DECOMPOSITION PROBLEMS IN COMPUTATIONAL GEOMETRY

Leonidas Palios

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

June 1992

© Copyright by Leonidas Palios 1992
All Rights Reserved

$\Sigma\tau o\upsilon\varsigma\ \gamma o\nu\varepsilon\acute{\iota}\varsigma\ \mu o\upsilon,$	To my parents,
$\sigma\tau o\upsilon\varsigma\ \delta\alpha\sigma\kappa\acute{\alpha}\lambda o\upsilon\varsigma\ \mu o\upsilon,$	to my teachers,
$\sigma\tau o\upsilon\varsigma\ \phi\acute{\iota}\lambda o\upsilon\varsigma\ \mu o\upsilon.$	to my friends.

ITHAKA

...

Arriving there is what you're destined for.
But don't hurry the journey at all.
Better if it lasts for years,
so you're old by the time you reach the island,
wealthy with all you've gained on the way,
not expecting Ithaka to make you rich.

Ithaka gave you the marvelous journey.
Without her you wouldn't have set out.
She has nothing left to give you now.

And if you find her poor, Ithaka won't have fooled you.
Wise as you will have become, so full of experience,
you'll have understood by then what these Ithakas mean.

Κωνσταντίνος Καβάφης (1911)
translation by Edmund Keeley and Philip Sherrard

Abstract

Decomposition problems are of major importance in computational geometry, as they allow us to express complicated objects in terms of simpler ones, which are in general easier to process, and often lead to more efficient algorithms. In this thesis, we present two decomposition algorithms on three-dimensional polyhedra, one to partition the boundary of a polyhedron of arbitrary genus into a small number of "well-behaved" pieces, the other to partition a polyhedron of zero genus into tetrahedra.

The first algorithm decomposes the boundary of a polyhedron of r reflex angles into at most $18r-2$ connected pieces, each of which lies on the boundary of its convex hull. A remarkable feature of this result is that the number of these convex-like pieces is independent of the number of vertices. Furthermore, it is linear in r, which contrasts with a quadratic worst-case lower bound on the number of convex pieces needed to decompose the polyhedron itself. The number of new vertices introduced in the process is $O(n)$, and the decomposition can be computed in $O(n+r\log r)$ time.

The second algorithm decomposes a polyhedron of zero genus (a polyhedron homeomorphic to a three-dimensional ball) that has n vertices and r reflex angles into a collection of $O\left(n+r^2\right)$ tetrahedra. The algorithm runs in $O\left(\left(n+r^2\right)\log r\right)$ time and requires $O\left(n+r^2\right)$ space. Up to within a constant factor, the number of tetrahedra produced is optimal in the worst case.

Discussion on an implementation of the second algorithm in the programming language C concludes the thesis. It involves the issues of how to carry out some of the tasks outlined in the theoretical description, how to represent the entities participating in the different stages of the algorithm, and how to deal with finite precision arithmetic.

Acknowledgments

I would like to express my gratitude to my advisor, Bernard Chazelle, not only for his invaluable advice and encouragement during the course of my study at Princeton, but because it was one of his classes that attracted me to Computational Geometry, and he was always available for questions and discussion. His sense of humor, too, will be long remembered.

I am indebted to David Dobkin for letting me use his three-dimensional viewer in its early stages (and for listening patiently to my complaints afterwards), and for his criticisms and suggestions regarding the format and content of this thesis. Thanks are also due to to the second thesis reader, Joel Friedman, for his timely response and helpful comments.

I wish to thank all the graduate students at the Department of Computer Science, whose presence, jokes, or smiles made graduate life much more enjoyable. I am particularly thankful to Alvaro Campos and Marios Dikaiakos, and my officemates, past and present, Brandon Dixon, Sarantos Kapidakis, Makoto Kobayashi, Monika Rauch, and Arthur Watson, for bearing with me these years. Special thanks go to Sarantos Kapidakis for introducing me to Postscript that proved so useful for producing expressive figures.

I would like to thank also the Hellenic Association of Princeton whose activities did not fail to distract me from my thesis work, and the Engineering Library staff that made working at the library a very pleasant experience.

Finally, I am thankful to my family for their support all these years, to Princeton University for giving me the opportunity to study in this wonderful environment, to the Department of Computer Science for providing the resources, and to the staff, in

particular to Melissa Lawson and Sharon Rodgers, for their friendliness and guidance through the maze of bureaucracy.

Preface

Geometric problems emerged very early in man's quest for knowledge, as people, in their attempt to describe the world in which they lived, were in need of defining measures and modeling physical objects. The first questions came up naturally from the everyday life: how to compute or compare the dimensions or the area of a field or a solid object, or how to partition it into a number of pieces to satisfy certain conditions. As geometric problems kept increasing in number and variety, Geometry emerged as a new science and along with Algebra and Trigonometry constituted the field of Mathematics that even now, thousands of years later, has not ceased to grow.

The introduction and wide use of the computer as a problem solving tool could in no case fail to influence geometry. In fact, the combination of the fundamentals of the theory of algorithms with old and new geometric problems led to the birth of a new field of computer science, called Computational Geometry. Although fairly new — it is considered emerging in the early nineteen seventies — computational geometry has attracted a good number of researchers who provided it with considerable growth and diversity. Typical problems that fall in the territory of computational geometry involve intersections and partitions of geometric objects, point location or range searching, and computation of convex hulls or Voronoi diagrams, with applications to graphics, computer-aided design and manufacturing, VLSI, and robotics.

In particular, decomposition problems play a very important role in Computational Geometry, as, often, objects that need to be processed are too complicated to be dealt in their generality. In such cases, decomposing the objects into pieces that are easier to process becomes a necessary preprocessing step. Applications involve

simpler descriptions of objects for robotics, and computer-aided design and manufacturing, as well as mesh generations for the finite element method.

In this thesis, we deal with two decomposition problems on three-dimensional nonconvex polyhedra; namely, we discuss the problem of partitioning the boundary of such a polyhedron into a small number of "well-behaved" pieces, and the problem of partitioning a polyhedron of zero genus into tetrahedra. In Chapter 1, we present some of the standard terminology for the two- and three-dimensional space, and overview the previous work on the field of decompositions. Chapter 2 presents an algorithm for the first problem. The algorithm subdivides the boundary of a nonconvex polyhedron of r reflex angles into no more than $18r - 2$ connected pieces that belong to the boundary of their convex hull. Chapter 3 discusses the second problem and presents an algorithm that partitions a polyhedron of n vertices, r reflex angles and zero genus into $O(n + r^2)$ tetrahedra, which is asymptotically optimal in the worst case. Discussion on an implementation of the second algorithm is included in Chapter 4. The implementation is included in the Appendix.

Contents

Abstract v

Acknowledgments vi

Preface viii

1 Preliminaries 1

1.1 Setting the Terminology 1

1.1.1 The two-dimensional Case 1

1.1.2 The three-dimensional Case 3

1.2 Related Work 5

1.2.1 The Two-dimensional Case 6

1.2.2 The Three-dimensional Case 10

2 Decomposing the Boundary of a Polyhedron 13

2.1 A Geometric Framework 15

2.2 The Decomposition Algorithm 17

2.2.1 The First Phase. 18

2.2.2 The Second Phase. 27

2.2.3 Description of the patches produced. 32

3 Tetrahedralizing a Nonconvex Polyhedron 37

3.1 Basic Strategy 37

3.2 Cups, Crowns, Domes, and Other Widgets 40

3.3 Foundations of the Pull-Off Phase 46
3.4 The Tetrahedralization Algorithm 54
3.4.1 The Pull-Off Phase 55
3.4.2 The Witness Oracle 60
3.4.3 The Fence-Off Phase 64
3.5 Putting the Pieces Together 67

4 Implementing the Tetrahedralization Algorithm 69
4.1 Robustness 70
4.2 Polyhedron Representation 74
4.3 The Triangulation Algorithm 78
4.3.1 Describing the Input Polyhedron 79
4.3.2 Triangulating the Boundary 82
4.3.3 Collecting the Witness Facets 87
4.3.4 Pulling-off the Cups 97
4.3.5 Computing the Fences 100
4.3.6 Shattering the Facets 102
4.3.7 Completing the Triangulation 103

5 Conclusions 107

6 Bibliography 109

7 Appendix 115

Chapter 1

Preliminaries

1.1 Setting the Terminology

Before plunging into the discussion of decomposition algorithms, it is appropriate to formally define the objects with which we will be dealing. As the field of computational geometry evolved from an attempt to apply algorithmic techniques on geometric problems, it comes at no surprise that these objects have been borrowed from the classical geometry. The algorithms that we are going to present are set in the three-dimensional space; we begin, however, by reviewing the terminology for the two-dimensional case, to prepare the ground for a smoother transition to three-dimensions.

1.1.1 The two-dimensional Case

The two-dimensional space $\mathcal{R}^2$, also called the *plane*, is the Cartesian product $\mathcal{R} \times \mathcal{R}$, where $\mathcal{R}$ denotes the set of reals. A *point* in the plane is denoted by a 2-tuple (x, y), x and y being real numbers. Given two distinct points $p = (x_1, y_1)$ and $q = (x_2, y_2)$ the *line* passing by p and q is the set of points $p + \alpha(q - p) = (x_1 + \alpha(x_2 - x_1), y_1 + \alpha(y_2 - y_1))$, where $\alpha \in \mathcal{R}$. If α is ranging over the entire set of reals, as is the case above, the combination $p + \alpha(q - p)$ is called a *linear* combination of p and q. If α is restricted to belong to the closed interval $[0, 1]$, the combination is

called *convex*. The convex combination of two points p and q yields the *line segment* defined by p and q; p and q are called its *endpoints*.

Line segments on the plane can be linked at their endpoints to form a polygonal line. More formally, a *polygonal line* or *chain* associated with an ordered set of points $\{p_1, p_2, \ldots, p_{k+1}\}$ is a sequence $l_1 l_2 \ldots l_k$ of line segments such that segment l_i $(i = 1, \ldots, k)$ has points p_i and p_{i+1} as its endpoints. The points p_i are called *vertices*, and the line segments l_i are called *edges* of the polygonal line. A polygonal line is *closed* if vertex p_1 coincides with vertex p_{k+1}, and it is λ-*monotone* if no two points of the polygonal line can be connected by a line segment perpendicular to the direction λ. Finally, a polygonal line is said to be *self-intersecting* if a vertex is shared by more than two edges or any other point of the polygonal line belongs to more than one edge. Let us now introduce the notion of a polygon ([58], [22]).

> **Definition 1.1.1:** A *polygon* is a subset of the plane bounded by a finite set of closed polygonal lines that are not self-intersecting and do not intersect each other.

The vertices and edges of these polygonal lines are called *vertices* and *edges* of the polygon respectively. From the definition of the self-intersecting polygonal lines, it is easy to see that each vertex of the polygon is incident upon two edges, and is therefore associated with an angle, the interior angle that these edges exhibit at the vertex. If this angle is larger than π, then the vertex is called *cusp* or *notch*. For example, the vertices u and v of the polygon shown in Figure 1.1 are cusps. The polygonal lines form the *boundary* of the polygon, separating the *interior* from the *exterior* of the polygon. A polygon is called *simple*, if its boundary consists of a single closed polygonal line; if not, then the polygon looks like the one in Figure 1.1, and we informally say that it contains *holes*. A polygon is *convex* if it is a convex set, meaning that any two of its points can be connected by a line segment that lies entirely in the set (polygon). As a result, a convex polygon has no cusps. Convex polygons are particularly desirable, as they result into simple and efficient algorithms. Other classes of polygons as well may share this property, as is the case, for instance, with monotone and star-shaped polygons. A simple polygon is λ-*monotone* if for

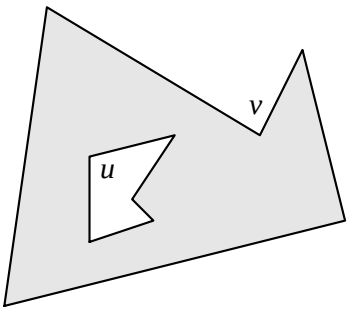


Figure 1.1: A nonsimple polygon

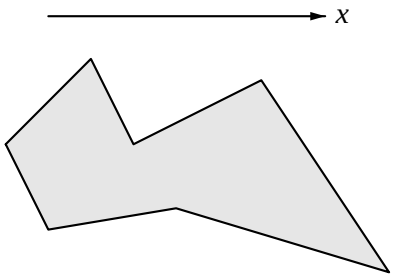


Figure 1.2: An x-monotone polygon

a direction λ its boundary can be decomposed into two polygonal lines that are λ-monotone. For example, the polygon of Figure 1.2 is x-monotone. More generally, we say that a simple polygon is *monotone* if it is λ-monotone for some direction λ. In order to define the star-shaped polygons, let us consider the set of points of the plane that can be connected to all the points of a polygon P by a line segment lying entirely in P. This set is called the *kernel* of P, and is obviously a subset of P. We say that a polygon is *star-shaped* if it has nonempty kernel. Any convex polygon is star-shaped, and it coincides with its kernel, whereas a nonsimple polygon clearly cannot be star-shaped.

Finally, the *convex hull* of a point set $\mathcal{P}$ is defined as the smallest convex subset of $\mathcal{R}^2$ that encloses all the points in $\mathcal{P}$. It is interesting to note that the vertices of the convex hull of a point set are members of that set.

1.1.2 The three-dimensional Case

A point of the three-dimensional space $\mathcal{R}^3$ is represented by a 3-tuple (x, y, z), where $x, y, z \in \mathcal{R}$. Two distinct points define a unique line, which is their linear combination, whereas their convex combination yields the line segment delimited by the two points. Unfortunately, unlike the two-dimensional case, there is not a widely accepted terminology for the three-dimensional equivalent of a polygon, and conflicting definitions are often encountered. We resort to algebraic topology to provide us with a formal and precise framework ([50], [59], [63]).

Definition 1.1.2: A *polyhedron* in three-dimensions is a piecewise linear 3-manifold with boundary that is connected and orientable.

Let us mention that, for a positive integer n, an *n-manifold with boundary* is a separable metric space such that each point has an open neighborhood homeomorphic either to the open ball U^n or to the space $U^n_+ = \{(x_1, x_2, \ldots, x_n) \in U^n \colon x_1 \geq 0\}$. The former set of points is called the *interior* of the manifold, whereas the latter is called its *boundary*. The boundary of a polyhedron is a collection of relatively open sets, the *faces*, which are called *vertices*, *edges*, or *facets*, if their affine closures have dimension 0, 1, or 2 respectively. The definition of a polyhedron ensures that each edge is incident upon two facets, and rules out intersecting, dangling, or abutting faces, as well as degeneracies like the one shown in Figure 1.3. On the other hand, the number of edges incident upon a vertex may vary, and it is referred to as the *degree* of that vertex.

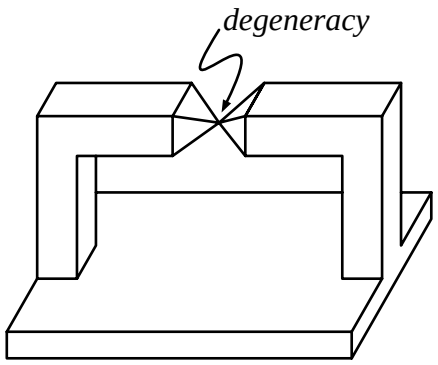


Figure 1.3

An important parameter of a three-dimensional polyhedron is its genus. Let us consider a surface in three-dimensions that is connected, that is, any two of its points can be connected by a path lying on the surface. Then, the *genus* of that surface is the maximum number of disjoined closed curves that can be drawn on the surface without disconnecting it. Extending the notion to the polyhedra, the genus of a polyhedron is the genus of its boundary. The sphere, for example, has genus 0, while a torus has genus 1. The genus of a polyhedron is related to the number n of its vertices, the number e of its edges and the number f of its facets by *Euler's Formula*: $n - e + f \geq 2 - 2g$. The inequality becomes equality under various conditions; this is the case, for instance, if all the facets of the polyhedron are simple polygons (or triangles in particular).

The definitions of a convex or star-shaped polygon and of the convex hull of a point set extend in a straightforward fashion to three dimensions. It is interesting to note that the interior dihedral angle defined by the two facets incident upon any edge of a convex polyhedron does not exceed π. Edges of a polyhedron whose incident facets exhibit interior dihedral angles larger than π are called *reflex* edges or *notches*.

1.2 Related Work

The problem of decomposing an object Q (viewed as a connected set of points in d-dimensional space) into simpler pieces is defined as the problem of producing a collection of simple pieces whose union is precisely Q. The problem is of major importance in computational geometry, as complex objects can be expressed in terms of simpler pieces that are in general easier to process and hence lead to more efficient algorithms. This is the case, for instance, with many algorithms for simple polygons, where triangulating the polygon is a standard preprocessing step (point-location [42], visibility and shortest paths [1, 13, 33], link distances [64], polygon separation [67]). Typical applications involve mesh generations for the finite element method, or simpler descriptions of objects for robotics, computer-aided design and manufacturing, while a large number of algorithms employs decomposition routines to preprocess the objects on which they are applied.

Decompositions are classified into two categories: coverings and partitions. In a partition, the intersection of any two pieces is either empty or belongs to the boundary of both pieces; in a covering, this restriction is relaxed, and the pieces are therefore free to overlap. A different characterization depends on whether there exist vertices of the decomposition that are not vertices of the initial complex object. Such vertices are called *Steiner points*, and decompositions that allow such points are often referred to as *Steiner decompositions*.

The extensive work on the subject provides us with decompositions into several different classes of simple pieces: convex, monotone, star-shaped, edge-visible, etc. See [9], [25], [56], [61]. Convex decompositions, and more specifically convex partitions, are usually the most appealing. Moreover, a decomposition into pieces of

constant size has the additional advantage that the processing of each such piece can be completed in constant time. The most common such decompositions are decompositions into simplices, that is, decompositions into triangles and tetrahedra in two and three dimensions respectively. Let us review some of the prior work on the subject.

1.2.1 The Two-dimensional Case

The problem that attracted the attention of the researchers more than any other in this category is the problem of triangulating a polygon. A *triangulation* of a polygon P is a partition of P into triangles whose vertices belong to the vertex set of P. Convex polygons can be easily triangulated in linear time by drawing the diagonals that connect a given vertex of the polygon to all its nonadjacent vertices. The same procedure does not apply, however, in the general case, as such diagonals may not necessarily lie entirely in the polygon.

The first subquadratic algorithm for the triangulation of a polygon was described by Garey, Johnson, Preparata and Tarjan in 1978 ([30]). The algorithm begins by employing an algorithm of Lee and Preparata ([45]) to partition the polygon into pieces that are monotone with respect to a fixed direction. Then each of the resulting monotone pieces can be easily triangulated. The idea is to sweep the piece with a line normal to the direction of monotonicity, and maintain a concave chain of vertices that bound the part of the piece that has not yet been triangulated. The cases that may arise when a new vertex is processed are shown in Figure 1.4. Triangulating the monotone pieces takes time linear in the number of vertices of the polygon, and so the time complexity of the entire algorithm is determined by the preprocessing phase which for a polygon of n vertices requires $O(n \log n)$ time.

Another $O(n \log n)$ algorithm was presented by Fournier and Montuno ([28]). The algorithm is also based on the sweep-line paradigm, and computes the horizontal (or vertical) *visibility map* of the polygon, that is, a decomposition of the polygon into trapezoids by means of horizontal (or vertical) line segments through the polygon's vertices. The visibility map yields a triangulation in time linear in the size of the polygon.

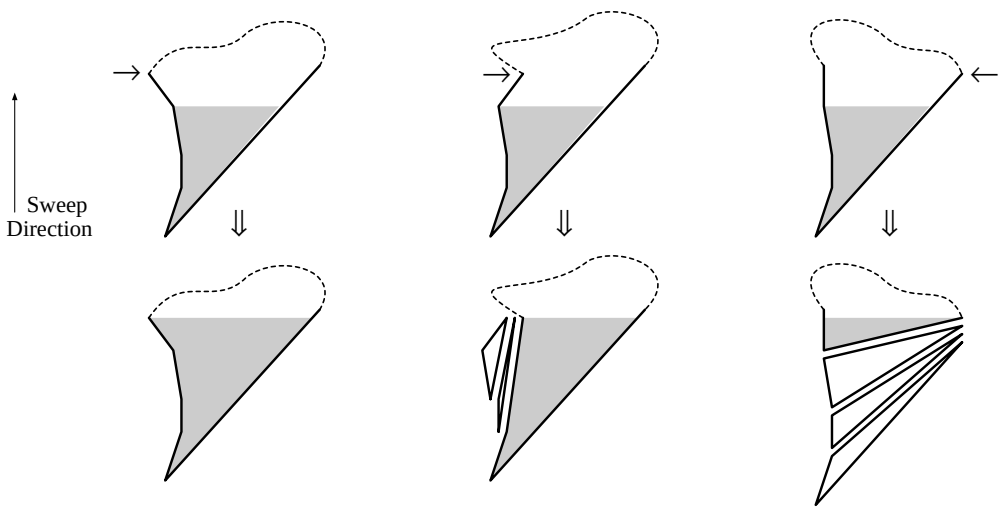


Figure 1.4: Triangulating a monotone polygon.

The quest for faster algorithms motivated researchers to focus on special classes of polygons. For example, Fournier and Montuno came up with linear time triangulation algorithms for star-shaped and spiral polygons ([28]), while Toussaint and Avis described linear time algorithms for edge-visible and monotone separable polygons ([68]). See also the survey paper of Toussaint ([66]).

A different trend involved associating the time complexity of the algorithm with some measure of the complexity of the polygon. Algorithms in this category were proposed by Hertel and Mehlhorn ([36], [51]), and Chazelle and Incerpi ([15]). In particular, the algorithm of Hertel and Mehlhorn combines the two phases of the algorithm of Garey, Johnson, Preparata and Tarjan into a single sweep through the polygon, making sure however that only a subset of the polygon's vertices with cardinality proportional to the number of cusps of the polygon needs to be sorted. If the numbers of vertices and cusps of the polygon are n and r respectively, the algorithm runs in $O(n + r \log r)$ time. The algorithm by Chazelle and Incerpi extends the notion of the visibility map to polygonal chains, where the segments that yield the decomposition may extend to infinity. It applies a bottom-up merging process to combine in a balanced fashion maps of smaller chains of the polygon's boundary. Its time complexity is $O(n \log s)$ time, where s is the *sinuosity* of the polygon, i.e., the number of times its boundary alternates between spirals of opposite orientation. The

final triangulation is obtained by the computed visibility map in linear time.

It should be noted that both the sinuosity s and the number of cusps r of a polygon can be as large as $\Omega(n)$, so that none of the algorithms presented so far breaks the $\Theta(n \log n)$ time barrier in the worst case. As a matter of fact, in the case of nonsimple polygons, a reduction from sorting yields that this bound is tight. For simple polygons, however, the question of whether computing their triangulation is indeed as hard as sorting remained open for many years. The question was finally settled negatively in 1986 by Tarjan and Van Wyk, who presented an $O(n \log \log n)$ time algorithm for the problem ([65]). Their algorithm, too, relies on the computation of the visibility map of a polygon. This is achieved by combining the divide-and-conquer paradigm with a linear time *Jordan sorting algorithm* ([38]), and by taking advantage of *finger search trees* that allow fast location, insertion and concatenation operations.

Although the previous algorithm has been implemented ([44]), its usefulness in practice is rather questionable, partly due to the use of finger search trees. In an attempt to simplify it, Kirkpatrick, Klawe and Tarjan presented a modified version that avoids the use of finger trees, without, however, compromising its time complexity ([43]). The new algorithm introduces the notion of a *wrap-around partition* of a polygon (or chain): that is the decomposition of the plane (viewed as an infinite vertical cylinder) induced by (horizontal) rays through the vertices of the polygon that extend to the left and right until they hit the polygon again. It works by computing the wrap-around partitions of the pieces in which the polygon's boundary is split during an appropriately tuned divide-and-conquer process. The partial partitions are then combined to yield the wrap-around partition of the entire polygon in linear time in a fashion similar to the one employed in [15].

As the complexities of the triangulation algorithms started approaching $O(n)$, the question that emerged was whether a simple polygon could indeed be triangulated in time linear in its size. Chazelle recently presented an algorithm that resolves the question affirmatively ([10]). The algorithm builds on the earlier bottom-up algorithm by Chazelle and Incerpi. The idea is to maintain only a portion of each computed visibility map, resulting in a partial visibility map for the entire polygon. The missing

information is provided in a later top-down phase. The algorithm makes use of the planar separator theorem, and the balanced decomposition of a tree to achieve its optimal performance.

Despite their theoretical superiority, the last three algorithms use constructions that are complicated enough to make them rather unattractive for practical implementation. Simpler versions have been obtained, however, thanks to randomization. The first probabilistic algorithm, which is based on the random sampling paradigm and divide-and-conquer, was proposed by Clarkson, Tarjan and Van Wyk ([19]). The algorithm begins by selecting a random sample of the polygon's edges, and constructing its visibility map, which is subsequently intersected with the remaining edges. As long as there is some vertex of the polygon whose associated visibility segment has not been drawn, the algorithm recurs on the region enclosing it. The selection of a random sample of the appropriate size, ensures that for a simple polygon of n vertices the recursion depth is $O(\log^* n)$. As it is expected that no more than $O(n)$ time is spent for the processing at each level of recursion, the expected time of the entire algorithm is $O(n \log^* n)$.

Newer probabilistic algorithms (Clarkson, Cole and Tarjan [18], Seidel [62]) even manage to avoid the use of Jordan sorting that the previous algorithm requires, while still maintaining the above expected time complexity.

We close this section by mentioning that decomposition problems gave rise to optimization questions in cases where more than one decomposition is possible. The complexity of these questions changes dramatically depending on whether the polygon to be decomposed is simple or not. As a general observation, for simple polygons dynamic programming is used in nearly all the optimization algorithms that we discuss below yielding superquadratic algorithms, whereas the same questions for nonsimple polygons are NP-hard.

For a simple polygon, Chazelle and Dobkin ([11]) in 1979 presented an $O(n + r^3)$ algorithm that partitions a polygon of n vertices, r of which are cusps, into the minimum number of convex pieces. The algorithm employs dynamic programming to determine the appropriate set of the so-called X_4- and Y-patterns, thus introducing Steiner points in the partition. If Steiner points are disallowed, Greene ([32]) gave an

$O(r^2n^2)$ time algorithm for the problem. Later, Keil ([41]) achieved an $O(r^2n\log n)$ time algorithm.

Asano and Asano ([3]) studied the problem of partitioning a simple polygon into the minimum number of trapezoids whose parallel edges are parallel to prespecified direction. They came up with an $O(n^3)$ time algorithm, by using an algorithm by Gavril to locate the maximum independent set of a *circle graph* ([31]). The time complexity of the algorithm was improved to $O(n^2)$ in a later paper of Asano, Asano and Imai, where a more economical routine to locate the above mentioned independent set was used ([4]).

In the case of nonsimple polygons all previous minimization questions are proven to be NP-hard ([48], [49], [40]), by means of reductions from 3SAT or planar-3SAT, both known to be NP-complete ([29], [47]). More NP-hardness results were also established by O'Rourke and Supowit for similar problems on coverings instead of partitions ([57]).

1.2.2 The Three-dimensional Case

Unlike their two-dimensional counterparts, decomposition problems in three dimensions tend to be much harder. As a result, the number of algorithms for these problems is fairly small.

The problem of partitioning a nonconvex polyhedron in convex pieces was first attacked by Chazelle ([8]). Since nonconvexity is due to the notches of the polyhedron, the algorithm resolves each notch in turn by slicing the polyhedron with a plane passing through the notch and forming angles with the notch's incident facets that do not exceed π. As more slicing planes are added, an unresolved notch may be cut into several sub-edges; so long as a single resolving plane is used to resolve all the sub-edges of the notch in all the pieces of the decomposition to which these sub-edges belong, the size of the final partition is $O(n^2)$, where n is the number of vertices of the polyhedron. (Note that if this requirement is not enforced, an exponential number of pieces may arise.)

Additionally, Chazelle proved a lower bound for the size of the convex partition

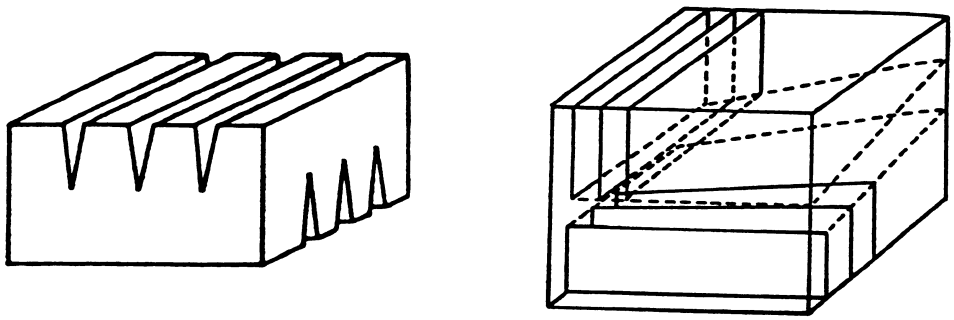


Figure 1.5

that a polyhedron may admit ([8]). In particular, he exhibited an infinite family of polyhedra that cannot be decomposed into fewer than a quadratic number of convex pieces. A typical member of this family is shown in Figure 1.5, where its notches lie on two hyperbolic paraboloids slightly apart from each other. This lower bound implies that the algorithm discussed earlier produces a partition whose size is asymptotically optimal in the worst case.

The problem of triangulating a simple polygon extends to three dimensions into the problem of partitioning a polyhedron into tetrahedra. The two cases, however, present some noteworthy differences. First, although the size of any triangulation of a simple polygon is unique, a polyhedron may admit partitions into a different number of tetrahedra. The simplest example is the bipyramid (shown in Figure 1.6) that can be partitioned into two or three tetrahedra ([60]). A more important difference involves the feasibility of these two partitioning problems. In particular, a polygon can always be triangulated. The same is true in three dimensions in the case that Steiner points are allowed in the partition; Chazelle's algorithm testifies to that as a convex partition can be easily refined into a partition into tetrahedra. However, not every polyhedron admits a partition into tetrahedra, if we require that their vertices are vertices of the polyhedron. A typical example is the "twisted" triangular prism: a triangular prism one of whose bases has been slightly twisted with respect to the other. Due to the distortion, no tetrahedron defined by any four vertices of the prism lies entirely in the prism. In fact, by means of a reduction from 3SAT, Ruppert and

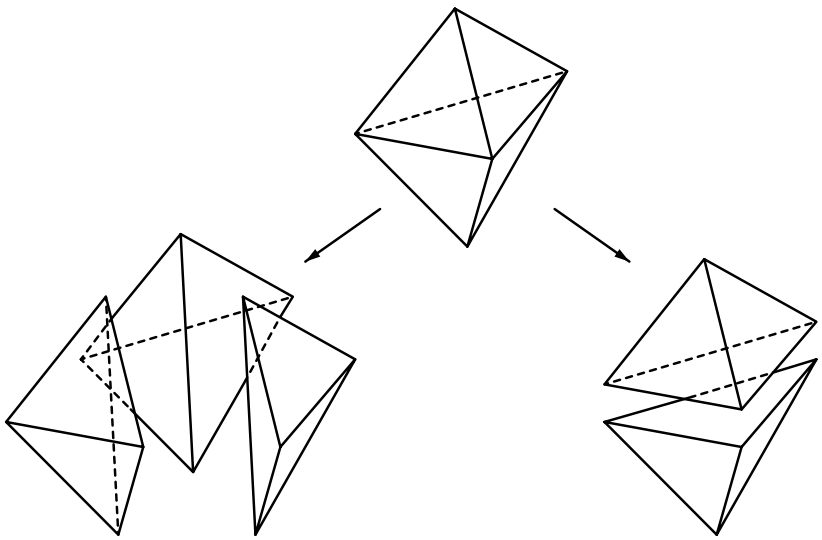


Figure 1.6: A bipyramid that admits two tetrahedralizations

Seidel showed that even the problem of deciding whether such a partition exists is NP-complete ([60]). The problem remains NP-complete even if restricted to the class of star-shaped polyhedra.

The need for three-dimensional decomposition algorithms in practice motivated work in practical issues, such as dealing with finite precision arithmetic, and guaranteeing "good quality" partitions, since it is important in some applications that the pieces of the decomposition are not long and skinny. For example, Bajaj and Dey presented a numerically robust algorithm for the problem of convex partitioning a nonconvex polyhedron by enhancing Chazelle's algorithm with primitives that deal with finite precision computations ([5]). On the other front, several algorithms have been proposed that try to rule out such cases by maximizing or bounding from below the minimum angle exhibited in any piece or the ratio of their edge lengths. We mention the work by Chew ([17]), Dey, Bajaj and Sugihara ([21]), and by Field ([26]).

Chapter 2

Decomposing the Boundary of a Polyhedron

The importance of decompositions into simpler objects is due to the fact that easier and faster algorithms are usually required to process simple objects. Indeed, it is often worth expressing a convoluted object in terms of a number of "well-behaved" components. In two dimensions, for example, polygon triangulation is a standard preprocessing step in many algorithms ([1], [13], [33], [42], [64], [67]); similarly, in three dimensions, a polyhedron can be expressed as a collection of convex pieces or tetrahedra in particular (see [5], [8], [48], [60], and the next chapter for discussions on such decompositions). Of course, the size of the decomposition is critical for the application that uses it. Unfortunately, a convex partition of a polyhedron may be of size quadratic in the description size of the polyhedron in the worst case ([8]), which makes it unattractive from an efficiency point of view. It would be, therefore, of interest to have partitions into a guaranteed small number of simple components.

In this chapter, we consider the problem of subdividing the boundary of a non-convex polyhedron of arbitrary genus into a small number of connected *convex-like* pieces. By convex-like piece, we mean a polyhedral surface which lies entirely on the boundary of its convex hull. Our result is that the boundary of a nonconvex polyhedron that has r reflex angles can be subdivided into no more than $18r - 2$ such pieces. It is interesting to note that the number of pieces is independent of the

number of vertices of the polyhedron, and it is linear in r. The algorithm proceeds in two phases. In the first phase, the boundary of the polyhedron is disassembled into at most $2r + 2$ pieces whose internal edges are not reflex edges of the polyhedron, and whose intersections with planes normal to a fixed plane Π are collections of chains monotone with respect to a fixed direction. The partition is achieved by cutting first along the reflex edges of the polyhedron, and then along the edges that are incident upon facets whose projections onto the fixed plane Π overlap; figuratively, we cut along the "ridges" and "keels" of the polyhedron. The second phase further splits the pieces into convex-like ones by clipping them with planes normal to Π that pass through the endpoints of the polyhedron's reflex edges. The clipping is carried out in such a way that it introduces only $O(n)$ new vertices. The entire algorithm runs in linear time, provided that the boundary of the given polyhedron has been triangulated. Boundary triangulation takes $O(n + r \log r)$ time.

In two dimensions, the problem is very simple, and admits a linear-time solution that produces the minimum number of polygonal curves into which the boundary of a polygon (possibly with holes) can be cut so that each such curve stays on the boundary of its convex hull. The algorithm first disconnects the boundary of the polygon at its *cusps* (the vertices whose incident edges form an interior angle larger than π), and then breaks each resulting piece in a greedy fashion to enforce the convexity condition. In other words, we start at one end of such a piece and keep walking along it for as long as each encountered edge lies on the convex hull of the subpiece traversed so far, disconnecting it otherwise. The whole process takes linear time. A second algorithm can be obtained by taking the two-dimensional equivalent of the first phase of our algorithm as outlined above. The boundary of the polygon is disassembled at the cusps and at the local extrema with respect to some fixed direction, say, the vertical direction z (Figure 2.1). It can be proven by induction that a polygon of r cusps has at most $r + 2$ local extrema, which implies that the total number of pieces produced cannot exceed $2r + 2$. This is almost optimal in the worst case, since for any r there is a polygon of r cusps whose boundary cannot be disassembled into fewer than $2r + 1$ pieces each lying on the boundary of its convex hull (Figure 2.2).

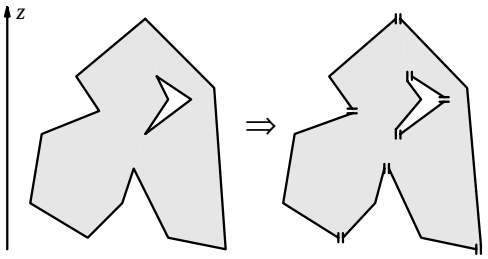


Figure 2.1

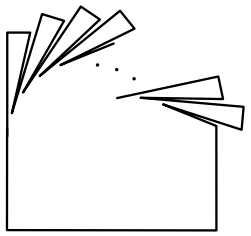


Figure 2.2

2.1 A Geometric Framework

We recall from the previous chapter that an edge e of a polyhedron is said to be *reflex* if the (interior) dihedral angle formed by its two incident facets exceeds π. By extension, we say that a vertex is *reflex* if it is incident upon at least one reflex edge. A *patch* of a polyhedron P is a collection of facets or subsets of facets of P with their adjoining edges and vertices. The edges of a patch that do not lie on its relative boundary are called *internal*. Let us try to extend to patches some of the definitions pertaining to polygons. A patch is said to be *connected* if any two of its points can be connected by two disjoint (except for their end-points) paths that lie on the patch. Under this definition, the two patches of Figure 2.3 are not considered connected. Unless it consists of a single facet, a connected patch has at least one internal edge. A connected patch is said to be *simple* if it is bounded by a single non-intersecting closed curve; otherwise, we say that the patch contains holes. A patch is called *monotone* with respect to a plane if no two distinct points of the patch project normally to the same point of the plane. Finally, a patch σ is *convex-like* if it lies on the boundary of the convex hull H_σ of its vertices and the interiors of both P and H_σ lie on the same side with respect to each of the facets of σ. The latter condition implies that none of the internal edges of a convex-like patch is a reflex edge of the given polyhedron. The following lemma presents the sufficient conditions for a patch to be convex-like.

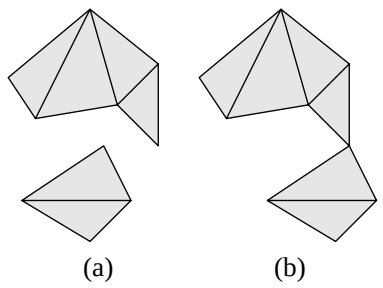


Figure 2.3

Lemma 2.1.1. *Let σ be a patch of a polyhedron P none of whose internal edges are reflex edges of P. If σ is simple and monotone with respect to a plane Π onto which it projects into a convex polygon, then σ is convex-like.*

Proof: (The lemma trivially holds if all the facets of σ are coplanar.) The monotonicity of σ with respect to Π implies that the projection of the boundary of σ onto Π coincides with the boundary of the projection of σ. Moreover, since the projection of σ is a convex polygon, σ's relative boundary lies entirely on the boundary of the convex hull of σ. Let us consider the unbounded cylinder whose axis is normal to Π and whose base is the projection of σ onto Π. Since the patch σ is simple, its relative boundary lies on the boundary of this cylinder, and σ separates the cylinder into two unbounded polyhedra, with respect to one of which, say T, the internal edges of σ are nonreflex. Then, the interiors of P and T lie on the same side with respect to each of the facets of σ. We need, therefore, only show that σ lies entirely on the boundary of its convex hull, or equivalently that T coincides with its convex hull. The latter can be easily established if we take into consideration the fact that T has no reflex edges.

∎

Finally, we introduce the notion of extrema. A point p of a d-dimensional set S of points is called an *extremum* of S with respect to an oriented line λ, or a λ-*extremum* for short, if S's intersection with a small enough d-ball centered at p lies entirely in the one of the two closed half-spaces defined by the hyperplane that is normal to λ and passes through p. The extrema are characterized as *negative* or *positive* depending on whether the above intersection lies in the nonnegative or nonpositive half-space respectively. For instance, for the polygon of Figure 2.4 and the indicated oriented line λ, the vertices A, B, C and D are negative λ-extrema, while X and Y are positive ones. (Note that the vertex Z is *not* an extremum.) Clearly, if no edge of a polygon or polyhedron is normal to λ, only vertices can be extrema.

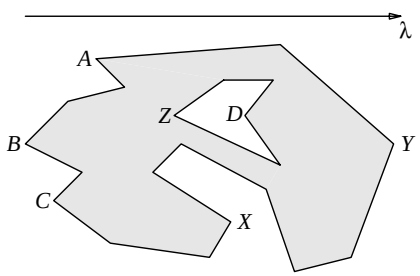


Figure 2.4

Very often in the following, we consider the intersection of the polyhedron with a plane normal to the x-axis. The intersection consists of several polygons (possibly with holes), and it will be referred to as a *yz-cross-section* of the polyhedron.

2.2 The Decomposition Algorithm

Our goal is to subdivide the boundary of a nonconvex polyhedron P of n vertices and r reflex edges into $O(r)$ connected convex-like patches. The polyhedron is given in any one of the standard representations, e.g., winged-edge (Baumgart [7]), doubly-connected-edge-list (Muller and Preparata [55]), quad-edge (Guibas and Stolfi [35]), so that all the face incidences either are explicitly stored or can be found in linear time. See Section 4.2 for a discussion of the models. To simplify the description of the algorithm, we assume that no facet of P is perpendicular to the z-axis, and no edge is normal to the x-axis. These assumptions are not restrictive; they can be checked in linear time, and, if necessary, enforced by rotating the system of reference. We also assume that the boundary of the polyhedron P is triangulated. Boundary triangulation can be achieved in $O(n + r \log r)$ time by employing the polygon triangulation algorithm of Hertel and Mehlhorn [36] on each nontriangular facet of P.

The algorithm consists of two phases, each of which splits already existing patches (the entire boundary of P is the initial patch to work on): the first phase guarantees that each of the resulting patches (i) does not have any internal edge that is a reflex edge of P, and (ii) its intersection with a plane normal to the x-axis is a collection of chains monotone with respect to the z-axis. Monotonicity with respect to the xz-plane is achieved in the second phase, where it is also ensured that the projections of the patches onto this plane are convex polygons. The resulting patches are therefore convex-like, by virtue of Lemma 2.1.1. Note that throughout the algorithm the patches are orientable, as they are subsets of the boundary of the polyhedron, a 2-manifold without boundary, which is orientable.

2.2.1 The First Phase.

As mentioned earlier, no internal edge of a convex-like patch can be a reflex edge of P. Therefore, we need to cut along each reflex edge, where cutting along an edge has the effect that its incident facets are no longer considered adjacent. (Note, however, that the internal edges of a patch can all be nonreflex edges of P, and still the patch may not be convex-like. Think of a patch spiraling around several times.) We are now tempted to embark on the second phase of the algorithm in which the patches are decomposed by means of cuts along the intersections of the boundary of the polyhedron with planes normal to the x-axis that pass through the reflex vertices of the polyhedron. This, however, will not necessarily produce the desired partition. Consider, for instance, the polyhedron of Figure 2.5, which is constructed by gluing two tetrahedra along a common facet uvw (where u and v are the vertices with the smallest and largest x-coordinates respectively), so that the edge uv is reflex. The above rules imply that the only cut introduced is along the single reflex edge uv. The boundary of the polyhedron will therefore still form a connected patch, which clearly is not convex-like. To rule out such cases, we sweep the polyhedron with a plane normal to the x-axis, and we further disassemble its boundary along the z-extrema of its current cross-section. Recall that no edge of P is normal to the x-axis; as a result, the x-extrema of the polyhedron are vertices of P. Moreover, no facet of P is normal to the z-axis, implying that the z-extrema in the cross-section are vertices, the intersections of edges of P with the sweep-plane. (Figure 2.13 depicts a typical cross-section of the polyhedron after all the cuts have been introduced.)

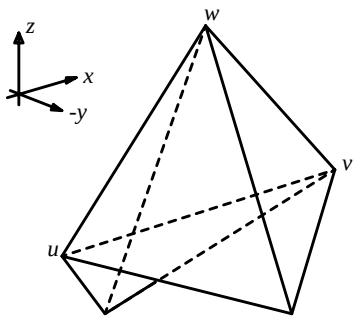


Figure 2.5

Although a direct implementation of the plane-sweep approach requires sorting, we can in fact avoid it. The reason is that determining whether an edge is a reflex edge of P or whether it contributes z-extrema in the corresponding cross-section of the polyhedron requires only information local to the edge, namely the relative position

of its two incident facets. We therefore process the edges in any order, cutting along those that satisfy the above criterion. Clearly, the entire phase takes time linear in the number of edges, while no more than linear space is needed.

Let σ be a patch at the end of this phase. The above procedure ensures that no internal edge of σ is a reflex edge of the polyhedron, and that σ's intersection with a plane normal to the x-axis consists of a number of z-monotone polygonal lines. Let us denote by σ_{xz} the projection of σ onto the xz-plane. Then, any point of σ that projects on the boundary $\partial\sigma_{xz}$ of σ_{xz} belongs to the closure of a reflex edge or an edge of the polyhedron contributing z-extrema in a yz-cross-section that intersects the edge: in either case, it belongs to the boundary $\partial\sigma$ of the patch. (Of course, not every point in $\partial\sigma$ projects into a point of $\partial\sigma_{xz}$.) Suppose that this were not true, that is, there exists a point p that does not belong to $\partial\sigma$, but whose projection belongs to $\partial\sigma_{xz}$. The latter condition implies that p lies in the closure of an edge e_p of P, and a line parallel to the y-axis that passes through p lies either above or below both facets of the polyhedron incident upon e_p. Then, either e_p is reflex, or it contributes z-extrema at the corresponding yz-cross-sections, which contradicts the assumption that p does not belong to $\partial\sigma$. In particular, for the nonreflex vertices of the polyhedron, the following lemma holds.

Lemma 2.2.1. *Let v be a nonreflex vertex of the polyhedron that projects into a point q on the boundary of the projection σ_{xz} of a patch σ onto the xz-plane. Then, q is not a cusp of the polygon σ_{xz}.*

Proof: In accordance with the above discussion, v is incident upon an edge e that contributes z-extrema at the corresponding yz-cross-section of the polyhedron. This implies that the intersection of the patch (and the polyhedron) with a small enough ball centered at the midpoint of e lies entirely in the closed halfspace above or below the plane defined by e and a line parallel to the y-axis. Moreover, since v is nonreflex, the same case applies to a small enough ball centered at v. The lemma follows from the fact that the above mentioned plane projects onto the xz-plane into the line supporting e's projection. ∎

Number of patches produced. We prove next that the number of patches that are produced at the end of this phase is no more than $2r + 2$. The key observation is that each patch has a vertex whose adjacent vertices that belong to the patch all have larger x-coordinates; the vertex of the patch with the smallest x-coordinate will do, in agreement with the assumption that no edge of the polyhedron is parallel to the yz-plane. In fact, a patch may have more than one such vertex; they are the negative x-extrema of the patch. Similarly, each patch has at least one vertex whose adjacent vertices that belong to the patch all have smaller x-coordinates; this is a positive x-extremum of the patch. These observations imply that if we compute the number of patches to which a vertex of the polyhedron contributes positive or negative x-extrema, add these numbers over all vertices, and divide by two, we obtain an upper bound in the number of patches that are produced.

The way to compute the number of patches to which a vertex v contributes positive or negative x-extrema can be more easily understood if we consider yz-cross-sections of the polyhedron. First, we consider the yz-cross-section $\mathcal{C}_{yz}(v)$ of the polyhedron at v; in $\mathcal{C}_{yz}(v)$, v is incident on a number of line segments, each of which corresponds to a facet f_i of the polyhedron, and ultimately to the patch containing f_i. Next, we consider the yz-cross-section $\mathcal{C}^+_{yz}(v)$ infinitesimally away from v towards increasing x-coordinates; $\mathcal{C}^+_{yz}(v)$ contains line segments that correspond to the same facets f_i too (recall the assumption that no edge of the polyhedron is parallel to the yz-plane), while new line segments may have just emerged. The patches to which v contributes negative x-extrema are precisely the patches corresponding to the latter line segments, and their number cannot exceed the number of disconnected polygonal chains of $\mathcal{C}^+_{yz}(v)$ to which the new line segments belong, (it is assumed that the previously described patch splitting process has been applied). For example, in each of the four cases shown in Figure 2.6, the left-hand side depicts a neighborhood N_v of a vertex v at the cross-section $\mathcal{C}_{yz}(v)$, while the right-hand side depicts what N_v may evolve into in $\mathcal{C}^+_{yz}(v)$; the shaded area indicates the interior of the polyhedron, and the boundary of the polyhedron is shown partitioned into patches. In Figure 2.6(a) in particular, the three new line segments that appear in $\mathcal{C}^+_{yz}(v)$ form two disconnected chains, which correspond to at most two patches (note that both chains may in fact belong to the

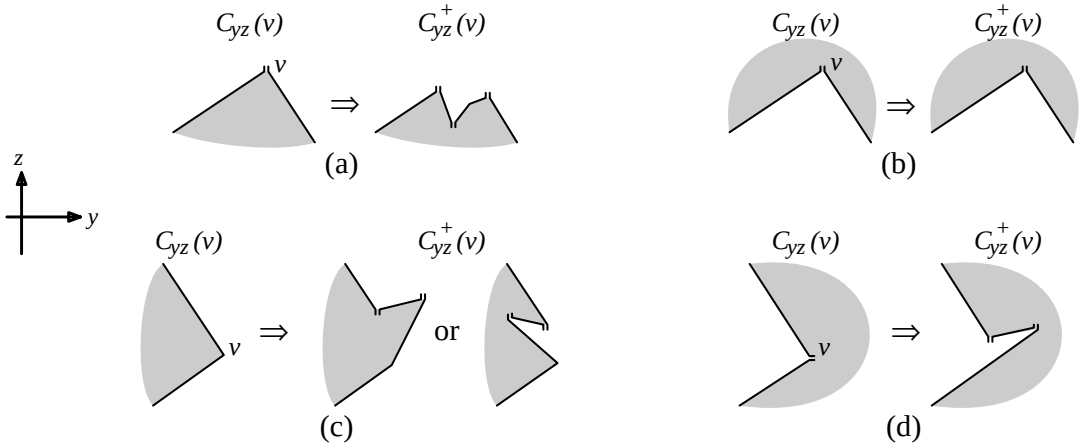


Figure 2.6: (a) at most 2 "new" patches; (b) 0 "new" patches; (c) at most 1 "new" patch; (d) at most 1 "new" patch.

same patch); in this case, therefore, v contributes negative x-extrema to at most 2 patches. Finally, we consider the yz-cross-section $\mathcal{C}_{yz}^{-}(v)$ infinitesimally away from v towards decreasing x-coordinates. The above discussion applies in a similar fashion to $\mathcal{C}_{yz}^{-}(v)$, implying that the number of patches to which v contributes positive x-extrema does not exceed the number of disconnected polygonal chains to which any new line segments in $\mathcal{C}_{yz}^{-}(v)$ belong, after the patch splitting process has been applied.

