.

In a plane sweep algorithm for two-dimensional objects, we solve the problem at hand by sweeping a vertical line from left to right over the scene. Sweeping on the upper hemisphere can be thought of as follows. (See Figure 1.) We move a half-circle from the left part of the equator, along the upper hemisphere to the right part of the equator, while keeping the two endpoints of the half-circle on the y-axis. Since we represent points on $\mathbb{S}^2$ as non-zero vectors having arbitrary lengths, we can also regard this as rotating a half-plane around the y-axis. It suffices to implement two types of *compare functions*.

3.1 Comparing two points

We are given two non-zero vectors $\mathbf{u}$ and $\mathbf{v}$, which represent points on the unit sphere, and want to decide which vector is visited first when rotating a half-plane around the y-axis clockwise by 360 degrees, starting from the negative x-axis. Assume $\mathbf{u}$ and $\mathbf{v}$ are visited simultaneously, and let H be the corresponding half-plane. That is, H is the half-plane that contains the y-axis, and the vectors $\mathbf{u}$ and $\mathbf{v}$. Then the order of $\mathbf{u}$ and $\mathbf{v}$ is determined by rotating a ray in H—starting at the negative y-axis—around the origin. Note that this is equivalent to rotating a half-plane that is orthogonal to H and that contains this ray.

To compute the VF- and EE-pairs, it suffices to be able to compare two vectors that are on or above the equator. For completeness, however, we define our compare function for any two non-zero vectors $\mathbf{u}$ and $\mathbf{v}$.

Note that vectors that represent the directions $(0, -1, 0)$ and $(0, 1, 0)$ are always contained in the rotating half-plane. It is natural to define $(0, -1, 0)$ as the minimum of all directions, and $(0, 1, 0)$ as the maximum of all directions.

The basic tool used when comparing two vectors is an orientation test, i.e., deciding whether a point is to the left, on, or to the right of a three-dimensional plane. The complete code for the compare function can be found at the end of this paper. (See also [Schwerdt and Smid 1999, pages 11–16].) We now briefly discuss

this code. A non-zero vector **u** is given as an instance `u` of type `sphere_point`. This point has homogeneous coordinates `u.X()`, `u.Y()`, `u.Z()`, and `u.W()`, which are of LEDA-type `integer`. The value of `u.W()` is always positive.

Consider the two `sphere_points` `u` and `v`. First, it is tested if one of `u` and `v` is the minimal or maximal direction. If this is not the case, then we compute `sweep`, which is a plane of LEDA-type `d3_rat_plane` containing `u` and the y-axis. The normal vector `sweep.normal()` of this plane is "in the sweep direction". Assume that `u.Z()` is positive. Using LEDA's orientation test `sweep.side_of(v)`, we find the position of `v` w.r.t. the plane `sweep`.

Case 1: `sweep.side_of(v)` is positive. Then `u` comes before `v` in the sweep process.

Case 2: `sweep.side_of(v)` is zero. Then `v` is contained in the plane `sweep`.
Case 2.1: `v.Z()` is positive. We compute `sp`, which is a point of LEDA-type `d3_rat_point` having the same coordinates as `u`. Then, we compute `Nsweep`, which is the plane through the origin and `sp`, and that is orthogonal to the plane `sweep`. The result of the comparison follows from the position of `v` w.r.t. `Nsweep`.
Case 2.2: `v.Z()` is less than or equal to zero. Since `v` is not the minimal or maximal direction, and `sweep` is not the plane $z = 0$, point `v` is below the plane $z = 0$, i.e., `v.Z()` is negative. Hence, `u` comes before `v` in the sweep process.

Case 3: `sweep.side_of(v)` is negative. In this case, the result of the comparison follows from the position of `v` w.r.t. the plane $z = 0$.

The cases when `u.Z()` is negative or zero are treated in a similar way.