It should be expected that the number of patches to which a vertex v contributes positive or negative x-extrema depends on the number r_v of reflex edges incident upon v. More specifically, the number of patches to which v contributes positive x-extrema should depend on the number r_v^{-} of reflex edges connecting v to vertices with smaller x-coordinates; similarly, the number of patches to which v contributes negative x-extrema should depend on the number r_v^{+} of reflex edges connecting v to vertices with larger x-coordinates. Figure 2.6 provides some intuition, by depicting details of $\mathcal{C}_{yz}(v)$ and $\mathcal{C}_{yz}^{+}(v)$ for a vertex v with $r_v^{+} = 1$; the single reflex vertex at the right-hand side figure of each of the four cases shown is the intersection of the reflex edge with the slicing plane. It turns out that one reflex edge may be accounted for no more than 2 patches having v as an x-extremum, so that a vertex v incident upon r_v reflex edges contributes (positive or negative) x-extrema to at most $2r_v + c$ patches, for some appropriate integer c. We consider the following cases:

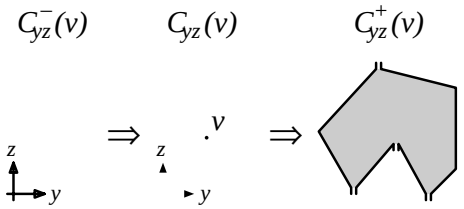


Figure 2.7

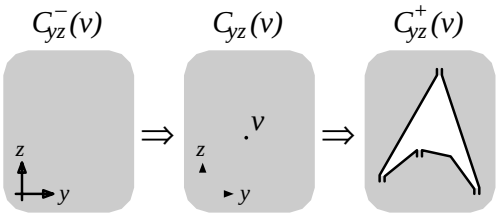


Figure 2.8

1. **The vertex v is of degree 0 in $\mathcal{C}_{yz}(v)$:** The vertex is either a point-polygon, or a point-hole in a polygon of $\mathcal{C}_{yz}(v)$.

   (a) point-polygon: In agreement with the definition of a polyhedron, v is an x-extremum of the polyhedron. If v is a negative x-extremum, then, as Figure 2.7 suggests, it may contribute negative x-extrema to no more than $2r_v^+ + 2$ patches. Moreover, the definition of a negative x-extremum of a polyhedron implies that v cannot possibly contribute positive x-extrema to any patch, as well as that $r_v = r_v^+$. Hence, v contributes x-extrema to at most $2r_v + 2$ patches. In a similar fashion, we find that v contributes x-extrema to at most $2r_v + 2$ patches as well, if it is a positive x-extremum of the polyhedron (the picture in this case is the same as Figure 2.7 where the pictures for $\mathcal{C}_{yz}^+(v)$ and $\mathcal{C}_{yz}^-(v)$ have been interchanged).

   (b) point-hole: Figure 2.8 depicts one of the two basic cases that may arise; the other one stems from Figure 2.8 after the pictures for $\mathcal{C}_{yz}^+(v)$ and $\mathcal{C}_{yz}^-(v)$ have been interchanged. In either case, $r_v \geq 3$, and v contributes x-extrema to no more than $2r_v - 2$ patches.

2. **The vertex v is of degree 2 in $\mathcal{C}_{yz}(v)$:** The fact that $\mathcal{C}_{yz}^-(v)$ and $\mathcal{C}_{yz}^+(v)$ look very similar in this case allows us to concentrate on the latter yz-cross-section for the moment; the results are later extended to the former one. If r_v^+ equals 0, the two-edge chain incident on v in $\mathcal{C}_{yz}(v)$ (that is, the intersection of the polyhedron's boundary with the slicing plane) evolves into a convex chain in $\mathcal{C}_{yz}^+(v)$, which implies that no new patches are intersected by the slicing plane in $\mathcal{C}_{yz}^+(v)$. Thus, no patch has v as a negative x-extremum. If, however, r_v^+ is positive, the number of such patches does not exceed $2r_v^+$. (Figure 2.9 depicts the four basic cases for $r_v^+ = 2$, where the two cusps at the right-hand side figure of each case are the intersections of the two reflex edges with the slicing plane;

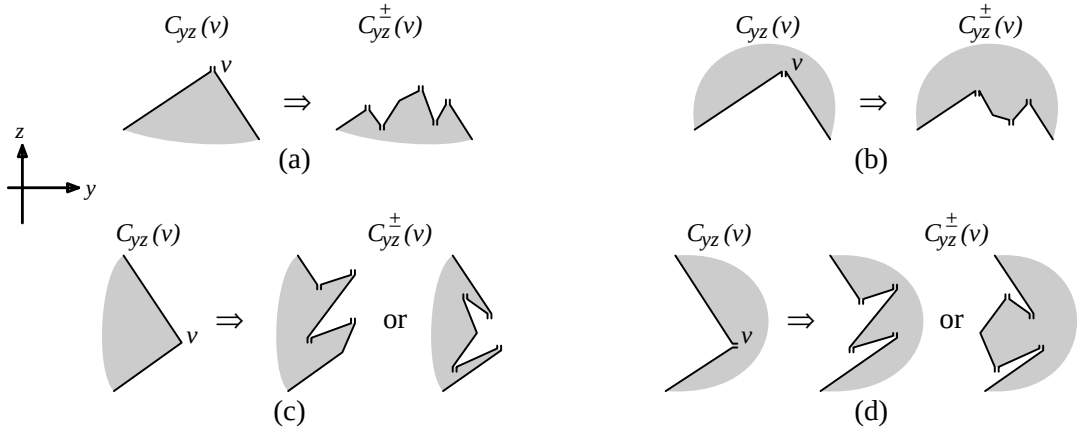


Figure 2.9: (a)-(d) at most 4, 2, 3, and 3 "new" patches respectively.

compare Figures 2.6 and 9 to see how cases with larger r_v^+ can be produced. In general, for the cases (a)-(d), the bounds are $2r_v^+$, $2r_v^+ - 2$, $2r_v^+ - 1$, and $2r_v^+ - 1$ respectively.) So, for all r_v^+, the number of patches to which v contributes negative x-extrema does not exceed $2r_v^+$.

The above arguments extend easily to the case of $\mathcal{C}_{yz}^-(v)$ implying that the number of patches to which v contributes positive x-extrema is at most $2r_v^-$. The combination of these results yields an upper bound of $2r_v^+ + 2r_v^- = 2r_v$ in the number of patches to which v contributes x-extrema.

3. **The vertex v is of degree larger than 2 in $\mathcal{C}_{yz}(v)$:** In this case, the neighborhood of v in $\mathcal{C}_{yz}(v)$ consists of $w_v \geq 2$ wedges touching at v ($w_v = 4$ in Figure 2.10), and it is a combination of the two cases shown in Figure 2.11, as well as those obtained from Figure 2.11 with the pictures corresponding to $\mathcal{C}_{yz}^-(v)$ and $\mathcal{C}_{yz}^+(v)$ interchanged (the dashed curves indicate that some of the wedges may belong to the same polygon of $\mathcal{C}_{yz}(v)$). (Note that wedges whose defining

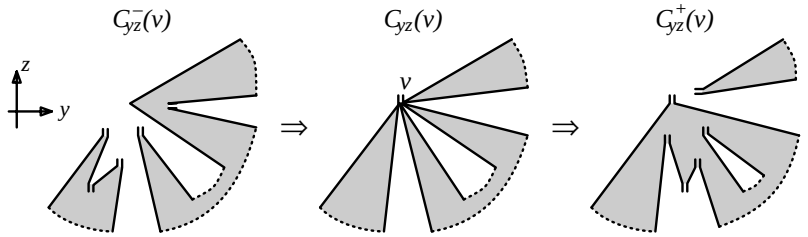


Figure 2.10

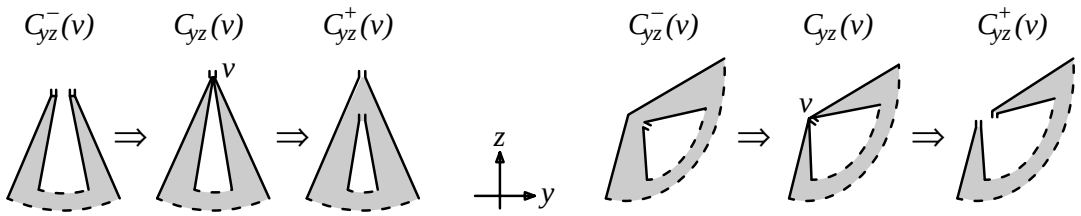


Figure 2.11

edges form a z-monotone chain are expected to contribute fewer patches, since the patch splitting method involves cutting at the z-extrema of the polygons in a yz-cross-section.) These wedges may either merge with or get detached from neighboring wedges in $\mathcal{C}_{yz}^{+}(v)$ and $\mathcal{C}_{yz}^{-}(v)$. In Figure 2.10 for instance, as $\mathcal{C}_{yz}(v)$ evolves into $\mathcal{C}_{yz}^{+}(v)$, the three bottom wedges merge into a single wedge cluster, while the top wedge forms a wedge cluster by itself. Let w_v^+ and k_v^+ (w_v^- and k_v^-) denote the number of such wedge clusters and the number of polygons containing these clusters in $\mathcal{C}_{yz}^{+}(v)$ ($\mathcal{C}_{yz}^{-}(v)$) respectively (in Figure 2.10, we have $w_v^- = 3$, $w_v^+ = 2$, and $k_v^- = k_v^+ = 2$). Clearly,

$$w_v \geq w_v^+ \geq k_v^+ \qquad \text{and} \qquad w_v \geq w_v^- \geq k_v^-. \tag{1}$$

Moreover, the definition of a polyhedron (see Section 2) implies the following crucial observation: if two wedges incident upon v in $\mathcal{C}_{yz}(v)$ merge in $\mathcal{C}_{yz}^{-}(v)$, they have to be detached from each other in $\mathcal{C}_{yz}^{+}(v)$, and vice versa (Figures 2.10 and 2.11 were drawn with this fact in mind); otherwise, we end up with degeneracies like the one exhibited at the solid object shown in Figure 1.3 or the one exhibited at the same point at this very object's complement with respect to $\Re^3$. In terms of w_v, w_v^-, and w_v^+, the observation can be expressed as follows:

$$(w_v - w_v^-) + (w_v - w_v^+) = w_v - 1 \iff w_v^- + w_v^+ = w_v + 1. \tag{2}$$

Finally, if v is incident upon w_v wedges in $\mathcal{C}_{yz}(v)$, then it is incident upon at least $w_v - 1$ reflex edges; recall that the neighborhood of v in $\mathcal{C}_{yz}(v)$ is a combination of the cases in Figure 2.11 (or their right-to-left counterparts), where in both cases shown no new line segments appear either in $\mathcal{C}_{yz}^{-}(v)$ or in $\mathcal{C}_{yz}^{+}(v)$, implying that v does not contribute x-extrema to any of the patches, and yet, in either case, v

is incident upon one reflex edge. Of course, it may be incident upon more such reflex edges, each of which leads to at most two patches to which v contributes an x-extremum, so that such a vertex v is accounted for no more than $2(r_v-(w_v-1))$ patches (see Figure 2.10).

If we sum all the contributions, we obtain the following bound in the number m of patches, where V_1, V_2, V_3, and V_4 denote the sets of vertices falling in cases 1a, 1b, 2, and 3 respectively.

$$\begin{aligned} m &\leq \frac{1}{2}\left(\sum_{v\in V_1}(2r_v+2)+\sum_{v\in V_2}(2r_v-2)+\sum_{v\in V_3}2r_v+\sum_{v\in V_4}2(r_v-w_v+1)\right) \\ &\leq \sum_{v\in\cup V_i} r_v+|V_1|-\sum_{v\in V_4}(w_v-1) \\ &\leq 2r+|V_1|-\sum_{v\in V_4}(w_v-1), \end{aligned} \tag{3}$$

since $\sum_{v\in\cup V_i} r_v = \sum_v r_v = 2r$. In light of the following lemma, inequality (3) implies that $m \leq 2r+2$. ∎

Lemma 3.2. *The total number $|V_1|$ of positive and negative x-extrema of a polyhedron does not exceed $\sum_{v\in V_4}(w_v-1)\ +2$, where V_1, V_4, and w_v are defined as in the previous paragraphs.*

Proof: Let us construct a graph that records the events that mark the history of the yz-cross-section of the polyhedron. Namely, we sweep the polyhedron with a plane normal to the x-axis, and whenever the number of polygons in the cross-section changes (as is the case when positive or negative x-extrema of the polyhedron and vertices in V_4 are encountered), we record the change appropriately (for completeness, we make sure to record all the vertices v in V_4 even those for which $k_v^- = k_v^+$). Note that events where the number of holes in a polygon of a yz-cross-section increases or decreases are not recorded. In particular,

1. at a negative x-extremum (vertex in V_1), we add to the graph two new nodes[1]

[1] We use the term *nodes* of a graph instead of vertices to avoid confusion with the vertices of the polyhedron.

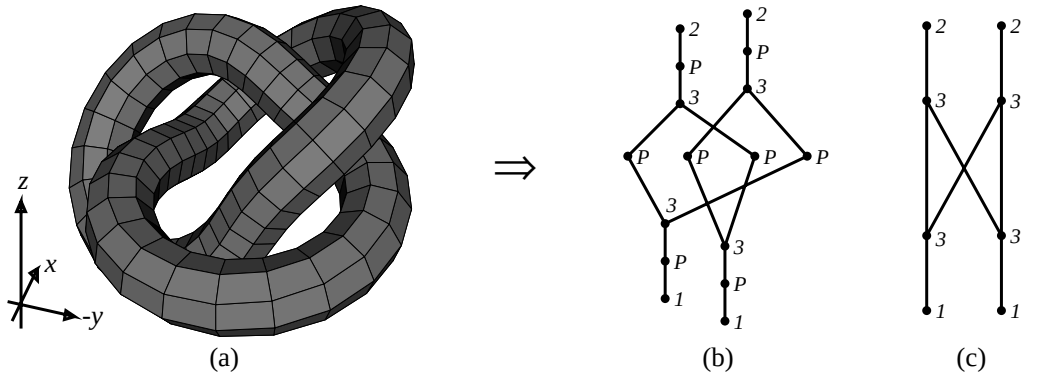


Figure 2.12

which we connect by an edge; the first node corresponds to the negative x-extremum while the second one is a *polygon-node* and corresponds to the series of polygons to which the negative x-extremum evolves;

2. at a positive x-extremum (vertex in V_1), we add one new node that corresponds to the positive x-extremum and we connect it to the polygon-node that represents the polygon which reduced to this positive x-extremum during the sweeping;

3. at a vertex v in V_4, we add one new node that corresponds to v and edges connecting it to the representatives of the k_v^- polygons in $\mathcal{C}_{yz}^-(v)$. Moreover, k_v^+ polygon-nodes are added, one for each of the k_v^+ polygons in $\mathcal{C}_{yz}^+(v)$, and edges are introduced between them and the node corresponding to v.

Since we are dealing with a single polyhedron, the resulting graph is connected. Figure 2.12(b) shows the graph that corresponds to the polyhedron of Figure 2.12(a). The numbers 1, 2, and 3 denote negative and positive x-extrema, and vertices in V_4 respectively (according to the cases above), while the letter P denotes polygon-nodes. Note that all polygon-nodes are of degree 2, and no two of them are adjacent. Moreover, the nodes corresponding to the negative and positive x-extrema of the polyhedron are of degree 1.

To simplify matters, we remove from the graph all polygon-nodes by coalescing their incident edges into a single edge; the resulting graph is a connected multigraph (see Figure 2.12(c)), whose node set is in one-to-one correspondence with the union

of V_1 and V_5. If we denote by $\mathcal{E}$ the total number of edges of the multigraph, we have

$$2\,\mathcal{E} \;=\; \sum_{\text{node } v} \text{degree}(v) \;=\; \sum_{v\in V_1} 1 \;+\; \sum_{v\in V_4} (k_v^- + k_v^+) \;\le\; |V_1| + \sum_{v\in V_4} (w_v + 1), \qquad (4)$$

since, for any vertex $v \in V_4$, inequalities (1) and (2) imply that $k_v^- + k_v^+ \le w_v^- + w_v^+ = w_v + 1$. Connectivity, on the other hand, implies that the number of edges is at least equal to 1 less than the number of nodes of the graph, that is, $\mathcal{E} \ge |V_1| + |V_4| - 1$, which combined with (4) yields

$$2\Big(|V_1| + |V_4| - 1\Big) \;\le\; |V_1| + \sum_{v\in V_4} (w_v + 1),$$

which proves the lemma. ∎

2.2.2 The Second Phase.

The previous phase produces patches that are not necessarily simple and may form spirals around the y-axis. Moreover, although the intersection of such a patch with any plane normal to the x-axis consists of a number of disconnected polygonal lines, each of which is monotone with respect to the z-axis, this is not sufficient to ensure that the patch is monotone with respect to the xz-plane; patch monotonicity is guaranteed only if the patch is decomposed into subpatches so that the polygonal lines that form the above intersection belong to different subpatches. This will be our goal in this phase, i.e., to decompose each of the patches produced in the previous phase into subpatches, the intersection of each of which with any plane normal to the x-axis is a single connected polygonal line monotone with respect to the z-axis. Then, making sure that the projections of the subpatches on the xz-plane are convex polygons implies that the final patches are convex-like (Lemma 2.1.1).

The method that we are going to use parallels the way a nonconvex polygon (that may contain holes) is partitioned into convex pieces by using cuts parallel to a chosen direction to resolve the polygon's cusps. Namely, each patch is split by means of cuts along the intersections of the patch with planes normal to the x-axis that pass through the reflex vertices of the given polyhedron. Let us consider the yz-cross-section of the polyhedron at its vertex v (after the first phase). Due to the first phase cuts, the

boundaries of the polygons of the cross-section are decomposed into polygonal lines monotone with respect to the z-axis. Unless v is a point hole in one of these polygons or a polygon reduced to a point (in which case we do nothing), v is incident upon at least two of these polygonal lines (Figure 2.13). If v is the vertex of such a chain with the smallest z-coordinate, we refer to the chain as the *up-chain* of v; otherwise, it is the *down-chain* of v. (In case v is not an endpoint of such a chain we split the chain into one up- and one down-chain, as shown in Figure 2.13.) If w wedges touch at v in the corresponding yz-cross-section, then v is incident upon at least $w-1$ reflex edges and $2w$ up- or down-chains. In other words, if v is incident upon r_v reflex edges, it is incident upon at most $2r_v+2$ chains. It should be noted that these chains may or may not belong to the same patch; eventually they will not.

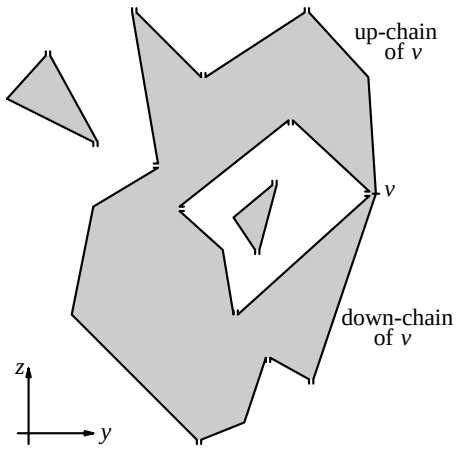


Figure 2.13

Cutting along the up- and down-chains of each reflex vertex will result in the desired decomposition. For simplicity, we cut along the up- and down-chains in separate passes, so that no cuts are advancing in opposite directions at the same time.

We describe below our method to generate cuts along the up-chains of the reflex vertices. Let us first remark that a brute-force approach may cut on the order of n facets every time we process a reflex vertex, which would produce an unacceptably large $\Omega(nr)$ number of patches. Our plan is to discontinue some of the cuts that might slice too many facets. The key idea is illustrated in Figure 2.14. Assume that we advance four cuts upwards through facet f. The point is to extend only the leftmost and rightmost ones through adjacent facets upwards past f; we stop the remaining two cuts at the edge incident upon f with the largest x-extent, and we generate a cut

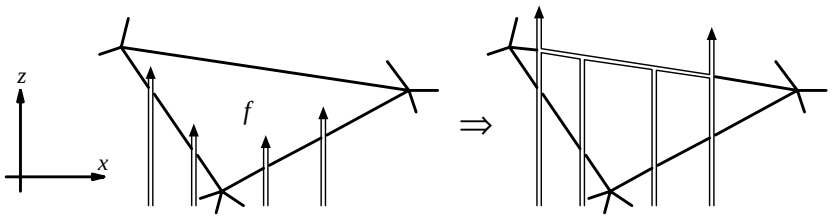


Figure 2.14

along that very edge, which extends between the intersections of the edge with the rightmost and leftmost cuts. Thus, we maintain the following invariant:

Cut-invariant: At most two cuts cross any edge propagating through a facet to an adjacent one.

We deal with each patch in turn; we sweep through it, moving upwards, advancing the cuts as we proceed. Each facet of the patch is processed after its adjacent facet(s) below it in the patch have been processed. Recall that if a cut has been generated along an edge during the first phase, the two facets incident upon the edge are no longer considered adjacent (although they may still belong to the same patch). Bearing this in mind, it is clear that some facets have no adjacent facets below them, and so these facets can be processed right away. Figure 2.15 depicts a patch with these facets shown highlighted.

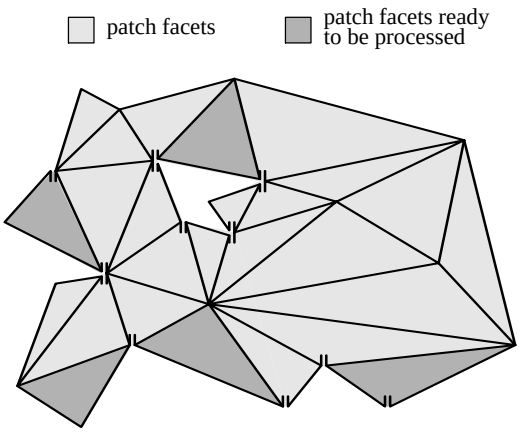


Figure 2.15

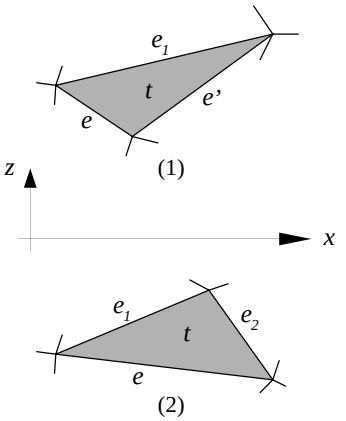


Figure 2.16

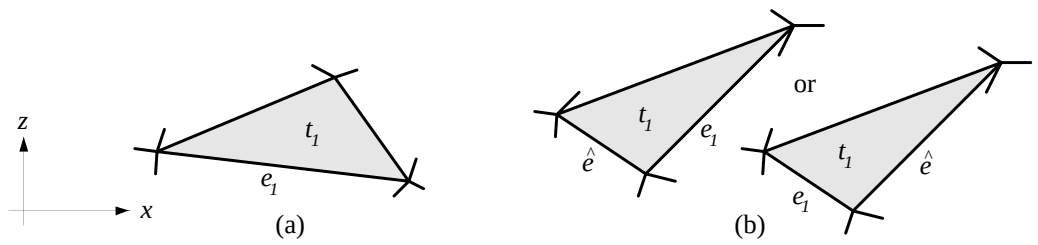


Figure 2.17

We start by inserting these facets in a queue Q intended to store the facets of the patch that are ready to be processed. Initially, our cut-invariant holds trivially. Then, for each element t of Q we iterate on the following procedure: depending on whether t is as shown in Figure 2.16(1) or 2.16(2), we execute step 1 or step 2 respectively.

1. The facet t is as shown in Figure 2.16(1), i.e., the edge e_1 incident upon t with the largest x-extent is "above" the other two edges e and e' incident upon t. Cuts may be propagating in t through the edges e and e', while an additional cut may emanate from the vertex incident to both e and e'. If the total number of cuts proceeding through t is no more than 2, then the cuts are simply extended all the way through t ready to proceed to adjacent facets, and they are associated with e_1. If, however, the number of cuts is larger than 2, we apply the idea illustrated in Figure 2.14. We extend only the leftmost and rightmost cuts all the way through t, and associate them with e_1. All remaining cuts are extended up to e_1 and are stopped there, while a cut is generated along e_1 between the intersection points of e_1 and the leftmost and rightmost cuts. So, our cut-invariant is maintained in this way. Finally, if no cut was made along e_1 during the first phase, we check whether the facet t_1, other than t, incident upon e_1 is a candidate for the queue Q. Namely, if t_1 is as in Figure 2.17(a), then it is inserted in Q; otherwise, t_1 is as in Figure 2.17(b) and is inserted in Q only if either a cut has been generated during the first phase along the edge $\hat{e}$, or the facet, other than t_1, incident upon $\hat{e}$ has already been processed.

2. The facet t is as shown in Figure 2.16(2), i.e., the edge e incident upon t with the largest x-extent is "below" the other two edges e_1 and e_2 incident upon t. If cuts

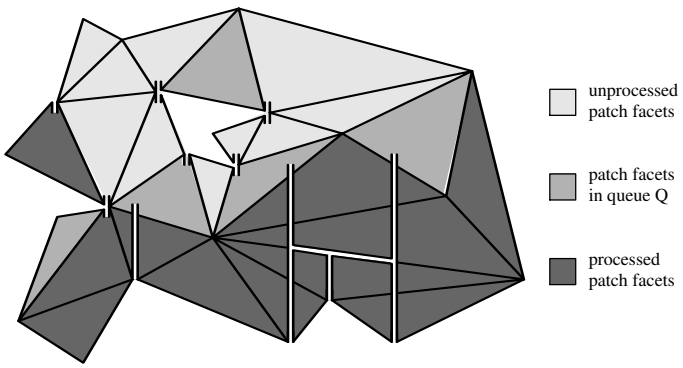


Figure 2.18

are propagating through e, we extend them through t and we associate them with e_1 or e_2 depending on which edge they intersect. By induction, our cut-invariant is maintained. Next, we test the facets t_1 and t_2 (other than t), that are incident upon e_1 and e_2 respectively as candidates for the queue Q. (Note that either one or both t_1 and t_2 may not exist as cuts along e_1 or e_2 during the first phase may have disconnected the patch at these edges.) The procedure is the same as that involving t_1 in the previous case. Specifically, if t_1 is as shown in Figure 2.17(a), i.e., e_1 is the edge incident upon t_1 with the largest x-extent, we insert t_1 in Q; if not, in which case t_1 is as in Figure 2.17(b), we insert t_1 in Q only if either we have generated a cut along the edge $\hat{e}$ during the first phase, or the facet, other than t_1, incident upon $\hat{e}$ has already been processed. The same test is also applied to t_2, if it exists.

Then, t is removed from Q and we proceed with the next facet in Q.

Figure 2.18 shows a snapshot as the procedure is applied to the patch of Figure 2.15. We can prove inductively that when the queue Q eventually becomes empty, all facets of the patch have been processed. Furthermore, our cut-invariant ensures that no more than two cuts are crossing an edge cutting both its incident facets.

This completes our work with the up-chains. Next we apply the same procedure with respect to the down-chains. Note that we take into consideration any cuts generated during the previous pass involving the up-chains. Consider, for instance, Figure 2.19. If a cut c_1 along a down-chain reaches an edge along which we generated

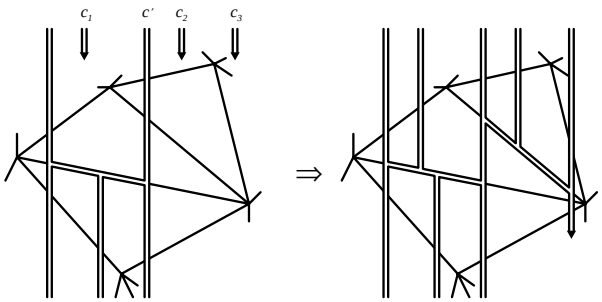


Figure 2.19

a cut during the previous pass, then the cut is stopped there and its processing is considered completed. Moreover, if two cuts c_2 and c_3 along two down-chains reach an edge e that has been cut by an up-chain cut c' and c_2 is between c' and c_3 (with respect to the x-axis), then c_2 is stopped there, and a cut between the intersections of e with c' and c_3 is generated along e. This establishes our cut-invariant for cuts along down-chains.

At the end, pieces of a single facet of the polyhedron may belong to several different patches. Interestingly, however, due to our cut-invariant, no more than four cuts through any given facet proceed to adjacent facets both up and down. These would be the leftmost and rightmost cuts through the facet along both up- and down-chains. Any other cuts stop at the edge incident upon the facet with the largest x-extent. The total time spent in this phase is linear in the number of facets of the polyhedron plus a constant overhead per segment of each cut. As the total number of these segments is proven in Section 2.2.3 to be linear in the size of polyhedron, the total time required for this second phase is $O(n)$.

2.2.3 Description of the patches produced.

Each of the patches that the algorithm produces consists of a portion of the boundary of the polyhedron that is clipped from left and right by two planes normal to the x-axis (cuts along up- or down-chains), and at the top and bottom by either an edge of the polyhedron (a reflex edge, for instance), or a polygonal line consisting of edges that contribute extrema in the cross-section of the polyhedron. Moreover, the patches

are simple and monotone with respect to the xz-plane. To prove this, we need only show that the intersection of any patch produced by the second phase with a plane normal to the x-axis is a single polygonal line monotone with respect to the z-axis.

Suppose, for contradiction, that this is not true. Then, since the patch is connected, there exist points p and q of the patch that have equal x-coordinates, and are such that a plane normal to the x-axis passing through them intersects the patch in a single polygonal line, whereas it intersects it into two disconnected polygonal lines if we move it slightly either to the left or to the right along the x-axis (to the right, in the case shown in Figure 2.20). (Note that the points p and q may coincide.) There exist therefore line segments e_1 and e_2 incident upon p and q respectively, along which cuts have been generated (e_1 and e_2 may coincide if p and q coincide). These cuts do not proceed along up- or down-chains of vertices, and thus e_1 and e_2 lie on edges of the polyhedron. Moreover, as the projections of patches onto the xz-plane exhibit internal angles at most equal to π at the projections of new vertices introduced during the second phase (see Figure 2.19), p and q must be vertices of the polyhedron. In fact, they must both be reflex ones, otherwise e_1 and e_2 would not have been cut. (Recall that e_1 and e_2 would have to contribute z-extrema at the corresponding yz-cross-sections, which is impossible if p and q are nonreflex vertices as shown in the figure.) This leads to contradiction, since, if this were the case, then cuts along the up-chain of p and the down-chain of q would have split the patch.

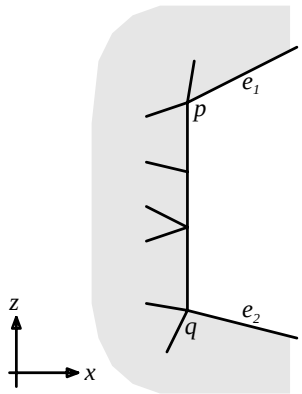


Figure 2.20

We prove next that the patches project on the xz-plane into convex polygons, which, in conjunction with Lemma 2.1.1, establishes that they are convex-like.

Lemma 2.3.1. *The projection of any patch on the xz-plane is a convex polygon.*

Proof: Since the patch is simple, its projection on the xz-plane is a simple polygon too. Let us now prove that the projection is indeed a convex polygon. Suppose, for

contradiction, that there exists a cusp, i.e., a vertex of the polygon such that the (interior) angle formed by its two incident edges exceeds π. The monotonicity of the patch with respect to the xz-plane implies that the cusp is the projection of some vertex v on the boundary of the patch. Since the second phase cuts contribute vertices that are the intersections of edges of the polyhedron with the "clipping" planes, no such vertex can project into a cusp on the xz-plane. This implies that v lies on a first phase cut, and in fact it is a nonreflex vertex of the polyhedron. But then, it cannot project into a cusp, according to Lemma 2.2.1, which leads to contradiction. ∎

Description size and total number of patches. Let us consider a polyhedron P of f facets and e edges, r of which are reflex. We compute the total number of edges of all the patches, where edges along the cuts are counted twice. The analysis proceeds in an incremental way by taking into account the new edges that each step of our algorithm introduces. The triangulation of the boundary of P does not affect the order of magnitude of the number of edges, so that this number is $O(e)$ before the beginning of the first phase. As the cuts of the first phase proceed along edges of P, the number of edges of all the patches at the end of the phase is at most twice their number after the boundary triangulation. During the second phase, several new edges are introduced in the following two ways: (i) an edge is split into two when a cut intersects it, or (ii) a new edge is introduced by cutting through a facet. Recall that for each reflex vertex we generate cuts along the intersections of the incident patches with a plane normal to the x-axis that passes through that vertex. As mentioned earlier, for a vertex that is incident upon r_v reflex edges, we generate at most $2r_v + 2$ cuts along up- or down-chains; since $\sum_v r_v = 2r$ and the total number of reflex vertices does not exceed $2r$, the number of cuts generated during the second phase of our algorithm is at most $8r$.

Our goal is to charge each new edge to either a facet or a cut generated in the second phase. Specifically, each facet is charged with the number of edges that result from cuts traversing it and proceeding through adjacent facets, while each cut is charged with the number of additional edges that its traversal through the very last facet causes. In particular, each facet is charged with 12 units; note that a cut that

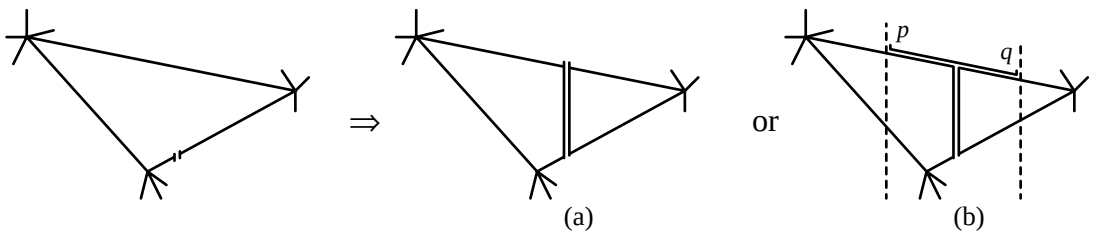


Figure 2.21

traverses a facet proceeding to adjacent ones leads to the creation of three new edges (see Figure 2.21(a)), and that at most 2 such cuts are allowed per facet, for each of the two passes of the second phase (cut-invariant). In a similar fashion, each cut is charged with 3 units to pay for the new edges created in the facet that the cut traverses last. Additionally, a fourth unit is charged to each cut to pay for the extra edge that is created in the case that a cut is generated along a portion of the edge with the largest x-extent incident upon a facet (edge pq in Figure 2.21(b)). It should be obvious that this charging scheme pays for all the edges introduced during the second phase. Summing everything up, we end up with a total of at most $O(e) + 4 \times 8r + 12f$ edges, which is linear in the size of the input.

We now estimate the number of patches that are finally produced. As mentioned already, the first phase produces no more than $2r + 2$ patches. We partition the set of cuts generated during the second phase into two classes:

(i) those that extend all the way to the boundary of the corresponding patch produced during the first phase, and

(ii) the remaining ones, that is, those that were stopped at an edge that had been cut at least twice (Figure 2.21(b)).

Each cut in class (i) increases the number of patches by at most 1, whereas each cut in class (ii) increases the number of patches by at most 2 (see Figure 2.21). As the total number of cuts is no more than $8r$, and the number of cuts in class (ii) is bounded above by $8r - 4$, the total number of patches produced cannot exceed $2r + 2 + 8r + 8r - 4 = 18r - 2$.

Our results are summarized in the following theorem.

Theorem 2.4.1. *The boundary of a nonconvex polyhedron of n vertices and r reflex edges can be subdivided into $18r - 2$ patches, each of which lies on the boundary of its convex hull. The decomposition can be carried out in $O(n + r \log r)$ time and $O(n)$ space.*

Chapter 3

Tetrahedralizing a Nonconvex Polyhedron

In this chapter, we deal with the problem of partitioning a three-dimensional polyhedron P into tetrahedra, that is, decomposing P into three-dimensional simplices, which do not overlap except on their boundaries, and whose union is P. We will refer to such a partition as a *tetrahedralization* of P. Note that we do not require that the tetrahedralization be a *cell complex*, that is, a partition in which the intersection of any two neighboring cells is a face of both these cells. Further discussion on this issue can be found in Section 3.5. Moreover, we are interested in partitions into pieces whose vertices may not necessarily be vertices of the given polyhedron P; disallowing such vertices, known as *Steiner points*, makes the problem much harder ([60]). The problem arises in mesh-generation for applications of the finite-element method, in computer-aided design and manufacturing, and in robotics.

3.1 Basic Strategy

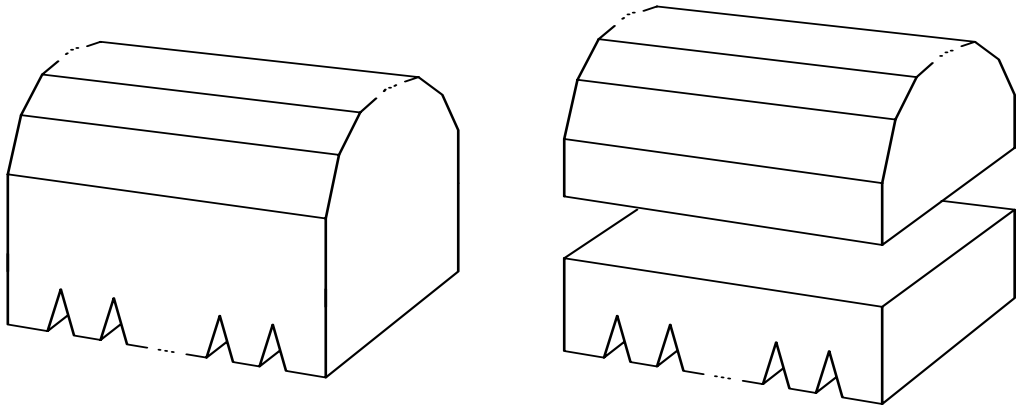
As in many other cases, convexity greatly simplifies the polyhedron tetrahedralization problem. Indeed, in order to partition a convex polyhedron into tetrahedra, we need only triangulate its boundary, which is particularly easy taking into account that the facets of a convex polyhedron are convex polygons. Then we pick an arbitrary vertex

q, and a tetrahedralization of the polyhedron can be readily produced by reporting one tetrahedron with vertices q, u, v and w, for each (triangular) facet (u, v, w) of the polyhedron, provided that q, u, v and w are not coplanar. The entire process requires linear time, and partitions a polyhedron of n vertices into at most $2n - 7$ tetrahedra.

Handling nonconvex polyhedra is, therefore, what makes the polyhedron tetrahedralization problem challenging. The problem is obviously closely related to the more general problem of partitioning a polyhedron into convex pieces, as a convex partition can be refined into a tetrahedralization in time linear in the combined size of all the convex pieces. For the latter problem, Chazelle described an algorithm that produces a convex cylindrical partition of a polyhedron, by slicing it with planes that resolve the polyhedron's reflex edges ([8]), meaning that the (interior) dihedral angles formed by the facets incident upon the reflex edges are split by the corresponding plane into two parts none of which exceeds π. The number of pieces in the partition is quadratic in the size of the polyhedron in the worst case. Moreover, he established a matching worst-case lower bound, by means of an infinite family of polyhedra, which interestingly are "highly nonconvex", in the sense that the number of their reflex edges is proportional to their sizes (see Section 1.2.2).

The asymptotic discrepancy between the convex and the highly nonconvex cases suggests that the size of the tetrahedralization of a polyhedron should depend on both the size of the polyhedron, and a measure of its nonconvexity; as such, we pick the number of its reflex edges. In this way, we will be able to take advantage, for instance, of the fact that a polyhedron is almost convex. A slight modification of Chazelle's construction yields a lower bound of $\Omega(n + r^2)$ for the size of the tetrahedralization that a polyhedron of r reflex edges and $n > 6r + 8$ vertices admits. If r is the number of reflex edges in Chazelle's polyhedron, we need only glue a convex block of $n - 6r - 8$ vertices on one of its facets.

Moreover, the same asymptotic discrepancy implies that it is to our advantage to try to chop convex chunks off the polyhedron before any attempt to resolve the reflex edges. To illustrate the idea, let us consider the polyhedron shown in Figure 3.1(a), where $\Theta(n)$ narrow rectangular facets form a convex upper hull. The nonconvexity



(a) (b)

Figure 3.1

of the polyhedron is due to a row of $\Theta(n)$ reflex edges carved in the lower hull along a direction perpendicular to the longest dimension of the upper hull facets. It is obvious that, each of these facets is cut up into $\Theta(n)$ pieces if vertical fences are erected through the reflex edges in order to resolve them, and the size of the convex decomposition and subsequent tetrahedralization is $\Theta(n^2)$. However, slicing the polyhedron into two pieces as shown in Figure 3.1(b), so that the upper and lower hulls of the polyhedron belong to different pieces, results in a tetrahedralization of size $\Theta(n)$, a gain of one order of magnitude over the previous case.

Of course, finding such slicing planes may be computationally expensive or even impossible. But the idea can be applied in a localized fashion too. Namely, we identify vertices that are incident upon nonreflex edges, and we remove them one by one, much like the way we would pull a ski hat off someone's head. In this way, the polyhedron gradually shrinks, eventually containing the nonconvex features and (hopefully) just a few more vertices. This constitutes the *pull-off phase*. Then, we enter the *fence-off phase*, in which vertical fences are erected through each edge of the polyhedron, producing a cylindrical decomposition that is next easily refined into a tetrahedralization. The algorithm partitions a polyhedron of n vertices and r reflex edges into $O(n + r^2)$ tetrahedra, which is asymptotically optimal in the worst case in light of the above mentioned lower bound. It runs in $O((n + r^2) \log r)$ time, and it

requires $O(n + r^2)$ space.

A preliminary version of the algorithm has been described in [16]. The algorithm is not difficult to code, and it has been implemented in C. Details on the implementation are included in the following chapter.

3.2 Cups, Crowns, Domes, and Other Widgets

Let P be a polyhedron. As mentioned in Section 1.1.2, an edge e of P is said to be *reflex* if the (interior) dihedral angle formed by its two incident facets exceeds π. By extension, we say that a vertex is *reflex* if it is incident upon at least one reflex edge. A nonreflex vertex is *flat* if all its incident facets lie in at most two distinct planes; otherwise, we say that it is *pointed* (Figure 3.2). It is easy to see that a pointed vertex cannot be incident upon two collinear edges, although it can be incident upon two coplanar (adjacent) facets of P.

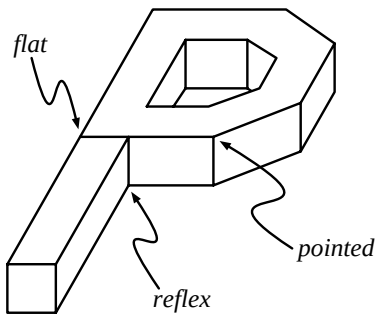
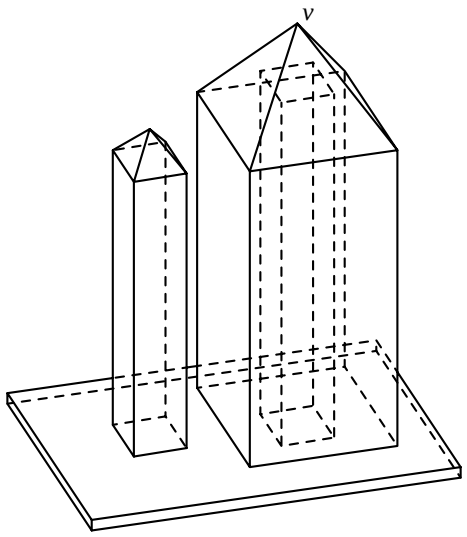
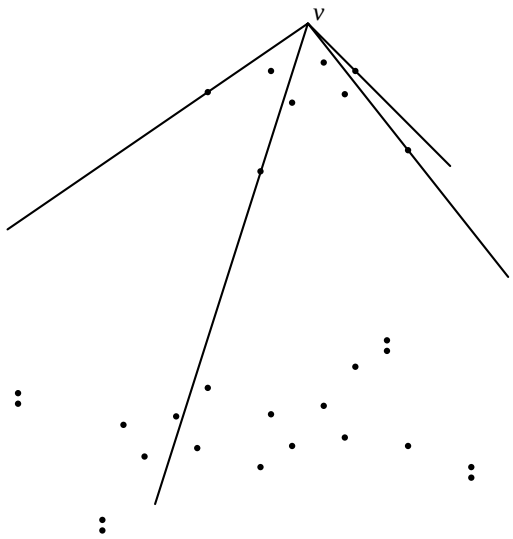


Figure 3.2: The different types of vertices.

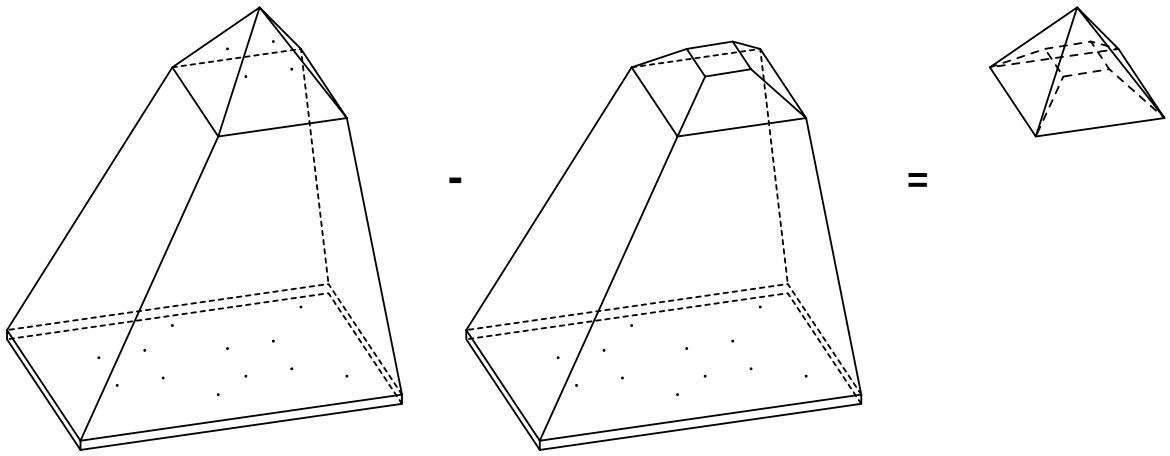
We are now ready to define the key notion of the "cup" of a pointed vertex. We start by defining the *cone* of a pointed vertex v as the unbounded convex polyhedron spanned by the edges incident upon v, or equivalently, $cone(v)$ is the locus of points $v + \sum_{1 \leq i \leq k} \alpha_i (w_i - v)$, where $w_1, \ldots, w_k$ are the vertices of P adjacent to v and the α_i's are arbitrary nonnegative reals (Figure 3.3b). The cone of v contains a subset S_v of the vertex set of the polyhedron P, whose elements may lie on the boundary of the cone or strictly in the cone's interior. Vertex v and its adjacent vertices obviously belong to S_v, and they all lie on the boundary of the cone. Let H_v and H_v^- be the convex hulls of the vertices in S_v and $S_v \setminus \{v\}$ respectively. The *cup* of v is defined as the polyhedron formed by the closure of $H_v \setminus H_v^-$ (Figure 3.3c). Roughly speaking,



(a) A polyhedron.



(b) The cone of the pointed vertex v.



(c) The cup of v is the difference of the convex hulls H_v and H_v^-.

Figure 3.3

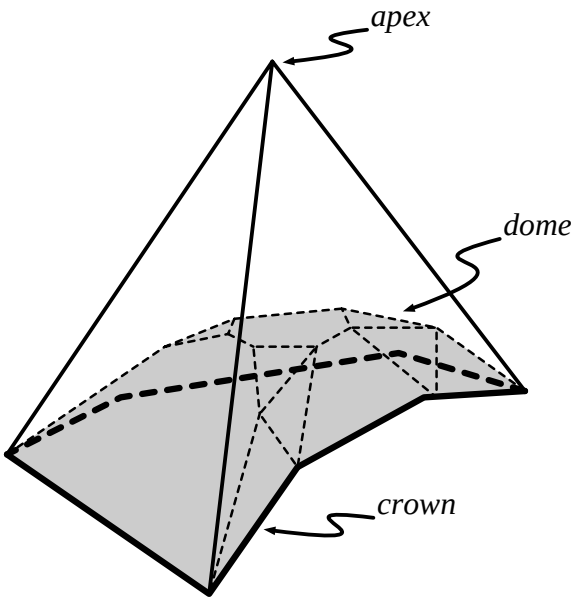


Figure 3.4: A cup.

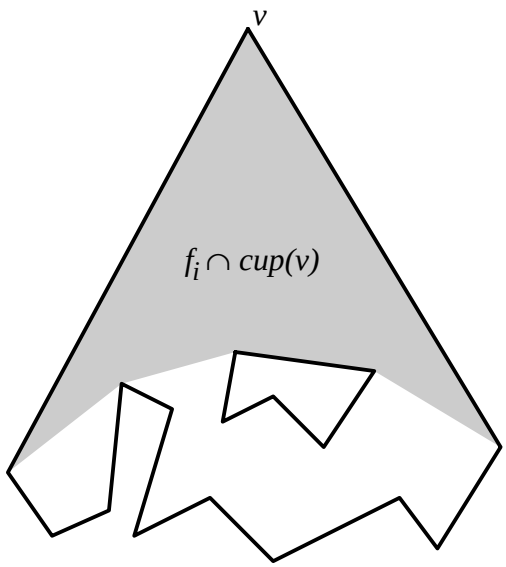


Figure 3.5: A facet of a cup.

the cup of v can also be defined as follows. Imagine that the boundary of P is a stretched rubber sheet, held in place by nails at the vertex locations. If we remove the nail at the pointed vertex v, the rubber sheet locally around v moves sweeping a portion of P that is precisely the $cup(v)$. The notion of the cup of a pointed vertex is the basis of our algorithm, as the removal of a vertex, to which we have been alluding earlier, meant precisely the removal of its cup.

It follows from the definition that the cup of v is a star-shaped polyhedron whose kernel contains v (Figure 3.4). We will refer to the pointed vertex v as the *apex* of the cup. The boundary of the cup contains a number of polygons incident upon v which are glued to H_v^- to form the boundary of H_v. The gluing border is a closed simple polygonal curve, called the *crown* of v, and can be centrally projected (with respect to v) onto a plane into the boundary of a convex polygon. The crown acts as a Jordan curve on the boundary of the cup, which it separates into one piece on the boundary of the cone and a concave (with respect to the cup) polyhedral patch, which we call the *dome* of v. Edges and vertices of the dome that are not in the crown are called *internal*.

Let us prove that the facets of the cup that are incident upon v all lie in the boundary ∂P of P. We collect the facets of P incident upon v, and (mentally) merge any coplanar ones. This produces superfacets $f_1, \ldots, f_k$ (given in either circular order around v), such that the dihedral angle between any two adjacent f_i's is strictly less than π. Each facet of the cup incident upon the apex is the intersection of the cup with one of the f_i's, which implies that such a facet is the two-dimensional equivalent of the cup of v on f_i. In particular, the facet is the closure of the difference $h_i \setminus h_i^-$, where h_i and h_i^- are the two-dimensional convex hulls of the vertices of f_i, and the vertices of f_i but v respectively (see Figure 3.5). The construction proves that the facets of the cup incident upon the apex lie in ∂P, and in fact any point in them belongs to the closure of a facet of P incident upon v. However, a facet of a cup does not necessarily lie within any given facet of P. Its boundary consists of a two-edge convex chain at v — v is a pointed vertex of P, and therefore exhibits an angle less than π in each of the f_i's — followed by a weakly concave chain possibly reduced to a single edge. (By a *convex* (resp. *concave*) chain we mean a piece of a polygon's boundary which always turns strictly right (resp. left) when traversed clockwise. In a *weekly* convex or concave chain we allow also collinear adjacent edges.) Note that the concatenation of these concave chains in sequence yields the crown of v.