3.2 Comparing two edges

Here, we are given two edges s_1 and s_2, which represent great arcs on the upper hemisphere. Each edge is specified by its two endpoints, which are given as instances of type `sphere_point`. Both these edges have at least one point in common with the sweep half-circle, and at least one of their endpoints is on the sweep half-circle. We want to determine the order of s_1 and s_2 along the sweep half-circle.

The implementation of this compare function is based on the corresponding function in [Bartuschka et al. 1997]. It uses orientation tests, and the compare function of Section 3.1.

Our implementations of the batched point location algorithm and the spherical segment intersection algorithm use the above compare functions, and closely follow that of Bartuschka et al. [Bartuschka et al. 1997]. Since the latter implementation works for line segments in the plane, we had to re-code it.

4. EXPERIMENTAL RESULTS

In this section, we report on the experiments we did on a SUN Ultra (300 MHz, 512 MByte RAM).

First, we tested our implementation on real-world polyhedral models obtained from Stratasys, Inc. 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 7 decimal digits of precision. Table 1 gives the test results for ten models, which were chosen to encompass different geometries. For

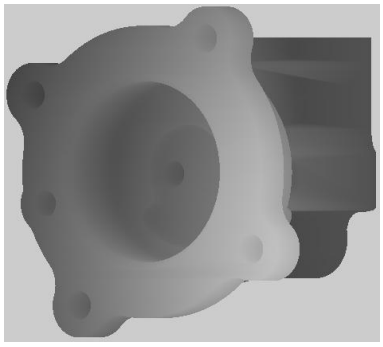

Fig. 2. *The model* `eaton_sp.stl` *in its original orientation.*

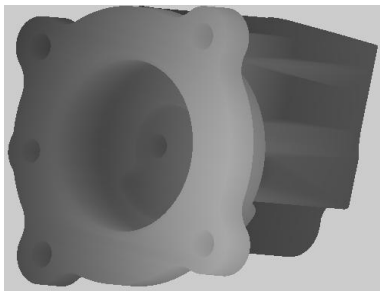

Fig. 3. *The same model in its width-realizing orientation. The coordinate system is as follows: The x-axis is increasing rightwards, the y-axis is increasing upwards, and the z-axis is increasing out of the paper. The optimal direction is along the y-axis.*

example, `tod21.stl` is a bracket, consisting of a hollow quarter-cylinder, with two flanges at the ends, and a through-hole drilled in one of the flanges. This model has 1,128 facets. The model `mj.stl` is an anvil shaped like a pistol with a square barrel. This model has 2,832 facets. The model `eaton_sp.stl`, which has 41,318 facets, is depicted in its original orientation in Figure 2. Figure 3 shows the optimal orientation for this part as found by our algorithm. The largest model tested is `fishb.stl`, which has 213,384 facets. Our program computed the width of the latter model within ten minutes. As can be seen in Table 1, the actual running time of the program heavily depends on the number, h, of facets of the convex hull. This is not surprising, because our compare functions are fairly complex. For each model that we tested, the value of h is much smaller than the number, n, of facets of the model.

Since we only have a limited number of polyhedral models, we also tested our implementation on random point sets. First, we used LEDA's point generator `random_d3_rat_points_in_cube` to generate points whose coordinates are inte-

model	n	h	k	time
`tod21.stl`	1,128	87	20	13
`mj.stl`	2,832	356	29	31
`triad1.stl`	11,352	3,874	3,769	504
`daikin_trt321.stl`	19,402	1,638	3,036	155
`impller.stl`	30,900	414	288	42
`eaton_sp.stl`	41,318	1,065	1,207	114
`4501005.stl`	50,626	2,306	2,063	165
`frame_29.stl`	67,070	388	312	31
`sa600280.stl`	74,350	3,899	2,669	375
`fishb.stl`	213,384	5,459	5,845	566

Table 1. *Performance of our implementation on some polyhedral models. n denotes the number of facets of the model; h and k denote the number of convex hull facets, and the number of EE-pairs, respectively;* time *denotes the time in seconds.*

N	h	k	min	max	average	variance
1,000	133	67	2.1	3.0	2.5	0.870
5,000	205	97	3.2	5.0	4.1	0.266
10,000	239	110	4.0	5.2	4.8	0.144
20,000	254	120	4.8	6.4	5.4	0.177
30,000	269	119	5.0	6.9	6.0	0.300
40,000	265	121	5.3	7.3	6.1	0.384
50,000	269	121	6.0	7.7	6.5	0.293
60,000	267	113	5.9	7.3	6.6	0.167
70,000	280	123	6.4	8.8	7.2	0.620
80,000	266	115	5.8	8.3	7.1	0.459
90,000	268	117	6.0	8.8	7.4	0.819
100,000	274	118	7.2	8.6	7.8	0.213

Table 2. *Performance of our implementation for points randomly chosen in a cube. For each value of N, we randomly generated ten point sets of size N. h and k denote the average number of convex hull facets, and the average number of EE-pairs, respectively. Although k could be $\Theta(h^2)$, this table shows that in practice, it is slightly less than $h/2$.* min, max, average, *and* variance *denote the minimum, maximum, and average time in seconds, respectively, and the variance.*

gers, from a uniform distribution in the cube $[-1000, 1000]^3$. For each value of $N \in \{10^3, \ldots, 10^5\}$, we generated ten point sets of size N. We measured the time of our program after these points were generated. Table 2 shows the minimum, maximum, and average running time in seconds, as well as the variance. This variance was computed using the formula [Graham et al. 1989, Section 8.2]

$$\frac{t_1^2 + t_2^2 + \ldots + t_m^2}{m-1} - \frac{(t_1 + t_2 + \ldots + t_m)^2}{m(m-1)},$$

where t_i denotes the time of the i-th run, and m denotes the total number of generated point sets (which is ten in our case).

Also in Table 2, the average values of h (the number of facets of the convex hull), and k (the number of EE-pairs) are given. Note that for this distribution, the expected value of h is bounded by $O(\log^2 N)$, see Section 4.1 in [Preparata and Shamos 1988].

Although the worst-case running time of the algorithm is $\Theta(N^2 \log N)$, our experimental results show that on random inputs, the algorithm is much faster. As we saw before, the actual worst-case performance is bounded by $O(N^2 + k \log h)$. As we can see in Table 2, the value of h is much smaller than N. Also, the value of k—which could be as large as $\Theta(h^2)$—is in fact slightly less than $h/2$. Table 2 shows that in practice, the running time is not proportional to N^2: otherwise, doubling N would increase the running time by at least a factor of four. This implies that the constant factor corresponding to the term $k \log h$ is large, and this term basically determines the running time in practice.

Next, we generated points with integer coordinates from a uniform distribution in the ball centered at the origin and having radius 1000, using LEDA's point generator `random_d3_rat_points_in_ball`. For each value of $N \in \{10^3, \ldots, 10^5\}$, we generated ten point sets of size N, and measured the time after these points were generated. The results are given in Table 3. For this distribution, the expected value of h is bounded by $O(\sqrt{N})$, see Section 4.1 in [Preparata and Shamos 1988]. In this case, the value of k is slightly larger than $h/2$. Again, the running time in

N	h	k	min	max	average	variance
1,000	258	141	4.9	5.9	5.3	0.870
5,000	607	325	12.2	14.7	13.4	0.694
10,000	884	477	19.2	20.7	20.0	0.188
20,000	1256	673	28.4	34.9	30.1	3.407
30,000	1535	818	34.6	39.9	37.5	1.815
40,000	1782	962	43.4	46.4	44.1	0.808
50,000	1976	1053	47.1	52.5	50.2	2.493
60,000	2165	1161	53.3	56.9	55.5	1.338
70,000	2346	1254	59.4	64.4	61.4	2.192
80,000	2528	1356	64.7	68.0	66.3	1.352
90,000	2677	1440	68.3	72.7	70.3	1.773
100,000	2810	1501	72.0	79.0	75.0	5.323

Table 3. *Performance of our implementation for points randomly chosen in a ball. For each value of N, we randomly generated ten point sets of size N. h and k denote the average number of convex hull facets, and the average number of EE-pairs, respectively. In this case, the value of k is slightly larger than $h/2$.* min, max, average, *and* variance *denote the minimum, maximum, and average time in seconds, respectively, and the variance.*

practice is not proportional to N^2, but is determined by the term $k \log h$, which has a large constant.