A more interesting observation is that the cup lies inside P. This follows from the fact that we have just proven. Indeed, it would be impossible for a facet or an edge of P to intersect the interior of the cup, unless a vertex of P did, which is, of course, ruled out by the very definition of the cup.

Let us now investigate the relationship between vertices and edges of a cup and the vertices and edges of the polyhedron. Clearly, all cup vertices are vertices of P, although the same cannot be said of cup edges. It is easy to see that the edges internal to the dome do not have to be edges of P, since a dome does not have to intersect the boundary ∂P of P except at the crown. But not even the crown edges bound necessarily any facet of P, although they certainly belong to ∂P (see Figure 3.5). However, as the convex hull operation merges coplanar facets, the edges of the cup incident upon the apex are not incident upon coplanar facets, and therefore are (nonreflex) edges of P. Turning to vertices now, we observe that the way the

dome is constructed implies that for each of its internal vertices there exists a small open half-ball centered at the vertex that lies entirely in the corresponding cup, and hence in P. This implies that vertices internal to domes are contributed either by reflex vertices of P, or by flat vertices whose incident facets are all coplanar, and consequently if an edge of P is internal to a dome then it is either reflex or incident upon two coplanar facets. On the other hand, the crown vertices may be reflex, flat, or pointed vertices of the polyhedron, although pointed vertices can only contribute the endpoints of the concave chains that compose the crown. Since the edges that connect these endpoints to the apex are also edges of P, a pointed vertex can be on the crown of another pointed vertex only if these two vertices are adjacent in P. Note however, that the converse is not always true. In particular, if two pointed vertices are connected by an edge which is incident upon two coplanar facets, none of them lies on the crown of the other.

Before concluding this section, we prove the following useful lemma:

Lemma 3.2.1. *Any reflex vertex w of P which is internal to a dome has at least three reflex edges of P incident upon it.*

Proof: Since w is an internal vertex of the dome of some pointed vertex, it is a vertex of the convex hull C_w of the point-set consisting of w and all its adjacent vertices in P. Note that all facets and edges incident upon w lie either in the interior or on the boundary of C_w. The definition of a polyhedron implies that w is incident upon at least three edges of this convex hull, and the lemma follows from the observation that the convex hull edges incident upon w are in fact reflex edges of the polyhedron. ∎

Removing the cup of a pointed vertex of the polyhedron P has the effect of reducing the number of vertices of P by one. Doing this repeatedly, we hope we will end up with a polyhedron of "small" size, on which we can apply the quadratic-size fencing process incurring a small, compared to the initial size of the polyhedron, cost. The underlying danger, however, is that removing cups may produce a solid object that is not a polyhedron, as is the case, for instance, with the polyhedra of Figure 3.6 when the cup of v is removed. (In the case of Figure 3.6(c), the resulting object has two coinciding facets.) These examples show that we should beware of cups whose domes

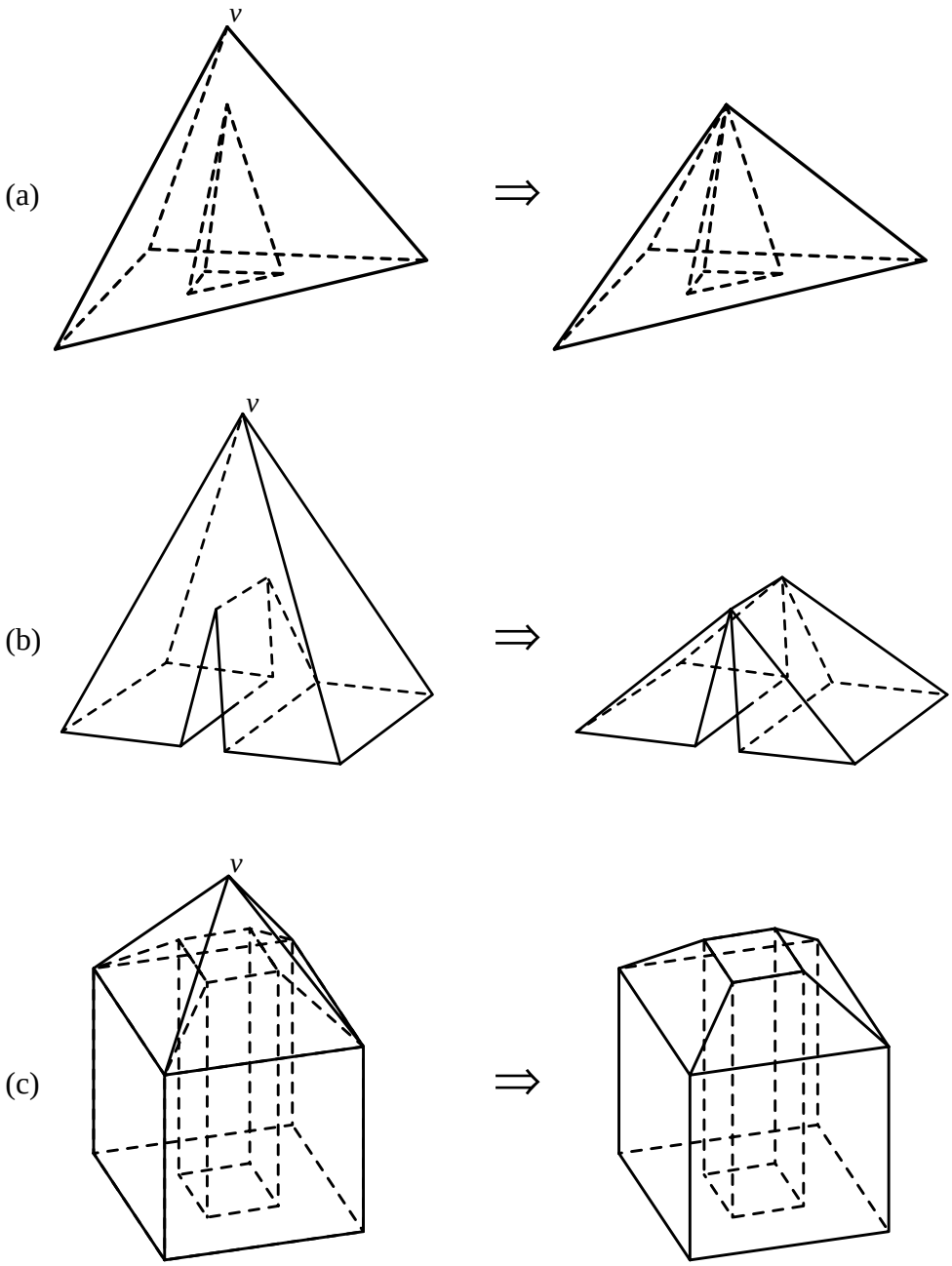


Figure 3.6: Extracting a hindered cup.

have either internal vertices, or internal edges that are also edges of the polyhedron. We refer to such cups as *hindered*. The removal of the cup of a pointed vertex that is not hindered guarantees that the resulting three-dimensional object will still be a polyhedron.

3.3 Foundations of the Pull-Off Phase

In this section that provides the theoretical background of our algorithm, we concentrate in the pull-off phase, and we present a few technical lemmas that establish its feasibility and efficiency. Namely, we prove that all but at most $2r$ of the pointed vertices of a polyhedron of n vertices and r reflex edges have their cups unhindered. Assuming that the polyhedron has no flat vertices, this means that the cups of at least $n-4r$ vertices can be removed while the resulting object remains a polyhedron. This result leads to an efficient way of removing all but a small multiple of r of the polyhedron's vertices.

In the following, we assume that P is a polyhedron of arbitrary genus with n vertices and m edges, exactly r of which are reflex.

Lemma 3.3.1. *If a vertex z of P is internal to the domes of two distinct pointed vertices v and v' of P, then v belongs to the crown of v', and vice versa. The same is true, if an edge of P is internal to both domes.*

Proof: Assume that the intersection Q of the interiors of the cups of v and v' is empty. Since z is internal to the dome of v, there exists a small open half-ball centered at z that lies entirely within the cup of v. A similar statement holds for v' as well, and since the two half-balls are nonintersecting, the domes of both v and v', locally around z, have to lie on the plane separating the two half-balls, which comes into contradiction with the fact that P is a polyhedron with z on its boundary.

Hence, the intersection of the interiors of the two cups is nonempty. The closure of Q must have at least one vertex outside the dome of v, otherwise it would have empty interior. The only such vertex can be the apex v, however, since the interior

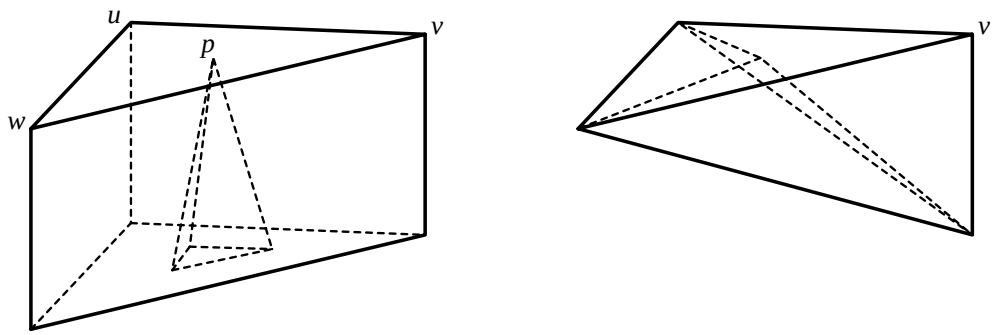


Figure 3.7: The reflex vertex p hinders the cups of u, v and w. On the right, the cup of v.

of a cup is free of vertices. Thus, v lies in the cup of v'. Since it is a pointed vertex of P, it must lie on the crown of v'.

This proves the first part of the lemma. The second part follows trivially, if we let z be the midpoint of the edge. ∎

Let us now try to assess the damage that vertices and edges of P can cause to the cups of pointed vertices by introducing internal vertices and edges in their domes respectively.

Lemma 3.3.2. *A vertex of P can contribute internal vertices to the domes of at most three pointed vertices.*

Proof: Suppose, for contradiction, that a vertex p is internal to the domes of four pointed vertices u, v, w, and z of P. It follows from Lemma 3.3.1 that the crown of each of the four apexes contains the other three as vertices. As a result, the six edges of the tetrahedron T formed by u, v, w, and z all lie in ∂P. This tetrahedron cannot have empty interior, otherwise one of the four apexes would be either reflex or flat. Furthermore, since p is an internal vertex of all four domes, and the edges of T are nonreflex edges of P, p must lie in T. However, it cannot lie on any of T's edges, otherwise there would be two nonadjacent apexes.

Since p is internal to the domes of all four apexes, there exists a small ball centered at p that lies entirely in each of their cones. The polyhedron's definition implies that

this ball intersects P's complement, and consequently it contains a point q outside P that avoids each of the six planes defined by p and any two of the four apexes. Then, p must lie in the relative interior of one of the four tetrahedra defined by q and any three of the apexes. All four vertices of that tetrahedron, however, lie in the cone of the fourth apex. Thus, p cannot be a vertex of that apex's dome, which gives us a contradiction.

Figure 3.7 shows that this bound is tight. ∎

Lemma 3.3.3. *Given an edge pq of P, there exist at most two pointed vertices u and v, such that pq is internal to the domes of both u and v, and p and q lie on the crowns of both domes.*

Proof: Suppose that pq is internal to the domes of u, v, and w, and that p and q lie on all three crowns. Then, all six line segments pu, qu, pv, qv, pw, qw lie in ∂P, and all three triangles puq, pvq, pwq lie entirely in P. Furthermore, it follows from Lemma 3.3.1 that each of u, v, and w lies on the crown of the other two, making it impossible for the boundary of each of the triangles puq, pvq, pwq to contain more than one of u, v, and w. We can thus establish an ordering of these triangles clockwise around pq; let it be puq-pvq-pwq.

Let us denote by $[x, l, y]$ the dihedral angle that is bounded by the two halfplanes passing through the line l and containing the points x and y respectively, and is traversed when the former halfplane rotates clockwise around l until it coincides with the latter one. From the definition of the dome, and the fact that the edge pq is internal to the dome of u, we conclude that there is a plane S passing through pq such that u lies in one of the open halfspaces defined by S while v and w lie either in the other one or on S. This precludes any of the dihedral angles $[u, pq, v]$, $[v, pq, w]$, and $[w, pq, u]$ from being larger than π. None of them can be equal to π either, however, as the adjacency of the apexes suggests.

Therefore, all three dihedral angles $[u, pq, v]$, $[v, pq, w]$, $[w, pq, u]$ are less than π. As a result, the union of the three tetrahedra defined by pq and any two of the apexes completely contains a small open ball centered at the midpoint of pq. Since the edge pq is a reflex edge of P, there is a point z outside P that lies in that ball and avoids

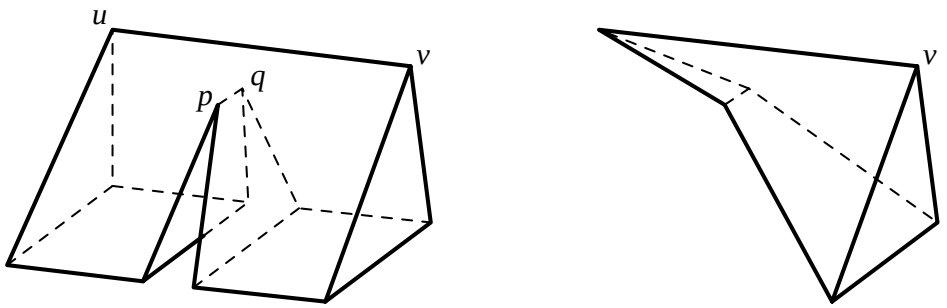


Figure 3.8: The internal edge pq has its endpoints on the crowns of u and v. On the right, the cup of v.

all three supporting planes of puq, pvq, and pwq. Then, the interior of one of the triangles uvz, vwz, wuz intersects pq. If this were uvz, then pq cannot possibly be internal to the dome of w, as the interior of uvz lies completely outside the cup of w.

Note that the statement of the lemma is tight, as shown in Figure 3.8. ∎

Summarizing the results of the previous lemmas, we derive:

Lemma 3.3.4. *The polyhedron P contains at most $2r$ pointed vertices whose domes are hindered.*

Proof: We recall again that vertices and edges of P that are internal to domes are contributed only by reflex or flat vertices and reflex edges or edges incident upon coplanar facets respectively. However, the cup of a pointed vertex v of P that is hindered remains so, if we eliminate from P all flat vertices and edges incident upon coplanar facets. The number of hindered cups is therefore the number of cups that reflex vertices and reflex edges of P alone hinder. Let us partition the reflex edges of P into three classes:

1: those with at least one endpoint being an internal vertex of some dome,

2: internal edges of a dome with both endpoints on the crown, and

3: all remaining reflex edges.

We prove by contradiction that classes 1 and 2 are disjoint. Assume that there exists a reflex edge e of P, internal to the domes of two pointed vertices u and v, such that

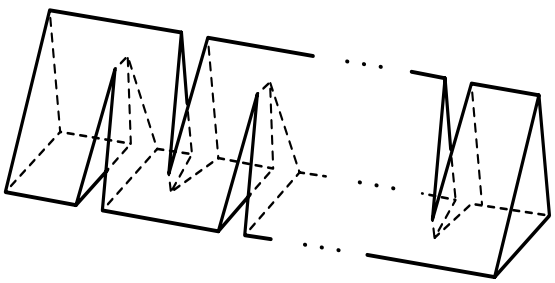


Figure 3.9

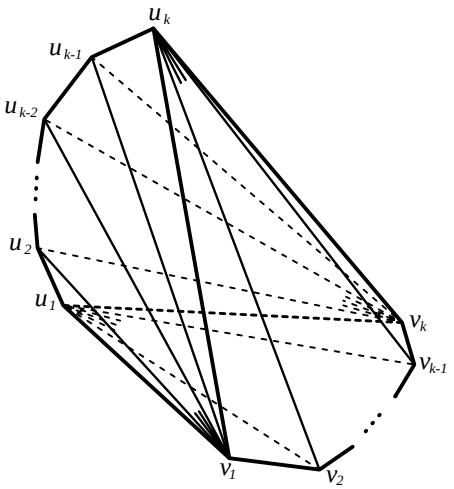


Figure 3.10

one of its endpoints, say p, lies on the crown of u and is internal to the dome of v. Then, there exists a small open half-ball centered at p that lies entirely in the cup of v, and hence in P. Since e is internal to the dome of u, it is not collinear with pu. With pu nonreflex, it follows that the unique plane defined by e and pu intersects the half-ball in a half-disk centered at p. The internal angle between e and pu is strictly less than π however, therefore the half-disk cannot lie entirely in P: a contradiction. We conclude that no vertex can be internal to the dome of a pointed vertex and on the crown of another one.

According to Lemma 3.3.2, an edge in class 1 may hinder at most three domes by contributing internal vertices through one given endpoint q (so the total might be as high as six). Note that if an endpoint of such an edge lies on the crown of some pointed vertex, then, according to the argument above, it cannot be internal to, and thus cannot hinder any dome. The endpoint q, however, will be incident upon at least two additional reflex edges of P in class 1 (Lemma 3.2.1). Therefore, the edges in class 1 can hinder at most $(2 \times 3)/3\ r_1$ domes, where r_1 is the cardinality of class 1. Additionally, Lemma 3.3.3 implies that each edge in class 2 may hinder at most two domes. The lemma follows readily. The stated bound is in fact achievable, taking into account the polyhedron shown in Figure 3.9. ∎

As a result, a polyhedron of n vertices (none of which is flat) and r reflex edges has at least $n - 4r$ pointed vertices whose cups are not hindered. These cups can be removed one by one, and be decomposed into tetrahedra, resulting in a polyhedron of at most $4r$ vertices and a collection of $\sum_i (d_i - 2)$ tetrahedra, where d_i is the degree of the apex of the i-th cup at the moment of its removal. We note that as a cup is a star-shaped polyhedron with the apex in its kernel, its tetrahedralization can be carried out very easily by triangulating the dome and reporting tetrahedra that are determined by the apex and each triangular facet of the dome. However, the order in which the cups are removed plays a crucial role in the size of the final tetrahedralization. Consider the convex polyhedron of $2k$ vertices, shown in Figure 3.10. Removing the cups of the vertices u_k, u_{k-1}, ..., u_2 in that order produces a total number of $(k-1)+(k-1)+\cdots+(k-1) = (k-1)^2$ tetrahedra. If, however, the cups of these same vertices are removed in the reverse order, we end up with $2+2+\cdots+2+(k-1) = 3k-5$ tetrahedra only. This observation and the fact that the degree of a pointed vertex v of P is an upper bound on the degree of the apex of $cup(v)$ if the boundary of P is triangulated suggest that we should try to remove the cups of the pointed vertices of small degree first. In fact, we can fix a constant upper bound d on the degree of the pointed vertices that we are going to remove, process all the vertices that qualify, and then reiterate on the resulting polyhedron as the removal may have decreased the degrees of some of the remaining vertices. Clearly, by restricting our attention to pointed vertices of degree bounded by a constant, we guarantee that the removal and tetrahedralization of their cups will result in a linear number of tetrahedra. The larger d is, the more vertices will be removed, and therefore, the more time the pull-off phase will take and the "skinnier" the polyhedron (that results after the unhindered cups of all pointed vertices of degree at most d have been exhaustively removed) will be.

By restricting our attention to polyhedra of zero genus, we can prove that the size of the polyhedron that results after the completion of the cup removal phase is proportional to the number of reflex edges of the initial polyhedron. This is derived from the fact (established in the following lemma) that the number of pointed vertices of a polyhedron P of zero genus which are of small degree and whose cups are unhindered

is at least a constant fraction of the total number of vertices of P provided that the number of reflex edges of P is "relatively" small.

Lemma 3.3.5. *Let $d \geq 6$ be a fixed integer, and $c < 1/4$ be a prespecified positive real. If no vertex of a polyhedron P of zero genus is flat, and $r \leq c(d-5)n/(d-1)$, then P contains at least $(1-4c)(d-5)n/(d-2)$ pointed vertices of degree at most equal to d whose cups are not hindered.*

Proof: Since a pointed vertex is incident upon nonreflex edges only, the sum of the degrees d_i of all pointed vertices is

$$\sum_i d_i \ \leq \ 2(m-r), \tag{1}$$

where m is the total number of edges of P. Let us denote by N and n' the number of pointed vertices of degree at most d, and the number of reflex vertices, respectively. We can then write

$$\sum_i d_i \ = \ \sum_{3 \leq d_i \leq d} d_i + \sum_{d_i > d} d_i \ \geq \ 3N + (n - n' - N)(d+1). \tag{2}$$

The combination of (1) and (2) yields $3N + (n-n'-N)(d+1) \ \leq \ 2(m-r)$, or

$$N \ \geq \ \frac{(d+1)(n-n') - 2(m-r)}{d-2}. \tag{3}$$

Since each reflex vertex is incident upon at least one reflex edge, we have $n' \leq 2r$, while Euler's formula for a polyhedron of zero genus provides us with the bound $m \leq 3n-6$. Substituting these bounds for n' and m in (3), we derive

$$N \ \geq \ \frac{(d+1)(n-2r) - 2(3n-6-r)}{d-2} \ = \ \frac{(d-5)n - 2dr + 12}{d-2}.$$

Among these vertices, at most $2r$ can have their cups hindered (Lemma 3.3.4). So, in order to guarantee the presence of pointed vertices with unhindered cups, it suffices that

$$\frac{d-5}{d-2}n - \frac{2d}{d-2}r + \frac{12}{d-2} \ > \ 2r.$$

	$d=6$	$d=8$	$d=10$	$d=12$	$d=15$	$d=20$
$c=1/7$	1/35	1/21	5/63	1/11	5/49	15/133
	3/28	3/14	15/56	3/10	30/91	5/14
$c=1/6$	1/30	1/14	5/54	7/66	5/42	5/38
	1/12	1/6	5/24	7/30	10/39	5/18
$c=1/5$	1/25	3/35	1/9	7/55	1/7	3/19
	1/20	1/10	1/8	7/50	2/13	1/6
$c=2/9$	2/45	2/21	10/81	14/99	10/63	10/57
	1/36	1/18	5/72	7/90	10/117	5/54
$c=3/13$	3/65	9/91	5/39	21/143	15/91	45/247
	1/52	1/26	5/104	7/130	10/169	5/78

Table 3.1: Values of $c(d-5)/(d-1)$ and $(1-4c)(d-5)/(d-2)$.

The number of such vertices will then be at least

$$\begin{aligned} \frac{d-5}{d-2}n - \frac{2d}{d-2}r + \frac{12}{d-2} - 2r &= \frac{(d-5)n - 4(d-1)r + 12}{d-2} \\ &\geq \frac{(1-4c)(d-5)n + 12}{d-2}. \end{aligned}$$ ∎

Lemma 3.3.5 implies that, if $r \leq c(d-5)n/(d-1)$, we can always find some vertices to remove, thus reducing the value of n. We can then apply the lemma on the resulting polyhedron, and so on until the above inequality does not hold any more. Consequently,

Corollary 3.3.1. *The exhaustive removal of the unhindered cups of the pointed vertices whose degree is no larger than some fixed constant d yields a polyhedron whose size is at most equal to a multiple of the number of reflex edges of the given polyhedron.*

In Table 3.1, we give the values of $c(d-5)/(d-1)$ and $(1-4c)(d-5)/(d-2)$ for a small sample of values of c and d. To clarify the meaning of these numbers, let us consider a concrete example. Let us select, for instance, c equal to $1/5$. The table indicates that if we pick d equal to 6, i.e., we intend to remove the cups of pointed vertices of degree at most 6, then if the ratio of reflex edges over vertices of the polyhedron is no more than $1/25$, at least one out of 20 vertices will qualify

for removal. If instead, d is chosen equal to 10, then for a ratio of reflex edges over vertices at most 1/9, the ratio of "qualified" vertices increases to 1 out of every 8. In general, for a fixed d, the closer c is chosen to 1/4, the larger the upper bound on the ratio r/n becomes, allowing the pull-off phase to be triggered for more polyhedra, and the smaller the ratio of "good" vertices becomes, cutting down on the number of vertices we will be able to pull off the current polyhedron. For a fixed c, the larger d becomes, both ratios increase, which is very desirable, but each cup is now bigger requiring more time, and producing more tetrahedra. The figures in Table 3.1, which in fact represent worst case bounds, persuade us that the algorithm can indeed be useful in practice.

3.4 The Tetrahedralization Algorithm

Given a polyhedron P of zero genus that has n vertices and r reflex edges, we show how to partition P into $O(n + r^2)$ tetrahedra. Taking into account the lower bound mentioned in Section 3.1, the number of tetrahedra produced is optimal in the worst case, up to within a constant factor. The tetrahedralization algorithm consists of two phases, figuratively termed *pull-off* and *fence-off*. In the pull-off phase, we reduce the size of the polyhedron to $O(r)$. The idea is to locate pointed vertices of small degree whose cups are not hindered, and to remove them by pulling off and tetrahedralizing their cups, and by replacing the boundary of P incident upon them by their domes. This shelling step reduces the vertex count, without increasing the number of reflex edges. Then, in the fence-off phase, we erect vertical fences through each edge of P, achieving a decomposition of P into cylindrical pieces that can be easily partitioned into tetrahedra. The algorithm runs in $O(n \log r + r^2 \log r)$ time and requires $O(n + r^2)$ space.

We assume that the polyhedron P is given to us in such a way that all the incidencies can be quickly retrieved. For this purpose, we can use any of the standard polyhedral representations given in the literature, e.g., winged-edge (Baumgart [7]), doubly-connected-edge-list (Muller and Preparata [55]), quad-edge (Guibas and Stolfi [35]). See Section 4.2 for descriptions of these three models.

As a preprocessing step, we normalize P, meaning that we remove all flat vertices (flat vertices do not add anything), and we triangulate its boundary. Not only does the boundary triangulation simplify any tests involving facets of the polyhedron, but it ensures that the degree of a pointed vertex v is an upper bound on the degree of v at its cup; as a result, unhindered cups of pointed vertices of constant degree will be of constant size. Furthermore, the boundary triangulation implies that the cylindrical pieces that are produced if vertical fences are erected through each and every edge of the polyhedron (as in the fence-off phase) are of zero genus, making them easier to partition into tetrahedra. The boundary of a polyhedron can be triangulated using, say, Hertel and Mehlhorn's triangulation algorithm ([36]). To do so, we triangulate each facet by sweeping a line L across its supporting plane, stopping and doing some extra processing at vertices whose incident edges both lie in the same halfplane with respect to L. For a facet of n_i vertices, r_i of which exhibit reflex angles, the procedure takes $O(n_i + r_i \log r_i)$, amounting to a total of $O(n + r \log r)$ for the entire polyhedron. The flat vertices can then be identified and removed in time linear in the size of the polyhedron. Summarizing, the normalization step takes $O(n + r \log r)$ time, and although it may increase the number of nonreflex edges, it decreases n by the number of flat vertices and does not affect r at all.

3.4.1 The Pull-Off Phase

The goal in the pull-off phase is to reduce the number of vertices of the given polyhedron P to the same order as the number of its reflex edges, while producing a linear in the size of P number of tetrahedra. Since our strategy will be to remove the cups of a number of pointed vertices that are not hindered, we need to be able to tell whether the cup of a pointed vertex is hindered or not. Let us consider a pointed vertex v of P. The boundary of the convex hull of v's crown consists of two polyhedral patches separated by the crown itself, one of which isolates the other from v. Let K_v be the polyhedron bounded by the patch in question and the cone of v (compare Figure 3.11 to Figure 3.3). The cup of v will be unhindered if and only if every reflex vertex and every reflex edge of P lies either outside K_v or on the crown of v. Furthermore, if the cup of v is unhindered, v's cup is precisely K_v.

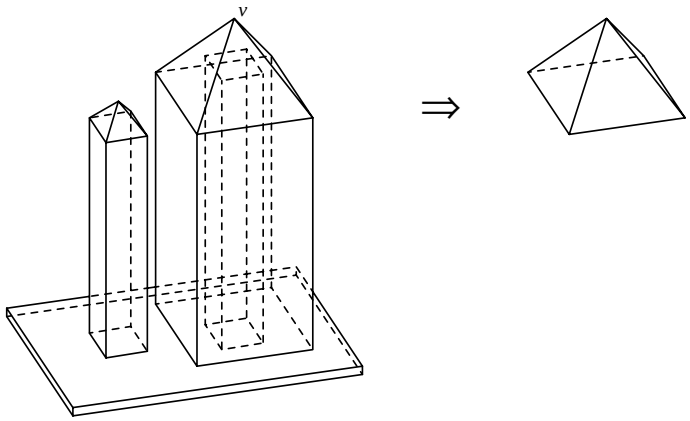


Figure 3.11: A polyhedron and the polyhedron K_v of its vertex v.

So, in order to decide whether the cup of a pointed vertex v is hindered or not, we have to check K_v against both the reflex vertices and the reflex edges of P. We can unify the two cases by replacing each reflex edge by a dummy reflex vertex at its midpoint. The reflex vertices and the midpoints of the reflex edges are collectively called *puncturing* vertices, and their number is no more than $3r$. Note that these vertices alone can determine the pointed vertices whose cups are hindered, since the dome of every hindered cup contains at least one puncturing vertex that does not lie on the crown. At the outset of the pull-off phase, we assume that every puncturing vertex is potentially hindering a cup. Local tests on the incidence structures of the vertices might allow us to weed out many candidates. Although this might be a practical step to take, it is not necessary strictly speaking. In light of the definition of puncturing vertices, the test for unhindered cups is stated as follows: first, K_v is computed; then, if no puncturing vertex lies inside K_v or on K_v's boundary that is not incident upon v, the cup of v is not hindered; in this case, the cup of v is precisely K_v.

To achieve the desired reduction in the size of P, we will have to remove $\Omega(n)$ unhindered cups. Verifying that each of them is unhindered or equivalently checking the corresponding K_i against each of the puncturing vertices, however, amounts to an undesirable $\Omega(nr)$ time complexity for this phase. To improve on this, we assume

for the moment that an oracle, the *Witness Oracle*, provides us with a set W of facets of P, after ∂P has been triangulated. The members of W are referred to as *witness* facets, and have the following properties:

(i) the set of vertices incident upon all witness facets is a superset of the pointed vertices that have hindered cups, and

(ii) a puncturing vertex potentially hinders the cups of vertices of at most four witness facets, which implies that the cardinality of W is $O(r)$.

Furthermore, each witness facet f is associated with a subset of puncturing vertices that potentially hinder the cups of f's vertices. The details of the Witness Oracle are given in the next subsection.

The pull-off phase is an iterative process. In the general step, we are working on a polyhedron P_i ($i \geq 1$), where P_1 is the given polyhedron P. Unless the number of vertices of P_i is not "much" larger than the number of its reflex edges (see Lemma 3.3.5), we remove from P_i a number of pointed vertices whose cups are not hindered, producing P_{i+1}, the next polyhedron to work on. The removal is carried out as follows: first, we go through all the pointed vertices of P_i, looking for those of degree below some fixed constant d, and we insert them in a queue of *favorable* vertices; once all vertices have been processed, we repeat the following steps until the queue is empty.

1. Let v be the favorable vertex referenced by the top of the queue. If v is marked to be skipped, we remove v from the queue, and we reiterate. Otherwise, we go through the facets of P_i that are incident upon v, and determine v's crown, without, however, modifying the boundary of P_i. Then, we compute the convex hull of the crown, and from that the polyhedron K_v, as described in the opening paragraph of this subsection.

2. If v is incident upon a witness facet f, we check whether the puncturing vertices associated with f lie outside K_v or on the part of K_v's boundary that is incident upon v, thus determining whether the cup of v is hindered or not. Since the cup of v is of constant size, this can be done in time proportional to the number of puncturing vertices probed. If the cup of v is found hindered, we remove v from the queue, and proceed with step 1.

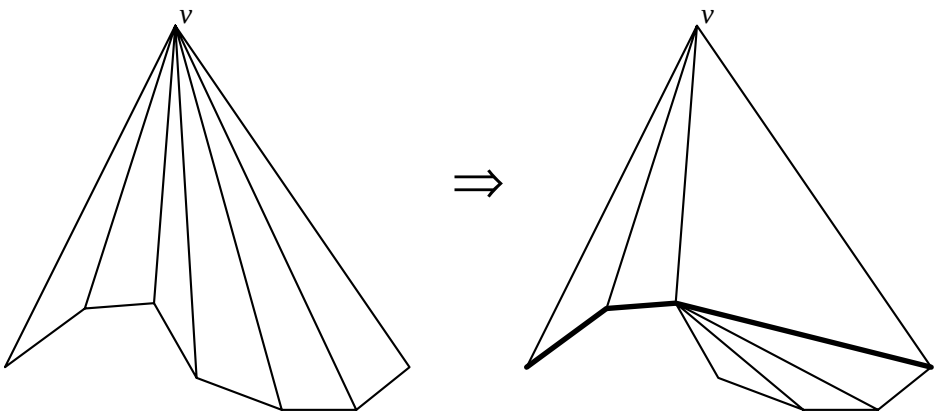


Figure 3.12: Retriangulating six coplanar facets incident upon v during the computation of v's crown.

3. Now, we are ready to extract and tetrahedralize the cup of v, which is in fact K_v; its size is constant, as the degree of v is no more than the fixed constant d. First, the boundary of P_i is retriangulated, by replacing the edge (v, w) by a new edge (u, z) whenever two adjacent coplanar triangular facets (u, v, w) and (w, v, z) incident upon v form a convex quadrilateral (u, v, z, w). The resulting polyhedral patch formed by the facets of P_i incident upon v is precisely $(\partial cup(v)) \setminus dome(v)$ (Figure 3.12). (Note that the retriangulation involves only the facets of P_i that are incident upon v.) This process may affect the degrees of the vertices that were adjacent to v before the crown computation. It is interesting to note, however, that the degrees of the vertices that do not become crown vertices will either decrease or, at worst, remain the same (Figure 3.12). In addition to that, every time that an edge is flipped, a pair of facets of P_i is substituted by a new pair. In case a witness facet f is replaced in this substitution, the information in W must be updated, and the puncturing vertices associated with f must be appropriately distributed among the facets in the new pair.

   After that, the cup of v can be pulled off by removing the facets of P_i that are incident upon v, and substituting them with facets of the dome of v. The dome of v is easily retrieved from K_v. Again, if any of the removed facets, say f, belongs to W, the new witness facets that result from f's removal need to be determined

(these are the facets of the dome of v that lie vertically above or below f), and lists of puncturing vertices associated with them need to be set up. Finally, the cup of v, or equivalently K_v, is tetrahedralized. The decomposition consists of the tetrahedra determined by v and each nonincident facet of K_v's triangulated boundary.

4. We remove v from the queue, and mark the vertices on its crown, so that they are skipped if met later in the queue.

Since each puncturing vertex is associated with no more than four witness facets, it will not be probed more than a constant number of times during a single pass through all favorable vertices. As a result, such a pass over a polyhedron of n_i vertices will take a total of $O(dn_i + r)$ time. From Lemma 3.3.5 and the fact that at most $d+1$ favorable vertices are removed from the queue or marked each time step 4 is executed, we derive that this process removes at least

$$\frac{1}{d+1} \times \frac{(1-4c)(d-5)}{(d-2)} n_i \geq \alpha n_i$$

vertices from the polyhedron, where α is a fixed positive constant less than 1. Thus, we are left with a polyhedron of $n_{i+1} \leq (1-\alpha)n_i$ vertices and at most r reflex edges. Repeating this pruning pass until $n_i = O(r)$ takes time proportional to $\sum_i (n_i + r) = O(n + r \log n)$. Note that as the cups of the vertices are being pulled off, some reflex vertices may become pointed. Therefore, the set of puncturing vertices, and consequently the set of witness facets, may need updating. We choose, however, to make no updates at all, possibly ending up with some unnecessary work being done. In any case, the stated time complexity is not affected.

Summarizing, the total time required by the pull-off phase, the witness oracle included, is $O((n + r^2) \log r)$, taking into account that the time complexity of the oracle is $O(n \log r + r^2 \log r)$ and that $r \log n = O((n + r^2) \log r)$ [1]. The total space needed amounts to $O(n + r^2)$. Finally, since each pulled-off cup produces at most $d-2$ new tetrahedra, by the end of this phase, P is decomposed into a collection of $O(n)$ tetrahedra and a polyhedron of $O(r)$ vertices.

[1] The function $h(x) = x/\log x$ is strictly increasing for $x > e \approx 2.718$. Thus, $h(r) < h(n + r^2)$, which directly implies that $r \log n < (n + r^2) \log r$.

3.4.2 The Witness Oracle

For a given polyhedron P of n vertices and r reflex edges whose boundary is triangulated, the oracle comes up with a set of $O(r)$ facets of P, whose vertices form a superset of the pointed vertices that have hindered cups. The following lemma, which takes advantage of the geometry of the cup, provides us with the idea to carry out this computation.

Lemma 3.4.1. *Let v be a pointed vertex of a polyhedron P, and p any point of $dome(v) \setminus crown(v)$. Then, for every line passing through p, at least one of the two rays from p must intersect the closure of a facet of the triangulated boundary of the polyhedron incident upon v. Moreover, the line segment delimited by p and the point of intersection lies entirely in P.*

Proof: By definition, the dome of v is a concave polyhedral patch with respect to v's cup. Therefore, either the line lies on the dome (in this case, the dome consists of coplanar facets) and thus intersects the crown, or at least one of the two rays from p intersects the interior of the cup. In either case, the line intersects the closure of one of the facets of the cup incident upon v, and the line segment with endpoints the point p and the point of intersection with the facet lies in the cup of v, and consequently in P. The lemma follows from the observation that any point of the closure of a facet of a cup incident upon the apex v belongs to the closure of a facet of P incident upon v. ∎

The lemma implies that a set of facets of P, whose vertices form a superset of the pointed vertices that have hindered cups, can be computed, if, for each puncturing vertex p, we compile a list of the facets of P whose closures are intersected by the vertical line through p, and, among them, we retain only those for which the line segment delimited by p and the point of intersection lies entirely in P. For convenience, we assume that no vertex of the polyhedron lies vertically above or below any of the puncturing vertices. Note that this assumption can be checked in $O(n \log r)$ time and can be relaxed with little extra effort. This assumption narrows down the number of facets intersected by a vertical line through a puncturing vertex to at most two per boundary crossing; as a result, the number of facets in the final list will not

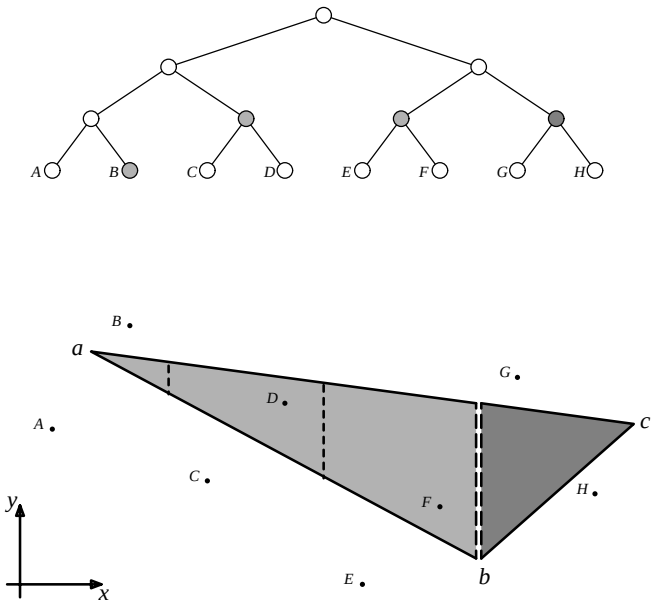


Figure 3.13

exceed $(2+2) \times 3r = 12r$. So, our task is to determine the (triangular) facets of P whose closures are traversed by a vertical line passing through a puncturing vertex. Selecting among these facets those that are closest to the puncturing vertex without passing through it involves a constant time test for each such facet.

If we take projections on the xy-plane, we can restate the problem in terms of the facets of P as follows: Which projections of puncturing vertices (on the xy-plane) lie in a given triangle? To answer such a question, we set up a complete binary tree whose leaves are in one-to-one correspondence, from left to right, with the puncturing vertices of P, ordered by nondecreasing x-coordinates. With each internal node t of the tree, we associate the canonical strip $\{(x,y) \,|\, x_{min} \leq x \leq x_{max}\}$, where x_{min} and x_{max} are respectively the smallest and largest x-coordinates of the vertices associated with the leaves descending from t. Let abc be the projection of a triangular facet f of P on the xy-plane, with $a \leq b \leq c$ in x-order. Following standard segment tree partition (Preparata and Shamos [58]), and using the x-order of the tree leaves, we partition the x-extents of ab and bc into $O(\log r)$ intervals, which in turn induces a partition of the triangle abc into $O(\log r)$ trapezoids, every one of which is associated

with a distinct node of the tree and spans the width of this node's canonical strip. For the x-extent of ab, for example, we traverse the tree starting from its root according to the following rules. If the x-extent of ab spans the entire width of the canonical strip H associated with the current tree node, then a trapezoid of abc (the intersection of H and abc) is associated with the current tree node and the descent from this node is stopped. The descent is also stopped in the case where the x-extent of ab does not intersect H. Otherwise, we proceed both to the left and right children of the tree node. (Figure 3.13 shows a set of eight points and the partition of a triangle into trapezoids.) Since we deal with continuous intervals on the x-axis, it is not difficult to see that this tree traversal will visit at most four tree nodes per level, no more than two of which may finally be associated with trapezoids of the triangle abc. Then, for each such node t, we must determine which points among those stored in t's descending leaves lie inside the corresponding trapezoid. The advantage of having trapezoids that span the entire width of the associated canonical strips is that, for each such computation, we can substitute the trapezoid by the double wedge that is determined by the nonparallel sides of the trapezoid and covers it. In fact, although we have many different trapezoids, each one of them can be replaced by one of two fixed double wedges, determined by the supporting lines of ab and ac, and bc and ac respectively.

Let us dualize the problem using the following asymmetric transformation: a point (a, b) is mapped to the line $y = ax + b$; a line $y = kx + d$ is mapped to the point $(-k, d)$. Note that the transformation maps vertical lines to the point at infinity, and this is what makes it attractive for our purposes. The reason is that no vertical line lies entirely in any of the double wedges with which we are dealing, and therefore, the chosen transformation maps such wedges to finite length line segments. (A different transformation might map such a double wedge to a pair of nonintersecting collinear rays.) In turn, the application of the duality transformation results in each node of the tree being associated with a certain arrangement of lines (the duals of the projections of the vertices stored at the leaves below the node in question). Building these arrangements can be achieved in $O(r^2)$ time and space, using the techniques of Chazelle et al. ([14]), or Edelsbrunner et al. ([24]).

In dual space, the problem of finding which members of the point-set that is stored in a tree node lie in a double wedge is the same as computing which lines of the line arrangement associated with this tree node intersect a line segment. This requires the ability to locate a point in the line arrangement fast. So, we start by processing the root arrangement, that is the arrangement of the dual images of all $O(r)$ puncturing vertices, to support $O(\log r)$-time point location (Kirkpatrick [42], Edelsbrunner et al. [23]). This takes only a linear amount of work through the arrangement. There is no need, however, to do the same processing to the remaining nodes of the tree, as we can take advantage of the fact that the arrangement of a node is a portion of the arrangement of its father. More specifically, each region of a father's arrangement lies entirely within one region of either child's. Thus, we only need to provide each region of a node's arrangement with two pointers, one directed toward the enclosing region for each of its children. Setting up all these pointers can be done in $O(r^2)$ time by ensuring that the work at a given node is at most proportional to the square of the number of its descending leaves. In the general step, we have at our disposal the full arrangement at a given node, and we color the lines green or red, depending on whether it is the left or right child that inherits them. To compute, say, the green arrangement with the pointers directed to it, we begin by merging collinear green edges into edges of the green arrangement. By traversing the full arrangement, we can now collect each region lying within a given region of the green arrangement. The same process applied to the red edges completes our work. This pointer scheme allows us to locate a point in the line arrangement associated with any node of the tree in logarithmic overall time.

Let us return to the problem of finding which lines of the line arrangement associated with a tree node t intersect a line segment. First, we need to locate the endpoints p and q of the line segment in the line arrangement. To do that, we locate p and q in the root arrangement, and then using region-to-region pointers, we go down the tree stopping at the node t. The point location and the tree traversal take only $O(\log r)$ time. If p and q lie in the same region, no further work need be done, and the answer is the empty set; otherwise, we pursue our search in the two children of t taking into account that the set of lines intersecting the line segment is the (disjoint) union of

the sets of lines with that property in t's children. This process eventually takes us to a certain subset of the leaves, where the desired intersections can be found directly. If k intersections are finally reported for the node t, the descent to the leaves takes no more than $(k+2)\log r$ time. Since a vertical line can cut only $O(r)$ facets, the total amount of time spent computing intersections between lines through puncturing vertices and facets of P is no more than $O(n \log r + r^2 \log r)$.

As mentioned earlier, of all the intersections between a vertical line through a given puncturing vertex p and the facets of P, only those for which the line segment connecting p to the point of intersection on the facet lies in P need be retained for further consideration in light of Lemma 3.4.1, amounting to at most four facets per puncturing vertex. Furthermore, during the selection process, pointers to the corresponding puncturing vertices are included in the records of the facets that are retained. As a result, in the end, the record of each such facet contains a list of the puncturing vertices that potentially hinder the cups of its vertices. From a time complexity standpoint, the Witness Oracle accounts for $O((n + r^2)\log r)$ in the running time of the algorithm. The space needed amounts to $O(r^2)$.

3.4.3 The Fence-Off Phase

Our goal here is to tetrahedralize a polyhedron of n vertices into $O(n^2)$ tetrahedra, the method being satisfactory only if at least a fixed fraction of the edges are reflex. The tetrahedralization is carried out by partitioning P into cylindrical pieces, and then by tetrahedralizing each piece separately. To build the cylindrical partition we attach vertical fences to each edge, reflex and nonreflex, one at a time. Let us say that a point p is *visible* from an edge e if it can be connected to a point of e by a vertical segment lying in P. The set of points visible from e is easily seen to be a monotone polygon: it is called the *fence* of the edge e and is to be attached to it (Figure 3.14). Our fences are similar to the *walls* used in the *slicing theorem* of Aronov and Sharir ([2]): the difference is that while fences project vertically onto their attaching edges, walls flood all over the free portion of the vertical plane passing through the edge.

Although a fence may have exposed edges "sticking out", erecting all the fences

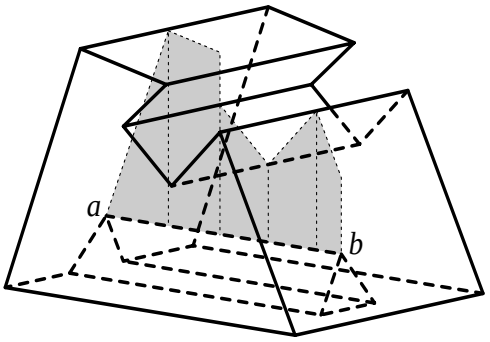


Figure 3.14: The fence of ab.

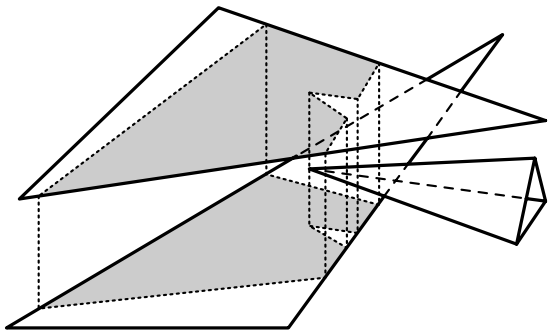


Figure 3.15: A nonconvex cylindrical piece.

partitions P into cylindrical pieces that are polyhedra. Each such piece can be defined by (i) specifying a horizontal base polygon, (ii) lifting it vertically into an infinite cylinder, and (iii) clipping the cylinder between two planes (which do not intersect inside the cylinder). Unfortunately, the cylindrical pieces do not always form a convex decomposition of P. Of course, every reflex edge of P is "resolved" in the sense that, because of its fence, the angles between the facets incident upon the sub-edges into which the reflex edge is split in the decomposition cease to be reflex. The problem is that new reflex edges might be created between two fences (Figure 3.15). Let us examine this phenomenon in some detail. Let e be a vertical edge of a fence. Since the edge e is incident upon at least one vertex of P, and ∂P is triangulated, the edge e cannot be left exposed after the fencing. It is conceivable, however, that e coincides with an edge of another fence which results in a reflex edge. What we can say at this point is that the fences partition P into cylindrical pieces, which are free of nonvertical reflex edges. But then, a triangulation of the base polygons of each cylindrical piece refines the partition into one consisting of cylindrical pieces whose base polygons are triangles. A decomposition of P into tetrahedra follows trivially.

The fence-off phase begins by computing the fence of each edge of P. Although, the edge may have been cut into a number of sub-edges due to previously introduced fences, we shall still treat each edge as one entity, and deal with all its sub-edges in one fell swoop. We give a very simple, albeit slightly inefficient, method for computing the fence of an edge e of P. Let Σ be the set of segments obtained by computing the

intersection of the facets of P with the projection of e on their supporting planes. In general, Σ consists of a number of disjoint polygonal boundaries. Then, the fence of e is the trapezoidal map induced by the visibility relation between e and the segments of Σ. The map consists of trapezoids that are bounded by e, the segments in Σ, and vertical segments through endpoints that connect points of e to their visible segments in Σ (Figure 3.16). The fence may have an upper, lower, or both an upper and a lower part, depending on whether there exist visibility trapezoids vertically above, below, or both above and below respectively. Note that the fence of e may degenerate to e if the portion of the three-dimensional space locally above and below e lies outside P. Since each trapezoid is bounded by a vertical segment through an endpoint that corresponds to an edge of P, and there are $O(n)$ edges, the size of the trapezoidal map is $O(n)$. The fence is computed by sorting the segments in Σ, and by sweeping them in order, to determine the visibility polygons. The entire computation for each edge can be carried out in $O(n \log n)$ time.

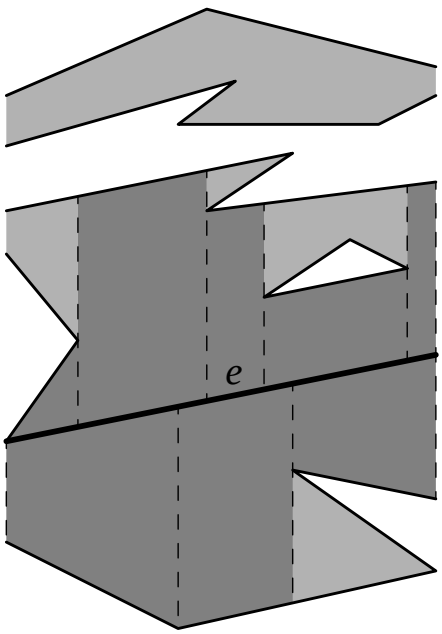


Figure 3.16: Computing the fence of e.

We do not, however, need to store each fence explicitly. Instead, we include each nonvertical bounding segment s of each fence in a list associated with the facet whose intersection with the fence is precisely s. After all fences have been computed, and their bounding segments have been attached to the appropriate facet records, each facet of P is scanned in turn, and its constrained triangulation is computed (the constraints are the edges incident upon the facet and the fence segments associated with it). The triangles reported are inserted in a global list of triangles, which, after all

facets have been processed, consists of pairs of triangles such that (i) the projections of the members of a pair on the xy-plane are identical, and (ii) the cylindrical polyhedron that is determined and bounded by such a pair lies in P. Note that the collection of these cylindrical pieces is nothing but the refined decomposition of P into cylindrical pieces that we mentioned earlier. The pairs of triangles will not necessarily be in order in the list. This can be achieved, however, by sorting the triangles in the list with respect to the x- and y-coordinates of their vertices (this will bring together triangles with the same projection on the xy-plane), and, in case of ties, with respect to the z-coordinate of their centroids, which will do the final matching. Each cylindrical piece determined by such a pair of triangles can be trivially decomposed into at most three tetrahedra.