Finally, we generated random point sets with integer coordinates that are close to the sphere centered at the origin and having radius 1000, using LEDA's point generator `random_d3_rat_points_on_sphere`. For each value of $N \in \{10^2, \ldots, 10^4\}$, we generated ten point sets of size N, and measured the time after these points were generated. The results are given in Table 4. For this distribution, (almost) all points are on the convex hull. (Recall that h denotes the number of convex hull facets.) In this case, the value of k is about $2h/3$. By doubling the number of points, the running time in practice increases by a factor that is slightly larger than two.

N	h	k	min	max	average	variance
100	196	122	5.8	6.3	6.0	0.021
500	996	635	34.6	36.2	35.3	0.293
1,000	1,996	1,277	73.8	80.7	76.4	4.715
2,000	3,995	2,538	156.6	162.6	158.2	3.089
3,000	5,992	3,809	239.7	245.2	242.5	3.584
4,000	7,982	5,072	325.0	334.9	330.3	9.343
5,000	9,968	6,315	409.5	424.2	416.0	20.160
6,000	11,962	7,549	496.7	533.6	504.1	121.824
7,000	13,933	8,788	583.4	635.0	596.6	320.541
8,000	15,899	9,997	660.0	696.5	677.7	108.740
9,000	17,882	11,226	740.9	822.4	768.6	443.838
10,000	19,814	12,400	842.2	934.6	864.3	850.981

Table 4. *Performance of our implementation for points randomly chosen close to a sphere. For each value of N, we randomly generated ten point sets of size N. h and k denote the average number of convex hull facets, and the average number of EE-pairs, respectively. In this case, the value of k is about $2h/3$.* min, max, average, *and* variance *denote the minimum, maximum, and average time in seconds, respectively, and the variance.*

5. CONCLUDING REMARKS

We have given a robust, exact and efficient implementation of an algorithm that solves an important problem in computational geometry. This problem has applications to Layered Manufacturing and motion planning.

Determining a good build direction in Layered Manufacturing leads to several other problems for objects on the unit sphere. For example, in [Majhi et al. 1998], we show how spherical Voronoi diagrams can be used to compute among all directions that minimize the width, a direction for which the so-called stair-step error is minimum. We plan to implement these Voronoi diagrams, again by implicitly representing points on $\mathbb{S}^2$ as vectors.

Recently, we implemented an algorithm that computes a description of all build directions for which a prescribed facet of the polyhedral model is not in contact with support structures. This leads to the problems of computing spherical convex hulls and the union of spherical polygons. For details, see [Schwerdt et al. 1999; Schwerdt et al. 2000].

The ideas presented in this paper can be used to solve problems involving line segments in the plane that are possibly unbounded (e.g., point location queries in a Voronoi diagram). If we solve such a problem directly in the plane, then we have to deal with different types of points at infinity. Therefore, a better approach may be to use the inverse central projection, which maps the line segments to great arcs on the upper hemisphere. Then, we can apply our techniques for solving the problem at hand on the unit sphere.

In [Mehlhorn et al. 1999], Mehlhorn et al. argue that *program checking* should be used when implementing geometric algorithms. We leave open the problem of designing a fast algorithm that checks whether the output of a width-minimizing algorithm is correct.

Acknowledgements

We thank Stratasys, Inc. for allowing us to test our implementation on their polyhedral models.

REFERENCES

AGARWAL, P. K. AND SHARIR, M. 1996. Efficient randomized algorithms for some geometric optimization problems. *Discrete Comput. Geom. 16*, 317–337.

BARTUSCHKA, U., MEHLHORN, K., AND NÄHER, S. 1997. A robust and efficient implementation of a sweep line algorithm for the straight line segment intersection problem. In *Proc. Workshop on Algorithm Engineering* (Venice, Italy, 1997), pp. 124–135.

BENTLEY, J. L. AND OTTMANN, T. A. 1979. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput. C-28*, 643–647.

CHAZELLE, B., EDELSBRUNNER, H., GUIBAS, L. J., AND SHARIR, M. 1993. Diameter, width, closest line pair and parametric searching. *Discrete Comput. Geom. 10*, 183–196.

GRAHAM, R. L., KNUTH, D. E., AND PATASHNIK, O. 1989. *Concrete Mathematics*. Addison-Wesley, Reading, MA.

HOULE, M. E. AND TOUSSAINT, G. T. 1988. Computing the width of a set. *IEEE Trans. Pattern Anal. Mach. Intell. PAMI-10*, 761–765.

JACOBS, P. F. 1992. *Rapid Prototyping & Manufacturing: Fundamentals of StereoLithography*. McGraw-Hill, New York.

MAJHI, J., JANARDAN, R., SCHWERDT, J., SMID, M., AND GUPTA, P. 1999a. Minimizing support structures and trapped area in two-dimensional layered manufacturing. *Comput.*

Geom. Theory Appl. 12, 241–267.

MAJHI, J., JANARDAN, R., SMID, M., AND GUPTA, P. 1999b. On some geometric optimization problems in layered manufacturing. *Comput. Geom. Theory Appl. 12*, 219–239.

MAJHI, J., JANARDAN, R., SMID, M., AND SCHWERDT, J. 1998. Multi-criteria geometric optimization problems in layered manufacturing. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.* (1998), pp. 19–28.

MEHLHORN, K. AND NÄHER, S. 1999. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, U.K.

MEHLHORN, K., NÄHER, S., SEEL, M., SEIDEL, R., SCHILZ, T., SCHIRRA, S., AND UHRIG, C. 1999. Checking geometric programs or verification of geometric structures. *Comput. Geom. Theory Appl. 12*, 85–103.

MEHLHORN, K., NÄHER, S., SEEL, M., AND UHRIG, C. *The LEDA user manual*. `http://www.mpi-sb.mpg.de/LEDA/MANUAL/MANUAL.html`: Max-Planck-Institute for Computer Science.

PREPARATA, F. P. AND SHAMOS, M. I. 1988. *Computational Geometry: An Introduction*. Springer-Verlag, New York, NY.

RAMSEY, N. 1994. Literate programming simplified. *IEEE Software 11*, 97–105.

SCHWERDT, J. AND SMID, M. 1999. Computing the width of a three-dimensional point set: documentation. Report 4, Department of Computer Science, University of Magdeburg, Magdeburg, Germany. `http://isgwww.cs.uni-magdeburg.de/~schwerdt/Cwidth.ps`.

SCHWERDT, J., SMID, M., JANARDAN, R., AND JOHNSON, E. 2000. Protecting critical facets in layered manufacturing: implementation and experimental results. In *Proc. 2nd Workshop on Algorithm Engineering and Experiments* (2000), pp. 43–57.

SCHWERDT, J., SMID, M., JANARDAN, R., JOHNSON, E., AND MAJHI, J. 1999. Protecting facets in layered manufacturing. In *Proc. 19th Conf. Found. Softw. Tech. Theoret. Comput. Sci.*, Volume 1738 of *Lecture Notes Comput. Sci.* (1999), pp. 281–292. Springer-Verlag.