The introduction of fences results in a partition of P that involves a total of $O(n^2)$ vertices, edges, and facets. The triangulation of the facets and the extraction of tetrahedra merely adds a constant multiplicative factor to the size of the decomposition. Therefore, the description size of the final partition is $O(n^2)$, and consequently $O(n^2)$ tetrahedra are produced. As mentioned earlier, the computation of all the fences takes $O(n^2 \log n)$ time. The triangulation of a facet that is associated with k_i constraining segments can be carried out in $O(1+k_i \log k_i)$ time, so that triangulating all the facets will take a total of $O(n^2 \log n)$ time, since $\sum_i k_i = O(n^2)$. Finally, sorting the triangles takes another $O(n^2 \log n)$ time, while processing the pairs does not take more than $O(n^2)$. Summarizing, the entire fence-off phase requires $O(n^2 \log n)$ time and $O(n^2)$ space.

3.5 Putting the Pieces Together

Given a polyhedron of zero genus with n vertices and r reflex edges, we start by (i) triangulating the boundary, and (ii) removing all flat vertices, while doing the obvious clean-up. Then, in case n greatly exceeds r, we apply the pull-off phase. We finish the decomposition by going through the fence-off phase. The running time of the algorithm is $O(n \log r + r^2 \log r)$. We summarize our results below.

Theorem 3.1. *In $O((n + r^2) \log r)$ time it is possible to partition a polyhedron of zero genus with n vertices and r reflex edges into $O(n + r^2)$ tetrahedra. The time bound includes the cost of producing a full-fledged tetrahedralization with an explicit description of its facial structure. Up to within a constant factor, the number of tetrahedra produced by the algorithm is optimal in the worst case.*

Of independent interest is the Lemma 3.3.4, which implies that the size of a polyhedron of r reflex edges can be reduced to $4r$ without its genus being altered. This, for instance, can help us to translate bounds on the genus of a polyhedron in terms of the number of its vertices into bounds in terms of the number of its reflex edges. The same idea can be applied to other parameters of the polyhedron as well, as long as they are not affected by the cup removal process.

Although the pieces of the partition fit nicely to assemble the given polyhedron, the partition will not necessarily be a cell complex. Assuming that collinear edges and coplanar facets are not merged during the constrained triangulation of a shattered facet, and that the vertical segments that bound trapezoids in the upper and lower parts of a fence are extended to the lower and upper parts respectively, the only possibility where a violation of the above property would occur is when dome facets that contribute facets to tetrahedra during the cup tetrahedralizations are shattered and subsequently triangulated during the fence-off phase. A minor change in our algorithm can alleviate this problem. Namely, if such a case occurs, the corresponding cup tetrahedron is partitioned into tetrahedra induced by the retriangulation of the facet. This change does not affect the time or space complexity of our algorithm but produces a cell complex. Moreover, we recall that the algorithm introduces Steiner points, that is, points that become vertices of the pieces in the partition. The problem of partitioning a polyhedron into tetrahedra without using Steiner points is NP-hard ([60]).

Finally, we point out that it is often desirable in practice to avoid long, skinny tetrahedra. (See the paper by Baker et al. ([6]) for similar concerns in two dimensions.) Our algorithm unfortunately does not employ a qualitative criterion that would ensure such a "good" tetrahedralization.

Chapter 4

Implementing the Tetrahedralization Algorithm

There is no doubt that finding an efficient algorithm to solve a problem is a difficult task; implementing it, however, poses some challenging questions too. First is the question of coding the steps of an algorithm. There are always more than one ways to code up a procedure. The best one does not have to be the first that comes to mind, and finding it may require a good deal of time. A second question, almost always intertwined with the previous one, involves the selection of the data structures to represent the entities that appear in the algorithm. In most cases, efficient implementations greatly rely on the careful selection of these data structures, as inappropriate representations usually result in complicated code, or waste of space. Last but not least, dealing with finite precision arithmetic imposes its own difficulties. Using a computer implies that computations do not necessarily produce exact results, a problem that the theoretical description of an algorithm does not consider. The problem is admittedly of great difficulty and critical importance.

In this chapter, we present details of the implementation of the triangulation algorithm discussed in the previous chapter. The algorithm has the advantage of not requiring intricate data structures; the most complicated one is a balanced binary tree needed in the facet-shattering process. Although coding insertion, deletion and location routines for such a tree is much more complicated than the corresponding

routines for an array or a list, some variants of balanced binary trees, red-black trees in particular, are becoming increasingly popular in applications.

Of course, not every single line of our implementation can be presented and discussed. Instead, we concentrate on the description of the data structures used, and include brief parts of the actual program to show how specific tasks are carried out and how the information stored in the data structures is exploited. To simplify the exposition, diagnostic messages and comments in these parts have been suppressed. The entire C program is attached in the Appendix.

4.1 Robustness

The theoretical description of an algorithm gives the liberty to the researchers to assume that all computations can be carried out with infinite precision and the results of all mathematical operations are exact. Unfortunately, this is not true in practice, as the word size of the computer on which the implementation of an algorithm will run imposes an unsurmountable obstacle. As a result, the issue of how the application can be made to cope with the shortcomings of finite precision arithmetic is indeed critical in practical applications. Algorithms that take finite precision into account, and manage to produce results consistent with the geometry of the input instance, or in some cases, of a slightly perturbed instance of the input are called *robust*.

The need for robust computations in practice has attached particular importance to the issue and spawned a lot of effort towards finding ways to avoid inconsistencies due to imprecise arithmetic and/or imprecise data. The amount of invested work is too big to review here. Early general-purpose techniques were borrowed from mathematics or engineering: symbolic computation, rational arithmetic, interval arithmetic ([53], [39], [54]). More recently, another general-purpose technique, the *Epsilon geometry*, was introduced ([34]), while solutions tailored to specific problems, involving mainly convex hull and line arrangement computations, were also suggested ([5], [27], [37], [52], [46]).

With the exception of symbolic computation and rational arithmetic, the remaining methods all build around tolerance margins associated with the objects on which

the algorithm at hand operates; they use and appropriately update them as these objects participate in numerical computations. We use this idea in the implementation of the polyhedron triangulation algorithm, although the error margin that we use remains fixed throughout the process. Namely, if the Euclidean distance between two faces of the polyhedron is less than some fixed positive ϵ, they are either collapsed (if they are both of the same dimensionality), or they are considered touching otherwise. The quantity ϵ provides the necessary margin to overcome round-off errors. Informally speaking, the idea seems natural, in the sense that if the distance of the two faces is so small that one cannot tell them apart, one might as well consider them as a single object or incident upon each other. To enforce this scheme, all comparisons are carried out through the C macro `E_COMPARE`, which is equal to 0 if the compared quantities fall within `EPSILON` from each other, and -1 and 1 if the first quantity is less than `EPSILON` less than the second, and larger than `EPSILON` larger than the second respectively. The macro is shown along with other useful macros in Figure 4.1. The value of `EPSILON` that we used in the implementation was 10^{-6} (see file `defs.h` in the Appendix).

Although a tolerance margin is adequate for an individual comparison, cascaded operations may easily produce inconsistencies. We distinguish between two cases. First, consider, for instance, the test that determines whether a vertex is flat. The test involves moving along the cycle of facets around the vertex and deciding whether they are coplanar. It may be the case that any two such adjacent facets are "coplanar within the tolerance margin," whereas this may not be true for all pairs of such facets. Obviously, testing only adjacent facets will produce an incorrect decision, while testing every pair of facets may be computationally expensive. An easy shortcut is to keep comparing against the same facet, which is thus selected as the representative of all these "coplanar" facets. We apply this idea throughout the implementation trying to minimize the effect of cascaded errors. Second, if in a chain of computations results of previous computations are used to produce new results, a small error in the first computation may increase rapidly resulting into inconsistences or erroneous decisions. An interesting feature of our algorithm is that no more than six cascaded

```
#define E_COMPARE(x, y) \
    ( ((x) > (y) + EPSILON) - ((x) < (y) - EPSILON) )

#define MY_ALLOC(ptr, str, n) \
    ( (((ptr) = (str *) malloc((n) * sizeof(str))) == NULL) ? \
                        ERROR("...out of memory") :  0 )

#define MY_CALLOC(ptr, str, n) \
    ( (((ptr) = (str *) calloc((n), sizeof(str))) == NULL) ? \
                        ERROR("...out of memory") :  0 )

#define MY_REALLOC(ptr, str, n) \
    ( (((ptr) = (str *)realloc(ptr, (n) * sizeof(str))) == NULL) ? \
                        ERROR("...out of memory") :  0 )

#define MAKE_POINT_2(p, xc, yc) \
    ( (p).x = (xc), (p).y = (yc) )

#define COPY_POINT_2(p, q) \
    ( (p).x = (q).x, (p).y = (q).y )

#define MAKE_POINT_3(p, xc, yc, zc) \
    ( (p).x = (xc), (p).y = (yc), (p).z = (zc) )

#define COPY_POINT_3(p, q) \
    ( (p).x = (q).x, (p).y = (q).y, (p).z = (q).z )

#define MAKE_MIDPOINT_3(p, a, b) \
    ( (p).x = ((a).x + (b).x) / 2.0, \
      (p).y = ((a).y + (b).y) / 2.0, \
      (p).z = ((a).z + (b).z) / 2.0 )

#define MOD(n, base) \
    ( ((n + base) % base) )

#define ERROR(string) \
    ( printf("%s.\n", (string)), exit(1) )
```

Figure 4.1: Useful Macros (cont'd)

```
#define EVAL_2(p, q) \
    ( (p).x * (q).a + (p).y * (q).b + (q).c )

#define EVAL_3(p, q) \
    ( (p).x * (q).a + (p).y * (q).b + (p).z * (q).c + (q).d )

#define FROM_x_COMP_z(xval, q1, q2) \
    ( (E_COMPARE(xval, q1.x) == 0) ?  (q1.z) :  \
        (E_COMPARE(xval, q2.x) == 0) ?  (q2.z) :  \
            (q1.z + (xval - q1.x) * (q2.z - q1.z) / (q2.x - q1.x)) )

#define FROM_y_COMP_z(yval, q1, q2) \
    ( (E_COMPARE(yval, q1.y) == 0) ?  (q1.z) :  \
        (E_COMPARE(yval, q2.y) == 0) ?  (q2.z) :  \
            (q1.z + (yval - q1.y) * (q2.z - q1.z) / (q2.y - q1.y)) )

#define FROM_FACET_COMP_z(p, f) \
    ( -((f).a * (p).x + (f).b * (p).y + (f).d) / (f).c )

/*** general-purpose table handling ***/
#define MAKE_TABLE(table, str, size) \
    {   str *element; \
        MY_CALLOC(element, str, (table)->max = (size)); \
        (table)->n = 0; \
        (table)->to_elem = (char *)element; \
    }

#define INS_TABLE(entry, str, table) \
    {   str *element; \
        element = (str *)(table)->to_elem; \
        if ((table)->n == (table)->max){ \
            (table)->max *= 2; \
            MY_REALLOC(element, str, (table)->max); \
            (table)->to_elem = (char *)element; \
        } \
        element[(table)->n++] = (entry); \
    }
```

Figure 4.1: Useful Macros

computations are ever performed. Indeed, the computations performed in the pull-off phase of our algorithm involve determining the relative position of a vertex with respect to a facet, which essentially reduces to comparing the inner product of the vertex coordinates with the computed facet normal against the computed offset. This involves only two cascaded computations. In the fence-off phase, however, computing the intersection of a (triangular) facet with the vertical projection of an edge onto the facet may result into as many as three cascaded computations. The intersection operation is performed with respect to the x- and y-coordinates of the line segments and edges involved, so a fourth computation is needed to yield the z-coordinates of the endpoints of the intersection. The line segments that are in this way produced are then fed to the facet shattering process, where a comparison function determines the order in which the segments will be processed. Processing may require the comparison of the slopes of the segments, adding the sixth and last cascaded step.

4.2 Polyhedron Representation

The polyhedron triangulation algorithm requires that the incidences between any two types of faces of the polyhedron, except for vertex-to-edge incidences, can be retrieved from the data structure. Indeed, triangulating the boundary of the polyhedron requires that one is able to find the edges that bound any given facet. Moreover, determining whether an edge is reflex requires that the two facets incident upon an edge can be located. Computing the cup of a vertex assumes the ability to walk along the cycle of facets around the vertex, which necessitates that one is able first to move from a vertex to at least one of its incident facets, and second to find the vertices of a facet. Finally, computing the fence of an edge implies the computation of intersections between the edge and the facets of the polyhedron; although storing the coordinates of the endpoints of each edge in the edge record will do, including pointers from the edge to its incident vertices avoids duplicate storage of information.

In general, requiring that the incidences between any two types of faces of a three-dimensional object can be easily extracted from its representation is not uncommon in implementations of algorithms dealing with such objects. That is why the standard

polyhedral representations described in the literature were designed so that they can cope with any such requests. Three such typical representations are: the *winged-edge* model suggested by Baumgart ([7]), the *doubly connected edge list* by Muller and Preparata ([55]), and the *quad-edge* structure by Guibas and Stolfi ([35]). All three representations are equivalent, in the sense that the same amount of information can be extracted from each of them, and any one can be derived from any other in time linear in the overall number of faces.

The winged-edge model, the first of the three chronologically, stores in the record of an edge e pointers to e's incident vertices and facets, and four more pointers to the edges that precede and follow e on the cycle of edges bounding e's incident facets. In turn, the record of a vertex or a facet simply points to one of the vertex's or facet's incident edges. Moreover, a consistent orientation of the edge cycles that bound the facets provides information about the relative position of the polyhedron with respect to any of its facets. The incidence information associated with an edge seems to form a wing on either of its sides, hence the name of the model.

Although the doubly linked edge list was introduced to represent planar graphs, it can without change be used to represent the boundary of a polyhedron as well. This representation too is centered around the edge record, which, in addition to the pointers to the edge's incident vertices and facets, contains pointers to the records of two adjacent edges; namely to the edge that precedes e in a counterclockwise traversal of the edges incident upon one of e's incident vertices (as seen from the outside), and to the edge that follows e in the corresponding edge cycle associated with the other vertex incident upon e.

Finally, the quad-edge representation is closely related to the winged-edge model in the sense that the record of an edge e contains pointers to the records of the four edges that precede and follow e in the circular traversal of the facets that are incident upon e. However, the way to look at them is different: namely, the two edges that follow e around the facets are indeed understood as such; the other two, however, are perceived as the edges that follow e in a circular traversal of the edge cycles around e's incident vertices. The vertices and the facets are both represented as rings of the records of their incident edges, and their records need only include a

pointer to one of these edge records marking a starting point. The quad-edge model has the advantage of treating both the vertices and the facets of a polyhedron or a graph uniformly, which makes it particularly suitable for applications where one wants to easily process both the given polyhedron or graph and its dual. It has the disadvantage, however, that it cannot represent polyhedra whose facets are nonsimple polygons. Let us close the description of this model by mentioning that reference [35] goes beyond simply describing the above model; it also includes an *edge algebra*, a formal way to navigate through the representation.

Our representation is different from either of the above three representations, as it focuses on the facet records than on the edge records. This should not come at a surprise, since the triangulation of the polyhedron's boundary, the very first operation we perform, greatly simplifies the facet structure, and this should be exploited. The records of the different faces include the necessary incidence information and special fields useful to computations pertaining to specific operations of the algorithm. Let us take a closer look at the data structures (Figure 4.2).

The record of a vertex v contains a `degree` field to store v's degree, a `type` field to record information about whether v is reflex, pointed, witness or garbage, three fields to store v's coordinates, and a pointer to one of v's incident facets after the triangulation of the boundary of the polyhedron. The edge records contain the minimum information possible as far as incidence to other faces is concerned, namely pointers to the two incident vertices and facets. Let us note that the vertices are stored in lexicographic order. Additionally, the record contains a `type` field that marks whether the edge is garbage (stores 0) or not (stores 1), and the coefficients of the projection of the edge onto the xy-plane, `a` and `b` being the coefficients of x and y respectively, and `c` being the offset. In fact, in order to facilitate the duality transformation used in the Witness Oracle implementation, we require that `b` is either 0 or -1. Each facet f that results from the triangulation has very simple structure, and its record, in addition to the pointers to f's incident vertices and edges, contains a `sign` field storing 1 if the interior of the polyhedron lies above f with respect to the z-axis, -1 if it lies below f, and 0 if the facet is garbage (for a facet that is parallel to the z-axis and is not garbage, the `sign` field may contain any nonzero integer),

```
typedef struct vertex_type {
                int                     degree, type;
                double                  x, y, z;
                struct facet_type       *facet;
} Vertex;

typedef struct edge_type {
                int                     type;
                double                  a, b, c;
                struct vertex_type      *vertex[2];
                struct facet_type       *facet[2];
} Edge;

typedef struct facet_type {
                int                     sign;
                double                  a, b, c, d;
                struct vertex_type      *vertex[3];
                struct edge_type        *edge[3];
} Facet;

typedef struct v_bord_type {
                struct vertex_type      *v;
                struct edge_type        *prev_e, *next_e;
} BoundVertex;

typedef struct bord_type {
                int                     size;
                struct v_bord_type      *first;
} Boundary;

typedef struct old_facet_type {
                int                     sign, bound_num, vertex_num;
                double                  a, b, c, d;
                struct bord_type        *boundary;
} OldFacet;

typedef struct polytope {
                int                     vertex_num, edge_num, facet_num;
                struct vertex_type      *vertices;
                struct edge_type        *edges;
                struct facet_type       *facets;
                struct old_facet_type   *old_facets;
                struct v_bord_type      *v_list;
                struct bord_type        *border;
} Polytope;
```

Figure 4.2: Polyhedron Representation

and four fields `a`, `b`, `c` and `d` that store the coefficients of f's supporting plane. The vertices incident upon f, as in the case of the edge record, are stored in lexicographic order, and the plane coefficients are normalized so that `c` is nonnegative, and the normal vector `(a, b, c)` is of unit measure. The latter requirement implies that the quantity $D = \mathtt{a} \times x + \mathtt{b} \times y + \mathtt{c} \times z + \mathtt{d}$ is the signed distance of the point (x, y, z) from the supporting plane of f, meaning that the absolute value of D is equal to the Euclidean distance of the point from the supporting plane of f, and, if D is not equal to zero, its sign is 1 (-1) if the point lies above (below respectively) f's supporting plane.

Each facet before the boundary triangulation step is represented as a collection of rings of vertices. Its record has fields `sign` and `a, b, c, d` with the same meaning as above, and a pointer to an array of `bound_num` simple closed polygonal lines that bound the corresponding facet. An additional field `vertex_num` stores the total number of its vertices. Each bounding line is represented by an array of `size` records, each of which contains pointers to a vertex and its two incident edges. The first element of such an array is pointed by `first`.

Finally, the record of a polyhedron contains three fields that store the numbers of its vertices, edges and facets, and pointers to the arrays of all its faces, including the polyhedron's facets before and after the boundary triangulation step, as well as the auxiliary arrays required for the facet representation.

4.3 The Triangulation Algorithm

Let us summarize the basic steps of the triangulation algorithm. The algorithm begins by inputing the polyhedron and appropriately initializing the fields of the data structures that are used to represent it. Next, the boundary of the polyhedron is triangulated, and we enter the pull-off phase. The phase removes any flat vertices and the unhindered cups of a number of vertices of the polyhedron, reducing its size to proportional to the number of its reflex edges. Then, the fence-off phase is applied, which first produces a cylindrical partition of the polyhedron by erecting vertical fences through all the edges of the polyhedron, and subsequently decomposes

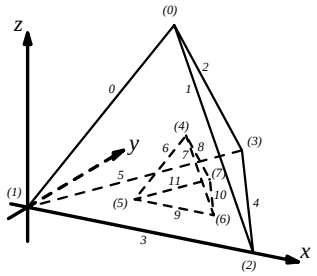


Figure 4.3: A punctured pyramid

it into tetrahedra completing the triangulation. The algorithm terminates printing the collected tetrahedra.

In the following, we describe each major step of the algorithm in a separate section. When the complexity of such a step is discussed, we assume that we are dealing with a polyhedron of n vertices and r reflex edges. The structure of the rest of the chapter roughly matches the structuring of the implementation of the algorithm into files.

4.3.1 Describing the Input Polyhedron

In order to input a polyhedron P, the user is requested to provide the coordinates of P's vertices and the necessary incidence information. Moreover, since this local information may determine two polyhedra, P and its complement, the user should provide one bit per facet f of the polyhedron specifying whether the interior of the polyhedron lies above or below the supporting plane of f.

Let us consider a concrete example, namely the punctured pyramid depicted in Figure 4.3. Let us assume that the vertices, edges and facets of the polyhedron have been numbered by consecutive nonnegative integers starting from 0, that will be used as their indices in the tables of vertices, edges and facets respectively. Figure 4.3 shows such numberings for the vertex and edge sets of the pyramid, while Figure 4.4 presents an input file that describes it by exploiting these numberings. The first line of the file contains the numbers of vertices, edges, and facets of the pyramid in that order. Then, the coordinates of the vertices are included in order of increasing vertex index, followed by a list of pairs of vertex indices that define the edges in order of

```
8    12   7

 8.0    4.0    8.0
 0.0    0.0    0.0
16.0    0.0    0.0
 8.0    8.0    0.0
 8.0    4.0    4.0
 4.0    2.0    0.0
12.0    2.0    0.0
 8.0    6.0    0.0

0    1
0    2
0    3
1    2
2    3
3    1
4    5
4    6
4    7
5    6
6    7
7    5

 1        2        3       1    3  4  5        3       5    11   10   9
-1        1        3       1    3  1  0
-1        1        3       2    1  2  4
-1        1        3       1    0  2  5
 1        1        3       5    6  7  9
 1        1        3       6    7 10  8
 1        1        3       5    6 11  8
```

Figure 4.4: Input file for the punctured pyramid of Figure 4.3

increasing index.

The list of facets, in order of increasing index again, completes the polyhedron's description. The description of a facet begins with an integer that is 1 if the interior of the polyhedron lies above the facet with respect to the z-direction, and -1 otherwise. The value is stored in the `sign` field of the facet record. In fact, the routine `init_polyhedron()` that inputs the polyhedron information assigns by default -1 to the `sign` field if a number other than 1 is read. Of course, telling whether the interior of the polyhedron lies above a facet parallel to the z-axis is not really meaningful. In this case, however, the contents of the field `sign` is never tested, and it can be any nonzero integer (the value of 0 is reserved to indicate a facet that became garbage). Let us point out that the z-direction is special to our algorithm since all fences are erected parallel to it.

The facet description concludes with the number of nonintersecting closed polygonal lines that bound the facet, followed by the descriptions of these lines. Such a polygonal line l is described by its size, the index of a vertex v, and the sequence of the indices of the edges that are encountered while walking along the line starting and stopping at v. In the case that a facet is bounded by more than one such lines, caution should be exercised so that the walks along these lines are consistent, meaning that the facet is either always to the left or always to the right. (This does not have to be true, however, for walks along lines bounding different facets.) Consider, for instance, the bottom facet (facet 0) of the punctured pyramid of Figure 4.3 that has two bounding lines. The edge sequences in the description of the facet in the input file of Figure 4.4 are determined by a counterclockwise walk (as seen from outside the polyhedron) along the bounding line determined by the edges with indices 9, 10 and 11, and a clockwise walk around the line determined by the edges with indices 3, 4 and 5, so that the facet remains always to the left during either of these walks.

It is assumed that no two nonincident faces of the input polyhedron are less than 10^{-6} (our threshold to discern nonincident faces) apart. The procedure `init_polyhedron()` included in the file `init.c` reads in all this information, stores it appropriately in the corresponding records, and fills in the remaining fields by assigning pointers and computing the necessary quantities. It calls the routines `set_e()` and `set_f()`

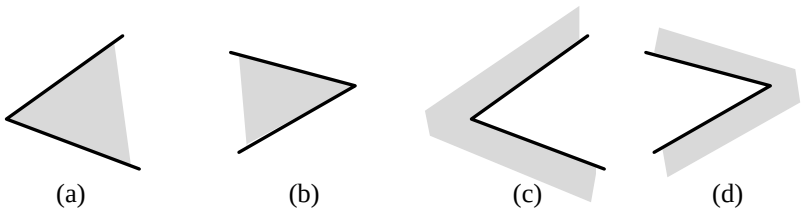


Figure 4.5: Different types of special points

that ensure that the vertices in the edge and facet records are stored in lexicographic order, and that the coefficients of the edges' projections onto the xy-plane and the facets' supporting planes meet the requirements that we presented in the previous section.

4.3.2 Triangulating the Boundary

In order to triangulate the boundary of the polyhedron, we apply the polygon triangulation algorithm of Hertel and Mehlhorn ([36]) on each facet of the polyhedron. Let us assume, for simplicity, that the polygon (facet) lies on the xy-plane. The algorithm collects those of its vertices whose incident edges form a nonmonotone chain with respect to the x-direction, each of which will be of one of four types: start point, stop point, left-turning point, and right-turning point (Figure 4.5). Then, each of these special points is processed in x-order, which corresponds to a sweeping of the polygon along the x-axis, while a list containing the regions in which the processed part of the facet has been decomposed is maintained in y-order at all times. Depending on the type of the special point, its processing may result in (i) the creation of a new region that is subsequently inserted in order in the list of existing ones, (ii) the termination of a region, which is then deleted from the list of regions, (iii) the splitting of a region into two new ones, and finally (iv) the merging of two adjacent regions into one. An interesting feature of the algorithm is that each region has its own local sweep-line that may lag behind, since it is updated in a lazy fashion. More specifically, only when one needs to find out whether a new special point lies in that region or not, its sweep-line is advanced along the x-axis until it reaches the x-coordinate of the point. Note that it is during this sweeping operation that the rest of the vertices of

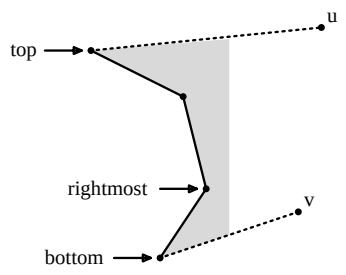
```
typedef struct v_list_node {
                struct vertex_type     *v;
                struct edge_type       *prev_e, *next_e;
                struct v_list_node     *previous, *next;
} VertexChain;

typedef struct region1_type {
                int                    up_ind, low_ind;
                struct bord_type       *up_border, *low_border;
                struct v_list_node     *rightmost, *top, *bottom;
                struct region1_type    *prev, *next;
} Region1;
```

Figure 4.6: Representing the regions

the polygon (that are not special points) are encountered.

For a polygon of n_i vertices, r_i of which are cusps, i.e., exhibit reflex interior angles, the algorithm takes $O(n_i + r_i \log r_i)$ time. To achieve this performance, a balanced binary search tree or equivalent data structure (to store the regions) is used. We can, however, afford to spend $O(n_i + r_i^2)$ time, since the entire polyhedron tetrahedralization algorithm has an overall time complexity of $O((n + r^2) \log r)$ for a polyhedron of n vertices and r reflex edges, and a reflex vertex contributes cusps to at most a constant number of facets. In other words, a doubly linked list whose entries are kept in y-order is an adequate data structure to store the x-monotone regions.

Let us study the data structure to represent such a region (Figure 4.6). A region is represented by a doubly linked list of `VertexChain` records each corresponding to a vertex of the already swept part of the region. Each such record contains a pointer to the corresponding vertex record, pointers to the incident edges (that may very well have been introduced by the triangulation process), and the fields `previous` and `next` that establish the double linking between these records. Now, the record of the



```
void     adj_upchain(regptr, v_b_ptr)
Region1     *regptr;
BoundVertex    *v_b_ptr;
{
    Vertex        *v;
    Edge        *e, *e_temp;
    VertexChain    *q, *r;

    v = v_b_ptr->v;
    for (q = regptr->top, e_temp = q->prev_e; ; q = r, e_temp = e){
        if ((r = q->next) == NULL|| angle(r->v, q->v, v) <= 0)
            break;        /* non convex angle */
        /* convex angle.  report a triangle */
        set_e(e = new_e(), r->v, v);
        set_f(new_f(), &info_f, q->next_e, e_temp, e);
    }
    q->previous = NULL;    /* all used q's are unlinked */
    q->prev_e = e_temp;
    regptr->top = q;
}
```

Figure 4.7: The routine `adj_upchain()`

region needs to have pointers to the first and last record of that chain (fields `top` and `bottom`), so that the chain can be easily updated and extended from either end. For this very purpose, the region record also stores the indices `up_ind` and `low_ind` of the vertices u and v adjacent to the two ends of the chain, and pointers `up_border` and `low_border` to the records of the bounding lines to which u and v belong. It should be noted that the vertex associated with the record pointed by the `top` field and u belong to the same bounding line; similarly for the vertex associated with the record pointed by `bottom` and v. Finally, a pointer to the rightmost node of the chain of the region, to be used if the region needs to be split, and pointers `prev` and `next` that tie the region records in a doubly linked list complete the data structure.

A typical operation over a region is to try to advance its sweep-line by extending its chain either from its upper or lower end. The invariant maintained is that the chain forms a weekly concave polygonal line, that is, it never turns left when traversed

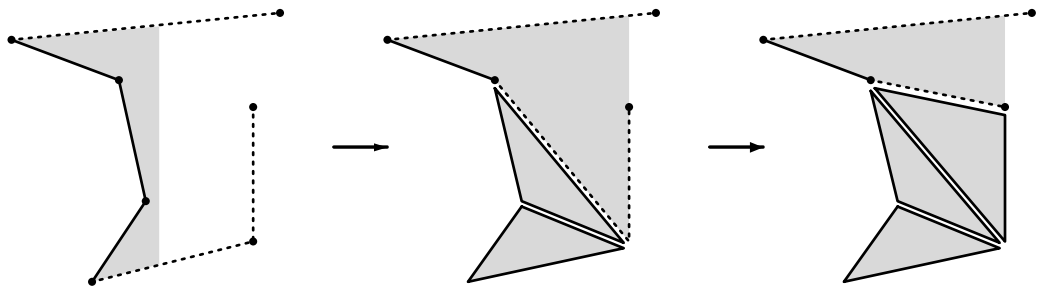


Figure 4.8

from its upper to its lower end. If extending the chain violates this requirement, then triangles are reported. Figure 4.7 presents the routine `adj_upchain()` that helps extend the chain from its upper end by including the vertex `v_b_ptr->v`. If a convex angle is formed by the new vertex v, the topmost vertex p_1 and the second from the top vertex p_2 of the chain, a new edge joining v to p_2 is created, and a new triangle (facet) determined by v, p_1 and p_2 is reported and inserted in the array of (triangular) facets. After that, vertex p_1 is discarded, and the above process is repeated for as long as convex angles between v and the currently two topmost vertices of the chain are encountered. The corresponding routine `adj_lowchain()` that performs the task of extending the chain of a region from its lower end is similar.

The two routines `adj_upchain()` and `adj_lowchain()` are the heart of the routine `advance()` that advances the sweep-line of a region up to the x-coordinate of the current point `vb_ptr->v`. The chain of a region needs to be extended from both ends in x-order, so we add one vertex at one of the ends at a time. Special care is needed when the boundary along which the chain of a region is extended folds inwards along a direction perpendicular to the x-axis. In this case, we need to make sure that the vertices that exhibit the fold are never inserted in the chain, otherwise overlapping triangles may be reported. Instead, triangles (facets) are reported cutting off the fold (Figure 4.8). Another fine point is that the routine `advance()` treats the two ends of the chain asymmetrically. This is witnessed only when the boundary along which the chain of the region is extended folds outwards along a direction perpendicular to the x-axis. Then, although we do not proceed to extend the lower end of the chain,

we do so for the upper end. This stems from the way the left-turning points are collected; namely, if a left-turning point needs to be collected among the vertices of a piece of the polygon's boundary that is normal to the x-axis, the one with the largest y-coordinate will qualify. Although the routine `advance()` is lengthy (see Appendix), it is fairly straightforward, as is the case with the rest of the routines that contribute to the triangulation process.

We close this section with a few remarks. In the above, we assumed that the polygon to be triangulated lies on the xy-plane. It would be unrealistic to believe that this will be true in general. Of course, one could rotate the system of reference to make the supporting plane of the given polygon parallel to the xy-plane, in which case the above process is readily applicable, but this involves the overhead for the computation of the rotated images of the vertices of the polygon. An easier way out that does not require transformation of the reference frame relies on the following observation. Consider the normal to the supporting plane of the polygon, and assume that its x- and z-coordinates are the smallest in absolute value. Then, the area of the projection of the polygon onto the xz-plane will be the largest among the areas of the polygon's projections on the three coordinate planes, and its ratio to the area of the given polygon will be no less than $1/\sqrt{3}$. So, we only need to find this favorable plane, and triangulate the polygon by considering its projection on that plane. As we will now be dealing with the xy-, xz- or yz-plane, the functions `diff1()` and `diff2()` that will return the differences of the primary and secondary coordinates (corresponding to the x- and y-coordinates in our previous discussion) of their arguments respectively need to be appropriately assigned. This is the task that the routine `select_funcs()` performs.

The final touch is to make sure that the orientation with respect to which the vertex chains that bound the given polygon (facet) match the one assumed by the triangulation routine. Our `triangulate()` routine assumes that the polygon is always to the left as we move along the vertex chains that bound it. This can be easily checked by locating the leftmost and lowermost vertex and investigating the relative position of its two incident edges. If the orientation is not the desired one the order of the edges along the bounding lines is reversed (see routine `check_orientation()`).

4.3.3 Collecting the Witness Facets

The process to collect the witness facets (a subset of the facets of the polyhedron whose incident vertices form a superset of the vertices whose cups are hindered) is the most demanding part of the implementation, as it involves duality ([58]) and fractional cascading ([12]). The implementation follows the method outlined in Section 3.4.2.

The process begins by collecting the puncturing vertices, the reflex vertices and the midpoints of the reflex edges. In terms of the puncturing vertices, the witness facets can be defined as the facets of P that are the first to be hit by rays parallel to the z-axis that emanate from the puncturing vertices. The puncturing vertices collection relies on the function `reflex()` that uses the information stored in an edge record to decide whether its argument, an edge e of the polyhedron, is reflex or not (Figure 4.9). Let f and g be the facets of the polyhedron incident upon e. Taking into account that no two facets of the given polyhedron are supposed to fall on each other, we conclude that if both f and g are parallel to the z-axis, then e has to be nonreflex. The remaining case is when at least one of the facets incident upon e, say f, is not parallel to the z-axis. Then, the edge is nonreflex, if either f and g are coplanar, or the sign of the signed distance of the vertex v of g that is not incident upon e from f is equal to the content of the field `sign` of f's record. In other words, e is nonreflex if v lies above the supporting plane of f if the interior of the polyhedron lies above f, and lies below f's supporting plane if the interior of the polyhedron lies below f. In fact, the function `reflex()` returns more information than a simple yes or no answer to the question whether e is reflex. Indeed, it returns 0 if e is nonreflex; otherwise, it returns 1, 2, or 3 depending on the relative position of the polyhedron's interior with respect to e. In particular, the returned value is 1 if the interior of the polyhedron lies vertically above e but not vertically below it, 2 if it lies both vertically above and below e, and 3 in the remaining case. This extra information is stored in the puncturing vertices record, and is used during the witness facet collection. Namely, if a reflex edge e is such that the interior of the polyhedron lies vertically above it but not vertically below it, then while collecting the candidates for the witness facets associated with the puncturing vertex contributed by e's midpoint, we ignore all facets of the polyhedron such that the interior of the polyhedron lies above them.

```
int     reflex(e)
Edge        *e;
{
    int         i, term;
    double          temp;
    Vertex        *v, *v1, *v2;
    Facet        *f, *g;

    f = e->facet[0];    g = e->facet[1];
    v1 = e->vertex[0];    v2 = e->vertex[1];
    if (E_COMPARE(f->c, 0.0) == 0){
        if (E_COMPARE(g->c, 0.0) != 0){ /* g not vertical */
            /* locate vertex v of facet f not incident upon e */
            for (i = 3; --i >= 0; )
                if ((v = f->vertex[i]) != v1 && v != v2)
                    break;
            temp = EVAL_3(*v, *g);
            if (E_COMPARE(temp, 0.0) * g->sign == -1)
                return(2 - g->sign);
        }
    }
    else {        /* f is not vertical */
        if ((term = -f->sign) == g->sign)
            term = 0;
        /* locate vertex v (of facet g) not incident upon e */
        for (i = 3; --i >= 0; )
            if ((v = g->vertex[i]) != v1 && v != v2)
                break;
        temp = EVAL_3(*v, *f);
        if (E_COMPARE(temp, 0.0) * f->sign == -1)
            return(2 + term);
    }
    return(0);
}
```

Figure 4.9: The routine `reflex()`

After the puncturing vertices have been collected, their projections onto the xy-plane are mapped into lines by means of the duality transformation. The line arrangement $\mathcal{A}$ that is thus formed is subsequently processed to derive a logarithmic time point location data structure, as well as a balanced binary tree of line arrangements as discussed in Section 3.4.2. Since the construction of this tree has been adequately explained in Section 3.4.2, we next describe in some detail the point location data structure. Point location is based on the *layers* of the arrangement $\mathcal{A}$. These are polygonal lines, each tagged with a distinct integer from 0 to 1 less than the number of lines in $\mathcal{A}$, such that if a ray perpendicular to the x-axis is shot from any point of the k-th layer towards increasing ys, it intersects exactly k of the lines of $\mathcal{A}$. (Figure 4.10 shows the layers of a line arrangement of three lines.) It is interesting to remark how well the duality transformation guarantee that no line is normal to the x-axis fits the layers definition. The layers are connected x-monotone chains each separating the arrangement into two pieces. As they do not cross each other, their tags provide an ordering along the y-axis, which implies that the layers can be structured in a balanced binary search tree. The tree allows for $O(\log^2 r)$ time point location in the arrangement: we walk down the tree, and at each tree node t we spend $O(\log r)$ time to determine the relative position of the query point with respect to the layer associated with t, which will indicate whether we will proceed to t's left or right descendant. The time complexity can be improved to $O(\log r)$ if fractional cascading is applied to the x-monotone layers. Namely, the layer of a tree node t is refined with the introduction of new vertices along it at the x-coordinates of some of the vertices of the layers stored at t's direct descendants. In fact, we use every other vertex of these

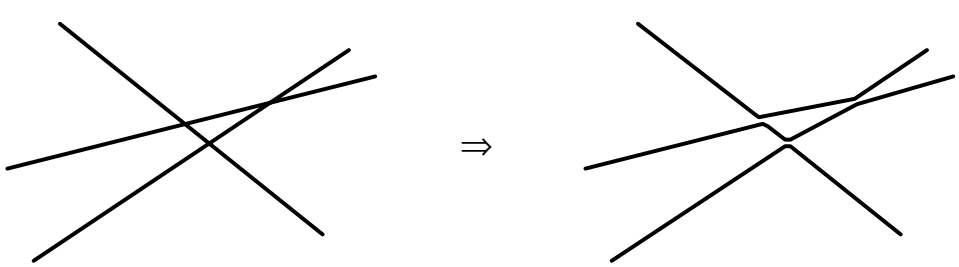


Figure 4.10: The layers of an arrangement of three lines

```
typedef struct vertex_rec {
                int                     up_reg_no, low_reg_no;
                double                  x, y;
                struct edge_rec         *next_edge;
} ArrangV;

typedef struct edge_rec {
                int                     punct_no, region_no;
                struct vertex_rec       *vertex;
                struct edge_rec         *next, *twin, *more;
} ArrangE;
```

Figure 4.11: The data structure to represent an arrangement

layers to introduce new vertices on t's layer. The process is performed from bottom to top, which implies that no backtracking is needed. The scheme guarantees that at the end the total size of the data structure is still linear in the number of vertices of the line arrangement $\mathcal{A}$, and that the x-extent of each edge in any of the resulting layers intersects the x-extents of at most two edges of either of the layers stored at its direct tree descendants. The latter observation and pointers between edges of the layers are the key tools on which the point location `locate_reg()` routine is based.

Let us describe how a line arrangement is represented (Figure 4.11). First, the vertices of the arrangement are stored in an array. Although a vertex may appear in as many as $O(\log r)$ arrangements due to the tree decomposition of the arrangement $\mathcal{A}$ of all the lines (dual images of the projections of all the puncturing vertices), it is stored only once. Its record contains its x- and y-coordinates, two integer fields `up_reg_no` and `low_reg_no`, and a pointer `next_edge` to an edge of an arrangement. The two integer fields store the numbers assigned to the regions that lie above and below the vertex (along the y-direction) in the global line arrangement $\mathcal{A}$, and are used only at the point-location process. The pointer field `next_edge` is used to help us navigate along the layers of a line arrangement. The field is extensively used throughout the tree decomposition and we make sure that it is cleared to `NULL` after all the edges incident upon the corresponding vertex have been visited. Let us mention that we

declare and use two special vertices `L_INF` and `R_INF` to mark artificial vertices at the $-\infty$ and $+\infty$ points of the x-axis.

The edge chains around the regions of the line arrangement are oriented in a counterclockwise direction, and so each edge assumes two opposite orientations, each associated with one of its two incident regions. Based on this, we represent each edge e of the arrangement as a pair of edge records q_1 and q_2 that point to each other by means of their `twin` fields. Each of the q_is is associated with one of the two regions, say r_i, incident upon e, and additionally stores the index `punct_no` of the puncturing vertex whose dual image contains e, the number `region_no` assigned to the region r_i, a pointer `next` to the corresponding record of the edge e' that follows e in the circular walk around r_i, and a pointer `v` to the record of the vertex incident upon both e and e'. A pointer field `more` that completes the edge record is used to produce a list of all the edges in the arrangement by linking together one of its twin records per edge. to link some of the edge records in a singly linked list; the list is exploited during the tree decomposition of the line arrangement, where it is scanned to determine which regions of the parent arrangement are merged in each of the children's arrangements.

The line arrangement is built incrementally, one line at a time, in decreasing order of their slopes and increasing offsets in case of equal slopes. Figure 4.13 shows the part of function `get_arrangmnt()` that is responsible for the insertion of a new point of intersection in the line arrangement, while Figure 4.12 shows the corresponding pointer updates. Note that the twin records of the edge that is split cease to be twin, and acquire new partners; the reason is that other edge records point to them, and this is the way to maintain the correct chains representing the boundaries of

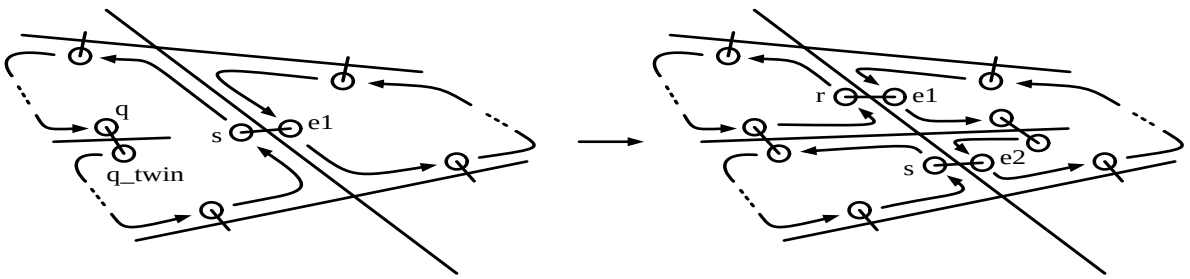


Figure 4.12: Inserting a new line in the arrangement.

```
int     get_arrangmnt(...)
{
    ...

        switch (cond){
        case 0:  /* line passes through a vertex */
            ...
            break;

        case 1:  /* line crosses edge */
            e1 = s->twin;
            get_edge_pair(&r, &e2, k);
            /* update new edge record r, twin to e1 */
            r->vertex = p;        /* p = s->vertex */
            r->next = s->next;
            s->next = q_twin;
            r->twin = e1;    e1->twin = r;
            q->next = r;
            /* update new edge record e2, twin to s */
            e2->vertex = e1->vertex;
            e2->next = e1->next;
            e2->twin = s;    s->twin = e2;
            /* compute point of intersection */
            x = -(offset - punct[k].y)/(slope - punct[k].x);
            p = get_vertex(x, slope * x + offset);
            q->vertex = s->vertex = e1->vertex = p;
            break;
        }

        /* update edge-records for new edge in new region */
        get_edge_pair(&q, &q_twin, i);
        q->twin = q_twin;    q_twin->twin = q;
        e1->next = q;
        q_twin->next = e2;
        q_twin->vertex = p;
        /* remaining fields of q to be set later */

    ...
}
```

Figure 4.13: Inserting a new line in the arrangement.

the regions without having to walk all the way around the regions. A final note on the line arrangement representation is that all unbounded regions are represented by chains of edge records that are open at the open end of the region, except for the ones open towards the $-\infty$ point of the x-axis. In this way, insertion of new lines is facilitated as the bounding chain of these regions is closed.

We turn now to the $O(\log r)$ depth binary tree, each of whose nodes stores the line arrangement of a subset of the dual images of the projections of the puncturing vertices onto the xy-plane. The arrangement $\mathcal{A}$ of the entire set is stored at the root, and the rule is that if a tree node stores the arrangement of a set S of lines, then its children store the arrangements of two disjoint subsets S_1 and S_2 of S, which partition S and whose cardinalities differ by at most 1. As mentioned in Section 3.4.2, the arrangement of S is a refinement of the arrangements of S_1 and S_2, and consequently each region of the parent arrangement lies entirely within a single region of the arrangements of either child. In fact, pointers are established from each region of an arrangement to its enclosing regions in the arrangements of its children. Using the logarithmic time point location data structure described earlier and this pointer scheme, we are able to locate the puncturing vertices whose projections on the xy-plane fall in the closure of the projection of a (triangular) facet in $O(\log r)$ time per reported puncturing vertex after an $O(\log r)$ time overhead per facet has been spent for point location in the root arrangement. Then, the set of facets that, for a particular puncturing vertex p, are closest to p with respect to the z-direction both above and below constitute the set of witness facets.

The tree of arrangements and the region pointers are constructed by means of a recursive routine `build_tree()` (Figure 4.14). In the general step, we have a line arrangement $\mathcal{L}$ from which we want to derive the arrangements $\mathcal{L}_1$ and $\mathcal{L}_2$ of two sets of lines which partition the set of lines of $\mathcal{L}$ and whose cardinalities differ by at most 1. Unlike Section 3.4.2, in the implementation we use Union-Find routines to merge regions of the parent arrangement to those of its children. The routines, although asymptotically superlinear, do not alter the overall asymptotic complexity of the algorithm, and are simpler and easier to code. The new approach works as follows. The routine `build_tree()` is supplied with a list of edge records of the arrangement

```
DualTreeNode    *build_tree(imin, imax, table, table_size, list)
int         imin, imax, table_size;
ArrangE        *list;
RegEntry    *table;
{
    int         i, mid_layer, separator, l_no, r_no;
    ArrangE        *left_list, *right_list;
    RegEntry    *left_table, *right_table;
    DualTreeNode    *t;

    t = du_tree + v_count++;
    if (imin == imax){
        t->left = NULL;        /* this marks a leaf */
        i = layer[imin].edge_ptr->punct_no;
        t->right = (DualTreeNode *)(punct + i);    /*ptr to punct*/
        /* reserve 2 slots in table of regions (playing it safe) */
        table->d[0].ptr = table->d[1].ptr = table;
        table++;
        table->d[0].ptr = table->d[1].ptr = table;
        next_avail += 2;
        return(t);
    }
    mid_layer = (imin + imax) / 2;
    separator = layer[mid_layer].edge_ptr->punct_no;
    t->x_maxleft = punct[separator].x;
    t->x_minright = punct[layer[mid_layer+1].edge_ptr->punct_no].x;
    next_avail += table_size;    /* reserve space in region table */
    merge(list, separator, table, table_size);
    l_no = equiv_name(table, table_size, 0);
    r_no = equiv_name(table, table_size, 1);
    left_list = right_list = NULL;
    split_arrang(imin, imax, mid_layer, separator, table, &left_list,
                                &right_list);
```

Figure 4.14: The routine `build_tree()` (cont'd)

```
    left_table = next_avail;
    t->left = build_tree(imin, mid_layer, left_table, l_no, left_list);
    right_table = next_avail;
    t->right = build_tree(mid_layer+1, imax, right_table, r_no,
                                right_list);
    /* establish the pointers between the tables of regions */
    for (i = 0; i < table_size; i++){
        table[i].d[0].ptr = left_table + table[i].d[0].dad;
        table[i].d[1].ptr = right_table + table[i].d[1].dad;
    }
    return(t);
}
```

Figure 4.14: The routine `build_tree()`

$\mathcal{L}$; for each edge in $\mathcal{L}$, the list contains one of the twin records that represent it. The list is passed to a routine `merge()` that visits each edge in the list to determine which regions of the parent arrangement are merged in the child arrangement that does not inherit the line supporting this edge, and uses Union-Find with path compression and load balancing to compute the sets of regions that are merged into the same region of $\mathcal{L}_1$ and $\mathcal{L}_2$. Next, the routine `equiv_name()` tags each such set with a distinct nonnegative number, which becomes the index of the corresponding region of $\mathcal{L}_1$ or $\mathcal{L}_2$ in the region table. The routine `split_arrang()` is then called to split the representation of $\mathcal{L}$ into the representations of $\mathcal{L}_1$ and $\mathcal{L}_2$. To achieve that, it uses the routine `disconnect()` to separate the arrangements at the intersection points of lines that do not all belong to the same $\mathcal{L}_i$. At the same time, `split_arrang()` compiles the lists of edges of $\mathcal{L}_1$ and $\mathcal{L}_2$ preparing for the recursive call on them. Indeed, after `split_arrang()` is done, the routine `build_tree()` recurses on both $\mathcal{L}_1$ and $\mathcal{L}_2$ and establishes the pointers that create the tree.

In the end, each tree node is associated with the arrangement of some set of lines. In particular, each of the leaves of the tree is associated with a single line. A tree node is represented by a record that, in addition to pointers to the node's left and right children, contains two fields `x_maxleft` and `x_minright` that store the x-coordinates of the largest puncturing vertex of its left child and the smallest puncturing vertex

of its right child respectively. These fields help us navigate through the tree to find the puncturing vertices whose projections on the xy-plane lie in the closure of the projection of a facet. We note that, while processing the polyhedron's facets at tree nodes, we will deal with both the set of associated puncturing vertices and its dual image, the corresponding line arrangement.

After the data structures have been prepared, we start processing the facets of the polyhedron in turn. As described in Section 3.4.2, each facet f is partitioned into at most two triangles with a plane parallel to the yz-plane. As a result, the projection of each of these triangles on the xy-plane can be viewed as the intersection of a double wedge and a strip parallel to the y-axis.