STOLFI, J. 1991. *Oriented Projective Geometry: A Framework for Geometric Computations*. Academic Press, New York, NY.

```
int compare(const sphere_point& u, const sphere_point& v)
{
  if( u.X() == 0 && u.Z() == 0)
    {
      if( u.Y() < 0 )      // is u the minimal point?
        {
          if( v.X() == 0 && v.Y() < 0 && v.Z() == 0 )
            {
              return(0);  // u == v
            }
          else
            {
              return(-1);  // u < v
            }
        }
      else
        {  // u.Y() > 0   u is the maximal point
          if( v.X() == 0 && v.Y() > 0 && v.Z() == 0 )
            {
              return(0);  // u == v
            }
          else
            {
              return(1);  // u > v
            }
        }
    }
  if( v.X() == 0 && v.Z() == 0 )
    {
      if( v.Y() < 0 )     // is v the minimal point?
        {
          return(1);  // u > v
        }
      else
        {
          return(-1);  // u < v
        }
    }

  d3_rat_plane sweep( d3_rat_point(0,1,0,1),
                      d3_rat_point(0,-1,0,1),
                      u.rat_point() );

  if(u.Z() > 0)
    {
      if( sweep.side_of( v.rat_point() ) > 0 )
        {
          return(-1);  // u < v
        }
      else
        {
          if( sweep.side_of( v.rat_point() ) == 0 )
            {
              if( v.Z() > 0 )
                {
```

```
                d3_rat_point sp( u.X(), u.Y(), u.Z(), u.W() );
                d3_rat_plane Nsweep( sp + sweep.normal(),
                                     d3_rat_point(0,0,0,1),
                                     sp );
                return(Nsweep.side_of(v.rat_point()));
              }
            else
              {
                return(-1);  // u < v
              }
          }
        else  // sweep.side_of(v) < 0
          {
            if( v.Z() < 0 )
              {
                return(-1);  // u < v
              }
            else
              {
                return(1);  // u > v
              }
          }
      }
  }

else   // u.Z() <= 0
  {
    if( u.Z() < 0 )
      {
        if( sweep.side_of( v.rat_point() ) < 0 )
          {
            return(1);  // u > v
          }
        else
          {
            if( sweep.side_of( v.rat_point() ) == 0 )
              {
                if( v.Z() < 0 )
                  {
                    d3_rat_point sp( u.X(), u.Y(), u.Z(), u.W() );
                    d3_rat_plane Nsweep( sp + sweep.normal(),
                                         d3_rat_point(0,0,0,1),
                                         sp );
                    return(Nsweep.side_of(v.rat_point()));
                  }
                else
                  {
                    return(1);  // u > v
                  }
              }
            else  // sweep.side_of(v) > 0
              {
                if( v.Z() < 0 )
                  {
                    return(-1);  // u < v
                  }
```

```
                else
                  {
                    return(1);  // u > v
                  }
              }
          }
      }

    else  // u.Z() == 0
      {
        if( u.X() < 0 )
          {
            if( v.Z() == 0 && v.X() < 0 )
              {
                d3_rat_point sp(u.X(), u.Y(), u.Z(), u.W());
                d3_rat_plane Nsweep( sp + sweep.normal(),
                                     d3_rat_point(0,0,0,1),
                                     sp );
                return(Nsweep.side_of(v.rat_point()));
              }
            else
              {
                return(-1);  // u < v
              }
          }
        else
          {
            if( u.X() > 0 )
              {
                if(v.Z() < 0)
                  {
                    return(-1);  // u < v
                  }
                else
                  {
                    if( v.Z() == 0 && v.X() > 0 )
                      {
                        d3_rat_point sp(u.X(), u.Y(), u.Z(), u.W());
                        d3_rat_plane Nsweep( sp + sweep.normal(),
                                             d3_rat_point(0,0,0,1),
                                             sp );
                        return(Nsweep.side_of(v.rat_point()));
                      }
                    else
                      {
                        return(1);  // u > v
                      }
                  }
              }
          }
      }
  }
}
```