Now, all we need to do is to locate the puncturing vertices whose projections on the xy-plane lie in the closure of the projections of these triangles, and to associate them with the facet f (routine `visit()`). Consider such a triangle T whose projection is the intersection of a double wedge and a strip H, and let Seg be the line segment that is the dual image of this double wedge. We first test whether the x-extent of H intersects the interval $[x_{min}, x_{max}]$, where x_{min} and x_{max} are the smallest and largest x-coordinates of the puncturing vertices. If not, then H falls either to the left or the right of the projections of all the puncturing vertices, and the processing of T is complete. Otherwise, we locate the endpoints of Seg in the line arrangement stored at the root of the tree, and we traverse the tree processing each node t as follows. If the endpoints were found to fall in the same region of the arrangement of t, we stop; the projections of T and the puncturing vertices associated with t on the xy-plane do not intersect. If this is not the case, then if H spans entirely the range of x-coordinates of any of the sets of puncturing vertices associated with t's children, routine `exhaust()` is called to visit the subtree rooted at that child of t. If H intersects but does not span the corresponding range of x-coordinates of any of t's children, `visit()` recurses on that tree node. The region pointer scheme helps us locate in constant time the regions of the new arrangements that enclose the endpoints of Seg. The routine `exhaust()` performs essentially the same processing as `visit()`, but it does not test the x-coordinates, and recurses to both children of the current tree node, since when it is called we know that H spans the entire x-extent of the point set associated with

the node, and consequently its children's too.

When a leaf is reached, and the endpoints of Seg do not belong to the same halfplane determined by the dual image of the single puncturing vertex p associated with the leaf, we know that the projection of p falls in the projection of the facet f that we process. The information about the witness facets associated with p may need update. Suppose that the interior of the polyhedron falls below f. (The case is similar if it falls above f.) Then if we collect facets vertically above p, and p lies below f, we test whether f is the lowermost facet above p. If f lies below the current lowermost facet(s) associated with p, it becomes the new current lowermost facet; if there is a tie, f is inserted in an array of current lowermost facets associated with p.

After all the facets have been processed, the set of witness facets is retrieved by collecting the facets associated with all the puncturing vertices.

4.3.4 Pulling-off the Cups

Before proceeding with the discussion of the implementation of this operation, we must point out that the approach we pursued in the implementation is slightly different from the one described in Section 3.4.1. Namely, in the previous chapter the nonreflex vertices incident upon witness facets are eligible to be pulled off provided that their cups are not hindered. In the implementation, however, these vertices are marked and are not allowed to participate in the process that removes unhindered cups. This greatly simplifies the routine that extracts the cup of a vertex v, as no facet incident upon v can be a witness facet, and thus no distribution of associated puncturing vertices between disappearing and emerging facets is needed. Moreover, this change does not affect the asymptotic complexity of the algorithm, as the number of vertices that are marked in this way is $O(r)$. The disadvantage is that a number of vertices, whose cups would have otherwise been pulled off, will not be removed, and they will be inherited to the fence-off phase, increasing the running time of the program and the number of tetrahedra produced.

The process starts by compiling a list of vertices that (i) have not been marked as reflex or incident upon a witness facet, and (ii) whose degree is no more than a

fixed constant `D` (set to 12 for our tests; see file `defs.h` in Appendix), and proceeds by removing the cups of the elements of the list one by one. If in the removal process the degree of an unmarked vertex drops below `D` + 1, the vertex is inserted in the list of vertices to be removed. For this reason, this list is implemented as a queue. Removing the cup of a vertex v consists of two stages. The first computes the crown of v potentially retriangulating the part of the polyhedron boundary incident upon v (coded in the routine `pointed_remove()`), while the second substitutes the boundary of the polyhedron incident upon v by the dome of v (coded in the routine `repair()`).

The crown of the vertex v is represented as a stack of records `CrownRec`, each corresponding to a crown edge; their order in the stack corresponds to a walk around the crown. The record of a crown edge e contains pointers to e, to e's incident vertices $v1$ and $v2$ in the order they are met along the walk, to the facet f incident upon e and v, and to the edge $e2$ connecting v to $v2$. Storing pointers to f and $e2$ allows us to collect the facets and edges of the polyhedron that are discarded due to the removal of the cup.

Let us take a look at the `pointed_remove()` routine. Note that the routine is not called unless the degree of the vertex to be removed is again verified to be at most equal to `D`, as preceding cup removals may have increased its degree (see `phase1()` in file `main.c`). The routine is supplied with a starting point `v2` of a facet `f2` incident upon the apex. The facet `f2` and the other facet incident upon both the apex and `v2` are not coplanar. Note that if no such a starting point exists, then the vertex is flat, and the routine `flat_remove()` is called instead.

The routine `pointed_remove()` starts by computing the corresponding crown. Every time a facet that is not coplanar with previous facets incident upon the `apex` is discovered, an appropriate projection plane is selected in the way described in the boundary triangulation process, that is, the plane defined by the coordinate axes along which the facet normal has the smallest and next smallest projections. The need to locate this plane stems from the fact that we want to test whether convex or concave angles are exhibited, and it would be more complicated to perform this test on the supporting plane of the current facet. Moreover, a crown record for that edge is pushed in the stack that stores the computed crown, and the stack position is

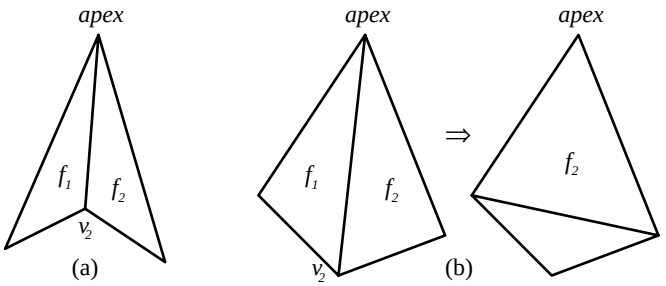


Figure 4.15: The cases that occur in crown computation

marked (by the pointer `first`) meaning that any preceding stack entries correspond indeed to the desired crown. After that, we proceed to the next facet in the ring of facets around `apex`. For as long as the current facet is coplanar to the first facet in the current set of coplanar facets (pointed by `f_sentnl`) the piece of the crown on their common supporting plane is getting assembled edge by edge. If the above condition is false, the crown computation starts on the new set of coplanar facets. The crown assembly is the direct coding of the cases shown in Figure 4.15. In the general step, we have a facet `f2` adjacent to the facet `f1` that has just been processed. The two facets form a quadrilateral Q that obviously exhibits an angle less than π at `apex`. If the interior angle of the quadrilateral at the vertex $v2$ diagonally from `apex` is at least equal to π (case(a)), a crown record for the edge of `f2` that is not incident upon `apex` is pushed at the stack, and we move to the next facet in order. If the angle is less than π, then the topmost record of the stack is popped, and the diagonal of the quadrilateral is flipped. Then, considering the facet incident upon `apex` that emerged by the diagonal flipping as the new facet `f2` , we move backwards in the stack performing the same operation. This continues for as long as angles less than π are exhibited at the vertex diagonally from the `apex` in the quadrilateral formed by the current facet `f2` and the facet at the top of the stack. When either an angle no less than π is found, or all the facets of the current set of coplanar facets have been traversed, a crown record with information from `f2` is inserted in the stack, and we proceed to the next facet in order.

Eventually, after traversing the entire cycle of facets around the apex, we will

return to the starting vertex, marked by `v_sentnl`, in which case the crown computation is complete. The routine terminates by collecting the facets and edges incident upon `apex` at the end of the crown computation step. These facets and edges will become garbage, as the part of the boundary of the polyhedron incident upon the apex will be substituted by the corresponding dome. In fact, all but three of the collected edge records and all but two of the collected facet records will be reused by the `repair()` routine to store the edges and facets of the dome. After calling the routine `repair()` that computes the dome and stitches it in the boundary, the routine `pointed_remove()` terminates.

The routine `repair()` finds the dome of the vertex `apex` passed as its argument by using a brute-force way to compute convex hulls. In fact, it does not compute the entire convex hull of the crown, but only the dome, that is the part of the convex hull's boundary that lies between `apex` and the rest of the convex hull. The idea is to try to find for each edge e of the crown in turn a vertex p of the crown such that no crown vertex lies in the same open halfspace (defined by the plane that passes by e and p) as the apex. The triangle defined by e and p is a dome facet, and it is subsequently stitched in the boundary of the polyhedron, while a tetrahedron defined by the facet and the apex is reported. The correct stitching requires caution, as one needs to check which edges of the facet already belong to the boundary of the polyhedron. The routine uses an economical strategy recycling the edges and facets that became garbage due to the cup removal step.

4.3.5 Computing the Fences

The routines that enable us to compute the fence of an edge e are straightforward implementations of the method described in Section 3.4.3. Namely, first, routine `fence_parts()` determines whether e's fence has an upper part, a lower part, both, or none.

This information is then passed to the routine `collect_segments()` that collects e's projections vertically onto the facets of the polyhedron and stores them appropriately into an `above` or a `below` list. The `above` list contains segments that are

contributed by facets that lie vertically above e and leave the interior of the polyhedron underneath them; if e's fence does not have an upper part, as determined by `fence_parts()`, the list is empty. The case is similar for the `below` list. Note that vertical facets, as well as the facets incident upon e are ignored, while projections that are either empty or degenerate to a single point are discarded. For the selection process, the routine `collect_segments()` uses the information pertaining to whether the fence of e has an upper part, lower part, or both, and the `sign` field of the probed facets.

Finally, routine `compute_fence()` processes separately the provided lists of segments, and selects among the elements of each such list the segments that bound the fence of e, which it associates with the facets that contribute them. In particular, for the `above` list for instance, `compute_fence()` sorts the endpoints of the list elements by increasing x-coordinate (or y-coordinate if the edge is perpendicular to the x-axis), by increasing z-coordinate, and in case of ties right endpoints precede left endpoints. The endpoints are then processed in order, and the collection of active line segments (the segments whose only the left endpoint has been encountered so far) needs to be appropriately updated. The update may involve insertion of a new segment, or deletion of an already existing element; no other change is needed, since the segments do not intersect. Moreover, the lowermost line segment should be located at small computational cost. These operations perfectly fit a heap data structure, which is what we use to store the active segments. Processing a new endpoint has the following effect on the heap. If the current endpoint is a right endpoint, the corresponding segment is simply removed from the heap if it is not the currently lowermost segment. Otherwise, the segment needs to be reported first, as it is one of the segments that bound the upper part of e's fence; and the floor is ready for a new lowermost segment. In the case where the current endpoint is a left endpoint of a segment, the segment needs to be inserted in the heap. If it does not become the lowermost segment, nothing needs to be done; otherwise a segment bounding the fence is reported.

4.3.6 Shattering the Facets

Shattering a facet is precisely triangulating the polygons in which the facet has been partitioned by the traces of fences. These polygons are not, however, explicitly represented; instead, we just have an unordered list of the line segments that bound them. So, our first concern is to try to relate the line segments with each other. We locate, therefore, their distinct endpoints, and associate the segments with them using pointers. Then, sorting the segments according to their left endpoints and slopes, produces clusters of segments incident upon the same left endpoint that are ordered by slope.

The representation and routines that are used to carry out the triangulation are the same as the ones discussed in Section 4.3.2. Briefly, we maintain a collection of regions, each of which is represented as a weakly concave chain bounding from the left the untriangulated portion of the facet. Unlike Section 4.3.2, however, we cannot afford to use an array or list to store the entire collection of regions in y-order; instead, a balanced binary search tree is necessary. (Our implementation uses the red-black tree routines included in [20].) In addition to that, the regions are connected in a doubly linked list in y-order; the sequential traversal of the list results in an in-order traversal of the red-black tree.

We are now ready to process the endpoints in x-order. Processing the current endpoint p involves locating the region in which it falls distinguishing cases depending on whether p lies in the interior or the boundary of the region, and updating the regions that start at p, terminate at p, or have p on their boundaries. In case p falls in the interior of a region is particularly easy (see routine `in_region()` in file `shatter.c` of the Appendix); the region needs to be split into two pieces, and new regions, if any, delimited by line segments incident upon p need to be inserted between them.

The case where p falls on the boundary of a region, is more complex, although it does not involve any special tricks (see routine `on_boundary()`). Our first action is to include p in the weekly concave chain that bounds from the left the untriangulated part of this region. In fact, p may lie on the boundaries of several regions, in which case the above operation has to be done to all these regions resulting in the complete

sweeping of some of them and their deletion from the list and the tree. Next, the segments, if any, that have p as their left endpoint need to be inserted, leading to the creation of new regions and their subsequent insertion to the regions' data structure. Attention is needed, if p is on the boundary of some region without actually being one of the regions vertices. There could be only two such regions, both bounded by the segment in the interior of which p falls. In this case, this segment needs to be interfiled in order among the segments incident upon p.

The reported triangles are collected in an array `triangles` whose processing is described in the next section. Let us remark that the `on_boundary()` routine is more general than what this particular problem instance requires. Indeed, the way the segments are defined by the fences imposes some restrictions on them. Namely, we do not expect to have segments that share more than one point, or equivalently collinear segments that intersect in a line segment of nonzero length. Moreover, if the left endpoint, say q, of a number of segments falls in the interior of another segment s, then s is expected to have slope that is strictly larger or strictly smaller than the slopes of all the segments incident upon q. So, if it were guaranteed that no errors would occur in the computations, the above restrictions could be incorporated in the code, simplifying it. We choose, however, to code the routine in its generality, since the above restrictions may be violated due to the tolerance threshold `EPSILON` that we use.

4.3.7 Completing the Triangulation

Completing the triangulation of the polyhedron is a straightforward implementation of the theoretical discussion in Section 3.4.3. The task is greatly facilitated thanks to the table of triangles that we have compiled during the shattering of the polyhedron's facets. All that is needed is that the matching pairs of triangles are brought together, and the (at most three) tetrahedra that partition the cylindrical piece determined by such a pair of triangles are reported (see function `tetrahedralize()` in the file `tetrahedra.c`, in the Appendix).

```
int     triang_compare(tr1, tr2)
Triangle    *tr1, *tr2;
{
    int         i, c, xy_p_compare();
    double          sum1, sum2;
    Point_3d    *p, *q;

    for (i = 0, sum1 = sum2 = 0.0, p = tr1->p, q = tr2->p; i < 3;
                 sum1 += p->z, sum2 += q->z, i++, p++, q++)
        if ((c = xy_p_compare(p, q)) != 0)
            return(c);
    if ((c = E_COMPARE(sum1, sum2)) == 0)
        ERROR("triang_compare> double occurence of triangle");
    return(c);
}
```

Figure 4.16: The routine `triang_compare()`

The matching of the triangles is carried out by a sorting routine, where the ordering is determined by a comparison function `triang_compare()` (Figure 4.16) that orders two triangles depending on the x- and y-coordinates of the three pairs of their corresponding vertices. In this way, triangles with the same projection onto the xy-plane are guaranteed to end up in adjacent positions in the table of triangles, and the final match-up is based on the z-coordinate of their centroids.

The algorithm concludes with the extraction of the tetrahedra. The way to partition a cylindrical piece determined by two matching triangles $t1$ and $t2$ is exemplified in Figure 4.17, where caution is needed to distinguish the cases where $t1$ and $t2$ touch on a vertex or along an edge. The code for `extract_tetrahedra()` is included in Figure 4.18.

The program terminates by outputing a list of the tetrahedra computed, each reported as its four vertices. For the punctured pyramid shown in Figure 4.3, the program outputs 18 tetrahedra, as shown in the Figure 4.19.

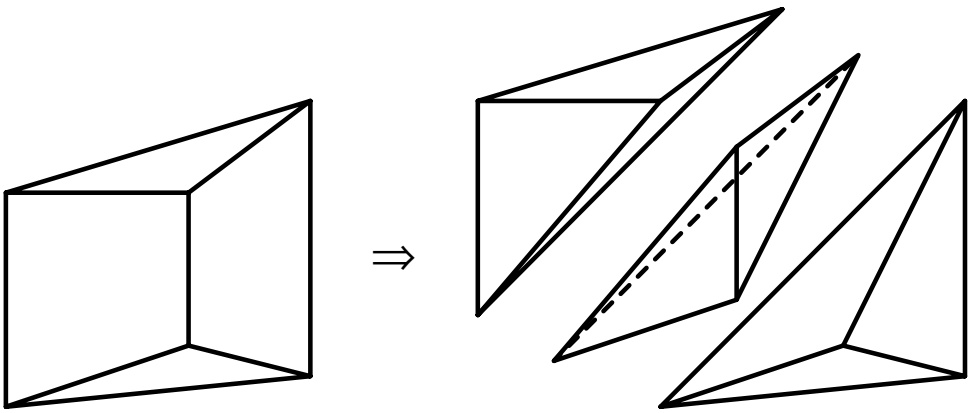


Figure 4.17: Tetrahedralizing a cylindrical piece

```
void     extract_tetrahedra(tr1, tr2)
Triangle    *tr1, *tr2;
{
    int         i;
    Point_3d    *q;
    Tetrahedron    t;

    for (i = 3; --i >= 0; )
        t.p[i] = tr1->p[i];
    /* report the one/two/three tetrahedra */
    for (i = 3, q = tr2->p + 2; --i >= 0; t.p[i] = *q, q--)
        if (E_COMPARE(t.p[i].z, q->z) != 0){    /* remove vertex p[i] */
            t.p[3] = *q;
            INS_TABLE(t, Tetrahedron, &tetrhdr_table);
        }
}
```

Figure 4.18: The routine `extract_tetrahedra()`

```
Polyhedron initialization is complete.
Witness identification is complete.
Pull-off phase is complete.
Computation of the fences is complete.
Shattering the polyhedron's facets is complete.
Fence-off phase is complete.

     Tetrahedra Produced (total number = 18)
(0.00, 0.00, 0.00)-(4.00, 2.00, 0.00)-(8.00, 8.00, 0.00)-(4.00, 2.00, 4.00)
(0.00, 0.00, 0.00)-(4.00, 2.00, 0.00)-(12.00, 2.00, 0.00)-(12.00, 2.00, 4.00)
(0.00, 0.00, 0.00)-(4.00, 2.00, 0.00)-(12.00, 2.00, 4.00)-(4.00, 2.00, 4.00)
(0.00, 0.00, 0.00)-(12.00, 2.00, 0.00)-(16.00, 0.00, 0.00)-(12.00, 2.00, 4.00)
(4.00, 2.00, 0.00)-(8.00, 4.00, 4.00)-(8.00, 6.00, 0.00)-(8.00, 6.00, 4.00)
(4.00, 2.00, 0.00)-(8.00, 4.00, 4.00)-(8.00, 6.00, 4.00)-(8.00, 4.00, 8.00)
(4.00, 2.00, 0.00)-(8.00, 4.00, 8.00)-(8.00, 6.00, 4.00)-(4.00, 2.00, 4.00)
(4.00, 2.00, 0.00)-(8.00, 4.00, 4.00)-(12.00, 2.00, 0.00)-(12.00, 2.00, 4.00)
(4.00, 2.00, 0.00)-(8.00, 4.00, 4.00)-(12.00, 2.00, 4.00)-(8.00, 4.00, 8.00)
(4.00, 2.00, 0.00)-(8.00, 4.00, 8.00)-(12.00, 2.00, 4.00)-(4.00, 2.00, 4.00)
(4.00, 2.00, 0.00)-(8.00, 6.00, 0.00)-(8.00, 8.00, 0.00)-(8.00, 6.00, 4.00)
(4.00, 2.00, 0.00)-(8.00, 6.00, 4.00)-(8.00, 8.00, 0.00)-(4.00, 2.00, 4.00)
(8.00, 4.00, 4.00)-(8.00, 6.00, 0.00)-(12.00, 2.00, 0.00)-(12.00, 2.00, 4.00)
(8.00, 4.00, 4.00)-(8.00, 6.00, 0.00)-(12.00, 2.00, 4.00)-(8.00, 6.00, 4.00)
(8.00, 4.00, 4.00)-(8.00, 6.00, 4.00)-(12.00, 2.00, 4.00)-(8.00, 4.00, 8.00)
(8.00, 6.00, 0.00)-(8.00, 8.00, 0.00)-(12.00, 2.00, 0.00)-(12.00, 2.00, 4.00)
(8.00, 6.00, 0.00)-(8.00, 8.00, 0.00)-(12.00, 2.00, 4.00)-(8.00, 6.00, 4.00)
(8.00, 8.00, 0.00)-(12.00, 2.00, 0.00)-(16.00, 0.00, 0.00)-(12.00, 2.00, 4.00)
```

Figure 4.19: Tetrahedralization of the pyramid of Figure 4.3

Chapter 5

Conclusions

In this thesis we studied two decomposition algorithms for three-dimensional polyhedra, namely, an algorithm to partition the boundary of a polyhedron into a small number of convex-like pieces, and an algorithm to partition a polyhedron of zero genus into tetrahedra. Moreover, issues on implementing the latter algorithm were discussed and an implementation in C was included. We consider below some interesting questions that arose from the above problems and can motivate future work.

For the problem of decomposing the boundary of the polyhedron into convex-like pieces, we assumed that we are allowed to cut through facets of the polyhedron. This has the disadvantage of introducing new vertices into the resulting decomposition. It would be of interest, therefore, to achieve a boundary decomposition into convex-like pieces by means of cuts along edges of the given polyhedron only. A different question is to find an algorithm that produces the minimum number of convex-like pieces. Is this problem NP-complete, as are many optimization questions in decompositions of three-dimensional objects?

Regarding the second problem of tetrahedralizing a polyhedron, we would like to extend our algorithm to polyhedra of nonzero genus. The only piece of the puzzle that is missing is an upper bound on the genus of a polyhedron in terms of the number of its reflex edges, as all lemmas of Chapter 3, with the exception of Lemma 3.3.5, apply to polyhedra of arbitrary genus. In fact, because of Lemma 3.3.4 and the pull-off process, we need only have an upper bound on the genus in terms of the number

of vertices of the polytope. For a polyhedron of n vertices and r reflex edges, we conjecture that its genus cannot exceed $r/3$. This may be difficult to prove, but an upper bound of n, which will translate into a bound of $4r$, should be more tractable.

Additionally, note that not all n-vertex polyhedra with r reflex edges necessitate $\Omega(n + r^2)$ tetrahedra to form a tetrahedralization. This motivates the following questions: Are there simple heuristics that one could use to guarantee that the tetrahedralization size does not exceed the minimum by more than a fixed constant in *all* cases? Is there a polynomial-time algorithm for such an approximation scheme? Are there preferred heuristics to keep the number of Steiner points as low as possible?

Finally, an interesting project would be to combine our implementation with a three-dimensional viewer in order to animate the algorithm. Preliminary results in this direction, where an experimental three-dimensional viewer written by David Dobkin and Steven North at Princeton University was used to animate the fence-off phase of our algorithm, showed that such an undertaking is worth pursuing.

Chapter 6

Bibliography

1. E. Arkin, R. Connelly, and J.S.B. Mitchell, On Monotone Paths Among Obstacles, with Applications to Planning Assemblies, *Proc. 5th Annual ACM Symposium on Computational Geometry* (1989), 334–343.
2. B. Aronov, and M. Sharir, Triangles in Space, or Building and Analyzing Castles in the Air, *Proc. 4th Annual ACM Symposium on Computational Geometry* (1988), 381–391.
3. T. Asano, and T. Asano, Partitioning Polygonal Regions into Trapezoids, *Proc. 24th Annual IEEE Symposium on Foundations of Computer Science* (1983), 233–241.
4. T. Asano, T. Asano, and H. Imai, Partitioning a Polygonal Region into Trapezoids, *Journal of the ACM* 33 (1986), 290–312.
5. C.L. Bajaj, and T.K. Dey, Robust Decompositions of Polyhedra, *Technical Report* (1989), Dept. of Computer Science, Purdue University.
6. B.S. Baker, E. Grosse, and C.S. Rafferty, Non-Obtuse Triangulation of Polygons, *Discrete and Computational Geometry* 3 (1988), 147–168.
7. B.G. Baumgart, A Polyhedron Representation for Computer Vision, *Proc. 1975 National Comput. Conference*, AFIPS Conference Proceedings, Vol. 44 (1975), AFIPS Press, Montvale, NJ, 589–596.
8. B. Chazelle, Convex Partitions of Polyhedra: A Lower Bound and Worst Case Optimal Algorithm, *SIAM Journal of Computing* 13 (1984), 488–507.

9. B. Chazelle, Approximation and Decomposition of Shapes, *Advances in Robotics* 1 (1987), (J.T. Schwartz and C.K. Yap, ed), Erlabaum, 145–185.

10. B. Chazelle, Triangulating a Simple Polygon in Linear Time, *Discrete and Computational Geometry* 6 (1991), 485–524.

11. B. Chazelle, and D.P. Dobkin, Optimal Convex Decompositions, *Computational Geometry*, North Holland, (1985), 63–133.

12. B. Chazelle, and L.J. Guibas, Fractional Cascading: I. A Data Structuring Technique; II. Applications, *Algorithmica* 1 (1986), 133–191.

13. B. Chazelle, and L.J. Guibas, Visibility and Intersection Problems in Plane Geometry, *Discrete and Computational Geometry* 4 (1989), 551–581.

14. B. Chazelle, L.J. Guibas, and D.T. Lee, The Power of Geometric Duality, *BIT* 25 (1985), 76–90.

15. B. Chazelle, and J. Incerpi, Triangulation and Shape-Complexity, *ACM Transactions on Graphics* 3 (1984), 135–152.

16. B. Chazelle, and L. Palios, Triangulating a Nonconvex Polytope, *Discrete and Computational Geometry* 5 (1990), 505–526.

17. L.P. Chew, Guaranteed-Quality Triangulation Meshes, *Technical Report*, Dept. of Computer Science, Cornell University.

18. K.L. Clarkson, R. Cole, and R.E. Tarjan, Randomized Parallel Algorithms for Trapezoidal Diagrams, *Proc. 7th Annual ACM Symposium on Computational Geometry* (1991), 152–165.

19. K.L. Clarkson, R.E. Tarjan, and C.J. Van Wyk, A Fast Las Vegas Algorithm for Triangulating a Simple Polygon, *Discrete and Computational Geometry* 4 (1989), 423–432.

20. T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms* (1990), McGraw Hill.

21. T.K. Dey, C.L. Bajaj, and K. Sugihara, *On Good Triangulations in Three Dimensions*, manuscript.

22. H. Edelsbrunner, *Algorithms in Combinatorial Geometry* (1987), Springer-Verlag.

23. H. Edelsbrunner, L.J. Guibas, and J. Stolfi, Optimal Point Location in a Monotone Subdivision, *SIAM Journal on Computing* 15 (1986), 317–340.

24. H. Edelsbrunner, J. O'Rourke, and R. Seidel, Constructing Arrangements of Lines and Hyperplanes with Applications, *Siam Journal on Computing* 15 (1986), 341–363.

25. H. Feng, and T. Pavlidis, Decomposition of Polygons into Simpler Components: Feature Generation for Synthetic Pattern Recognition, *IEEE Transactions on Computers* 24 (1975), 636–650.

26. D.A. Field, Implementing Watson's Algorithm in Three Dimensions, *Proc. 2nd Annual ACM Symposium on Computational Geometry* (1986), 246–259.

27. S. Fortune, and V. Milenkovic, Numerical Stability of Algorithms for Line Arranggements, *Proc. 7th Annual Symposium on Computational Geometry* (1991), 334–341.

28. A. Fournier, and D.Y. Montuno, Triangulating Simple Polygons and Equivalent Problems, *ACM Transactions on Graphics* 3 (1984), 153–174.

29. M.R. Garey, and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, 1979.

30. M.R. Garey, D.S. Johnson, F.P. Preparata, and R.E. Tarjan, Triangulating a Simple Polygon, *Information Processing Letters* 7 (1978), 175-179.

31. F. Gavril, Algorithms for a Maximum Clique and a Maximum Independent Set of a Circle Graph, *Networks* 3 (1973), 261–273.

32. D.H. Greene, *The Decomposition of Polygons into Convex Parts*, manuscript, Xerox PARC, 1982.

33. L.J. Guibas, J. Hershberger, D. Leven, M. Sharir, and R.E. Tarjan, Linear Time Algorithms for Visibility and Shortest Path Problems Inside Triangulated Simple Polygons, *Algorithmica* 2 (1987), 209–233.

34. L.J. Guibas, D. Salesin, and J. Stolfi, Epsilon Geometry: Building Robust Algorithms from Imprecise Computations, *Proc. 5th Annual Symposium on Computational Geometry* (1989), 208–217.

35. L.J. Guibas, and J. Stolfi, Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams, *ACM Transactions on Graphics* 4 (1985), 75–123.

36. S. Hertel, and K. Mehlhorn, Fast Triangulation of a Simple Polygon, *Lecture Notes on Computer Science* 158 (1983), 207–218.

37. C.M. Hoffman, J.E. Hopcroft, and M.S. Karasick, Towards Implementing Robust Geometric Computations, *Proc. 4th Annual Symposium on Computational Geometry* (1988), 106–117.

38. K. Hoffman, K. Mehlhorn, P. Rosenstiehl, and R.E. Tarjan, Sorting Jordan Sequences in Linear Time Using Level-Linked Search Trees, *Information and Control* 68 (1986), 170–184.

39. R. Kearfott, Interval arithmetic techniques in the computational solution of nonlinear systems of equations, *Lectures in Applied Mathematics* 26 (1990), 337–357.

40. J.M. Keil, *Decomposing Polygons into Simpler Components*, Ph.D. Thesis (1983), Dept. of Computer Science, University of Toronto.

41. J.M. Keil, Decomposing a Polygon into Simpler Components, *SIAM Journal on Computing* 14 (1985), 799–811.

42. D.G. Kirkpatrick, Optimal Search in Planar Subdivisions, *SIAM Journal on Computing* 12 (1983), 28–35.

43. D.G. Kirkpatrick, M.M. Klawe, and R.E. Tarjan, Polygon Triangulation in $O(n \log\log n)$ Time with Simple Data Structures, *Proc. 6th Annual ACM Symposium on Computational Geometry* (1990), 34–43.

44. A. Knight, J. May, J. McAffer, T. Nguyen, and J.-R. Sack, A Computational Geometry Workbench, *Proc. 6th Annual ACM Symposium on Computational Geometry* (1990), 370.

45. D.T. Lee, and F.P. Preparata, Location of a Point in a Planar Subdivision and its Applications, *SIAM Journal on Computing* 6 (1977), 594–606.

46. Z. Li, and V. Milenkovic, Constructing Strongly Convex Hulls Using Exact or Rounded Arithmetic, *Proc. 6th Annual Symposium on Computational Geometry* (1990), 235–243.

47. D. Lichtenstein, Planar Formulae and their Uses, *SIAM Journal on Computing* 11 (1982), 329–343.

48. A. Lingas, The Power of Non-Rectilinear Holes, *Lecture Notes in Computer Science* 140 (1982), 369–383.

49. A. Lingas, R. Pinter, R. Rivest, and A. Shamir, Minimum Edge Length Partitioning of Rectilinear Polygons, *Proc. 20th Annual Allerton Conference on Communication, Control, and Computing* (1982), 53–63.

50. W.S. Massey, *Algebraic Topology: An Introduction* (1977), Springer-Verlag.

51. K. Mehlhorn, *Data Structures and Algorithms*, Vol. 3 (1984), Springer-Verlag, Berlin.

52. V. Milenkovic, Calculating Approximate Curve Arrangements Using Rounded Arithmetic, *Proc. 5th Annual Symposium on Computational Geometry* (1989), 197–207.

53. R.E.Moore, *Methods and applications of interval analysis* (1979), SIAM.

54. S.P. Murdur, and P.A. Koparkar, Interval methods for processing geometric objects, *IEEE Computer Graphics and Applications* (1984), 7–17.

55. D.E. Muller, and F.P. Preparata, Finding the Intersection of two Convex Polyhedra, *Theoretical Computer Science* 7 (1978), 217–236.

56. J. O'Rourke, *Art Gallery Theorems and Algorithms*, Oxford University Press (1987).

57. J. O'Rourke, and K.J. Supowit, Some NP-Hard Polygon Decomposition Problems, *IEEE Transactions on Information Theory* IT-29 (1983), 181–190.

58. F.P. Preparata, and M.I. Shamos, *Computational Geometry, An Introduction* (1985), Springer-Verlag.

59. C.P. Rourke, and B.J. Sanderson, *Introduction to Piecewise-Linear Topology* (1982), Springer-Verlag.

60. J. Ruppert, and R. Seidel, On the Difficulty of Tetrahedralizing 3-Dimensional Non-Convex Polyhedra, *Proc. 5th Annual ACM Symposium on Computational Geometry* (1989), 380–392.

61. B. Schachter, Decomposition of Polygons into Convex Sets, *IEEE Transactions on Computers* 27 (1978), 1078–1082.

62. R. Seidel, *A Simple and Fast Incremental Randomized Algorithm for Computing Trapezoidal Decompositions and for Triangulating Polygons*, manuscript.

63. J. Stillwell, *Classical Topology and Combinatorial Group Theory* (1980), Springer-Verlag.

64. S. Suri, A Linear Time Algorithm for Minimum Link Paths inside a Simple Polygon, *Computer Vision, Graphics, and Image Processing* 35 (1986), 99–110.

65. R.E. Tarjan, and C.J. Van Wyk, An $O(n \log\log n)$-time Algorithm for Triangulating a Simple Polygon, *SIAM Journal on Computing* 17 (1988), 143–178.

66. G.T. Toussaint, Pattern Recodnition and Geometrical Complexity, *Proc. 5th International Conference on Pattern Recognition* (1980), 1324–1347.

67. G.T. Toussaint, On Separating two Simple Polygons by a Single Translation, *Discrete and Computational Geometry* 4 (1989), 265–278.

68. G.T. Toussaint, and D. Avis, On a Convex Hull Algorithm for Polygons and its Application to Triangulation Problems, *Pattern Recognition* 15 (1982), 23–29.

Chapter 7

Appendix

Index of File Names

(in order of appearance in the appendix)

File Name	*Number of Pages*
include.h	1
defs.h	1
structs.h	2
macros.h	1
main.c	6
utils.c	5
init.c	1
triangulate.c	8
dualize.c	11
pull_off.c	6
fence.c	3
heap.c	2
regions.c	6
shatter.c	4
tetrahedra.c	2

Function Index (in alphabetical order)

Function Name	*File Name*	*Page Number*
adj_lowchain()	triangulate.c	4
adj_upchain()	triangulate.c	3
advance()	triangulate.c	4
all_segs_dump()	main.c	2
build_tree()	dualize.c	9
chain_merge()	regions.c	5
chain_split()	regions.c	6
chainbottom_insert()	regions.c	5
chaintop_insert()	regions.c	5
check_orientation()	triangulate.c	2
check_p()	dualize.c	3
clean_regions()	regions.c	6
cleanup_heap()	heap.c	1
collect_4hedron()	pull_off.c	1
collect_segments()	fence.c	1
collinear()	utils.c	2
compute_fence()	fence.c	2
compute_intersection()	utils.c	4
del_region()	regions.c	4
disconnect()	dualize.c	8
downheap()	heap.c	1
du_clean()	dualize.c	10
dualize()	dualize.c	9
endp_print()	shatter.c	1
equiv_name()	dualize.c	7
exhaust()	dualize.c	10
extract_tetrahedra()	tetrahedra.c	1
fence_parts()	fence.c	1

Function Name	*File Name*	*Page Number*
flat_remove()	pull_off.c	2
flip_diagonal()	pull_off.c	1
fr_cascade()	dualize.c	6
from_x_comp_z()	utils.c	3
from_y_comp_z()	utils.c	3
get_arrangmnt()	dualize.c	4
get_e_rec()	dualize.c	3
get_edge_pair()	dualize.c	3
get_list_rec()	regions.c	2
get_node()	triangulate.c	2
get_reg_rec()	regions.c	2
get_tables()	dualize.c	3
get_values()	pull_off.c	1
get_vertex()	dualize.c	3
heap_delete()	heap.c	1
heap_insert()	heap.c	1
heap_size()	heap.c	1
heap_top()	heap.c	1
in()	triangulate.c	6
in_region()	shatter.c	2
init_polyhedron()	init.c	1
init_regions()	regions.c	1
init_segheap()	heap.c	2
ins_region()	regions.c	4
intersection()	utils.c	4
left_rotate()	regions.c	2
locate()	triangulate.c	6
locate_reg()	dualize.c	7
low_witness()	dualize.c	3
lowchain_insert()	triangulate.c	4

Function Name	*File Name*	*Page Number*
main()	main.c	5
merge()	dualize.c	7
merge_chain()	triangulate.c	7
merge_left()	dualize.c	5
merge_right()	dualize.c	6
mult_ins_regions()	shatter.c	1
new_e()	main.c	2
new_f()	main.c	2
new_region()	regions.c	2
on_boundary()	shatter.c	2
phase1()	main.c	4
phase2()	main.c	5
plane_coef_compute()	utils.c	3
pointed_remove()	pull_off.c	5
pr_edge()	main.c	1
pr_facet()	main.c	1
pr_final_arrang()	dualize.c	2
pr_old_facet()	main.c	1
pr_polyhedron()	main.c	2
pr_regions()	triangulate.c	1
pr_special()	triangulate.c	1
pr_temp_arrang()	dualize.c	1
pr_vertex()	main.c	1
preprocess()	shatter.c	1
print()	regions.c	1
print_cascade()	dualize.c	2
print_crown()	pull_off.c	1
print_layers()	dualize.c	2
print_reg_table()	dualize.c	2
print_up_envelope()	dualize.c	1

Function Name	*File Name*	*Page Number*
print_vertices()	dualize.c	2
printtable()	fence.c	1
punct_collect()	main.c	3
punct_comp()	main.c	3
punct_print()	main.c	2
query()	dualize.c	10
rb_del_fix()	regions.c	4
rb_ins_fix()	regions.c	3
reflex()	main.c	2
reg_locate()	regions.c	3
region_delete()	triangulate.c	6
region_insert()	triangulate.c	6
reglist_print()	regions.c	1
regtree_print()	regions.c	1
repair()	pull_off.c	4
report()	fence.c	2
report_4hedra()	main.c	3
right_rotate()	regions.c	3
seg_compare()	utils.c	1
seg_line_clip()	utils.c	4
seg_print()	shatter.c	1
seg_seg_clip()	utils.c	3
segheap_compare()	heap.c	2
select_funcs()	triangulate.c	2
set_e()	utils.c	5
set_f()	utils.c	5
shatter()	shatter.c	3
sp_collect()	triangulate.c	3
sp_compare()	triangulate.c	2
sp_process()	triangulate.c	7

Function Name	*File Name*	*Page Number*
split_arrang()	dualize.c	8
split_chain()	triangulate.c	7
test_match()	tetrahedra.c	1
tetrahedralize()	tetrahedra.c	1
tri_dump()	tetrahedra.c	1
triang_compare()	tetrahedra.c	1
triangle_report()	regions.c	6
triangulate()	triangulate.c	8
up_witness()	dualize.c	3
upchain_insert()	triangulate.c	4
upd_bottomchain()	regions.c	5
upd_save()	dualize.c	8
upd_topchain()	regions.c	5
upheap()	heap.c	1
vert_rel_pos()	utils.c	3
visit()	dualize.c	10
wrt_boundr()	regions.c	2
wrt_region()	regions.c	2
x_v_diff()	utils.c	2
xy_ep_angle()	utils.c	2
xy_ep_compare()	utils.c	1
xy_p_compare()	utils.c	1
xy_v_angle()	utils.c	2
xyz_v_compare()	utils.c	1
xz_ep_compare()	utils.c	1
xz_v_angle()	utils.c	2
y_v_diff()	utils.c	2
yz_ep_compare()	utils.c	1
yz_v_angle()	utils.c	2
z_v_diff()	utils.c	2

```
/*   include.h   */

#include <stdio.h>
#include <math.h>
#include "defs.h"
#include "structs.h"
#include "macros.h"
```

```
/*   defs.h   */


#define DIAGNOSTICS

#define DEBUG

#define XALLDEBUG

/* define DEBUG, in case ALLDEBUG is defined */
#ifdef ALLDEBUG
#define DEBUG
#endif


#define EPSILON 0.000001

#define D 12


#define SMALL_SIZE   8          /* table sizes */
#define MEDIUM_SIZE  16
#define LARGE_SIZE   32
```

```
/*   structs.h   */


typedef struct vertex_type {
                int                     degree, type;
                double                  x, y, z;
                struct facet_type       *facet;
} Vertex;

typedef struct edge_type {
                int                     type;
                         /* type = 0 if edge is garbage, 1 otherwise */
                double                  a, b, c;
                        /* coefs of edge projection on xy-plane; b is {0,-1} */
                struct vertex_type      *vertex[2];
                             /*  vertices 0 and 1 are in lexigographic order  */
                struct facet_type       *facet[2];
} Edge;

typedef struct facet_type {
                int                     sign;
                                /* sign = 0, if facet is garbage           */
                                /*        1, if polyhedron is above facet  */
                                /*       -1, otherwise                     */
                double                  a, b, c, d;
                               /* coefs of facet's supporting plane;  c >= 0 */
                struct vertex_type      *vertex[3];
                struct edge_type        *edge[3];
} Facet;

typedef struct v_bord_type {
                struct vertex_type      *v;
                struct edge_type        *prev_e, *next_e;
} BoundVertex;

typedef struct bord_type {
                int                     size;   /* length of boundary */
                struct v_bord_type      *first; /* ptr to 1st vertex */
} Boundary;

typedef struct old_facet_type {
                             /* sign is similar to sign in facet_type   */
                int                     sign, bound_num, vertex_num;
                double                  a, b, c, d;
                struct bord_type        *boundary;
} OldFacet;

typedef struct polyhedron {
                int                     vertex_num, edge_num, facet_num;
                struct vertex_type      *vertices;
                struct edge_type        *edges;
                struct facet_type       *facets;
                struct old_facet_type   *old_facets;
                struct v_bord_type      *v_list;
                struct bord_type        *border;
} Polyhedron;



typedef struct point_2d {
                double          x, y;
} Point_2d;

typedef struct point_3d {
                double          x, y, z;
} Point_3d;

typedef struct punct_type {
                int                     type, up_i, up_max, low_i, low_max;
                double                  x, y, z, z_above, z_below;
                struct facet_type       **up_f;
                struct facet_type       **low_f;
} Puncturing;

typedef struct endpoint_type {
                int                     kind, seg_no, facet_no;
                                /*  "kind"= 0 if left endpoint, 1 if right  */
                double                  x, y, z;
} Endpoint;

typedef struct line_segment {
                int                     facet_no;
                struct point_3d         endpoint[2];
} LineSegment;

typedef struct triang_type {
                struct point_3d         p[3];
} Triangle;

typedef struct tetrahedr_type {
                struct point_3d         p[4];
} Tetrahedron;

typedef struct table_head {
                int             n, max; /* current and maximum available size */
                char            *to_elem;          /* pointer to actual array */
} Table;



/* for the dual tree of puncturing vertices */

typedef struct du_reg_type {
                union {
                        int                     dad;
                        struct du_reg_type      *ptr;
                } d[2];
} RegEntry;

typedef struct du_t_root {
                int                     l_num, p_num;
                double                  x_min, x_max;
                struct du_reg_type      *table;
                struct du_t_node        *t_node;
} DualTreeRoot;

typedef struct du_t_node {
                double                  x_maxleft, x_minright;
                struct du_t_node        *left, *right;
} DualTreeNode;



/* for the shattering process */

typedef struct tree_node{
                int                     color;
                struct region_type      *reg_ptr;
                struct tree_node        *parent, *left, *right;
```

```
} rbTreeNode;

typedef struct chain_node {
                struct endpoint_type    *p;
                struct chain_node       *previous, *next;
} ChainNode;

typedef struct region_type {
                struct endpoint_type    *up_p, *low_p;
                                        /* pointers to points array */
                struct chain_node       *rightmost, /* ptr to rightmost point */
                                        *top,       /* ptrs to top and bottom */
                                        *bottom;   /* of a doubly linked list */
                struct region_type      *next_reg, *prev_reg;
                struct tree_node        *tnode_ptr;
} Region;
```

```
/*    macros.h   */


#define E_COMPARE(x, y) \
        ( ((x) > (y) + EPSILON) - ((x) < (y) - EPSILON) )



#define MY_ALLOC(ptr, str, n) \
        ( (((ptr) = (str *) malloc((n) * sizeof(str))) == NULL) ? \
                                                ERROR("...out of memory") : 0 )

#define MY_CALLOC(ptr, str, n) \
        ( (((ptr) = (str *) calloc((n), sizeof(str))) == NULL) ? \
                                                ERROR("...out of memory") : 0 )

#define MY_REALLOC(ptr, str, n) \
        ( (((ptr) = (str *)realloc(ptr, (n) * sizeof(str))) == NULL) ? \
                                                ERROR("...out of memory") : 0 )



#define MAKE_POINT_2(p,xc,yc) \
        ( (p).x = (xc),  (p).y = (yc) )

#define COPY_POINT_2(p,q) \
        ( (p).x = (q).x,  (p).y = (q).y )

#define MAKE_POINT_3(p,xc,yc,zc) \
        ( (p).x = (xc), (p).y = (yc), (p).z = (zc) )

#define COPY_POINT_3(p,q) \
        ( (p).x = (q).x, (p).y = (q).y, (p).z = (q).z )

#define MAKE_MIDPOINT_3(p, a, b) \
        ( (p).x = ((a).x + (b).x) / 2.0, \
          (p).y = ((a).y + (b).y) / 2.0, \
          (p).z = ((a).z + (b).z) / 2.0 )



#define EVAL_2(p, q) \
        ( (p).x * (q).a + (p).y * (q).b  + (q).c )

#define EVAL_3(p, q) \
        ( (p)->x * (q).a + (p)->y * (q).b  + (p)->z * (q).c + (q).d )



#define FROM_x_COMP_z(xval, q1, q2) \
        ( (E_COMPARE(xval, q1.x) == 0) ? (q1.z) : \
                (E_COMPARE(xval, q2.x) == 0) ? (q2.z) : \
                        (q1.z + (xval - q1.x) * (q2.z - q1.z) / (q2.x - q1.x)) )

#define FROM_y_COMP_z(yval, q1, q2) \
        ( (E_COMPARE(yval, q1.y) == 0) ? (q1.z) : \
                (E_COMPARE(yval, q2.y) == 0) ? (q2.z) : \
                        (q1.z + (yval - q1.y) * (q2.z - q1.z) / (q2.y - q1.y)) )

#define FROM_FACET_COMP_z(p, f) \
        ( -((f).a * (p).x + (f).b * (p).y + (f).d) / (f).c )
```

```
#define ERROR(string) \
        ( printf("%s.\n", (string)),   exit(1) )



#define MOD(n, base) \
        ( ((n + base) % base) )



/***   general-purpose table handling   ***/

#define MAKE_TABLE(table, str, size) \
        { str *element; \
          MY_CALLOC(element, str, (table)->max = (size)); \
          (table)->n = 0; \
          (table)->to_elem = (char *)element; \
        }


        /* if array full: double maximum table size and reallocate space */
#define INS_TABLE(entry, str, table) \
        { str *element; \
          element = (str *)(table)->to_elem; \
          if ((table)->n == (table)->max){ \
                (table)->max *= 2; \
                MY_REALLOC(element, str, (table)->max); \
                (table)->to_elem = (char *)element; \
          } \
          element[(table)->n++] = (entry); \
        }
```

```
   /*********                   main.c                            *********/
   /*    The core of Chazelle & Palios' tetrahedralization algorithm    */
   /*  After the polyhedron has been read in (init_polyhedron), its     */
   /*  boundary is triangulated (triangulate).  Next the pull-off phase */
   /*  is launched (phase1), followed by the fence-off phase (phase2).  */


#include "include.h"


void            triangulate(), dualize(), du_clean();


int             e_count, e_no_max, f_count, f_no_max, reflex_no;
Polyhedron      P;
Table           tetrhdr_table, *segs, triangles;



#ifdef DEBUG
/*****************************************************************************/

void    pr_vertex(v, f_table)
Vertex          *v;
Facet           *f_table;
{
        fprintf(stderr, "  (%.2f, %.2f, %.2f)", v->x, v->y, v->z);
        if (v->type == 0)
                fprintf(stderr, " has been pulled off\n");
        else{
                fprintf(stderr," of type %d and degree %d", v->type, v->degree);
                if (v->facet == NULL)
                        fprintf(stderr, "  points to NULL facet\n");
                else
                        fprintf(stderr, "  points to facet %d\n",
                                                v->facet - f_table);
        }
}


void    pr_edge(e, v_table, f_table)
Edge            *e;
Vertex          *v_table;
Facet           *f_table;
{
        if (e->type == 0){
                fprintf(stderr, "  garbage\n");
                return;
        }
        if (e->vertex[0] == NULL)
                fprintf(stderr, "  V= [NULL,");
        else
                fprintf(stderr, "  V= [%d,", e->vertex[0] - v_table);
        if (e->vertex[1] == NULL)
                fprintf(stderr, " NULL],");
        else
                fprintf(stderr, " %d],", e->vertex[1] - v_table);
        if (e->facet[0] == NULL)
                fprintf(stderr, "  F= [NULL,");
        else
                fprintf(stderr, "  F= [%d,", e->facet[0] - f_table);
        if (e->facet[1] == NULL)
                fprintf(stderr, " NULL],");
        else
                fprintf(stderr, " %d],", e->facet[1] - f_table);
        fprintf(stderr, "  (a=%.2f, b=%.2f, c=%.2f)\n", e->a, e->b, e->c);
}


void    pr_facet(f, v_table, e_table)
Vertex          *v_table;
Edge            *e_table;
Facet           *f;
{
        if (f->sign == 0)
                fprintf(stderr, "  garbage\n");
        else{
                if (f->vertex[0] == NULL)
                        fprintf(stderr, "  V= [NULL,");
                else
                        fprintf(stderr, "  V= [%d,", f->vertex[0] - v_table);
                if (f->vertex[1] == NULL)
                        fprintf(stderr, " NULL,");
                else
                        fprintf(stderr, " %d,", f->vertex[1] - v_table);
                if (f->vertex[2] == NULL)
                        fprintf(stderr, " NULL],");
                else
                        fprintf(stderr, " %d],", f->vertex[2] - v_table);
                if (f->edge[0] == NULL)
                        fprintf(stderr, "  E= [NULL,");
                else
                        fprintf(stderr, "  E= [%d,", f->edge[0] - e_table);
                if (f->edge[1] == NULL)
                        fprintf(stderr, " NULL,");
                else
                        fprintf(stderr, " %d,", f->edge[1] - e_table);
                if (f->edge[2] == NULL)
                        fprintf(stderr, " NULL],");
                else
                        fprintf(stderr, " %d],", f->edge[2] - e_table);
                fprintf(stderr, "  sgn=%d,  (a=%.2f, b=%.2f, c=%.2f, d=%.2f)\n",
                                        f->sign, f->a, f->b, f->c, f->d);
        }
}


void    pr_old_facet(f, v_table, e_table)
Vertex          *v_table;
Edge            *e_table;
OldFacet        *f;
{
        int             j, k;
        Boundary        *bp;
        BoundVertex     *vlp;

        fprintf(stderr,"\t(a=%.4f, b=%.4f, c=%.4f, d=%.4f)\n",
                                                f->a, f->b, f->c, f->d);
        for (j = f->bound_num, bp = f->boundary;  j > 0;  j--, bp++){
                fprintf(stderr,"\tBorder line index= %d,  size= %d\n\t",
                                                bp - P.border, bp->size);
                for (k = bp->size, vlp = bp->first;  k > 0;  k--, vlp++){
                        fprintf(stderr,"  {[%d],(%d),[%d]}",vlp->prev_e-e_table,
                                        vlp->v - v_table, vlp->next_e -e_table);
                }
                fprintf(stderr,"\n");
        }
        fprintf(stderr,"\tTotal number of vertices= %d\n", f->vertex_num);
}
```

```
void    pr_polyhedron(Pol, new)
int                     new;    /* 'new' selects old or new facets */
Polyhedron              Pol;
{
        int             i;
        Vertex          *vp, *v_table = Pol.vertices;
        Edge            *ep, *e_table = Pol.edges;
        Facet           *new_fp, *f_table = Pol.facets;
        OldFacet        *old_fp, *old_f_table = Pol.old_facets;

        fprintf(stderr, "\n\tTable of vertices\n");
        for (i = 0, vp = v_table;  i < Pol.vertex_num;  i++, vp++){
                fprintf(stderr, "vertex %d:", i);
                pr_vertex(vp, f_table);
        }

        fprintf(stderr, "\n\tTable of edges\n");
        for (i = 0, ep = e_table;  i < Pol.edge_num;  i++, ep++){
                fprintf(stderr, "edge %d:", i);
                pr_edge(ep, v_table, f_table);
        }

        if (new){
                fprintf(stderr, "\n\tTable of facets\n");
                for (i = 0, new_fp = Pol.facets;  i < Pol.facet_num;
                                                        i++, new_fp++){
                        fprintf(stderr, "facet %d:", i);
                        pr_facet(new_fp, v_table, e_table);
                }
        }
        else{
                fprintf(stderr, "\n\tTable of old facets\n");
                for (i = 0, old_fp = old_f_table;  i < Pol.facet_num;
                                                        i++, old_fp++){
                        fprintf(stderr, "facet %d:", i);
                        pr_old_facet(old_fp, v_table, e_table);
                }
        }
}


void    punct_print(punct_array)
Table           *punct_array;
{
        int             i;
        Puncturing      *p;

        fprintf(stderr, "\n\tTable of puncturing vertices\n");
        for (i = punct_array->n, p = (Puncturing *)(punct_array->to_elem);
                                                i > 0;  i--, p++)
                fprintf(stderr, "point (%.3f, %.3f, %.3f)\n", p->x,p->y,p->z);
}


void    all_segs_dump(seg_table, size)
                /* print all segments  (for debugging purposes) */
int             size;
Table           *seg_table;
{
        int             i, j;
        Table           *table;
        LineSegment     *s;

        for (i = 0, table = seg_table;  i < size;  i++, table++){
                fprintf(stderr, "facet# %d:\n", i);
                s = (LineSegment *)table->to_elem;
                for (j = table->n;  j > 0;  j--, s++){
                        fprintf(stderr, "\tfrom facet %d :", s->facet_no);
                        fprintf(stderr,"(%.3f, %.3f, %.3f)-(%.3f, %.3f, %.3f)\n",
                          s->endpoint[0].x, s->endpoint[0].y, s->endpoint[0].z,
                          s->endpoint[1].x, s->endpoint[1].y, s->endpoint[1].z);
                }
        }
}

/*****************************************************************************/
#endif



Edge    *new_e()
{
        if (++e_count >= e_no_max)
                ERROR("new_e> not enough edge records");
        return(P.edges + e_count);
}


Facet   *new_f()
{
        if (++f_count >= f_no_max)
                ERROR("new_f> not enough facet records");
        return(P.facets + f_count);
}


int     reflex(e)
        /* returns: 0 if edge is non-reflex,                            */
        /*          1 if edge is reflex & polyhedron is above it,       */
        /*          2 if edge is reflex & polyhedron is above & below   */
        /*          3 if edge is reflex & polyhedron is below it.       */
        /*** WE TAKE ADVANTAGE OF THE ASSUMPTION THAT facet->c >= 0 ! ***/
Edge            *e;
{
        int             i, term;
        double          temp;
        Vertex          *v, *v1, *v2;
        Facet           *f, *g;

        f = e->facet[0];        g = e->facet[1];
        v1 = e->vertex[0];      v2 = e->vertex[1];
        if (E_COMPARE(f->c, 0.0) == 0){
                /* note the ASSUMPTION: a reflex edge can not be vertical */
                if (E_COMPARE(g->c, 0.0) != 0){  /* g not vertical */
                        /* locate vertex v of facet f not incident upon e */
                        for (i = 3;  --i >= 0; )
                                if ((v = f->vertex[i]) != v1 && v != v2)
                                        break;
                        temp = EVAL_3(v, *g);
                        if (E_COMPARE(temp, 0.0) * g->sign == -1)
                                return(2 - g->sign);
                }
        }
        else{           /* f is not vertical */
                if ((term = -f->sign) == g->sign)
                        term = 0;
                /* locate vertex v of facet g that is not incident upon e */
                for (i = 3;  --i >= 0; )
```

```
                        if ((v = g->vertex[i]) != v1 && v != v2)
                                break;
                temp = EVAL_3(v, *f);
#ifdef DIAGNOSTICS
                if (term == 0 && E_COMPARE(temp, 0.0) == 0)
                        /* polyhedron on different sides of coplanar f and g */
                        ERROR("reflex> inconsistent polyhedron");
#endif
                if (E_COMPARE(temp, 0.0) * f->sign == -1)
                        return(2 + term);
        }
#ifdef DEBUG
        fprintf(stderr,"edge (%.3f,%.3f,%.3f)-(%.3f,%.3f,%.3f) is NOT reflex\n",
                                v1->x, v1->y, v1->z, v2->x, v2->y, v2->z);
#endif
        return(0);
}


int     punct_comp(elem1, elem2)
char    *elem1, *elem2;
{
        int             cond;
        Puncturing      *p1, *p2;

        p1 = (Puncturing *)elem1;
        p2 = (Puncturing *)elem2;
        if ((cond = E_COMPARE(p1->x, p2->x)) == 0)
                return( E_COMPARE(p2->y, p1->y) );
        return(cond);
}


void    punct_collect(edge_num, punct_array)
int             edge_num;
Table           *punct_array;
 /* collect puncturing vertices (reflex vertices & midpoints of reflex edges) */
{
        int                     i, type;
        Vertex          *v1, *v2;
        Edge            *e;
        Puncturing      punct;

        punct.up_i = punct.low_i = 0;
        punct.up_max = punct.low_max = 4;
        /* process each edge in turn, testing whether it is reflex */
        for (i = edge_num, e = P.edges, reflex_no = 0;  --i >= 0;  e++)
                if (type = reflex(e)){          /* reflex edge */
                        reflex_no++;

                        /* A reflex vertex needs up-witnesses if there is an */
                        /*  is an incident reflex edge that needs "up-fence" */
                        /* (NOT needed that ALL such edges require up-fences)*/
                        /* The case is similar for the down-witnesses   */
                        switch ((v1 = e->vertex[0])->type){
                        case  1: /* vertex not yet marked as reflex */
                                 v1->type = -type;
                                 break;
                        case -1: /* vertex already needs up witness */
                                 if (type == 3)
                                         v1->type = -2;
                                 break;
                        case -2: /* vertex already needs up & down witnesses */
                                        /* do nothing */
                                 break;
                        case -3: /* vertex alreaady needs down witness */
                                 if (type == 1)
                                        v1->type = -2;
                                 break;
                        }
                        switch ((v2 = e->vertex[1])->type){
                        case  1: /* vertex not yet marked as reflex */
                                 v2->type = -type;
                                 break;
                        case -1: /* vertex already needs up witness */
                                 if (type == 3)
                                        v2->type = -2;
                                 break;
                        case -2: /* vertex already needs up & down witnesses */
                                        /* do nothing */
                                 break;
                        case -3: /* vertex alreaady needs down witness */
                                 if (type == 1)
                                        v2->type = -2;
                                 break;
                        }
#ifdef DEBUG
        fprintf(stderr,"edge (%.3f,%.3f,%.3f)-(%.3f,%.3f,%.3f) is REFLEX (%d)\n",
                                v1->x, v1->y, v1->z, v2->x, v2->y, v2->z, type);
#endif
                        /* A reflex edge midpoint needs up-witnesses if the */
                        /*      reflex edge needs "up_fence"            */
                        /* The case is similar for the down-witnesses   */
                        MAKE_MIDPOINT_3(punct, *v1, *v2);
                        punct.type = type;
                        MY_ALLOC(punct.up_f, Facet *, 4);
                        MY_ALLOC(punct.low_f, Facet *, 4);
                        INS_TABLE(punct, Puncturing, punct_array);
                }
        for (i = P.vertex_num, v1 = P.vertices;  --i >= 0;  v1++)
                if ((type = v1->type) < 0 && type >= -3){
                        COPY_POINT_3(punct, *v1);
                        punct.type = type;
                        MY_ALLOC(punct.up_f, Facet *, 4);
                        MY_ALLOC(punct.low_f, Facet *, 4);
                        INS_TABLE(punct, Puncturing, punct_array);
                }
}


void    report_4hedra()
{
        int             i, j;
        Point_3d        *p;
        Tetrahedron     *t;

        printf("\n\n\tTetrahedra Produced  (total number = %d)\n\n",
                                                        i = tetrhdr_table.n);
        for (t = (Tetrahedron *)(tetrhdr_table.to_elem);  --i >= 0;  t++){
                for (j = 3, p = t->p;  --j >= 0;  p++)
                        printf("(%.2f, %.2f, %.2f)-", p->x, p->y, p->z);
                printf("(%.2f, %.2f, %.2f)\n", p->x, p->y, p->z);
        }
}


void    phase1(m)
        /* pull-off phase:  m is the number of edges of initial polyhedron */
int     m;
{
```

```
        int             i, j, nv;
        Vertex          *v, **good_v;
        Edge            *e;
        Facet           *f, *g;
        Puncturing      *p;
        Table           punct_table;
        DualTreeRoot    dt_root;

        /* allocate space for the puncturing vertices; initialize to 0  */
        MAKE_TABLE(&punct_table, Puncturing, MEDIUM_SIZE);
        /* collect puncturing vertices processing the first "m" edges   */
        /* In this way, we avoid dealing with the nonreflex edges that  */
        /* are introduced by the triangulation */
        punct_collect(m, &punct_table);
        if ((i = punct_table.n) > 0){
#ifdef DEBUG
                punct_print(&punct_table);
#endif
                /* sort puncturing vertices, or in dual space lines */
                /* by increasing slope & decreasing offset (for equal slopes) */
                qsort(punct_table.to_elem, i, sizeof(Puncturing), punct_comp);

                /* compute tree of dual images of puncturing vertices */
                dualize(&dt_root, &punct_table);

                /* find the witness facets */
                for (j = P.facet_num, f = P.facets;  --j >= 0;  f++)
                        if (E_COMPARE(f->c, 0.0) != 0)
                                query(f, &dt_root);     /* non-vertical facet */
                du_clean(&dt_root);

                /* mark the witness vertices */
                for (p = (Puncturing *)punct_table.to_elem;  --i >= 0;  p++){
                        for (j = p->up_i;  --j >= 0;  ){
                                f = p->up_f[j];
                                f->vertex[0]->type = f->vertex[1]->type
                                        = f->vertex[2]->type = -5;
                        }
                        for (j = p->low_i;  --j >= 0;  ){
                                f = p->low_f[j];
                                f->vertex[0]->type = f->vertex[1]->type
                                        = f->vertex[2]->type = -5;
                        }
                        (void) free((char *)p->up_f);
                        (void) free((char *)p->low_f);
                }
        }
        printf("\nWitness identification is complete.\n");
        (void) free(punct_table.to_elem);
        MAKE_TABLE(&tetrhdr_table, Tetrahedron, LARGE_SIZE);

        /* collect the good vertices */
        i = m = nv = P.vertex_num;
        MY_CALLOC(good_v, Vertex *, ++m);
        for (v = P.vertices, j = 0;  --i >= 0;  v++)
                if (v->type > 0 && v->degree <= D){
                        /* vertex is not reflex or witness & has small degree */
#ifdef DEBUG
                        fprintf(stderr, "good vertex  <%d>\n", v - P.vertices);
#endif
                        v->type = 2;    /* mark vertex "in-list" */
                        good_v[j] = v;
                        j = MOD(j + 1, m);
                }
        if (j != 0){            /* process the good vertices */
                int             k, first = 1;
                double          temp;
                Vertex          *v1, *v2, *v3;
                Edge            *e2;
                Facet           *f_sentnl;

                for (i = 0;  i != j;  i = MOD(i + 1, m)){
                        if ((v = good_v[i])->degree > D || v->type == 3)
                                continue;
#ifdef DEBUG
                        pr_polyhedron(P, 1);
#endif
                        /* find an edge incident upon the vertex good_v[i] */
                        for (f= v->facet, k= 3;  --k >= 0 && (e= f->edge[k]); ){
                                                /* NOTE:  YES, ASSIGNMENT */
                                if ((v3 = e->vertex[0]) == v){
                                        v3 = e->vertex[1];
                                        break;
                                }
                                else
                                        if (e->vertex[1] == v)
                                                break;
                        }
#ifdef DIAGNOSTICS
                       if (k < 0)
                                ERROR("phase1> illegal vertex-to-facet pointer");
#endif

                        f_sentnl = f;
                        do{   /* move around to find two non-coplanar facets */
                                g = f;    v2 = v3;    e2 = e;
                                if ((f = e2->facet[0]) == g)
                                        f = e2->facet[1];
                                for (e = f->edge[k = 2];
                                     e->vertex[0] == v2 || e->vertex[1] == v2; )
                                        e = f->edge[--k];
                                if ((v3 = e->vertex[0]) == v)
                                        v3 = e->vertex[1];
                                temp = EVAL_3(v3, *f_sentnl);
                        } while (E_COMPARE(temp, 0.0) == 0 && f != f_sentnl);
                        if (f == f_sentnl)      /* flat vertex */
                                flat_remove(v, v2, v3, e, f, good_v, &j, m);
                        else{
                                if (first){     /* mark v & incident vertices */
                                  /* dirty trick ensuring that the polyhedron */
                                  /* will not be completely tetrahedralized   */
                                  /* at the pull-off phase                    */
                                        v->type = 3;
                                        first = 0;
                                        for ((v1 = v2)->type = 3;  v3 != v1;
                                                v3 = v2, e = e2, f = g){
                                                v3->type = 3;
                                                if ((g = e->facet[0]) == f)
                                                        g = e->facet[1];
                                                for (e2 = g->edge[k = 2];
                                                          e2->vertex[0] == v3 ||
                                                          e2->vertex[1] == v3; )
                                                        e2 = g->edge[--k];
                                                if ((v2 = e2->vertex[0]) == v)
                                                        v2 = e2->vertex[1];
                                        }
                                }
                                else
                                        pointed_remove(v, v2, e2, f, good_v,
                                                                        &j, m);
```

```
                }
        }

        /* compact facet table; don't compact edge, vertex tables */
        for (g = (f = P.facets) + P.facet_num;  (--g)->sign == 0;  )
                ;       /* skip garbage facets at end of table */
        for (;  f < g;  f++){
                if (f->sign == 0){      /* garbage */
                        for (i = 3;  --i >= 0;  )
                                if ((v = g->vertex[i])->facet == g)
                                        v->facet = f;
                        for (i = 3;  --i >= 0;  )
                                if ((e = g->edge[i])->facet[0] == g)
                                        e->facet[0] = f;
                                else
                                        e->facet[1] = f;
                        *f = *g;
                        g->sign = 0;
                        while ((--g)->sign == 0)
                                ;       /* skip garbage facets */
                }
        }
        P.facet_num = g - P.facets + 1;
    }
    (void) free((char *)good_v);
}


void    phase2()
{
    int             i, k, cond, e_parallel_yz;
    Edge            *eptr;
    Facet           *fptr;
    Table           *s, above, below;
    LineSegment     lseg;

    MY_ALLOC(segs, Table, P.facet_num);
    for (k = 0, s = segs;  k < P.facet_num;  k++, s++)
        MAKE_TABLE(s, LineSegment, SMALL_SIZE);

    for (k = 0, eptr = P.edges;  k < P.edge_num;  k++, eptr++){
        if (eptr->type == 0)
            continue;       /* garbage edge */
#ifdef DEBUG
        fprintf(stderr, "\nCompute fence of edge %d.\n", k);
#endif

        if ((cond = fence_parts(eptr)) <= 1){
            /* allocate space for upper & lower fence, if needed */
            if (cond >= 0){         /* need upper fence */
                MAKE_TABLE(&above, LineSegment, 2);
            }
            if (cond <= 0){         /* need lower fence */
                MAKE_TABLE(&below, LineSegment, 2);
            }
            e_parallel_yz = (E_COMPARE(eptr->b, 0.0) == 0);

            collect_segments(eptr, cond, &above, &below);
            if (cond >= 0){         /*  UPPER  FENCE  */
#ifdef DEBUG
                fprintf(stderr, "Upper part of fence:\n");
#endif
                compute_fence(1, &above, e_parallel_yz);
                (void) free(above.to_elem);
            }
            if (cond <= 0){         /*  LOWER  FENCE  */
#ifdef DEBUG
                fprintf(stderr, "Lower part of fence:\n");
#endif
                compute_fence(-1, &below, e_parallel_yz);
                (void) free(below.to_elem);
            }
        }
#ifdef DEBUG
        else
            fprintf(stderr, "needs NO fence at all\n");
#endif
    }
    printf("\nComputation of the fences is complete.\n");
#ifdef DEBUG
    all_segs_dump(segs, P.facet_num);
#endif

    /* shatter facets */
    MAKE_TABLE(&triangles, Triangle, LARGE_SIZE);
    for (k= 0, s= segs, fptr= P.facets; k < P.facet_num; k++, s++, fptr++){
        if (E_COMPARE(fptr->c, 0.0) == 0)
            continue;       /* don't bother about vertical facets */

        /* include the incident edges of each facet in its segs array */
        lseg.facet_no = k;
        for (i = 0;  i < 3;  i++){
            eptr = fptr->edge[i];
            COPY_POINT_3(lseg.endpoint[0], *(eptr->vertex[0]));
            COPY_POINT_3(lseg.endpoint[1], *(eptr->vertex[1]));
            INS_TABLE(lseg, LineSegment, s);
        }
#ifdef DEBUG
        fprintf(stderr, "\nFACET no %d\n", k);
#endif
        shatter(s);
        (void) free(s->to_elem);
    }
    (void) free((char *)segs);
    printf("\nShattering the polyhedron's facets is complete.\n");

    /* sort triangles, match the opposite ones, and produce tetrahedra */
    tetrahedralize();

    (void) free(triangles.to_elem);
}


void    main()
{
    int             i, num;
    OldFacet        *old_f;

    /* read polyhedron info in, allocate space and store it */
    init_polyhedron(&P, &e_no_max, &f_no_max);
    e_count = (num = P.edge_num) - 1;
    f_count = -1;
    printf("\nPolyhedron initialization is complete.\n");

#ifdef DEBUG
    pr_polyhedron(P, 0);
#endif

    /* triangulate each facet (even the vertical ones) of the polyhedron */
    for (old_f = P.old_facets, i = P.facet_num;  --i >= 0;  old_f++)
```

```
                triangulate(old_f);
        P.edge_num = ++e_count;
        P.facet_num = ++f_count;

#ifdef DEBUG
        pr_polyhedron(P, 1);
#endif

        phase1(num);            /* pull-off phase */
        printf("\nPull-off phase is complete.\n");
#ifdef DEBUG
        pr_polyhedron(P, 1);
#endif

        phase2();               /* fence-off phase */
        printf("\nFence-off phase is complete.\n");

        report_4hedra();

        (void) free(tetrhdr_table.to_elem);
        (void) free((char *)P.vertices);
        (void) free((char *)P.edges);
        (void) free((char *)P.facets);
        (void) free((char *)P.old_facets);
        (void) free((char *)P.border);
        (void) free((char *)P.v_list);
}
```

```
   /*********                   utils.c                            *********/
   /*                   Generally useful routines                          */


#include "include.h"


double          sqrt();



/*****  comparison functions   *****/

int     xyz_v_compare(v1, v2)
Vertex          *v1, *v2;
{
        int     c;

        if ((c = E_COMPARE(v1->x, v2->x)) != 0)
                return(c);
        else
                if ((c = E_COMPARE(v1->y, v2->y)) != 0)
                        return(c);
                else
                        return( E_COMPARE(v1->z, v2->z) );
}



int     xy_p_compare(p1, p2)
        /* returns 1/0/-1 iff p1 is lex larger/equal/smaller than p2,   */
        /*              with respect to their x- and y-coordinates      */
        /* returns 0 ONLY if points have "identical" x- & y-coordinates */
Point_3d        *p1, *p2;
{
        int             c;

        if ((c = E_COMPARE(p1->x, p2->x)) != 0)
                return(c);
        else            /* p1 and p2 have "same" x_coordinate */
                return( E_COMPARE(p1->y, p2->y) );
}



int     xy_ep_compare(ep1, ep2)
        /* returns 1/0/-1 iff ep1 is lex larger/equal/smaller than ep2, */
        /*              with respect to their x- and y-coordinates      */
        /* returns 0 ONLY if points have identical x- & y-coordinates   */
Endpoint        *ep1, *ep2;
{
        int                     c;

        if ((c = E_COMPARE(ep1->x, ep2->x)) != 0)
                return(c);
        else
                return( E_COMPARE(ep1->y, ep2->y) );
}



int     xz_ep_compare(ep1, ep2)
        /* returns 1/-1 iff ep1 is lex larger/smaller than ep2,         */
        /*              with respect to their x- and z-coordinates      */
        /* right endpoints precede left endpoints in case of tie        */
        /* returns 0 if points of same kind have same x- & z-coords     */
Endpoint        *ep1, *ep2;
{
        int     d;

        if ((d = E_COMPARE(ep1->x, ep2->x)) != 0)
                return(d);
        else            /* ep1->x and ep2->x are 'almost' equal */
                if ((d = E_COMPARE(ep1->z, ep2->z)) != 0)
                        return(d);
                else
                        return(ep2->kind - ep1->kind);
}



int     yz_ep_compare(ep1, ep2)
        /* returns 1/-1 iff ep1 is lex larger/smaller than ep2,         */
        /*              with respect to their y- and z-coordinates      */
        /* right endpoints precede left endpoints in case of tie        */
        /* returns 0 if points of same kind have same y- & z-coords     */
Endpoint        *ep1, *ep2;
{
        int     d;

        if ((d = E_COMPARE(ep1->y, ep2->y)) != 0)
                return(d);
        else            /* ep1->y and ep2->y are 'almost' equal */
                if ((d = E_COMPARE(ep1->z, ep2->z)) != 0)
                        return(d);
                else
                        return(ep2->kind - ep1->kind);
}



int     seg_compare(s1, s2)
        /* returns 1/0/-1 iff the line_segment that corresponds to s1   */
        /*              lies above/on/below that corresponding to s2    */
LineSegment     *s1, *s2;
{
        int             c, first_vertical;
        double          slope1, slope2;
        Point_3d        *p11, *p12, *p21, *p22;

        p11 = s1->endpoint;
        p21 = s2->endpoint;
        if ((c = xy_p_compare(p11, p21)) != 0)
                return(c);
        else{   /* left endpoints "coincide" */
                p12 = p11 + 1;
                p22 = p21 + 1;
                first_vertical = (E_COMPARE(p11->x, p12->x) == 0);
                if (E_COMPARE(p21->x, p22->x) == 0)
                        if (first_vertical)    /* both segments vertical */
                                /* give priority to the longer line-segment */
                                return( -E_COMPARE(p12->y, p22->y) );
                        else    /* only second segment vertical */
                                return(1);
                else
                        if (first_vertical)    /* only first segment vertical */
                                return(-1);
                        else{   /* give priority to the one with larger slope */
                                slope1 = (p11->y - p12->y) / (p11->x - p12->x);
```

```
                        slope2 = (p21->y - p22->y) / (p21->x - p22->x);
                        return( -E_COMPARE(slope1, slope2) );
                }
        }
}



/*****   functions for geometric purposes   *****/

double  x_v_diff(v1, v2)        /* return difference of x-coords of v1 and v2 */
Vertex          *v1, *v2;
{
        return(v1->x - v2->x);
}


double  y_v_diff(v1, v2)        /* return difference of y-coords of v1 and v2 */
Vertex          *v1, *v2;
{
        return(v1->y - v2->y);
}


double  z_v_diff(v1, v2)        /* return difference of z-coords of v1 and v2 */
Vertex          *v1, *v2;
{
        return(v1->z - v2->z);
}



int     collinear(v1, v2, v3)
                /* returns 1 if v1, v2, v3 are collinear, 0 otherwise */
Vertex          *v1, *v2, *v3;
{
        double  dx1, dx2, dy1, dy2, dz1, dz2;

        dx1 = v1->x - v2->x;    dx2 = v1->x - v3->x;
        dy1 = v1->y - v2->y;    dy2 = v1->y - v3->y;
        dz1 = v1->z - v2->z;    dz2 = v1->z - v3->z;
        if (E_COMPARE(dy1 * dz2, dy2 * dz1) != 0)
                return(0);
        if (E_COMPARE(dz1 * dx2, dz2 * dx1) != 0)
                return(0);
        if (E_COMPARE(dx1 * dy2, dx2 * dy1) != 0 )
                return(0);
        return(1);
        /* return( E_COMPARE(dy1 * dz2, dy2 * dz1) == 0
                && E_COMPARE(dz1 * dx2, dz2 * dx1) == 0
                && E_COMPARE(dx1 * dy2, dx2 * dy1) == 0 ); */
}



int     xy_ep_angle(p1, p2, p3)
        /* return: 0 if projections of p1,p2,p3 on xy-plane are collinear, */
        /*        -1 if we turn left moving from p1 to p2 to p3,        */
        /*         1 if we turn right moving from p1 to p2 to p3.       */
Endpoint        *p1, *p2, *p3;
{
        double                  a, b;

        if (E_COMPARE(p1->x, p2->x) == 0)
                return(E_COMPARE(p2->x, p3->x) * E_COMPARE(p1->y, p2->y));
        else{           /* line p1-p2 non-vertical */
                a = -(p1->y - p2->y);
                b = p1->x - p2->x;
                return(E_COMPARE(a * p3->x + b * p3->y, a * p1->x + b * p1->y));
        }
}



int     xy_v_angle(v1, v2, v3)
        /* return: 0 if projections of v1,v2,v3 on xy-plane are collinear, */
        /*        -1 if we turn left moving from v1 to v2 to v3,        */
        /*         1 if we turn right moving from v1 to v2 to v3.       */
Vertex          *v1, *v2, *v3;
{
        double                  a, b;

        if (E_COMPARE(v1->x, v2->x) == 0)
                return(E_COMPARE(v2->x, v3->x) * E_COMPARE(v1->y, v2->y));
        else{           /* line v1-v2 non-vertical */
                a = -(v1->y - v2->y);
                b = v1->x - v2->x;
                return(E_COMPARE(a * v3->x + b * v3->y, a * v1->x + b * v1->y));
        }
}



int     xz_v_angle(v1, v2, v3)
        /* return: 0 if projections of v1,v2,v3 on xz-plane are collinear, */
        /*        -1 if we turn left moving from v1 to v2 to v3,        */
        /*         1 if we turn right moving from v1 to v2 to v3.       */
Vertex          *v1, *v2, *v3;
{
        double                  a, b;

        if (E_COMPARE(v1->x, v2->x) == 0)
                return(E_COMPARE(v2->x, v3->x) * E_COMPARE(v1->z, v2->z));
        else{           /* line v1-v2 non-vertical */
                a = -(v1->z - v2->z);
                b = v1->x - v2->x;
                return(E_COMPARE(a * v3->x + b * v3->z, a * v1->x + b * v1->z));
        }
}



int     yz_v_angle(v1, v2, v3)
        /* return: 0 if projections of v1,v2,v3 on yz-plane are collinear, */
        /*        -1 if we turn left moving from v1 to v2 to v3,        */
        /*         1 if we turn right moving from v1 to v2 to v3.       */
Vertex          *v1, *v2, *v3;
{
        double                  a, b;

        if (E_COMPARE(v1->y, v2->y) == 0)
                return(E_COMPARE(v2->y, v3->y) * E_COMPARE(v1->z, v2->z));
        else{           /* line v1-v2 non-vertical */
                a = -(v1->z - v2->z);
                b = v1->y - v2->y;
                return(E_COMPARE(a * v3->y + b * v3->z, a * v1->y + b * v1->z));
        }
}
```

```
void    plane_coef_compute(f, p, q, r)
        /* the computation of the coefficients of the plane that passes */
        /* through p, q, r can be carried out using least square method */
Facet           *f;
Vertex          *p, *q, *r;
{
        double          a, b, c, dx1, dx2, dy1, dy2, dz1, dz2, d1, d2, d3;

        dx1 = p->x - q->x;      dx2 = p->x - r->x;
        dy1 = p->y - q->y;      dy2 = p->y - r->y;
        dz1 = p->z - q->z;      dz2 = p->z - r->z;
        d1 = dy1 * dz2 - dy2 * dz1;
        d2 = dz1 * dx2 - dz2 * dx1;
        d3 = dx1 * dy2 - dx2 * dy1;
        if (E_COMPARE(d1, 0.0) == 0){
                a = 0.0;
                if (E_COMPARE(d2, 0.0) == 0){
                        b = 0.0;
                        c = 1.0;        /* d3 can not be "equal" to 0 */
                }
                else{
                        b = 1.0;
                        if (E_COMPARE(d3, 0.0) == 0)
                                c = 0.0;
                        else
                                c = d3 / d2;
                }
        }
        else{
                a = 1.0;
                if (E_COMPARE(d2, 0.0) == 0)
                        b = 0.0;
                else
                        b = d2 / d1;
                if (E_COMPARE(d3, 0.0) == 0)
                        c = 0.0;
                else
                        c = d3 / d1;
        }
        d1 = sqrt(a * a + b * b + c * c);       /* normalizing factor */
        if (E_COMPARE(c, 0.0) < 0)      /* make sure f->c >= 0 */
                d1 = -d1;
        f->a = a / d1;
        f->b = b / d1;
        f->c = c / d1;
        f->d = -(a * p->x + b * p->y + c * p->z) / d1;
}



double  from_x_comp_z(point, seg)
        /* return z-coord of point in line-segment 'seg' with x-coord=point.x */
Endpoint        point;
LineSegment     seg;
{
        double          x;
        Point_3d        p, q;

        p = seg.endpoint[0];   q = seg.endpoint[1];
        x = point.x;
        if (E_COMPARE(x, p.x) == 0)
                return(p.z);
        else
                if (E_COMPARE(x, q.x) == 0)
                        return(q.z);
                else
                        return( p.z + (x - p.x) * (q.z - p.z) / (q.x - p.x) );
}



double  from_y_comp_z(point, seg)
        /* return z-coord of point in line-segment 'seg' with y-coord=point.y */
Endpoint        point;
LineSegment     seg;
{
        double          y;
        Point_3d        p, q;

        p = seg.endpoint[0];   q = seg.endpoint[1];
        y = point.y;
        if (E_COMPARE(y, p.y) == 0)
                return(p.z);
        else
                if (E_COMPARE(y, q.y) == 0)
                        return(q.z);
                else
                        return( p.z + (y - p.y) * (q.z - p.z) / (q.y - p.y) );
}



int     vert_rel_pos(p, u, v)
   /* return 1/0/-1 if p z-above/on/below edge (u,v); plane (p,u,v) // z-axis */
Vertex          *p, *u, *v;
{
        double          a, b, t;

        a = u->z - v->z;
        b = -(u->x - v->x);
        if (E_COMPARE(b, 0.0) == 0){       /* edge parallel to the yz-plane */
                b = -(u->y - v->y);
                t = a * (p->y - u->y) + b * (p->z - u->z);
        }
        else
                t = a * (p->x - u->x) + b * (p->z - u->z);
        return( E_COMPARE(t, 0.0) );
}



int     seg_seg_clip(seg, edge)         /* COLLINEAR segment to edge clipping */
        /* returns -1, if segment is fully rejected, */
        /*          0, if segment is fully/partially accepted, & updates seg */
LineSegment     *seg;
Edge            *edge;
{
        double          c1, c2, lc1, lc2;
        Vertex          *v0, *v1;
        Point_3d        p0, p1;

        p0 = seg->endpoint[0];          p1 = seg->endpoint[1];
        v0 = edge->vertex[0];           v1 = edge->vertex[1];
        if (E_COMPARE(p0.x, p1.x) != 0){        /* seg not // to yz-plane */
                c1 = p0.x;              c2 = p1.x;
                lc1 = v0->x;            lc2 = v1->x;
        }
        else{
                c1 = p0.y;              c2 = p1.y;
```

```
                lc1 = v0->y;            lc2 = v1->y;
        }
        if (E_COMPARE(lc1, c1) > 0)
                if (E_COMPARE(lc1, c2) < 0)
                        COPY_POINT_3(seg->endpoint[0], *v0);
                else
                        return(-1);
        if (E_COMPARE(lc2, c2) < 0)
                if (E_COMPARE(lc2, c1) > 0)
                        COPY_POINT_3(seg->endpoint[1], *v1);
                else
                        return(-1);

        return(0);
}



Point_3d        intersection(seg, e)
        /* compute intersection of line-segment seg & edge e, on xy-plane  */
        /* The line segment and the edge MUST intersect in a SINGLE point  */
LineSegment     *seg;
Edge            *e;
{
        double          y, x1, y1, dx1, dy1, dx2, dy2, c;
        Point_3d        p;

        x1 = seg->endpoint[0].x;        y1 = seg->endpoint[0].y;
        dx1 = x1 - seg->endpoint[1].x;  dy1 = y1 - seg->endpoint[1].y;
        dx2 = e->b;                     dy2 = -(e->a);
        c = e->c;

        if (E_COMPARE(dy1, 0.0) == 0)   /*  seg // x-axis; note: dy2 <> 0 */
                MAKE_POINT_3(p, (dx2 * y1 + c) / dy2, y1, 0.0);
        else
                if (E_COMPARE(dy2, 0.0) == 0){  /* edge e // x-axis */
                        y = -c / dx2;           /* note: dx2 not equal to 0 */
                        MAKE_POINT_3(p, x1 + (dx1 * (y - y1)) / dy1, y, 0.0);
                }
                else{
                        y = (x1 * dy1 * dy2 - y1 * dx1 * dy2 - c * dy1) /
                                                        (dy1 * dx2 - dx1 * dy2);
                        MAKE_POINT_3(p, (dx2 * y + c) / dy2, y, 0.0);
                }
        return(p);
}



int     seg_line_clip(seg, line, select)
        /* clip segment seg against halfplane defined by line and sgn(select) */
        /* Return: -1 if segment is fully rejected,                     */
        /*          1 if segment is fully accepted,   and               */
        /*          0 if segment is partially accepted, and updates seg */
int             select;
LineSegment     *seg;
Edge            *line;
{
        int             i, sign[2];
        Point_3d        *q;

        for (i = 2;  --i >= 0;  )
                sign[i] = E_COMPARE(EVAL_2(seg->endpoint[i], *line), 0.0);

        if (sign[0] * sign[1] >= 0){
                /* endpoints of segment at same side with respect to line */
                if (select * (sign[0] + sign[1]) < 0)
                        return(-1);             /* trivial rejection */
                else      /* trivial acceptance */
                        return(1);
        }
        else{   /* endpoints of segment on different sides of line */
                q = seg->endpoint;
                if (select * sign[0] > 0)
                        q++;    /* update endpoint[1] and not endpoint[0] */
                *q = intersection(seg, line);
        }
        return(0);
}



int     compute_intersection(f, e, lseg)
    /* compute intersection of facet f and edge e, and return it in lseg */
    /* it returns -1 if the intersection is empty.                       */
    /* otherwise, it returns 1 if intersection is on an edge, or 0 else  */
Edge            *e;
Facet           *f;
LineSegment     *lseg;
{
        int             i, diff, val = 0;
        double          e_a, e_b, e_c, temp;
        Point_3d        p, *q;
        Edge            *line, *next_line;

        e_a = e->a;         e_b = e->b;          e_c = e->c;
        /* initialize lseg. Remember vertices 0 and 1 of edge are lex-sorted */
        for (i = 2;  --i >= 0;  )
                COPY_POINT_3(lseg->endpoint[i], *(e->vertex[i]));

        /* clip against the projections of the edges of the triangular facet */
        for (i = 0, next_line = f->edge[2];  i < 3;  i++){
                line = next_line;
                next_line = f->edge[i];
                MAKE_MIDPOINT_3(p, *next_line->vertex[0], *next_line->vertex[1]);
                if (E_COMPARE(e_a, line->a) == 0 && E_COMPARE(e_b, line->b) == 0)
                        /* projections of e and line on xy-plane are parallel */
                        if ((diff = E_COMPARE(e_c, line->c)) == 0){ /*collinear*/
                                /* this intersection reported twice, but...  */
                                /*      ...NONE is included in the shattering */
                                if (seg_seg_clip(lseg, line) == -1)
                                        return(-1);     /* rejection */
                                else{
                                        val = 1;      /* intersection on edge */
                                        break;
                                }
                        }
                        else{   /* projections parallel but not collinear */
                                temp = EVAL_2(p, *line);
                                if (diff * E_COMPARE(temp, 0.0) > 0)
                                        /* e in wrong halfplane: rejection */
                                        return(-1);
                        }
                else{           /* projections of e and line not parallel */
                        /* clip against the supporting line of 'line' */
                        /* determine interesting halfplane */
                        temp = EVAL_2(p, *line);
                        if (seg_line_clip(lseg, line, E_COMPARE(temp, 0.0)) == -1)
                                return(-1);     /* rejection */
                }
```

```
        }
        /* return back to facet's supporting plane */
        for (i = 2, q = lseg->endpoint;  --i >= 0;  q++)
                q->z = -(f->a * q->x + f->b * q->y + f->d) / f->c;
        return(val);
}



/*****  functions to assign values to edges and facets  *****/

void    set_e(e, p, q)
Edge            *e;
Vertex          *p, *q;
{
        double                  dx, dy;

        e->type = 1;            /* edge is not garbage */
        /* store vertices in lexicographic order */
        if (xyz_v_compare(p, q) <= 0){
                e->vertex[0] = p;
                e->vertex[1] = q;
        }
        else{
                e->vertex[0] = q;
                e->vertex[1] = p;
        }
        p->degree++;
        q->degree++;

        dx = p->x - q->x;       dy = p->y - q->y;
        if (E_COMPARE(dx, 0.0) == 0){
                e->b = 0.0;
                if (E_COMPARE(dy, 0.0) != 0)
                        e->a = 1.0;     /* edge nonvertical */
                else{
                        e->a = 0.0;     /* edge vertical */
#ifdef DIAGNOSTICS
                        if (E_COMPARE(p->z, q->z) == 0)
                                ERROR("set_e> edge too small");
#endif
                }
        }
        else{
                e->b = -1.0;
                if (E_COMPARE(dy, 0.0) == 0)
                        e->a = 0.0;
                else
                        e->a = dy / dx;
        }
        e->c = -(e->a * p->x + e->b * p->y);
}


void    set_f(f, info_f, e1, e2, e3)
Edge            *e1, *e2, *e3;
Facet           *f, *info_f;
{
        Vertex          *v1, *v2, *v3;

        f->edge[0] = e1;        f->edge[1] = e2;        f->edge[2] = e3;
        if ((v1 = e1->vertex[0]) == (v2 = e2->vertex[0])){
                f->vertex[0] = v1;
                f->vertex[1] = v2 = e3->vertex[0];
                f->vertex[2] = v3 = e3->vertex[1];
        }
        else
                if (e3->vertex[0] == v1){
                        f->vertex[0] = v1;
                        f->vertex[1] = v2;
                        f->vertex[2] = v3 = e2->vertex[1];
                }
                else{
                        f->vertex[0] = v2;
                        f->vertex[1] = v1;
                        f->vertex[2] = v3 = e1->vertex[1];
                }
        v1->facet = v2->facet = v3->facet = f;
        if (e1->facet[0] == NULL)
                e1->facet[0] = f;
        else
                e1->facet[1] = f;
        if (e2->facet[0] == NULL)
                e2->facet[0] = f;
        else
                e2->facet[1] = f;
        if (e3->facet[0] == NULL)
                e3->facet[0] = f;
        else
                e3->facet[1] = f;

#ifdef DEBUG
        fprintf(stderr,"# (%.2f,%.2f,%.2f)-(%.2f,%.2f,%.2f)-(%.2f,%.2f,%.2f)\n",
                v1->x, v1->y, v1->z, v2->x, v2->y, v2->z, v3->x, v3->y, v3->z);
#endif
#ifdef DIAGNOSTICS
        if (collinear(v1, v2, v3))      /* collinear points */
                ERROR("set_f> degenerate facet reported");
        if (info_f == NULL)
                ERROR("set_f> null facet info_f");
#endif
        f->sign = info_f->sign;
        f->a = info_f->a;       f->b = info_f->b;
        f->c = info_f->c;       f->d = info_f->d;
        if (E_COMPARE(f->c, 0.0) == 0)          /* "almost" vertical */
                f->sign = -1;
}
```

```
   /*********                   init.c                            *********/
   /*                Routines to read-in the polyhedron                    */


#include "include.h"



void    plane_coef_compute(), set_e();



void    init_polyhedron(Pol, e_max_ptr, f_max_ptr)   /* initialize polyhedron */
Polyhedron      *Pol;
int             *e_max_ptr, *f_max_ptr;
{
        int             nv, nf, ne, i, j0, j1, j2, k, v_count;
        Vertex          *vertex, *vp, *q, *r;
        Edge            *edge, *ep;
        Facet           temp_f;
        BoundVertex     *vlp, *temp_vlp;
        Boundary        *bp;
        OldFacet        *fp;

        (void)scanf("%d%d%d", &nv, &ne, &nf);
        Pol->vertex_num = nv;
        Pol->edge_num = ne;
        Pol->facet_num = nf;
        MY_CALLOC(vertex, Vertex, nv);  /* clearing fields */
        Pol->vertices = vertex;
        MY_ALLOC(Pol->old_facets, OldFacet, nf);
        MY_CALLOC(edge, Edge, (*e_max_ptr = 5 * (ne - nf)));
        Pol->edges = edge;
        MY_CALLOC(Pol->facets, Facet, (*f_max_ptr = 10*(ne-nf)/3));
        MY_ALLOC(Pol->v_list, BoundVertex, 2 * ne);
        MY_ALLOC(Pol->border, Boundary, 2 * ne / 3);

        for (vp = vertex, i = nv;  i > 0;  i--, vp++){
                (void)scanf("%lf%lf%lf", &vp->x, &vp->y, &vp->z);
                vp->type = 1;           /* valid vertex */
                vp->degree = 0;
        }

        for (vp = vertex, ep = edge, i = ne;  i > 0;  i--, ep++){
                (void)scanf("%d%d", &j0, &j1);
#ifdef DIAGNOSTICS
                if (j0 < 0 || j0 >= nv || j1 < 0 || j1 >= nv || j0 == j1)
                        ERROR("init> invalid vertex index in edge");
                else
#endif
                        set_e(ep, vp + j0, vp + j1);
        }

        fp = Pol->old_facets;
        for (bp = Pol->border, vlp = Pol->v_list, i = nf;  i > 0;  i--, fp++){
                fp->boundary = bp;
                (void)scanf("%d%d", &j0, &k);
                if ((fp->sign = j0) == 0){
                        fprintf(stderr, "init> facet sign is 0; set to -1.\n");
                        fp->sign = -1;          /* default value */
                }
                for (v_count = 0, fp->bound_num = k;  k > 0;  k--, bp++, vlp++){
                        bp->first = vlp;
                        (void)scanf("%d%d%d", &j0, &j1, &j2);
                        v_count += j0;
#ifdef DIAGNOSTICS
                        if (j1 < 0 || j1 >= nv)
                                ERROR("init> invalid vertex index in facet");
                        if (j2 < 0 || j2 >= ne)
                                ERROR("init> invalid edge index in facet");
#endif
                        vlp->prev_e = ep = edge + j2;
                        vp = vertex + j1;
                        if ((q = ep->vertex[0]) == vp)
                                q = ep->vertex[1];
                        for (bp->size = j0;  --j0 > 0;  ){
                                (void)scanf("%d", &j1);
#ifdef DIAGNOSTICS
                                if (j1 < 0 || j1 >= ne)
                                        ERROR("init> invalid edge index in facet");
#endif
                                vlp->v = q;
                                vlp->next_e = ep = edge + j1;
                                vlp++;
                                vlp->prev_e = ep;
                                if ((r = ep->vertex[0]) == q)
                                        q = ep->vertex[1];
                                else
                                        q = r;
                        }
                        vlp->v = vp;
                        vlp->next_e = edge + j2;
                }
                fp->vertex_num = v_count;
#ifdef DIAGNOSTICS
                if (v_count <= 2)
                        ERROR("init> degenerate facet");
#endif

                /* locate three non-collinear vertices */
                temp_vlp = fp->boundary->first;
                vp = temp_vlp->v;
                q = (++temp_vlp)->v;
                for (temp_vlp++, k = v_count, r = NULL;  --k >= 2;  temp_vlp++)
                        if (!collinear(vp, q, (r = temp_vlp->v)))
                                break;
#ifdef DIAGNOSTICS
                if (k <= 1)
                        ERROR("init> facet below e-threshold");
#endif

                plane_coef_compute(&temp_f, vp, q, r);
                fp->a = temp_f.a;       fp->b = temp_f.b;
                fp->c = temp_f.c;       fp->d = temp_f.d;
        }
        /* good to have a test to verify polyhedron is not self-intersecting */
}
```

```
   /*********                  triangulate.c                   *********/
   /*            Routines to triangulate an arbitrary polygon           */
   /*           using Hertel & Mehlhorn's triangulation algorithm       */
   /*  The routines locate the coordinate plane on which the polygon    */
   /*  has the largest projection, and use the corresponding coordina-  */
   /*  tes to simplify the process and avoid frame transformation.      */


#include "include.h"


/* declare functions used by the functions in this file */

int             xy_v_angle(), yz_v_angle(), xz_v_angle();
double          x_v_diff(), y_v_diff(), z_v_diff();
void            set_e(), set_f();
Edge            *new_e();
Facet           *new_f();
#ifdef DEBUG
void            pr_old_facet();
#endif



extern Polyhedron               P;



typedef struct sp_point {
                int                     kind, vb_ind;
                struct bord_type        *border;
} Special;
typedef struct v_list_node {
                struct vertex_type      *v;
                struct edge_type        *prev_e, *next_e;
                struct v_list_node      *previous, *next;
} VertexChain;
typedef struct region1_type {
                int                     up_ind, low_ind;
                   /* indexes to v_bord_type array of next vertices up & down */
                struct bord_type        *up_border, *low_border;
                         /* ptrs to records of upper and lower bounding lines */
                struct v_list_node      *rightmost, /* ptr to rightmost point */
                                        *top,       /* ptrs to top and bottom */
                                        *bottom;   /* of a doubly linked list */
                struct region1_type     *prev, *next;
} Region1;



static int              region_ind, v_list_ind,
                        (*angle)();             /* left/right turn test */
static double           (*diff1)(),     /* returns primary coords difference */
                        (*diff2)();     /* returns secondary coords difference*/
static Facet            info_f;
static OldFacet         *facet;
static Region1          *region_table, *region_list;
static VertexChain      *list;



#ifdef DEBUG
/*****************************************************************************/
```

```
void    pr_special(array_head)
Table           *array_head;
{
        int             i, j;
        Special         *sp;
        Vertex          *v;
        BoundVertex     *vb;

        fprintf(stderr, "\n\tTable of special points\n\n");
        for (i= array_head->n, j= 0, sp= (Special *)array_head->to_elem;
                                                        i > 0;  i--, j++, sp++){
                fprintf(stderr, "special point %d:  ", j);
                vb = sp->border->first + sp->vb_ind;
                v = vb->v;
                fprintf(stderr, "(%.3f, %.3f, %.3f),   ", v->x, v->y, v->z);
                fprintf(stderr, "type = %d\n", sp->kind);
        }
}


void    pr_regions(bord_array)
Boundary        *bord_array;
{
        Vertex          *v;
        VertexChain     *q;
        Region1         *regptr;

        for (regptr = region_list;  regptr != NULL;  regptr = regptr->next){
                fprintf(stderr, "region #%d:\n", regptr - region_table);
                v = (regptr->up_border->first + regptr->up_ind)->v;
                fprintf(stderr,"\tup_bord_ind = %d,  up_v = (%.2f,%.2f,%.2f)\n",
                                regptr->up_border - bord_array, v->x,v->y,v->z);
                v = (regptr->low_border->first + regptr->low_ind)->v;
                fprintf(stderr,"\tlow_bord_ind= %d,  low_v= (%.2f,%.2f,%.2f)\n",
                                regptr->low_border -bord_array, v->x,v->y,v->z);

                q = regptr->top;
                fprintf(stderr, "\tforward chain:  <%d>", q->prev_e - P.edges);
                do{
                        v = q->v;
                        fprintf(stderr, ", (%.2f,%.2f,%.2f), <%d>",
                                        v->x, v->y, v->z, q->next_e - P.edges);
                        q = q->next;
                }while (q != NULL);
                fprintf(stderr, "\n");

                q = regptr->bottom;
                fprintf(stderr, "\tbackward chain:  <%d>", q->next_e - P.edges);
                do{
                        v = q->v;
                        fprintf(stderr, ", (%.2f,%.2f,%.2f), <%d>",
                                        v->x, v->y, v->z, q->prev_e - P.edges);
                        q = q->previous;
                }while (q != NULL);
                v = regptr->rightmost->v;
                fprintf(stderr, "\n\trightmost:  (%.2f,%.2f,%.2f)\n",
                                                        v->x, v->y, v->z);
        }
}
/*****************************************************************************/
#endif



int     sp_compare(sp1, sp2)    /* lex-compare special points sp1, sp2 */
```

```
Special         *sp1, *sp2;
{
        int                     c;
        double                  diff;
        BoundVertex     *vbp1, *vbp2;

        vbp1 = sp1->border->first + sp1->vb_ind;
        vbp2 = sp2->border->first + sp2->vb_ind;
        diff = diff1(vbp1->v, vbp2->v);
        if ((c = E_COMPARE(diff, 0.0)) != 0)
                return(c);
        else{    /* primary coords of vbp1->v and vbp2->v are 'almost' equal */
                diff = diff2(vbp1->v, vbp2->v);
                return( E_COMPARE(diff, 0.0) );
        }
}


VertexChain     *get_node(list_root)
VertexChain     *list_root;
{
        if (--v_list_ind < 0)
                ERROR("get_node> list overflow");
        return(list_root + v_list_ind);
}


void    select_funcs(f)
OldFacet        *f;
{
        double  a, b, c;

        if ((a = f->a) < 0)     a = -a;
        if ((b = f->b) < 0)     b = -b;
        if ((c = f->c) < 0)     c = -c;

        /* select appropriate functions for triangulation */
        if (E_COMPARE(a, b) <= 0)
                if (E_COMPARE(b, c) <= 0){
                        diff1 = x_v_diff;
                        diff2 = y_v_diff;
                        angle = xy_v_angle;
                }
                else{
                        angle = xz_v_angle;
                        if (E_COMPARE(a, c) <= 0){
                                diff1 = x_v_diff;
                                diff2 = z_v_diff;
                        }
                        else{
                                diff1 = z_v_diff;
                                diff2 = x_v_diff;
                        }
                }
        else
                if (E_COMPARE(a, c) <= 0){
                        diff1 = y_v_diff;
                        diff2 = x_v_diff;
                        angle = xy_v_angle;
                }
                else{
                        angle = yz_v_angle;
                        if (E_COMPARE(b, c) <= 0){
                                diff1 = y_v_diff;
                                diff2 = z_v_diff;
                        }
                        else{
                                diff1 = z_v_diff;
                                diff2 = y_v_diff;
                        }
                }
}


void    check_orientation()
{
        int             i, j, h;
        double          d1, d2, nextslope;
        BoundVertex     *min, *next, *prev, temp;
        Vertex          *min_v, *next_v, *prev_v;
        Boundary        *minb, *bptr;
        Edge            *temp_e;

        minb = facet->boundary;
        min = minb->first;
        min_v = min->v;
        /*  locate leftmost lowermost vertex    */
        for (j = facet->bound_num, bptr = minb;  j > 0;  j--, bptr++)
                for (i = bptr->size, next = bptr->first;  i > 0;  i--, next++){
                        d1 = diff1(min_v, (next_v = next->v));
                        switch (E_COMPARE(d1, 0.0)){
                        case -1: break; /* do nothing */
                        case  0: d2 = diff2(min_v, next_v);
                                 if (E_COMPARE(d2, 0.0) <= 0)
                                        break;  /* else fall to next cae */
                        case  1: minb = bptr;
                                 min = next;
                                 min_v = next_v;
                                 break;
                        }
                }
        i = minb->size;
        j = min - minb->first;
        next = minb->first + MOD(j + 1, i);
        d1 = diff1((next_v = next->v), min_v);
        d2 = diff2(next_v, min_v);
        if (E_COMPARE(d1, 0.0) == 0){
                h = E_COMPARE(d2, 0.0);
#ifdef DIAGNOSTICS
                if (h == 0)
                        ERROR("check_orientation> edge too short");
#endif
        }
        else{
                nextslope = d2 / d1;
                prev = minb->first + MOD(j - 1, i);
                d1 = diff1(min_v, (prev_v = prev->v));
                d2 = diff2(min_v, prev_v);
                if (E_COMPARE(d1, 0.0) == 0){
                        h = E_COMPARE(d2, 0.0);
#ifdef DIAGNOSTICS
                        if (h == 0)
                                ERROR("check_orientation> edge too short");
#endif
                }
                else{
                        h = E_COMPARE(nextslope, d2/d1);
#ifdef DIAGNOSTICS
                        if (h == 0)
                                ERROR("check_orientation> degenerate facet");
```

```
#endif
                }
        }

        if (h == 1)
                return;
        /* reverse orientation */
        for (h = facet->bound_num, bptr= facet->boundary;  h > 0;  h--, bptr++){
                prev = bptr->first;
                next = prev + (j = bptr->size - 1);
                for (i = 0;  i < j;  i++, prev++, j--, next--){
                        temp_e = prev->prev_e;  /* swap incident edges */
                        prev->prev_e = prev->next_e;
                        prev->next_e = temp_e;
                        temp = *next;
                        temp_e = temp.prev_e;   /* swap incident edges */
                        temp.prev_e = temp.next_e;
                        temp.next_e = temp_e;
                        *next = *prev;          /* swap records */
                        *prev = temp;
                }
                if (i == j){
                        temp_e = prev->prev_e;  /* swap incident edges */
                        prev->prev_e = prev->next_e;
                        prev->next_e = temp_e;
                }
        }
}


void    sp_collect(sp_table)
Table           *sp_table;
{
        int             i, j, size, term, cond, prevcond, activecond;
        double          diff, slope, prevslope;
        Boundary        *bptr;
        BoundVertex     *v_b_table, *cur, *next;
        Vertex          *cur_v, *next_v;
        Special         temp;

        /* process each boundary line in turn, collecting special-points */
        for (j = 0, bptr= facet->boundary;  j < facet->bound_num;  j++, bptr++){
                /* find two adjacent vertices with different primary coords */
                i = size = bptr->size;
                next = cur = v_b_table = bptr->first;
                for (cur_v = cur->v;  i > 0;  cur_v = next_v, cur = next, i--){
                        if (++next == v_b_table + size)
                                next = v_b_table;
                        diff = diff1(cur_v, (next_v = next->v));
                        if ((activecond = E_COMPARE(diff, 0.0)) != 0)
                                break;
                }
#ifdef DIAGNOSTICS
                if (i == 0)     /* all pairs checked */
                        ERROR("sp_collect> degenerate facet");
#endif

                prevslope = diff2(cur_v, next_v) / diff1(cur_v, next_v);
                for (i = size, cur_v = next_v, cur= next, prevcond = activecond;
                                        i > 0;  cur_v = next_v, cur= next, i--){
                        /* note: activecond never assigned to 0 in for loop! */
                        if (++next == v_b_table + size)
                                next = v_b_table;
                        diff = diff1(cur_v, (next_v = next->v));
                        if ((cond = E_COMPARE(diff, 0.0)) == 0){
                                diff = diff2(cur_v, next_v);
                                if ((prevcond = 2 * E_COMPARE(diff, 0.0)) == 0)
                                        ERROR("sp_collect> edge too short");
                                continue;
                        }

                        slope = diff2(cur_v, next_v) / diff1(cur_v, next_v);
                        if (cond * activecond < 0){
                                temp.border = bptr;
                                temp.vb_ind = cur - v_b_table;
                                /* kind: 0 if start-point, 2 if stop-point,
                                        1/3 if left-/right-turning point */
                                switch (prevcond){
                                case -2: /* vertical upward previous edge */
                                        if (cond == -1)
                                                temp.kind = 0;
                                        else
                                                temp.kind = 3;
                                        break;
                                case  2: /* vertical downward previous edge */
                                        if (cond == -1)
                                                temp.kind = 1;
                                        else
                                                temp.kind = 2;
                                        break;
                                case -1:
                                case  1: if (prevslope > slope)
                                                term = 1;
                                        else
                                                term = 0;
                                        temp.kind = cond + term + 1;
                                        break;
                                }
                                INS_TABLE(temp, Special, sp_table);
                        }
                        prevcond = activecond = cond;
                        prevslope = slope;
                }
        }
}


void    adj_upchain(regptr, v_b_ptr)
Region1         *regptr;
BoundVertex     *v_b_ptr;
{
        Vertex          *v;
        Edge            *e, *e_temp;
        VertexChain     *q, *r;

        v = v_b_ptr->v;
        for (q = regptr->top, e_temp = q->prev_e;  ;  q = r, e_temp = e){
                if ((r = q->next) == NULL || angle(r->v, q->v, v) <= 0)
                        break;          /* non convex angle */

                /* convex angle.  report a triangle */
                set_e(e = new_e(), r->v, v);
                set_f(new_f(), &info_f, q->next_e, e_temp, e);
        }
        q->previous = NULL;     /* all used q's are unlinked */
        q->prev_e = e_temp;
        regptr->top = q;
}
```

```
void    upchain_insert(regptr, bptr, v_bord_ind)
Region1         *regptr;
Boundary        *bptr;
int                     v_bord_ind;
{
        Boundary        *up_bord;
        BoundVertex     *vb;
        VertexChain     *q, *r;

        up_bord = regptr->up_border;
        vb = up_bord->first + v_bord_ind;
        adj_upchain(regptr, vb);
        /* insert point in the point-chain associated with region */
        r = regptr->top;
        q = get_node(list);
        q->v = vb->v;
        q->prev_e = vb->next_e;
        q->next_e = r->prev_e;
        q->previous = NULL;
        q->next = r;
        r->previous = regptr->top = q;
        regptr->up_border = bptr;
        /*** CAUTION: CALLING function MUST update "rightmost" field ***/
}


void    adj_lowchain(regptr, v_b_ptr, caution)
Region1         *regptr;
BoundVertex     *v_b_ptr;
int                     caution;
        /* caution helps stop triangle reporting if the lower boundary */
        /*      proceeds vertically folding inwards */
        /* this is not needed for the adj_upchain, since we process the */
        /*      upper boundary first */
{
        double          diff;
        Vertex          *v;
        Edge            *e, *e_temp;
        VertexChain     *q, *r;

        v = v_b_ptr->v;
        for (q = regptr->bottom, e_temp = q->next_e;  ; q = r, e_temp = e){
                if ((r = q->previous) == NULL || angle(r->v, q->v, v) >= 0)
                        break;          /* non convex angle */
                if (caution){
                        diff = diff1(v, r->v);
                        if (E_COMPARE(diff, 0.0) == 0)
                                break;
                }

                /* convex angle.  report a triangle */
                set_e(e = new_e(), r->v, v);
                set_f(new_f(), &info_f, q->prev_e, e_temp, e);
        }
        q->next = NULL;         /* all used q's are unlinked */
        q->next_e = e_temp;
        regptr->bottom = q;
}


void    lowchain_insert(regptr, bptr, v_bord_ind)
Region1         *regptr;
Boundary        *bptr;
int                     v_bord_ind;
{
        Boundary        *low_bord;
        BoundVertex     *vb;
        VertexChain     *q, *r;

        low_bord = regptr->low_border;
        vb = low_bord->first + v_bord_ind;
        adj_lowchain(regptr, vb, 0);
        /* insert point in the point-chain associated with region */
        r = regptr->bottom;
        q = get_node(list);
        q->v = vb->v;
        q->prev_e = r->next_e;
        q->next_e = vb->prev_e;
        q->next = NULL;
        q->previous = r;
        r->next = regptr->bottom = q;
        regptr->low_border = bptr;
        /*** CAUTION: CALLING function MUST update "rightmost" field ***/
}


void    advance(regptr, vb_ptr)
        /* advance local sweep line up to the x-coordinate of point */
Region1         *regptr;
BoundVertex     *vb_ptr;
{
        int             h, k, l, up_size, low_size;
        double          d;
        BoundVertex     *up_vb_table, *low_vb_table, *up_vb, *low_vb, *next_vb;
        Vertex          *v, *new_v, *up_v, *low_v, *next_v, *leftmost,
                                                        *old_rightmost;
        Edge            *e, *e_temp;
        Boundary        *up_bord, *low_bord;

        old_rightmost = regptr->rightmost->v;
        new_v = vb_ptr->v;

        up_bord = regptr->up_border;
        up_size = up_bord->size;
        k = regptr->up_ind;
        up_vb = (up_vb_table = up_bord->first) + k;
        up_v = up_vb->v;
        low_bord = regptr->low_border;
        low_size = low_bord->size;
        l = regptr->low_ind;
        low_vb = (low_vb_table = low_bord->first) + l;
        low_v = low_vb->v;

        /* advance upper and lower boundaries IN LOCK STEP */
        d = diff1(up_v, low_v);
        if (E_COMPARE(d, 0.0) <= 0)
                leftmost = up_v;
        else
                leftmost = low_v;
        for (d = diff1(leftmost, new_v);  E_COMPARE(d, 0.0) < 0;
                                                d = diff1(leftmost, new_v)){
                if (leftmost == up_v){
                        next_vb = up_vb_table + (h = MOD(k + 1, up_size));
                        d = diff1((next_v = next_vb->v), leftmost);
                        if (E_COMPARE(d, 0.0) == 0){
                                d = diff2(next_v, up_v);
                                if (E_COMPARE(d, 0.0) < 0){
                                        /* boundary folds inwards */
                                        adj_upchain(regptr, up_vb);
                                        v = regptr->top->v;
```

```
                        e_temp = regptr->top->prev_e;
                        do{
                                set_e(e = new_e(), v, next_v);
                                set_f(new_f(), &info_f, e_temp,
                                        up_vb->next_e, e);
                                k = h;
                                up_vb = next_vb;
                                up_v = next_v;
                                next_vb = up_vb_table +
                                      (h = MOD(k + 1, up_size));
                                d = diff1((next_v=next_vb->v),
                                                leftmost);
                                e_temp = e;
                        } while (E_COMPARE(d, 0.0) == 0);
                        regptr->top->prev_e = e;
                        upchain_insert(regptr, up_bord, k);
                }
                else{   /* boundary folds outwards */
                        h = k;
                        do{
                                upchain_insert(regptr, up_bord, h);
                                next_vb = up_vb_table +
                                      (h = MOD(h + 1, up_size));
                                d = diff1((next_v = next_vb->v),
                                                leftmost);
                        } while (E_COMPARE(d, 0.0) == 0);
                }
        }
        else
                upchain_insert(regptr, up_bord, k);
        k = h;
        up_vb = next_vb;
        up_v = next_v;
}
else{
        next_vb = low_vb_table + (h = MOD(l - 1, low_size));
        d = diff1((next_v = next_vb->v), leftmost);
        if (E_COMPARE(d, 0.0) == 0){
                d = diff2(next_v, low_v);
                if (E_COMPARE(d, 0.0) > 0){
                        /* boundary folds inwards */
                        adj_lowchain(regptr, low_vb, 1);
                        v = regptr->bottom->v;
                        e_temp = regptr->bottom->next_e;
                        do{
                                set_e(e = new_e(), v, next_v);
                                set_f(new_f(), &info_f, e_temp,
                                        low_vb->prev_e, e);
                                l = h;
                                low_vb = next_vb;
                                low_v = next_v;
                                next_vb = low_vb_table +
                                     (h = MOD(l - 1, low_size));
                                d = diff1((next_v=next_vb->v),
                                                leftmost);
                                e_temp = e;
                        } while (E_COMPARE(d, 0.0) == 0);
                        regptr->bottom->next_e = e;
                        lowchain_insert(regptr, low_bord, l);
                }
                else{   /* boundary folds outwards */
                        h = l;
                        do{
                                lowchain_insert(regptr, low_bord, h);
                                next_vb = low_vb_table +
                                                (h = MOD(h - 1, low_size));
                                        d = diff1((next_v = next_vb->v),
                                                        leftmost);
                                } while (E_COMPARE(d, 0.0) == 0);
                        }
                        else
                                lowchain_insert(regptr, low_bord, l);
                        l = h;
                        low_vb = next_vb;
                        low_v = next_v;
                }
                d = diff1(up_v, low_v);
                if (E_COMPARE(d, 0.0) <= 0)
                        leftmost = up_v;
                else
                        leftmost = low_v;
        }
        regptr->low_ind = l;

        /* extend upper boundary */
        d = diff1(up_v, new_v);
        if (E_COMPARE(d, 0.0) == 0){
                next_vb = up_vb_table + (h = MOD(k + 1, up_size));
                d = diff1((next_v = next_vb->v), new_v);
                if (E_COMPARE(d, 0.0) == 0){
                        d = diff2(next_v, up_v);
                        if (E_COMPARE(d, 0.0) < 0){
                                /* boundary folds inwards */
                                adj_upchain(regptr, up_vb);
                                v = regptr->top->v;
                                e_temp = regptr->top->prev_e;
                                do{
                                        /* low_vb holds ptr to vb of low_ind */
                                        if (next_v == low_v){
                                                regptr->top->prev_e = e_temp;
                                                regptr->up_ind = h;
                                                adj_lowchain(regptr, low_vb, 0);
                                                e = regptr->bottom->next_e;
                                                set_f(new_f(), &info_f, e_temp,
                                                        up_vb->next_e, e);
                                                return;
                                        }
                                        set_e(e = new_e(), v, next_v);
                                        set_f(new_f(), &info_f, e_temp,
                                                        up_vb->next_e, e);
                                        k = h;
                                        up_vb = next_vb;
                                        up_v = next_v;
                                        next_vb = up_vb_table +
                                                 (h = MOD(k + 1, up_size));
                                        d = diff1((next_v = next_vb->v), new_v);
                                        e_temp = e;
                                } while (E_COMPARE(d, 0.0) == 0);
                                regptr->top->prev_e = e;
                        }
                        else{   /* boundary folds outwards */
                                for (;  ;  k = h){
                                        next_vb = up_vb_table +
                                                (h = MOD(k + 1, up_size));
                                        d = diff1(next_vb->v, new_v);
                                        if (E_COMPARE(d, 0.0) != 0)
                                                break;
                                        upchain_insert(regptr, up_bord, k);
                                }
```

```
                        }
                }
        }
        regptr->up_ind = k;

        /* set the rightmost pointer */
        d = diff1((up_v = regptr->top->v), (low_v = regptr->bottom->v));
        if (E_COMPARE(d, 0.0) >= 0){
                d = diff1(up_v, old_rightmost);
                if (E_COMPARE(d, 0.0) >= 0)
                        regptr->rightmost = regptr->top;
        }
        else{
                d = diff1(low_v, old_rightmost);
                if (E_COMPARE(d, 0.0) >= 0)
                        regptr->rightmost = regptr->bottom;
        }
}


int     in(regptr, point)
Region1         *regptr;
Vertex          *point;
{
        double                  diff;
        Vertex          *p;
        BoundVertex     *vb;

        vb = (regptr->up_border)->first + regptr->up_ind;
        p = vb->v;
        diff = diff1(p, point);
        if (E_COMPARE(diff, 0.0) == 0){
                diff = diff2(p, point);
                if (E_COMPARE(diff, 0.0) < 0)
                        return(1);
        }
        else
                if (angle(regptr->top->v, p, point) < 0)
                        return(1);

        vb = (regptr->low_border)->first + regptr->low_ind;
        p = vb->v;
        diff = diff1(p, point);
        if (E_COMPARE(diff, 0.0) == 0){
                diff = diff2(p, point);
                if (E_COMPARE(diff, 0.0) > 0)
                        return(-1);
        }
        else
                if (angle(regptr->bottom->v, p, point) > 0)
                        return(-1);

        return(0);
}


Region1 *locate(vb_ptr, place)
        /* returns: pointer to region that contains vb_ptr->v, or lies */
        /* immediately above vb_ptr->v;  NULL if no such region exists */
        /* place is set to 0 in the former case, 1 otherwise            */
int                     *place;
BoundVertex     *vb_ptr;
{
        Vertex          *v;
        Region1         *cur, *next;

        /* sequential search in the y-structure advancing local sweep lines */
        *place = 1;
        for (cur = NULL, next = region_list, v = vb_ptr->v;  next != NULL;
                                                        next = next->next){
                advance(cur = next, vb_ptr);
                switch (in(cur, v)){
                case  1: return(cur->prev);     /* point above cur region */
                case  0: *place = 0;            /* point in cur region */
                         return(cur);
                }
        }
        return(cur);            /* located below the lowermost region */
}


void    region_insert(prev, entry)
                /* insert entry in doubly linked list after prev */
Region1         *prev, entry;
{
        Region1         *regptr;

#ifdef DIAGNOSTICS
        if (region_ind <= 0)
                ERROR("region_insert> not enough region records");
#endif
        regptr = region_table + --region_ind;
        *regptr = entry;                /* copy information */

        if ((regptr->prev = prev) == NULL){
                regptr->next = region_list;     /* insert at head of list */
                region_list = regptr;
        }
        else{
                regptr->next = prev->next;
                if (prev->next != NULL)
                        prev->next->prev = regptr;
                prev->next = regptr;
        }
}


void    region_delete(regptr)
                /* delete from doubly linked list */
Region1         *regptr;
{
        Region1         *next;

        if ((next = regptr->next) != NULL)
                next->prev = regptr->prev;
        if (regptr->prev != NULL)
                regptr->prev->next = next;
        else{
                if (regptr != region_list)
                        ERROR("region_delete> inconsistent region list");
                region_list = next;
        }
}


void    merge_chain(regptr, bptr, vb_ind)
int             vb_ind;
Region1         *regptr;
Boundary        *bptr;
{
```

```
        BoundVertex     *vb;
        VertexChain     *q;
        Region1         *nextregptr;

        vb = bptr->first + vb_ind;
        nextregptr = regptr->next;

#ifdef DIAGNOSTICS
        if (regptr == NULL)
                ERROR("merge_chain> NULL upper region");
        if (regptr->low_border != bptr || regptr->low_ind != vb_ind)
                ERROR("merge_chain> upper region to merge is incorrect");
        if (nextregptr == NULL)
                ERROR("merge_chain> NULL lower region");
        if (nextregptr->up_border != bptr)
                ERROR("merge_chain> lower region to merge is incorrect");
#endif
        /* locate ensures that *regptr does not lag behind,             */
        /* but we need to advance local sweep-line of adjacent region   */
        advance(nextregptr, vb);
#ifdef DIAGNOSTICS
        if (nextregptr->up_ind != vb_ind)
                ERROR("merge_chain> top vertex does not match");
#endif

        adj_lowchain(regptr, vb, 0);
        adj_upchain(nextregptr, vb);

        /* merge chains */
        q = get_node(list);
        q->v = vb->v;
        q->prev_e = regptr->bottom->next_e;
        q->next_e = nextregptr->top->prev_e;
        q->previous = regptr->bottom;
        q->next = nextregptr->top;
        (regptr->bottom)->next = (nextregptr->top)->previous =
                                                        regptr->rightmost = q;
        regptr->low_ind = nextregptr->low_ind;
        regptr->low_border = nextregptr->low_border;
        regptr->bottom = nextregptr->bottom;

        region_delete(nextregptr);
}


void    split_chain(regptr, bptr, vb_ind, v)
int             vb_ind;
Region1         *regptr;
Boundary        *bptr;
Vertex          *v;
{
        int             size = bptr->size;
        VertexChain     *q, *cutpoint;
        Region1         temp;

        cutpoint = regptr->rightmost;

        q = get_node(list);
        *q = *cutpoint;

        temp = *regptr;
        regptr->low_ind = vb_ind;
        regptr->low_border = bptr;
        if (regptr->bottom == cutpoint)
                temp.bottom = q;
        else{
                regptr->bottom = cutpoint;
                cutpoint->next->previous = q;
        }
        cutpoint->next = q->previous = NULL;
        set_e(cutpoint->next_e = q->prev_e = new_e(), cutpoint->v, v);

        temp.up_ind = vb_ind;
        temp.up_border = bptr;
        temp.top = temp.rightmost = q;

        lowchain_insert(regptr, bptr, vb_ind);
        regptr->rightmost = regptr->bottom;
        regptr->low_border = bptr;
        regptr->low_ind = MOD(vb_ind - 1, size);

        region_insert(regptr, temp);
        upchain_insert(regptr = regptr->next, bptr, vb_ind);
        regptr->rightmost = regptr->top;
        regptr->up_border = bptr;
        regptr->up_ind = MOD(vb_ind + 1, size);
}


void    sp_process(sp_table)
Table           *sp_table;
{
        int             i, pos, ind, size;
        Boundary        *bptr;
        BoundVertex     *cur;
        Edge            *e1, *e2;
        Special         *sp;
        VertexChain     *q, *r;
        Region1         *regptr, temp;

        /* sweep the special points in order, advancing only the needed */
        /* sweep-lines until they reach the current point, and report   */
        for (i = sp_table->n, sp = (Special *)sp_table->to_elem;
                                                --i >= 0;  sp++){
                cur = (bptr = sp->border)->first + (ind = sp->vb_ind);

                /* locate index of region that contains point, or... */
                regptr = locate(cur, &pos);     /* ...lies above point */

#ifdef ALLDEBUG
        fprintf(stderr, "\nNew special point located");
        if (pos == 0)
                fprintf(stderr, " in region #%d\n", regptr - region_table);
        else
                fprintf(stderr, " below region #%d\n", regptr - region_table);
#endif
#ifdef DIAGNOSTICS
        if (sp->kind == 0 && pos == 0 || sp->kind != 0 && pos != 0)
                ERROR("sp_process> incorrect point location");
#endif
                switch (sp->kind){
                case 0: /* start point */
                        size = bptr->size;
                        /* create record of new region */
                        q = get_node(list);
                        q->next = q->previous = NULL;
                        q->v = cur->v;
                        q->prev_e = cur->next_e;        /* !!!!! */
                        q->next_e = cur->prev_e;
                        temp.top = temp.bottom = temp.rightmost = q;
```

```
                        temp.up_border = temp.low_border = bptr;
                        temp.up_ind = MOD(ind + 1, size);
                        temp.low_ind = MOD(ind - 1, size);

                        region_insert(regptr, temp);   /* insert after regptr */
                        break;
                case 1: /* left-turning point */
                        split_chain(regptr, bptr, ind, cur->v);
                        break;
                case 2: /* stop-point */
                        /* locate ensures that region has been fully swept */
#ifdef DIAGNOSTICS
                        if (regptr->up_ind != ind || regptr->low_ind != ind)
                                ERROR("sp_process> 'advance' internal error");
#endif
                        if ((q = regptr->top) != regptr->bottom){
                                for (e1 = q->prev_e;    /* report triangles */
                                        (r = q->next) != regptr->bottom;
                                                        q = r, e1 = e2){
                                        set_e(e2 = new_e(), r->v, cur->v);
                                        set_f(new_f(), &info_f, q->next_e, e1,
                                                                e2);
                                }
                                set_f(new_f(), &info_f, q->next_e, e1, r->next_e);
                        }

                        /* delete entry from region-table */
                        region_delete(regptr);
                        break;
                case 3: /* right-turning-point */
                        merge_chain(regptr, bptr, ind);
                        break;
                }
#ifdef DEBUG
                pr_regions(facet->boundary);
#endif
        }
}


void    triangulate(t)          /* facet t is decomposed into triangles */
OldFacet        *t;
{
        int             m;
        Table           special;

#ifdef DEBUG
        fprintf(stderr, "\nNew facet to triangulate\n");
#endif

        /* copy values of sign and a, b, c, d to info_f */
        info_f.sign = t->sign;
        info_f.a = t->a;        info_f.b = t->b;
        info_f.c = t->c;        info_f.d = t->d;

        select_funcs(facet = t);

        /*  the implementation assumes that the boundary components are */
        /*  oriented so that the facet lies always to the right, as we  */
        /*  walk along them.  We verify the assumption, and, if needed, */
        /*  we reverse the orientation of the bounding lines.           */
        check_orientation();
#ifdef DEBUG
        fprintf(stderr, "\nAfter verifying the orientation, the facet is:\n");
        pr_old_facet(facet, P.vertices, P.edges);
#endif

        MAKE_TABLE(&special, Special, MEDIUM_SIZE);
        sp_collect(&special);

        qsort(special.to_elem, (m = special.n), sizeof(Special), sp_compare);
#ifdef DEBUG
        pr_special(&special);
#endif

        /* allocate the space for the y-structure */
        /* array of at most m/2 entries  (m= total number of special points) */
        MY_ALLOC(region_table, Region1, (region_ind = m / 2));
        region_list = NULL;
        /* array of list_records, for the chain of vertices in each region */
        MY_ALLOC(list, VertexChain, (v_list_ind = facet->vertex_num +m));

        sp_process(&special);

        (void) free((char *)region_table);
        (void) free((char *)list);
        (void) free(special.to_elem);
}
```

```
   /*********                  dualize.c                        *********/
   /*          Routines necessary for the Witness Oracle.              */
   /*  The puncturing vertices are mapped to lines, their line arran-   */
   /*  gement is computed (get_arrangmnt), the layers of the arrange-   */
   /*  ment are extracted (get_tables), and they are used to build a    */
   /*  logarithmic point-location data structure using fractional-cas-  */
   /*  cading (merge_left/right, fr_cascade).  Then, the line arrange-  */
   /*  ment tree with the region pointers is constructed (build_tree),  */
   /*  and the facets are proceesed (query) concluding the oracle.      */


#include "include.h"



typedef struct vertex_rec {
        int                     up_reg_no, low_reg_no;
        double                  x, y;
        struct edge_rec         *next_edge;
        /* next_edge points to edge incident upon and to the right of vertex */
        /* and belongs to next layer */
} ArrangV;
typedef struct edge_rec {
        int                     punct_no, region_no;
        struct vertex_rec       *vertex;
        struct edge_rec         *next, *twin, *more;
        /* next points to the next edge record in the same facet, twin to the */
        /* twin edge record, more to another edge record in the list of edges */
} ArrangE;
typedef struct layer_rec {
        struct edge_rec         *edge_ptr;
} TempLayer;
typedef struct layer_entry {
        struct fc_e_rec         *fc_e_ptr;
        struct edge_rec         *edge_ptr;
} Layer;
typedef struct fc_e_rec {
        int                     v_type, punct_no, up_reg_no, low_reg_no;
        struct vertex_rec       *v;
        struct fc_e_rec         *up, *down, *next;
} FC_Edge;



static void             (*update_witness)();
static int              region_max, e_count, e_max, v_count, top_size;
static ArrangV          *vert, l_inf_rec, r_inf_rec, *L_INF, *R_INF;
static ArrangE          *edgeptr, *edge_list;
static TempLayer        *temp_layer;
static Layer            *layer;
static FC_Edge          *fc_e_list, *top;
static RegEntry         *region, *next_avail;
static DualTreeNode     *du_tree;
static Puncturing       *punct, *sentinel;



#ifdef DEBUG
/*****************************************************************************/
static int                      v_no;


void    print_up_envelope(start, stop)
ArrangE         *start, *stop;
{
        ArrangV         *p;

        for (;  start != NULL && start != stop;  start = start->next){
                fprintf(stderr, "\t edge along line %d ", start->punct_no);
                if ((p = start->vertex) == L_INF)
                        fprintf(stderr, "to vertex L_INF.");
                else if (p == R_INF)
                        fprintf(stderr, "to vertex R_INF.");
                else
                        fprintf(stderr, "to vertex (%.3f, %.3f)", p->x, p->y);
                fprintf(stderr, "   (r_no= %d, r_twin_no= %d)\n",
                                start->region_no, start->twin->region_no);
        }
}


void    pr_temp_arrang(t, size)
int             size;
TempLayer       *t;
{
        int             i, k;
        ArrangV         *p;
        ArrangE         *q, *nextq, *first_e, *last_e;

        if ((i = size) <= 0)
                return;

        fprintf(stderr, "\n\tLine arrangement of the lower %d line(s)\n", i);
        for (last_e = NULL, k = 0;  --i >= 0;  last_e = q->twin, t++){
                fprintf(stderr, "\nRegion (%d).\n", k++);
                for (first_e = q = t->edge_ptr;  (p = q->vertex) != R_INF;
                                                        q = nextq){
                        fprintf(stderr, "\t edge along line %d", q->punct_no);
                        fprintf(stderr, " to vertex (%.3f, %.3f)", p->x, p->y);
                        fprintf(stderr, "   (r_no= %d, r_twin_no= %d)\n",
                                        q->region_no, q->twin->region_no);
                        if ((nextq = p->next_edge) == NULL)
                                nextq = q->next;    /* first time thru vertex */
                        else{   /* a region has just finished */
                                print_up_envelope(q->next, first_e);
                                fprintf(stderr, "\nRegion (%d).\n", k++);
                                first_e = nextq;
                                if (nextq->twin->next == q->twin){
                                        p->next_edge = NULL;
                                        continue;
                                }
                        }
                        p->next_edge = nextq->twin->next;
                }
                fprintf(stderr, "\t edge along line %d", q->punct_no);
                fprintf(stderr, " to vertex R_INF.");
                fprintf(stderr, "   (r_no= %d, r_twin_no= %d)\n",
                                        q->region_no, q->twin->region_no);
                print_up_envelope(last_e, first_e);
        }
        /* print lowermost region */
        fprintf(stderr, "\nRegion (%d).\n", k++);
        print_up_envelope(last_e, NULL);
        fprintf(stderr, "\nTotal number of regions = %d\n", k);
}


void    pr_final_arrang(imin, imax)
int     imin, imax;
```

```
{
        int             diff;
        ArrangV         *p;
        ArrangE         *q, *nextq, *first_e, *last_e;
        Layer           *l;

        if ((diff = imax - imin + 1) <= 0)
                return;

        fprintf(stderr, "\nLine arrangement of the given %d line(s)\n", diff);
        for (l = layer + imin, last_e = NULL;  --diff >= 0;
                                                last_e = q->twin, l++){
                first_e = q = l->edge_ptr;
                fprintf(stderr, "\nRegion (%d).\n", q->region_no);
                for (;  (p = q->vertex) != R_INF;  q = nextq){
                        fprintf(stderr, "\t edge along line %d", q->punct_no);
                        fprintf(stderr, " to vertex (%.3f, %.3f)", p->x, p->y);
                        fprintf(stderr, "   (r_no= %d, r_twin_no= %d)\n",
                                        q->region_no, q->twin->region_no);
                        if ((nextq = p->next_edge) == NULL)
                                nextq = q->next;    /* first time thru vertex */
                        else{   /* a region has just finished */
                                print_up_envelope(q->next, first_e);
                                first_e = nextq;
                                fprintf(stderr, "\nRegion (%d).\n", nextq->region_no);
                                if (nextq->twin->next == q->twin){
                                        p->next_edge = NULL;
                                        continue;
                                }
                        }
                        p->next_edge = nextq->twin->next;
                }
                fprintf(stderr, "\t edge along line %d", q->punct_no);
                fprintf(stderr, " to vertex R_INF.");
                fprintf(stderr, "   (r_no= %d, r_twin_no= %d)\n",
                                q->region_no, q->twin->region_no);
                print_up_envelope(last_e, first_e);
        }
        /* print lowermost region */
        fprintf(stderr, "\nRegion (%d).\n", last_e->region_no);
        print_up_envelope(last_e, NULL);
}


void    print_vertices(v_number)
int     v_number;
{
        int             i;
        ArrangV         *v;

        if ((i = v_number) < 0)
                return;

        fprintf(stderr, "\n\tTable of vertices\n");
        for (v = vert;  i >= 0;  i--, v++){
                fprintf(stderr, "   x = %.3f,   y = %.3f,  ", v->x, v->y);
                if (v->next_edge == NULL)
                        fprintf(stderr, "NULL\n");
                else
                        fprintf(stderr, "not NULL !!!\n");
        }
}


void    print_layers(n)
int     n;
{
        int             i;
        ArrangV         *p;
        FC_Edge         *e;
        Layer           *l;

        fprintf(stderr, "\nLayers of the Line Arrangement.\n");
        for (i = 0, l = layer;  i < n;  i++, l++){
                fprintf(stderr, "\nLayer #%d:\n", i);
                for (e = l->fc_e_ptr;  e != NULL;  e = e->next){
                        fprintf(stderr, "\t edge on line %d", e->punct_no);
                        if ((p = e->v) == NULL)
                                fprintf(stderr, " to R_INF         ");
                        else
                                fprintf(stderr, " to (%.3f, %.3f)", p->x,p->y);
                        fprintf(stderr, "   (up_reg_no = %d,", e->up_reg_no);
                        fprintf(stderr, " low_reg_no = %d)\n", e->low_reg_no);
                }
        }
}


void    print_cascade(n)
int     n;
{
        int             i;
        ArrangV         *p;
        FC_Edge         *e;
        Layer           *l;

        fprintf(stderr, "\nFractional Cascading Data Structure.\n");
        for (i = 0, l = layer;  i < n;  i++, l++){
                fprintf(stderr, "\nLayer #%d:\n", i);
                for (e = l->fc_e_ptr;  e != NULL;  e = e->next){
                        fprintf(stderr, "\t E_%d:", e - fc_e_list);
                        fprintf(stderr, " on line %d", e->punct_no);
                        if ((p = e->v) == NULL)
                                fprintf(stderr, " to R_INF         ");
                        else
                                fprintf(stderr, " to (%.3f, %.3f)", p->x,p->y);
                        fprintf(stderr, " points to ");
                        if (e->up == NULL)
                                fprintf(stderr, " NULL/");
                        else
                                fprintf(stderr, " E_%d/", e->up - fc_e_list);
                        if (e->down == NULL)
                                fprintf(stderr, "NULL");
                        else
                                fprintf(stderr, "E_%d", e->down - fc_e_list);
                        fprintf(stderr, "  (R_%d/R_%d)\n", e->up_reg_no,
                                                                e->low_reg_no);
                }
        }
}


void    print_reg_table(table, num, t)
int             num, t;
RegEntry        *table;
{
        int     i;

        fprintf(stderr, "\nContents of table region.\n\n");
        if (t == 0)
```

```
                for (i = 0;  i < num;  i++)
                        fprintf(stderr, "Entry %d: %d, %d\n", i,
                                        table[i].d[0].dad, table[i].d[1].dad);
        else
                for (i = 0;  i < num;  i++)
                        fprintf(stderr, "Entry %d: ptr to %d, ptr to %d\n", i,
                              table[i].d[0].ptr-table, table[i].d[1].ptr-table);
}
/*****************************************************************************/
#endif



#ifdef DIAGNOSTICS
/*****************************************************************************/
void    check_p(p)              /* to assert correct execution */
ArrangV         *p;
{
        if (p == L_INF || p == R_INF || p == NULL)
                ERROR("check_p> incorrect vertex");
        if (p->next_edge != NULL)
                ERROR("check_p> next_edge field updated incorrectly");
}
/*****************************************************************************/
#endif


ArrangV         *get_vertex(x, y)
double  x, y;
{
        ArrangV         *v;

        v = vert + (++v_count);
        v->x = x;
        v->y = y;
        v->next_edge = NULL;
        return(v);
}


FC_Edge         *get_e_rec()
{
        if (++e_count > e_max)
                ERROR("set_edge_entry> edge table overflow");
        return(fc_e_list + e_count);
}

void    get_edge_pair(e1, e2, punct_no)
int             punct_no;
ArrangE         **e1, **e2;
{
        ArrangE         *q1, *q2;

        q1 = edgeptr + (++e_count);
        q2 = edgeptr + (++e_count);
        q1->region_no = q2->region_no = -1;
        q1->punct_no = q2->punct_no = punct_no;
        q1->more = edge_list;   /* insert ONLY q1 at front of valid edge list */
        edge_list = q1;
        *e1 = q1;
        *e2 = q2;
}


void    up_witness(p, f, z)
        /* test and update witness facets that lie above the puncturing */
        /* vertex, with the interior of the polyhedron being below them */
double          z;
Facet           *f;
Puncturing      *p;
{
        if (p->type == 3)
                return;
#ifdef DEBUG
        fprintf(stderr,"Puncturing vertex (%.3f, %.3f, %.3f)\n",p->x,p->y,p->z);
#endif
        if (p->up_i == 0 || E_COMPARE(p->z_above, z) < 0){
                p->up_i = 0;
                p->z_above = z;
        }
        p->up_f[p->up_i] = f;
        if (++(p->up_i) >= p->up_max)
                MY_REALLOC(p->up_f, Facet *, (p->up_max *= 2));
}


void    low_witness(p, f, z)
        /* test and update witness facets that lie below the puncturing */
        /* vertex, with the interior of the polyhedron being above them */
double          z;
Facet           *f;
Puncturing      *p;
{
        if (p->type == 1)
                return;
#ifdef DEBUG
        fprintf(stderr,"Puncturing vertex (%.3f, %.3f, %.3f)\n",p->x,p->y,p->z);
#endif
        if (p->low_i == 0 || E_COMPARE(p->z_below, z) > 0){
                p->low_i = 0;
                p->z_below = z;
        }
        p->low_f[p->low_i] = f;
        if (++(p->low_i) >= p->low_max)
                MY_REALLOC(p->low_f, Facet *, (p->low_max *= 2));
}


void    get_tables(t, num)
int             num;
TempLayer       *t;
{
        int             i, k, stop;
        ArrangV         *v, *p;
        ArrangE         *q, *nextq, *r, *first_e, *last_e;
        FC_Edge         *e1, *e;
        Layer           *l;

        /* we allocate some extra space to facilitate coding of fract-cascade */
        MY_ALLOC(layer, Layer, 3 * num / 2);
        /* allocate 2 n^2 edge records */
        MY_ALLOC(fc_e_list, FC_Edge, e_max = 2 * num * num);

        /* process topmost layer separately */
        for (l = layer, q = l->edge_ptr = t->edge_ptr, e_count = -1,
             l->fc_e_ptr = e1 = get_e_rec(), first_e = q->twin;
                                (v = q->vertex) != R_INF;  e1 = e1->next){
                e1->punct_no = q->punct_no;
                e1->up_reg_no = v->up_reg_no = q->region_no = 0;
                e1->up = e1->down = NULL;
```

```
                e1->v = v;
                e1->v_type = 1;                 /* real vertex */
                q = q->next;                    /* of course, first and ... */
                v->next_edge = q->twin->next;   /* not last time thru vertex */
                e1->next = get_e_rec();
        }
        e1->punct_no = q->punct_no;
        e1->up_reg_no = q->region_no = e1->v_type = 0;  /* 0th region; fake v */
        e1->v = NULL;
        e1->up = e1->down = e1->next = NULL;

        last_e = q->twin;
        /* process remaining layers */
        for (e = fc_e_list, k= 1, l++, t++, i = num;  --i > 0;
                        last_e = q->twin, e = l->fc_e_ptr, l++, t++, k++){
                for (q= l->edge_ptr= t->edge_ptr, l->fc_e_ptr= e1= get_e_rec();
                   (v = q->vertex) != R_INF;  e1 = e1->next, q = nextq){
                        e1->punct_no = q->punct_no;
                        e1->up_reg_no = q->region_no = k;
                        e1->up = e1->down = NULL;
                        e1->v = v;
                        e1->v_type = 1;                 /* real vertex */
                        if ((nextq = v->next_edge) == NULL){
                                nextq = q->next;    /* first time thru vertex */
                                v->up_reg_no = k;
                        }
                        else{   /* a region has just finished */
                                /* update region_no field of upper envelope */
                                do{
                                        e->low_reg_no = k;
                                        stop = (e->v == v);
                                        e = e->next;
                                } while (!stop);
                                r = q->next;
#ifdef DIAGNOSTICS
                                if (r->region_no >= 0)
                                        ERROR("get_tables> incorrect region");
#endif
                                r->region_no = k;
                                for (p= r->vertex;  (r= r->next)->region_no < 0;
                                                                p = r->vertex){
#ifdef DIAGNOSTICS
                                        check_p(p);
#endif
                                        r->region_no = p->low_reg_no = k;
                                }
                                k++;            /* increment count of regions */
                        }
                        if (nextq->twin->next == q->twin)
                                v->next_edge = NULL;    /* last time */
                        else
                                v->next_edge = nextq->twin->next;
                        e1->next = get_e_rec();
                }
                e1->punct_no = q->punct_no;
                e1->up_reg_no = q->region_no = k;
                e1->v = NULL;
                e1->v_type = 0;                 /* fake vertex */
                e1->up = e1->down = e1->next = NULL;
                /* update region_no field of upper envelope of region */
                do{
                        e->low_reg_no = k;
                } while ((e = e->next) != NULL);
                r = last_e;
#ifdef DIAGNOSTICS
                if (q->next != NULL || r->region_no >= 0)
                        ERROR("get_tables> incorrect open to the right region");
#endif
                r->region_no = k;
                for (p= r->vertex;  (r= r->next)->region_no < 0;  p= r->vertex){
#ifdef DIAGNOSTICS
                        check_p(p);
#endif
                        r->region_no = p->low_reg_no = k;
                }
                first_e->next = NULL;   /*break closing ptrs of left-open regs*/
                first_e = t->edge_ptr->twin;
        }
        region_max = k;   /* region_max incremented by 1 for lowermost region */
        /* set region_no field in boundary of lowermost region */
        do{
                e->low_reg_no = k;
        } while ((e = e->next) != NULL);
        r = last_e;
#ifdef DIAGNOSTICS
        if (r->region_no >= 0)
                ERROR("get_tables> incorrect lowermost region");
#endif
        r->region_no = region_max;
        for (p = r->vertex;  (r = r->next) != NULL;  p = r->vertex){
#ifdef DIAGNOSTICS
                check_p(p);
#endif
                r->region_no = p->low_reg_no = region_max;
        }

        (void) free((char *)temp_layer);
}


int     get_arrangmnt(num)
        /* computes the layers in the dual space (quadratic space and time) */
        /* the left-open regions have closing pointers facilitating insertion */
        /* these closing pointers are set to NULL by the routine get_tables */
int             num;
{
        int             i, j, k, cond, layer_count;
        double          slope, offset, x;
        ArrangV         *p;
        ArrangE         *e1, *e2, *q, *q_twin, *r, *s;
        TempLayer       *rec, *next_rec;

        /* allocate space for the list structure */
        MY_ALLOC(temp_layer, TempLayer, num);
        MY_ALLOC(edgeptr, ArrangE, 2 * num * num);
        MY_ALLOC(vert, ArrangV, num * (num - 1) / 2);
        edge_list = NULL;
        e_count = v_count = -1;

        /*  insert all lines, one at a time in decreasing slope */
        for (i = j = num;  --i >= 0;  next_rec = rec){
                for (slope = punct[i].x, offset = punct[i].y;  --i >= 0; )
                        if (E_COMPARE(slope, punct[i].x) != 0
                                        || E_COMPARE(offset, punct[i].y) != 0)
                                break;
                slope = punct[++i].x;
                offset = punct[i].y;

                /* allocate a pair of edge-records for the current edge */
                get_edge_pair(&q, &q_twin, i);
```

```
                q->twin = q_twin;       q_twin->twin = q;
                /* link the layer-record to the corresponding edge-record */
                rec = temp_layer + --j;
                rec->edge_ptr = q;

                q_twin->vertex = L_INF;
                if (j == num - 1){
                        q->vertex = R_INF;
                        /* TOPMOST and LOWERMOST regions are OPEN */
                        q->next = q_twin->next = NULL;
                        continue;       /* do not do anything else */
                }

                q_twin->next = s = next_rec->edge_ptr;
                /* note: q->vertex, q->next and ptr to q_twin to be set later */

                /* update line arrangement.  s points to 1st edge below rec */
                for (;  ;  s = s->next){
                        k = s->punct_no;
                        if ((p = s->vertex) == R_INF){
                                /* deal with rays infinite to the right */
                                if ((cond = E_COMPARE(punct[k].x, slope)) == 0)
                                        break;          /* exit for loop */
#ifdef DIAGNOSTICS
                                if (cond < 0)
                                        ERROR("get_arrangmnt> incorrect sorting");
#endif
                            /* note: if cond = 1, our line will cross an edge */
                            /* So, cond correctly used in switch stmnt below  */
                        }
                        else if (p == L_INF){   /* rays infinite to the left */
                                if (s->next == NULL)    /* lowermost region */
                                        break;
                                else
                                        continue;       /* keep moving around */
                        }
                        else if ((cond = E_COMPARE(p->y,
                                        slope * p->x + offset)) < 0)
                                continue;       /* keep moving around region */

                        switch (cond){
                        case 0: /* line passes through a vertex */
                                /* locate appropriate new region, set e1 & e2 */
                                q->vertex = p;
                                r = q->next = s->next;
                                s->next = q_twin;
                                for (r = r->twin;  r->punct_no != k;  )
                                        r = r->next->twin;
                                e1 = r;
                                k = (e2 = r->next)->punct_no;
                                break;
                        case 1: /* line crosses edge */
                                e1 = s->twin;
                                get_edge_pair(&r, &e2, k);
                                /* update new edge record r, twin to e1 */
                                r->vertex = p;          /* p = s->vertex */
                                r->next = s->next;
                                s->next = q_twin;
                                r->twin = e1;   e1->twin = r;
                                q->next = r;

                                /* update new edge record e2, twin to s */
                                e2->vertex = e1->vertex;
                                e2->next = e1->next;
                                e2->twin = s;   s->twin = e2;
                                /* compute point of intersection */
                                x = -(offset - punct[k].y)/(slope - punct[k].x);
                                p = get_vertex(x, slope * x + offset);
                                q->vertex = s->vertex = e1->vertex = p;
                                break;
                        }

                        /* update edge-records for new edge in new region */
                        /* e1, e2 point to edges before and after this edge */
                        /* p points to the previous vertex */
                        get_edge_pair(&q, &q_twin, i);
                        q->twin = q_twin;       q_twin->twin = q;
                        e1->next = q;
                        q_twin->next = e2;
                        q_twin->vertex = p;
                        /* q->vertex, q->next, ptr to q_twin to be set later */

                        /* invariant: k is equal to e2->punct_no */
                        if ((s = e2)->vertex == L_INF && s->next == NULL)
                                break;          /* lowermost region */
                }
                q->vertex = R_INF;
                q->next = NULL;

#ifdef ALLDEBUG
                pr_temp_arrang(rec, num - j);
                print_vertices(v_count);
#endif
        }
        layer_count = num - j;
#ifdef DIAGNOSTICS
        if (layer_count <= 0)
                ERROR("get_arrangmnt> layer table is empty");
#endif

        /* copy the edge list into a table to facilitate binary search */
        get_tables(rec, layer_count);

#ifdef DEBUG
        pr_final_arrang(0, layer_count - 1);
        print_vertices(v_no = v_count);
#endif
        return(layer_count);
}


void    merge_left(child, parent)
Layer           *child, *parent;
{
        int             cond;
        ArrangV         *v1, *v2;
        FC_Edge         *s, *s_new, *t, *old_t;

        for (s = parent->fc_e_ptr, t = child->fc_e_ptr;
                        (old_t = t)->v != NULL && (v1 = (t = t->next)->v) != NULL;
                                                t = t->next, s = s->next){
                for (;  ;  s = s->next){
                        s->up = old_t;
                        if ((v2 = s->v) == NULL){ /* parent's layer exhausted */
                                /* copy child's every other vertex to parent */
                                s->v_type = 0;          /* fake vertex */
                                do{
                                        s_new = get_e_rec();
                                        *s_new = *s;
```

```
                                s->v = v1;          /* only records */
                                s->next = s_new;        /* to change */
                                s = s_new;
                                s->up = t = t->next;
                        } while (t->v != NULL
                                && (v1 = (t = t->next)->v) != NULL);
                        s->v = NULL;
                        s->next = NULL;
                        return;
                }
                if ((cond = E_COMPARE(v2->x, v1->x)) >= 0)
                        break;
        }
        if (cond == 1){
                s_new = get_e_rec();
                *s_new = *s;
                s->v = v1;
                s->v_type = 0;          /* fake vertex */
                s_new->next = s->next;
                s->next = s_new;
        }
    }
    do{             /* make all parent up-pointers point to old_t */
        s->up = old_t;
    } while ((s = s->next) != NULL);
}


void    merge_right(child, parent)
Layer           *child, *parent;
{
        int             cond;
        ArrangV         *v1, *v2;
        FC_Edge         *s, *s_new, *t, *old_t;

        for (s = parent->fc_e_ptr, t = child->fc_e_ptr;
                (old_t = t)->v != NULL && (v1 = (t = t->next)->v) != NULL;
                                        t = t->next, s = s->next){
                for (;  ;  s = s->next){
                        s->down = old_t;
                        if ((v2 = s->v) == NULL){ /* parent's layer exhausted */
                                /* copy child's every other vertex to parent */
                                s->v_type = 0;          /* fake vertex */
                                do{
                                        s_new = get_e_rec();
                                        *s_new = *s;
                                        s->v = v1;          /* only records */
                                        s->next = s_new;        /* to change */
                                        s = s_new;
                                        s->down = t = t->next;
                                } while (t->v != NULL
                                        && (v1 = (t = t->next)->v) != NULL);
                                s->v = NULL;
                                s->next = NULL;
                                return;
                        }
                        if ((cond = E_COMPARE(v2->x, v1->x)) >= 0)
                                break;
                }
                if (cond == 1){
                        s_new = get_e_rec();
                        *s_new = *s;
                        s->v = v1;
                        s->v_type = 0;          /* fake vertex */
                        s_new->next = s->next;
                        s->next = s_new;
                }
        }
        do{             /* make all parent down-pointers point to old_t */
                s->down = old_t;
        } while ((s = s->next) != NULL);
}


void    fr_cascade(l_table, l_num)
int             l_num;
Layer           *l_table;
{
        int             halfstep, step;
        FC_Edge         *e, *t;
        Layer           *child, *parent, *next, *last;

        for (halfstep = 1, step = 2, last = (child = next = l_table) + l_num -1;
                                (next += halfstep) <= last;
                                child = next, halfstep = step, step <<= 1)
                for (;  child <= last;  child += step){
                        if ((parent = child + halfstep) > last){
                                /* make parent point to child's layer */
#ifdef ALLDEBUG
        fprintf(stderr, "copy %d to %d\n", child-l_table, parent-l_table);
#endif
                                *parent = *child;
                                last = parent;
                        }
                        else{
#ifdef ALLDEBUG
        fprintf(stderr, "merge %d into %d\n", child-l_table, parent-l_table);
#endif
                                merge_left(child, parent);
                                if ((child = parent + halfstep) > last){
#ifdef ALLDEBUG
        fprintf(stderr, "set down ptrs of %d to NULL\n", parent-l_table);
#endif
                                        e = parent->fc_e_ptr;
                                        do{ /* set parent's down-ptrs to NULL */
                                                e->down = NULL;
                                        } while ((e = e->next) != NULL);
                                }
                                else{
#ifdef ALLDEBUG
        fprintf(stderr, "merge %d into %d\n", child-l_table, parent-l_table);
#endif
                                        merge_right(child, parent);
                                }
                        }
                }

        /* copy topmost layer in array top for easy binary search */
        MY_ALLOC(top, FC_Edge, l_num * (l_num + 1) / 2);
        for (e = parent->fc_e_ptr, t = top;  e != NULL;  e = e->next, t++)
                *t = *e;
        top_size = t - top;
}


int     locate_reg(x_coord, y_coord, sel)
   /* locate and return index of region in which (x_coord, y_coord) falls */
   /* sel is 1 if we are interested in region above the point, -1 else,   */
   /* in case the point falls on an edge of the arrangement               */
int     sel;
```

```
double  x_coord, y_coord;
{
        int             left, right, middle, cond;
        ArrangV         *v;
        FC_Edge         *t;
        Puncturing      p;

        /* do binary search on the topmost layer */
        if ((right= top_size -2) < 0 || E_COMPARE(x_coord, top[right].v->x) > 0)
                /* a one-edge chain, or point to right of rightmost vertex */
                right++;
        else
                for (left = 0;  left < right;  ){
                        middle = (left + right) / 2;
                        switch (E_COMPARE(x_coord, top[middle].v->x)){
                        case -1: right = middle;
                                 break;
                        case  1: left = middle + 1;
                                 break;
                        case  0: left = right = middle;
                                 break;
                        }
                }
        t = top + right;

        /* use the fractional cascading structure to find answer */
        for (cond = 1;  ; ){
                if ((v=t->v) != NULL && (cond = E_COMPARE(x_coord, v->x)) > 0){
                        t = t->next;
                        if ((v = t->v) != NULL)
                                cond = E_COMPARE(x_coord, v->x);
                        /* otherwise cond will be equal to 1, which is fine */
                }
                p = punct[t->punct_no];
                switch (E_COMPARE(y_coord, p.x * x_coord + p.y)){
                case -1: if (t->down == NULL)
                                return(t->low_reg_no);
                         t = t->down;
                         break;
                case  1: if (t->up == NULL)
                                return(t->up_reg_no);
                         t = t->up;
                         break;
                case  0: if (cond == 0 && t->v_type != 0)
                                if (sel == 1)
                                        return(v->up_reg_no);
                                else
                                        return(v->low_reg_no);
                         else
                                if (sel == 1)
                                        return(t->up_reg_no);
                                else
                                        return(t->low_reg_no);
                }
        }
}


void    merge(e_list, separator, table, table_size)
int             separator, table_size;
ArrangE         *e_list;
RegEntry        *table;
{
        int             h, i, j, k1, k2, m;
        ArrangE         *q;

        for (q = e_list;  q != NULL;  q = q->more){
                if (q->punct_no <= separator)
                        m = 1;          /* union regions in right arrangement */
                else
                        m = 0;          /* union regions in left arrangement */

                h = q->region_no;
                for (i = h;  (k1 = table[i].d[m].dad) >= 0;  i = k1)
#ifdef DIAGNOSTICS
                        if (k1 >= table_size)
                                ERROR("merge> reference outside table bounds")
#endif
                        ;
                for (k1 = table[h].d[m].dad;  k1 >= 0;  ){
                        table[h].d[m].dad = i;  /* path compression */
                        h = k1;
                        k1 = table[h].d[m].dad;
                }
                h = q->twin->region_no;
                for (j = h;  (k2 = table[j].d[m].dad) >= 0;  j = k2)
#ifdef DIAGNOSTICS
                        if (k2 >= table_size)
                                ERROR("merge> reference outside table bounds")
#endif
                        ;
                for (k2 = table[h].d[m].dad;  k2 >= 0;  ){
                        table[h].d[m].dad = j;  /* path compression */
                        h = k2;
                        k2 = table[h].d[m].dad;
                }
                if (i != j)
                        if (k2 < k1){
                                table[j].d[m].dad += k1;
                                table[i].d[m].dad = j;
                        }
                        else{
                                table[i].d[m].dad += k2;
                                table[j].d[m].dad = i;
                        }
        }
}


int     equiv_name(table, table_size, c)
int             table_size, c;
RegEntry        *table;
{
        int     h, i, j, k, m;

        for (i = m = 0;  i < table_size;  i++){
                for (h = i;  (h = table[h].d[c].dad) > i; )
                        ;
#ifdef DIAGNOSTICS
                if (h == i)
                        ERROR("equiv_name> region pointer pointing to itself");
#endif
                if (h < 0){     /* new equivalence class */
                        for (k = table[i].d[c].dad;  k > i;  k = j){
                                j = table[k].d[c].dad;
                                table[k].d[c].dad = i;  /*make them point to i*/
                        }
                        table[i].d[c].dad = m++;
                }
                else{
```

```
                        for (k = table[i].d[c].dad;  k > i;  k = j){
                                j = table[k].d[c].dad;
                                table[k].d[c].dad = h;  /*make them point to h*/
                        }
                        table[i].d[c].dad = table[h].d[c].dad;
                }
        }
        return(m);
}


int     disconnect(e, separator)
int             separator;
ArrangE         *e;
{
        int             k;
        ArrangE         *q1, *q1_next, *q2, *q3, *q3_next;

        for (q1 = e, q2 = e->next;  q2->punct_no > separator;
                                        q1 = q2->twin, q2 = q1->next)
                ;       /* skip red edges */
        q1_next = q2;
        k = e->punct_no;        /* punct_no of topmost green edge thru vertex */
        for (q2 = q2->twin;  q2->punct_no != k;  q2 = q2->next->twin)
                ;
        k = q1->punct_no;       /* punct_no of last red edge thru vertex */
        for (q3 = q2->next->twin;  q3->punct_no != k;  q3 = q3->next->twin)
                ;
        q3_next = q3->next;

        if (e->next == q1->twin){       /* single red crossing line */
                q1->vertex = q2->next->vertex;
                q1->next = q2->next->next;
                q3->vertex = e->next->vertex;
                q3->next = e->next->next;
                q1->twin = q3;
                q3->twin = q1;
        }
        else{           /* more than one red line */
                q1->next = q2->next;
                q3->next = e->next;
        }

        if (q1_next->twin == q2){       /* single green line */
                e->vertex = q1_next->vertex;
                e->next = q1_next->next;
                q2->vertex = q3_next->vertex;
                q2->next = q3_next->next;
                e->twin = q2;
                q2->twin = e;
                return(1);
        }
        else{           /* more than one green line */
                e->next = q1_next;
                q2->next = q3_next;
        }
        return(0);
}


void    upd_save(c, q, listptr, table)
int             c;
ArrangE         *q, **listptr;
RegEntry        *table;
{
        /* update region_no field */
        q->region_no = table[q->region_no].d[c].dad;
        q->twin->region_no = table[q->twin->region_no].d[c].dad;
        /* insert in list of valid edges */
        q->more = *listptr;
        *listptr = q;           /* I do not have to insert q->twin too */
}


void    split_arrang(i_min, i_max, i_middle, separator, table, l_listptr,
                                                                r_listptr)
int             i_min, i_max, i_middle, separator;
ArrangE         **l_listptr, **r_listptr;
RegEntry        *table;
{
        int             i;
        ArrangV         *p;
        ArrangE         *q, *nextq;

        for (i = i_min;  i <= i_middle;  i++){
                for (q = layer[i].edge_ptr;  (p = q->vertex) != R_INF; ){
                if ((nextq = p->next_edge) == NULL){
                        /* first time through vertex */
                        if (q->next->punct_no > separator  /* red edge*/
                         && disconnect(q, separator) != 0)
                                continue;       /* process q again */
                        nextq = q->next;        /*to be assigned to q*/
                }
                upd_save(0, q, l_listptr, table);
                if (nextq->twin->next == q->twin)
                        p->next_edge = NULL;    /* last time */
                else
                        p->next_edge = nextq->twin->next;
                q = nextq;
                }
                upd_save(0, q, l_listptr, table);       /* last edge on layer */
        }

        /* go on with red_arrangement, updating ONLY region_no field */
        for (;  i <= i_max;  i++){
                for (q = layer[i].edge_ptr; (p=q->vertex) != R_INF; q = nextq){
                if ((nextq = p->next_edge) == NULL)
                        nextq = q->next;    /* first time thru vertex */
                upd_save(1, q, r_listptr, table);
                if (nextq->twin->next == q->twin)
                        p->next_edge = NULL;    /* last time */
                else
                        p->next_edge = nextq->twin->next;
                }
                upd_save(1, q, r_listptr, table);       /* last edge on layer */
        }
}


DualTreeNode    *build_tree(imin, imax, table, table_size, list)
int             imin, imax, table_size;
ArrangE         *list;
RegEntry        *table;
{
        int             i, mid_layer, separator, l_no, r_no;
        ArrangE         *left_list, *right_list;
        RegEntry        *left_table, *right_table;
        DualTreeNode    *t;

        t = du_tree + v_count++;
```

```
        if (imin == imax){
                t->left = NULL;         /* this marks a leaf */
                i = layer[imin].edge_ptr->punct_no;
                t->right = (DualTreeNode *)(punct + i); /*ptr to punct*/

                /* reserve 2 slots in table of regions  (playing it safe) */
                table->d[0].ptr = table->d[1].ptr = table;
                table++;
                table->d[0].ptr = table->d[1].ptr = table;
                next_avail += 2;
                return(t);
        }

        mid_layer = (imin + imax) / 2;
        separator = layer[mid_layer].edge_ptr->punct_no;
        t->x_maxleft = punct[separator].x;
        t->x_minright = punct[layer[mid_layer+1].edge_ptr->punct_no].x;

        /* reserve space in the table of regions */
        next_avail += table_size;
        merge(list, separator, table, table_size);

#ifdef ALLDEBUG
        print_reg_table(table, table_size, 0);
#endif

        l_no = equiv_name(table, table_size, 0);
        r_no = equiv_name(table, table_size, 1);

#ifdef ALLDEBUG
        print_reg_table(table, table_size, 0);
#endif

        left_list = right_list = NULL;
        split_arrang(imin, imax, mid_layer, separator, table, &left_list,
                                                              &right_list);
#ifdef ALLDEBUG
        pr_final_arrang(imin, mid_layer);
        pr_final_arrang(mid_layer + 1, imax);
        print_vertices(v_no);
#endif

        left_table = next_avail;
        t->left = build_tree(imin, mid_layer, left_table, l_no, left_list);
        right_table = next_avail;
        t->right = build_tree(mid_layer+1, imax, right_table, r_no, right_list);

        /* establish the pointers between the tables of regions */
        for (i = 0;  i < table_size;  i++){
                table[i].d[0].ptr = left_table + table[i].d[0].dad;
                table[i].d[1].ptr = right_table + table[i].d[1].dad;
        }
        return(t);
}


void    dualize(root, table)
DualTreeRoot    *root;
Table           *table;
        /* table of lines is assumed to be sorted by increasing slope and */
        /* decreasing offset for equal slopes */
{
        int                     i, n, layer_no;
#ifdef DIAGNOSTICS
        int                     reg_size;
#endif

        punct = (Puncturing *)(table->to_elem);
        sentinel = punct + (n = table->n);
        L_INF = &l_inf_rec;
        R_INF = &r_inf_rec;

        layer_no = get_arrangmnt(n);
#ifdef DEBUG
        print_layers(layer_no);
#endif

        fr_cascade(layer, layer_no);
#ifdef DEBUG
        print_cascade(layer_no);
#endif

        /* allocate space for the tables of regions (initialized to -1) */
        MY_ALLOC(region, RegEntry, (i = 3*layer_no*(layer_no+1)/2));
#ifdef DIAGNOSTICS
        reg_size = i;
#endif
        for (;  --i >= 0;  )    /* set all entries to -1 */
                region[i].d[0].dad = region[i].d[1].dad = -1;
        next_avail = region;
        MY_ALLOC(du_tree, DualTreeNode, 2 * layer_no);
        v_count = 0;    /* next available tree node */

        /* create tree of pointers between regions of arrangements */
        root->l_num = layer_no;
        root->p_num = n;
        root->x_min = punct[0].x;
        root->x_max = punct[n-1].x;
        root->table = region;
        root->t_node = build_tree(0, layer_no-1, region, region_max+1, edge_list);
        /* note : root->t_node == du_tree */
#ifdef DIAGNOSTICS
        if (next_avail > region + reg_size)
                ERROR("dualize> _region_ table overflow");
#endif
#ifdef DEBUG
        print_reg_table(region, next_avail - region, 1);
#endif

        /* throw away the space allocated by the edge list structure */
        free((char *)edgeptr);
}


void    du_clean(r)
DualTreeRoot    *r;
{
        free((char *)vert);
        free((char *)layer);
        free((char *)top);
        free((char *)fc_e_list);
        free((char *)r->table);
        free((char *)r->t_node);
}


void    exhaust(t, r1, r2, facet)
DualTreeNode    *t;
RegEntry        *r1, *r2;
```

```
Facet           *facet;
        /* r1 and r2 are assumed to point to different regions */
{
        double          x, y, z0;
        Puncturing      *p;
        RegEntry        *temp1, *temp2;

        if (t->left == NULL){   /* report intersection(s) */
                p = (Puncturing *)t->right;
                z0 = FROM_FACET_COMP_z(*p, *facet);
                if (E_COMPARE(p->z, z0) == facet->sign)
                        update_witness(p, facet, z0);
                for(x=p->x, y=p->y;  ++p != sentinel && E_COMPARE(x, p->x) == 0
                                                && E_COMPARE(y, p->y) == 0; )
                        if (E_COMPARE(p->z, z0) == facet->sign)
                                update_witness(p, facet, z0);
        }
        else{
                if ((temp1 = r1->d[0].ptr) != (temp2 = r2->d[0].ptr))
                        exhaust(t->left, temp1, temp2, facet);  /*left subtree*/
                if ((temp1 = r1->d[1].ptr) != (temp2 = r2->d[1].ptr))
                        exhaust(t->right, temp1, temp2, facet); /*right subtree*/
        }
}


void    visit(t, xleft, xright, c, l, r, r1, r2, f)
int             c, l, r;
        /* c is 1 if right side is in, 0 else */
        /* l <= 0 if left triangle boundary on or to left of strip, 1 else */
        /* r >= 0 if right triangle boundary on or to right of strip, -1 else */
double          xleft, xright;  /* x-bounds of triangle */
Facet           *f;
RegEntry        *r1, *r2;       /* r1 != r2 */
DualTreeNode    *t;
{
        RegEntry        *temp1, *temp2;

        for (;  ;  ){
#ifdef DIAGNOSTICS
                if (t->left == NULL)
                        ERROR("visit> incorrect tree structure");
#endif

                if (r < 1-c && E_COMPARE(xright, t->x_minright) <= -c){
                                /* triangle does not intersect right subtree */
                        if ((r1 = r1->d[0].ptr) == (r2 = r2->d[0].ptr))
                                return;
                        if ((r = E_COMPARE(xright, t->x_maxleft)) >= 1-c){
                                exhaust(t->left, r1, r2, f);
                                return;
                        }
                        t = t->left;
                }
                else if (l > 0 && E_COMPARE(xleft, t->x_maxleft) > 0){
                                /* triangle does not intersect left subtree */
                        if ((r1 = r1->d[1].ptr) == (r2 = r2->d[1].ptr))
                                return;
                        if ((l = E_COMPARE(xleft, t->x_minright)) <= 0){
                                exhaust(t->right, r1, r2, f);
                                return;
                        }
                        t = t->right;
                }
                else
                        break;
        }

        /* triangle intersects both left and right subtrees */
        if ((temp1 = r1->d[1].ptr) != (temp2 = r2->d[1].ptr))
                /* recurse right */
                if (r >= 1-c)
                        exhaust(t->right, temp1, temp2, f);
                else
                        visit(t->right, xleft, xright, c, 0, r, temp1, temp2, f);
        if ((temp1 = r1->d[0].ptr) != (temp2 = r2->d[0].ptr))
                /* recurse left */
                if (l <= 0)
                        exhaust(t->left, temp1, temp2, f);
                else{           /* note: no problem assigning new value to r */
                        r = E_COMPARE(xright, t->x_maxleft);
                        visit(t->left, xleft, xright, c, l, r, temp1, temp2, f);
                }
}


void    query(f, root)
Facet           *f;
DualTreeRoot    *root;
{
        int             c0, c1, c2;
        double          x1, x2, y2, x3, l_bound, r_bound;
        Edge            *e, *e1, *e2, *e3;
        RegEntry        *table, *r1, *r2, *r3;
        DualTreeNode    *tree;

        l_bound = root->x_min;
        r_bound = root->x_max;
        tree = root->t_node;
        table = root->table;

        /* Assumption : vertex[0], vertex[1] and vertex[2] lex-sorted */
        x1 = f->vertex[0]->x;   x2 = f->vertex[1]->x;   x3 = f->vertex[2]->x;
        y2 = f->vertex[1]->y;
        /* e1 joins vertex[0] & vertex[1], and similarly for e2 and e3 */
        for (c0 = 3;  --c0 >= 0;  )
                if ((e = f->edge[c0])->vertex[1] == f->vertex[1])
                        e1 = e;
                else
                        if (e->vertex[0] == f->vertex[1])
                                e2 = e;
                        else
                                e3 = e;
        switch (f->sign){
        case -1: update_witness = up_witness;   /* polyhedron below facet */
                 break;
        case  1: update_witness = low_witness;  /* polyhedron above facet */
                 break;
        }
        /* determine whether v2 is above or below edge v1-v3 */
        c0 = E_COMPARE(y2, e3->a * x2 + e3->c);

#ifdef DEBUG
        fprintf(stderr, "\nfacet:  left_x=%.3f, middle_x=%.3f, right_x=%.3f\n",
                                                                x1, x2, x3);
        fprintf(stderr, "\tL1=(%.3f, %.3f), L2=(%.3f, %.3f), L3=(%.3f, %.3f)\n",
                                e1->a, e1->c, e2->a, e2->c, e3->a, e3->c);
#endif
#ifdef DIAGNOSTICS
        if (E_COMPARE(x1, x2) > 0 || E_COMPARE(x2, x3) > 0)
```

```
                ERROR("query> vertices of facet not x-sorted");
        if (c0 == 0)
                ERROR("query> facet too small");
#endif

        /* break facet into at most two sub-triangles */
        r3 = NULL;
        if (E_COMPARE(x1, x2) >= 0)
                x2 = x1;        /* edge v1-v2 is vertical */
        else
                if (E_COMPARE(x2, x3) >= 0){
                        x2 = x1;        /* edge v2-v3 is vertical */
                        e2 = e1;
                }
                else{
                        /* call for half-open piece of facet; x1, x2 & e1, e3 */
#ifdef ALLDEBUG
fprintf(stderr, "   left part:  l_x=%.3f, r_x=%.3f, ", x1, x2);
fprintf(stderr, "L1=(%.3f, %.3f), L2=(%.3f, %.3f)\n", e1->a,e1->c,e3->a,e3->c);
#endif
                        if (E_COMPARE(x2, l_bound) > 0          /*not off left*/
                         && E_COMPARE(x1, r_bound) <= 0){      /*not off right*/
                                r1 = table + locate_reg(-e1->a, e1->c, c0);
                                r3 = table + locate_reg(-e3->a, e3->c,-c0);
#ifdef ALLDEBUG
fprintf(stderr,"   point (%.3f, %.3f) in region %d\n", -e1->a, e1->c, r1-table);
fprintf(stderr,"   point (%.3f, %.3f) in region %d\n", -e3->a, e3->c, r3-table);
#endif
                                if (r1 != r3){
                                        c1 = E_COMPARE(x1, l_bound);
                                        c2 = E_COMPARE(x2, r_bound);
                                        if (c1 <= 0 && c2 > 0)
                                                exhaust(tree, r1, r3, f);
                                        else
                                                visit(tree, x1, x2, 0, c1, c2,
                                                                r1, r3, f);
                                }
                        }
                }
        /* call for the closed piece of facet; x2, x3 & e2, e3 */
#ifdef ALLDEBUG
fprintf(stderr, "   right part:  l_x=%.3f, r_x=%.3f, ", x2, x3);
fprintf(stderr, "L1=(%.3f, %.3f), L2=(%.3f, %.3f)\n", e2->a,e2->c,e3->a,e3->c);
#endif
        if (E_COMPARE(x3, l_bound) >= 0         /* not off to left */
         && E_COMPARE(x2, r_bound) <= 0){       /* not off to right */
                r2 = table + locate_reg(-e2->a, e2->c, c0);
                if (r3 == NULL)
                        r3 = table + locate_reg(-e3->a, e3->c, -c0);
#ifdef ALLDEBUG
fprintf(stderr,"   point (%.3f, %.3f) in region %d\n", -e2->a, e2->c, r2-table);
fprintf(stderr,"   point (%.3f, %.3f) in region %d\n", -e3->a, e3->c, r3-table);
#endif
                if (r2 != r3){
                        c1 = E_COMPARE(x2, l_bound);
                        c2 = E_COMPARE(x3, r_bound);
                        if (c1 <= 0 && c2 >= 0)
                                exhaust(tree, r2, r3, f);
                        else
                                visit(tree, x2, x3, 1, c1, c2, r2, r3, f);
                }
        }
}
```

```
   /*********                   pull_off.c                        *********/
   /*            Routines for pointed and flat vertex removal           */
   /*  For a pointed vertex, its crown is first computed (pointed_re-   */
   /*  move), and then its dome is computed and stitched at the boun-   */
   /*  dary of the polyhedron (repair).                                 */
   /*  NOTE: The cup of the pointed vertex is known to be unhindered.   */
   /*  Removing a flat vertex is somehow similar (flat_remove).         */

#include "include.h"


typedef struct crown_rec {
                int             sf_no;
                Vertex          *v1, *v2;
                Edge            *e, *e2;
                Facet           *f;
} CrownRec;


void            plane_coef_compute(), set_e(), set_f();
int             xy_v_angle(), yz_v_angle(), xz_v_angle(), xyz_v_compare();

extern Polyhedron       P;
extern Table            tetrhdr_table;


static int              e_ind, f_ind;
static CrownRec         e_table[D];
static Edge             *free_edge[D];
static Facet            *free_facet[D];


#ifdef DEBUG
/*****************************************************************************/
void    print_crown(crown)
CrownRec        *crown;
{
        CrownRec        *e_ptr;

        fprintf(stderr, "\nCrown:\n");
        e_ptr = crown - 1;
        do{
                e_ptr++;
                fprintf(stderr, "sf_no = %d,   v1 = %d,",
                                        e_ptr->sf_no, e_ptr->v1 - P.vertices);
                if (e_ptr->v2 == NULL)
                        fprintf(stderr, "  v2 = NULL,");
                else
                        fprintf(stderr, "  v2 = %d,", e_ptr->v2 - P.vertices);
                fprintf(stderr, "   e = %d,  e2 = %d,",
                                        e_ptr->e - P.edges, e_ptr->e2 -P.edges);
                fprintf(stderr, "   f = %d\n", e_ptr->f - P.facets);
        } while (e_ptr->v2 != NULL);
}
/*****************************************************************************/
#endif



void    collect_4hedron(v1, v2, v3, v4)
Vertex          *v1,*v2,*v3,*v4;
{
        Tetrahedron     t;

        COPY_POINT_3(t.p[0], *v1);
        COPY_POINT_3(t.p[1], *v2);
        COPY_POINT_3(t.p[2], *v3);
        COPY_POINT_3(t.p[3], *v4);
        INS_TABLE(t, Tetrahedron, &tetrhdr_table);
#ifdef DEBUG
        fprintf(stderr, "tetrahedron: ");
        fprintf(stderr, " (%.2f, %.2f, %.2f),", v1->x, v1->y, v1->z);
        fprintf(stderr, " (%.2f, %.2f, %.2f),", v2->x, v2->y, v2->z);
        fprintf(stderr, " (%.2f, %.2f, %.2f),", v3->x, v3->y, v3->z);
        fprintf(stderr, " (%.2f, %.2f, %.2f)\n", v4->x, v4->y, v4->z);
#endif
}


void    get_values(apex, vrtx, edge, facet, newv_ptr, e1_ptr, e2_ptr)
Vertex          *apex, *vrtx, **newv_ptr;
Edge            *edge, **e1_ptr, **e2_ptr;
Facet           *facet;
{
        int             i;
        Vertex          *newv, *v;
        Edge            *e;

        for (i = 3;  (newv = facet->vertex[--i]) == apex || newv == vrtx; )
                ;
        *newv_ptr = newv;
        for (i = 2;  i >= 0;  i--){
                if ((e = facet->edge[i]) == edge)
                        continue;
                if ((v = e->vertex[0]) == apex)
                        *e1_ptr = e;
                else
                        if (v == vrtx)
                                *e2_ptr = e;
                        else            /* v == newv */
                                if (e->vertex[1] == apex)
                                        *e1_ptr = e;
                                else
                                        *e2_ptr = e;
        }
}



void    flip_diagonal(v, v1, v2, v3, e2, e3, d1, f1, f2)
Vertex          *v, *v1, *v2, *v3;
Edge            *e2, *e3, *d1;
Facet           *f1, *f2;
{
        int     i, j;

        if (v2->facet == f1)
                v2->facet = f2;
        if (e3->facet[0] == f2)
                e3->facet[0] = f1;
        else
                e3->facet[1] = f1;
        if (d1->facet[0] == f1)
                d1->facet[0] = f2;
        else
                d1->facet[1] = f2;
        set_e(e2, v1, v3);              /* update flipped diagonal */
        for (i = 3;  f1->vertex[--i] != v2;  )
                ;
```

```
        f1->vertex[i] = v3;     /* vertices not lex-sorted; don't bother */
        for (i = 3;  f1->edge[--i] != d1;  )
                ;
        f1->edge[i] = e3;
        /* substitute v by v1 in facet f2 and lex-sort vertices of f2 */
        for (i = 3;  (v2 = f2->vertex[--i]) != v;  )
                if (xyz_v_compare(v1, v2) > 0){
                        f2->vertex[i] = v1;
                        v1 = v2;
                }
        for (j = i;  --j >= 0;  i = j)
                if (xyz_v_compare((v2 = f2->vertex[j]), v1) > 0)
                        f2->vertex[i] = v2;
                else
                        break;
        f2->vertex[i] = v1;
        for (i = 3;  f2->edge[--i] != e3;  )
                ;
        f2->edge[i] = d1;
}



void    flat_remove(apex, v2, v3, e3, f2, list, j_ptr, size)
int     *j_ptr, size;
Vertex  *apex, *v2, *v3, **list;
Edge    *e3;
Facet   *f2;
{
        int             ind, cond, noncolnr, wrt_apex, (*angle)();
        double          a, b, c, temp, (*diff1)(), (*diff2)(), x_v_diff(),
                        y_v_diff(), z_v_diff();
        Vertex          *v1, *q;
        Edge            *e1, *e2, *d1, *d2;
        Facet           *f1, *temp_f, *sentinel;
        CrownRec        *crown, *e_ptr, *base;

#ifdef DEBUG
        fprintf(stderr, "\nRemoving flat vertex %d\n", apex - P.vertices);
#endif
        /* locate appropriate projective plane */
        if ((a = f2->a) < 0)    a = -a;
        if ((b = f2->b) < 0)    b = -b;
        if ((c = f2->c) < 0)    c = -c;
        if (E_COMPARE(a, b) <= 0)
                if (E_COMPARE(b, c) <= 0){
                        diff1 = x_v_diff;   diff2 = y_v_diff;
                        angle = xy_v_angle;
                }
                else{
                        diff1 = x_v_diff;   diff2 = z_v_diff;
                        angle = xz_v_angle;
                }
        else
                if (E_COMPARE(a, c) <= 0){
                        diff1 = x_v_diff;   diff2 = y_v_diff;
                        angle = xy_v_angle;
                }
                else{
                        diff1 = y_v_diff;   diff2 = z_v_diff;
                        angle = yz_v_angle;
                }

        /* locate leftmost lowermost vertex incident upon the apex */
        sentinel = temp_f = f2;
        do{             /* v2 stores ptr to current leftmost lowermost vertex */
                f1 = temp_f;   v1 = v3;   e1 = e3;
                if ((temp_f = e1->facet[0]) == f1)
                        temp_f = e1->facet[1];
                for (e3 = temp_f->edge[ind = 2];
                                e3->vertex[0] == v1 || e3->vertex[1] == v1; )
                        e3 = temp_f->edge[--ind];
                if ((v3 = e3->vertex[0]) == apex)
                        v3 = e3 ->vertex[1];
                if ((cond = diff1(v1, v2)) < 0 ||
                                        cond == 0 && diff2(v1, v2) <= 0){
                        v2 = v1;
                        e2 = e1;
                        f2 = temp_f;
                }
        } while (temp_f != sentinel);
#ifdef DIAGNOSTICS
        fprintf(stderr, "leftmost lowermost vertex [%d]\n", v2 - P.vertices);
#endif

        /* locate a triangle/quadrilateral containing apex; retriangulate */
        get_values(apex, v2, e2, f2, &v3, &e3, &d2);
        e_ptr = (crown = e_table) - 1;
        f_ind = 0;
        cond = angle((q = v2), v3, apex);       /* store leftmost vertex in q */
#ifdef DIAGNOSTICS
        if (cond == 0)
                ERROR("flat_remove> degenerate cup facet");
#endif

        for (base = ++e_ptr, noncolnr = 1, f_ind++;  ;  e_ptr++){
                /* update edge record in crown, and store values */
                e_ptr->v1 = v1 = v2;
                e_ptr->v2 = v2 = v3;
                e_ptr->e = d1 = d2;
                e_ptr->e2 = e2 = e3;
                e_ptr->f = f1 = f2;
                if (noncolnr == 0)              /* v1, v2, v3 collinear */
                        e_ptr->sf_no = -f_ind;
                else
                        e_ptr->sf_no = f_ind;
                if (v2 == q)
                        break;

                if ((f2 = e2->facet[0]) == f1)
                        f2 = e2->facet[1];
                get_values(apex, v2, e2, f2, &v3, &e3, &d2);
#ifdef DIAGNOSTICS
                temp = EVAL_3(v3, *sentinel);
                if (E_COMPARE(temp, 0.0) != 0)
                        ERROR("flat_remove> vertex is not flat");
#endif
                if (f_ind <= 1 && (wrt_apex = angle(q, v3, apex)) != cond){
                           /* lines q-v2 & q-v3 on different sides of apex */
                        ind = 1;
                        if (wrt_apex == 0)      /* q, v3, apex collinear */
                                ind++;
                        while(--ind >= 0){
                                e_ptr++;
                                e_ptr->v1 = v1 = v2;
                                e_ptr->v2 = v2 = v3;
                                e_ptr->e = d1 = d2;
                                e_ptr->e2 = e2 = e3;
                                e_ptr->f = f1 = f2;
                                e_ptr->sf_no = ++f_ind;
```

```
                        if ((f2 = e2->facet[0]) == f1)
                                f2 = e2->facet[1];
                        get_values(apex, v2, e2, f2, &v3, &e3, &d2);
                }
                base = e_ptr + 1;       /* prepare new chain */
                f_ind++;
                noncolnr = 1;
        }
        else    /* lines q-v2 and q-v3 on same side of apex */
                while ((noncolnr = angle(v1, v2, v3)) *
                                   angle(v1, v2, apex) > 0){
                        /* convex angle; update polyhedron */
                        flip_diagonal(apex, v1, v2, v3, e2, e3, d1, f1, f2);
                        /* update degree of v2; NOT of apex */
                        if (--(v2->degree) <= D && v2->type == 1){
                                v2->type = 2;
                                list[*j_ptr] = v2;
                                *j_ptr = MOD(*j_ptr + 1, size);
                        }
                        f2 = f1;

                        /* move backwards */
                        v2 = v1;
                        d2 = e2;  /* v3 and e3 do not change */
                        if (--e_ptr < base)
                                break;  /* note noncolnr != 0 */
#ifdef DIAGNOSTICS
                        if (v2 != e_ptr->v2)
                                ERROR("flat_remove> illegal crown");
#endif
                        v1 = e_ptr->v1;
                        e2 = e_ptr->e2;
                        f1 = e_ptr->f;
                        d1 = e_ptr->e;
                }
        }
        e_ptr->v2 = NULL;       /* indicates end of chain */

#ifdef DEBUG
        print_crown(crown);
#endif
        /* collect edges and facets incident upon the apex */
        e_ptr = crown + (ind = -1);
        do{
                free_edge[++ind] = e2 = (++e_ptr)->e2;
                e2->type = 0;           /* e2 is garbage */
                free_facet[ind] = f1 = e_ptr->f;
                f1->sign = 0;           /* f1 is garbage */
                if ((e2 = e_ptr->e)->facet[0] == f1)
                        e2->facet[0] = NULL;
                else
                        e2->facet[1] = NULL;
                v1 = e_ptr->v1;
                if (--(v1->degree) <= D && v1->type == 1){
                        v1->type = 2;           /* mark & insert in list */
                        list[*j_ptr] = v1;
                        *j_ptr = MOD(*j_ptr + 1, size);
                }
        } while (e_ptr->v2 != NULL);
        if ((cond = e_ptr->sf_no) < 0)
                cond = -cond;           /* store number of crown chains */
        f_ind = ind;            /* # of free facets equal to # of free edges */
        apex->type = 0;         /* mark apex as pulled off (= garbage) */
```

```
        /* finally, triangulate polygon bounded by crown */
        for (e_ptr = crown;  e_ptr->sf_no < 2;  e_ptr++)
                ;
        q = e_ptr->v2;    v1 = crown->v1;
        if (cond == 3){         /* generic triangle */
                Vertex  temp_v;

                MAKE_MIDPOINT_3(temp_v, *e_ptr->v1, *q);
                for (;  (++e_ptr)->v2 != NULL; )
                        ;
                base = crown;
                for (e1 = (base++)->e;  (v3=e_ptr->v1) != q;  v1 = v3, e1 = e2){
                        for (e3 = e_ptr->e;  angle(v3, v2=base->v1, base->v2) *
                                                angle(v3, v2, &temp_v) > 0;
                                        e3 = e2, v1 = v2, e1 = (base++)->e){
                                /* v3, v2, base->v2 form a convex angle */
                                if (base->v2 == q){ /* last edge of 1st chain */
                                        do{       /* report triangles from v2 */
                                                set_e(e2 = free_edge[ind--],
                                                                        v2, v3);
                                                set_f(free_facet[f_ind--],
                                                        &sentinel, e1, e2, e3);
                                                e1 = e2;
                                                e3 = (--e_ptr)->e;
                                        } while ((v3 = e_ptr->v1) != q);
                                        set_f(free_facet[f_ind--], &sentinel,
                                                        e1, base->e, e3);
#ifdef DIAGNOSTICS
                                        if (ind != 2 || f_ind != 1)
                                                ERROR("flat_remove> incorrect removal of flat vertex");
#endif
                                        return;
                                }
                                set_e(e2 = free_edge[ind--], v3, v2);
                                set_f(free_facet[f_ind--], &sentinel, e1, e2, e3);
                        }
                        /* concave angle */
                        set_e(e2 = free_edge[ind--], v3, v2);
                        set_f(free_facet[f_ind--], &sentinel, e1, e2, e3);
                        e_ptr--;
                }
                /* report all triangles from q */
                for (e3 = e_ptr->e;  base->v2 != q;  e3 = e2, e1 = (base++)->e){
                        set_e(e2 = free_edge[ind--], base->v1, q);
                        set_f(free_facet[f_ind--], &sentinel, e1, e2, e3);
                }
                set_f(free_facet[f_ind--], &sentinel, e1, e3, base->e);
        }
        else
                if (cond == 4){         /* generic quadrilateral */
                        base = e_ptr;
                        e2 = (++e_ptr)->e;
                        for (;  (v3 = (++e_ptr)->v2) != NULL;  e2 = e3){
                                set_e(e3 = free_edge[ind--], v3, q);
                                set_f(free_facet[f_ind--], &sentinel,
                                                        e2, e3, e_ptr->e);
                        }
                        set_e(e1 = free_edge[ind--], v1, q);
                        set_f(free_facet[f_ind--], &sentinel, e1, e2, e_ptr->e);
                        for (d2 = (e_ptr = crown)->e;  ++e_ptr != base;
                                                        d2 = e_ptr->e, e1 = e2){
                                set_e(e2 = free_edge[ind--], e_ptr->v1, q);
                                set_f(free_facet[f_ind--], &sentinel, e1, d2, e2);
                        }
                        set_f(free_facet[f_ind--], &sentinel, e1, d2, base->e);
```

```
                }
                else
                        ERROR("flat_remove> illegal crown for a flat vertex");
#ifdef DIAGNOSTICS
        if (ind != 2 || f_ind != 1)
                ERROR("flat_remove> incorrect removal of flat vertex");
#endif
}


void    repair(apex, crown)
Vertex          *apex;
CrownRec        *crown;
{
        int             h, i, crownf_no;
        double          temp;
        Vertex          *v1, *v2, *v3, *p, *q, *v_base = P.vertices;
        Edge            *e1, *e2, *e3, *temp_e;
        Facet           f_tmp;
        CrownRec        *cur, *next, *elem;

        for (cur = crown, v1 = cur->v1, v2 = cur->v2, crownf_no = cur->sf_no;  ;
                                        crownf_no = cur->sf_no = 0){
                /* skip crown edges in same superfacet */
                for (elem = cur + 1;
                        (i = elem->sf_no) == crownf_no || i == -crownf_no; )
                        elem++;
                v3 = elem->v2;

                /* compute coefficients of plane passing through v1, v2, v3 */
                plane_coef_compute(&f_tmp, v1, v2, v3);
                temp = EVAL_3(apex, f_tmp);
                /* h holds above/below info of apex w.r. to plane v1,v2,v3 */
                h = E_COMPARE(temp, 0.0);
#ifdef DIAGNOSTICS
                if (h == 0)
                        ERROR("repair> degenerate facet");
#endif

                /* find v3, so that facet (v1, v2, v3) belongs to convex hull */
                for (next = elem, i = (elem+1)->sf_no;
                                        (p = (++next)->v2) != NULL; ){
                        temp = EVAL_3(p, f_tmp);
                        if (h * E_COMPARE(temp, 0.0) == 1){
                                elem = next;
                                plane_coef_compute(&f_tmp, v1, v2, (v3 = p));
                                temp = EVAL_3(apex, f_tmp);
                                h = E_COMPARE(temp, 0.0);
#ifdef DIAGNOSTICS
                                if (h == 0)
                                        ERROR("repair> degenerate facet");
#endif
                                i = (elem+1)->sf_no;
                        }
                }
                f_tmp.sign = -h;                /* note f_tmp.c>=0 */

                if (i < 0)
                        i = -i;
                if ((h = next->sf_no) < 0)
                        h = -h;
                h = h - i;      /* stores number of facets from elem to end */

                /* plane (v1, v2, v3) belongs to convex hull */
                temp_e = cur->e;
                if (elem == cur + 1)    /* elem right next to cur */
                        e1 = e2 = elem->e;
                else{
                        set_e(e1 = e2 = free_edge[e_ind--], v2, v3);

                        /* process adjacent coplanar facets */
                        for (next = cur;  (p = (++next)->v2) != elem->v1;
                                                e2 = e3, v2 = p){
                                temp = EVAL_3(p, f_tmp);
                                if (E_COMPARE(temp, 0.0) != 0)
                                        break;
                                /* p coplanar with current facet */
                                set_e(e3 = free_edge[e_ind--], p, v3);
                                set_f(free_facet[f_ind--], &f_tmp, e2, next->e,
                                                                        e3);
                                /* report tetrahedron */
                                collect_4hedron(apex, p, v2, v3);
                        }
                        if (p == elem->v1){     /* first crown part exhausted */
                                temp = EVAL_3(p, f_tmp);
                                if (E_COMPARE(temp, 0.0) == 0)  /* coplanar */
                                        set_f(free_facet[f_ind--], &f_tmp, e2,
                                                        next->e, elem->e);
                                else{           /* determine new plane */
                                        Facet           f2_tmp;
                                        plane_coef_compute(&f2_tmp, v2, v3, p);
                                        temp = EVAL_3(apex, f2_tmp);
                                        f2_tmp.sign = -E_COMPARE(temp, 0.0);
#ifdef DIAGNOSTICS
                                        if (f2_tmp.sign == 0)
                                                ERROR("repair> degenerate facet");
#endif
                                        set_f(free_facet[f_ind--], &f2_tmp, e2,
                                                        next->e, elem->e);
                                }
                                /* report tetrahedron */
                                collect_4hedron(apex, p, v2, v3);
                        }
                        else{
                                /* recurse on first part of chain */
                                elem->v2 = NULL;        /* break chain */
                                next--;
                                next->sf_no = 0;
                                next->v1 = v3;
                                next->e = e2;
                                repair(apex, next);

                                elem->v2 = v3;  /* fix broken chain */
                        }
                }
                e2 = temp_e;
                if ((next = elem + 1)->v2 == NULL){     /* last in crown */
                        set_f(free_facet[f_ind--], &f_tmp, e2, e1, next->e);
                        /* report tetrahedron */
                        collect_4hedron(apex, v1, v2, v3);
                        break;
                }
                if (h > 0){             /* more than one facet */
                        /* process adjacent collinear crown edges */
                        for(;  next->sf_no < 0;  v2 = v3, v3 = p, e1 = next->e,
                                                e2 = e3, next++){
                                p = next->v2;           /* p != NULL */
#ifdef DIAGNOSTICS
                                if (p == NULL)
                                        ERROR("repair> incorrect facet numbers");
```

```
#endif
                                temp = EVAL_3(p, f_tmp);
                                if (E_COMPARE(temp, 0.0) != 0)
                                        break;
                                /* p coplanar with current facet */
                                set_e(e3 = free_edge[e_ind--], v1, v3);
                                set_f(free_facet[f_ind--], &f_tmp, e1, e2, e3);
                                /* report tetrahedron */
                                collect_4hedron(apex, v1, v2, v3);
                        }
                }

                /* process adjacent coplanar facets */
                for (;  (p = next->v2) != NULL;  v3 = p, e1 = e3, next++){
                        temp = EVAL_3(p, f_tmp);
                        if (E_COMPARE(temp, 0.0) != 0)
                                break;
                        /* p coplanar with current facet */
                        set_e(e3 = free_edge[e_ind--], p, v2);
                        set_f(free_facet[f_ind--], &f_tmp, e1, next->e, e3);
                        /* report tetrahedron */
                        collect_4hedron(apex, p, v2, v3);
                }
                if (p == NULL){         /* second part of crown exhausted */
                        set_f(free_facet[f_ind--], &f_tmp, e1, e2, next->e);
                        /* report tetrahedron */
                        collect_4hedron(apex, v1, v2, v3);
                        break;                  /* exit loop */
                }

                /* p not coplanar with facet */
                set_e(e3 = free_edge[e_ind--], v1, v3);
                set_f(free_facet[f_ind--], &f_tmp, e1, e2, e3);
                /* report tetrahedron */
                collect_4hedron(apex, v1, v2, v3);

                /* iterate on last part of crown */
                cur = next - 1;
                cur->v1 = v1;   /* cur->v2 is correct */
                cur->e = e3;
                v2 = v3;
        }
}



void    pointed_remove(apex, v2, e2, f2, list, j_ptr, size)
int     *j_ptr, size;
Vertex  *apex, *v2, **list;
Edge    *e2;
Facet   *f2;
{
        int             i, cond, noncolnr, sf_no, (*angle)();
        double          a, b, c, temp;
        Vertex          *v1, *v3, *v_sentnl;
        Edge            *e3, *d1, *d2;
        Facet           *f1, *f_sentnl;
        CrownRec        *crown, *e_ptr, *first;

#ifdef DEBUG
        fprintf(stderr, "\nPulling off vertex %d\n", apex - P.vertices);
#endif
        get_values(apex, v2, e2, f2, &v3, &e3, &d2);
        e_ptr = (crown = e_table) - 1;
        sf_no = 0;

        v_sentnl = v2;
        do {
                /* f2 starts new superfacet: find appropriate projection plane */
                if ((a = f2->a) < 0)    a = -a;
                if ((b = f2->b) < 0)    b = -b;
                if ((c = f2->c) < 0)    c = -c;
                if (E_COMPARE(a, b) <= 0)
                        if (E_COMPARE(b, c) <= 0)
                                angle = xy_v_angle;
                        else
                                angle = xz_v_angle;
                else
                        if (E_COMPARE(a, c) <= 0)
                                angle = xy_v_angle;
                        else
                                angle = yz_v_angle;
                cond = angle(v2, v3, apex);
#ifdef DIAGNOSTICS
                if (cond == 0)
                        ERROR("pointed_remove> degenerate cup facet");
#endif

                for (first= ++e_ptr, f_sentnl= f2, noncolnr=1, sf_no++; ; e_ptr++){
                        /* update edge record in crown, and store values */
                        e_ptr->v1 = v1 = v2;
                        e_ptr->v2 = v2 = v3;
                        e_ptr->e = d1 = d2;
                        e_ptr->e2 = e2 = e3;
                        e_ptr->f = f1 = f2;
                        if (noncolnr == 0)      /* collinear points */
                                e_ptr->sf_no = -sf_no;
                        else
                                e_ptr->sf_no = sf_no;

                        if ((f2 = e2->facet[0]) == f1)
                                f2 = e2->facet[1];
                        get_values(apex, v2, e2, f2, &v3, &e3, &d2);
                        temp = EVAL_3(v3, *f_sentnl);
                        if (E_COMPARE(temp, 0.0) != 0)
                                break;          /* f1 and f2 not coplanar */

                        /*f1, f2 coplanar: move backwards along crown & update*/
                        while ((noncolnr = angle(v1, v2, v3)) * cond > 0){
                                /* convex angle; update polyhedron */
                                flip_diagonal(apex, v1, v2, v3, e2, e3, d1, f1, f2);
                                /* update degree of v2, but not of apex */
                                if (--(v2->degree) <= D && v2->type == 1){
                                        v2->type = 2;
                                        list[*j_ptr] = v2;
                                        *j_ptr = MOD(*j_ptr + 1, size);
                                }
                                f2 = f1;

                                /* move backwards */
                                v2 = v1;
                                d2 = e2;        /* v3 and e3 do not change */
                                if (--e_ptr < first)
                                        break;          /* note noncolnr != 0 */
#ifdef DIAGNOSTICS
                                if (v2 != e_ptr->v2)
                                        ERROR("pull_off> illegal crown");
#endif
                                v1 = e_ptr->v1;
                                e2 = e_ptr->e2;
                                f1 = e_ptr->f;
```

```
                                d1 = e_ptr->e;
                        }
                }
        } while (v2 != v_sentnl);
        e_ptr->v2 = NULL;       /* indicates end of chain */

#ifdef DEBUG
        print_crown(crown);
#endif

        /* collect edges and facets incident upon the apex */
        e_ptr = crown + (i = -1);
        do{
                free_edge[++i] = e2 = (++e_ptr)->e2;
                e2->type = 0;           /* e2 is garbage */
                e2->facet[0] = e2->facet[1] = NULL;
                free_facet[i] = f1 = e_ptr->f;
                f1->sign = 0;           /* f1 is garbage */
                if ((e2 = e_ptr->e)->facet[0] == f1)
                        e2->facet[0] = NULL;
                else
                        e2->facet[1] = NULL;
                v1 = e_ptr->v1;
                if (--(v1->degree) <= D && v1->type == 1){
                        v1->type = 2;           /* mark & insert in list */
                        list[*j_ptr] = v1;
                        *j_ptr = MOD(*j_ptr + 1, size);
                }
        } while (e_ptr->v2 != NULL);
        apex->type = 0;         /* mark apex as pulled off (= garbage) */

        e_ind = f_ind = i;
        repair(apex, crown);    /* compute dome */
#ifdef DIAGNOSTICS
        if (e_ind != 2 || f_ind != 1)
                ERROR("pointed_remove> incorrect cup computation");
#endif
}
```

```
   /*********                   fence.c                          *********/
   /*      Routines to compute the fences of the polyhedron's edges     */
   /*  First, it is determined whether the fence will have an upper or  */
   /*  lower part or both (fence_parts), and then candidate segments    */
   /*  for visibility along the vertical direction are collected (col-  */
   /*  lect_segments).  The final fence is computed by the compute_fe-  */
   /*  nce routine that uses a heap for the active candidate segments.  */

#include "include.h"



extern Polyhedron       P;
extern Table            *segs;



#ifdef DEBUG
/*****************************************************************************/

void    printtable(p, size)             /* print contents of table */
int             size;
Endpoint        *p;
{
        int     i;

        for (i = 0;  i < size;  i++, p++)
                fprintf(stderr, "i= %d,  (%.3f, %.3f, %.3f, %d, %d)\n",
                                i, p->x, p->y, p->z, p->kind, p->seg_no);
}
/*****************************************************************************/
#endif



int     fence_parts(e)          /* determines needed parts of fence of edge e */
                                /* return 2 if e needs no fence,              */
                                /*        1 if it needs only upper fence,     */
                                /*        0 if it needs upper & lower parts,  */
                                /*       -1 if it needs only lower fence.     */
Edge    *e;
{
        int             k, upper_fence = 1, lower_fence = 1;
        double          temp;
        Vertex          *v1, *v2, *p;
        Facet           *f1, *f2;

        f1 = e->facet[0];       f2 = e->facet[1];
        v1 = e->vertex[0];      v2 = e->vertex[1];
        if (E_COMPARE(f2->c, 0.0) == 0)         /* f2 is vertical */
                if (E_COMPARE(f1->c, 0.0) == 0){
#ifdef DEBUG
        fprintf(stderr, "The edge's adjacent facets are vertical.\n");
#endif
                        return(2);
                }
                else{           /* f2 is vertical but not f1; swap them */
                        f1 = f2;
                        f2 = e->facet[0];
                }
        for (p = f1->vertex[k = 2];  p == v1 || p == v2;  )
                p = f1->vertex[--k];
        if (E_COMPARE(f1->c, 0.0) == 0){        /* f1 is vertical; but not f2 */
                switch (vert_rel_pos(p, v1, v2)){
                case  1:        /* p above edge v1-v2 */
                        upper_fence = 0;
                        if (f2->sign == 1)      /* polyhedron above f2 and e */
                                lower_fence = 0;
                        break;
                case  0:        /* p on edge v1-v2 */
#ifdef DIAGNOSTICS
                        ERROR("fence_parts> degenerate facet");
#endif
                        break;
                case -1:        /* p below edge v1-v2 */
                        lower_fence = 0;
                        if (f2->sign == -1)     /* polyhedron below f2 and e */
                                upper_fence = 0;
                        break;
                }
        }
        else{           /* neither f1 nor f2 is vertical */
                if ((k = f2->sign) * f1->sign < 0){
                        /* polyhedron above one of f1, f2 and below the other */
                        temp = EVAL_3(p, *f2);
                        if (E_COMPARE(temp, 0.0) * k > 0)
                                upper_fence = lower_fence = 0; /* convex edge */
                }
                else     /* normal vectors of facets point to same z */
                        if (k > 0)    /* P's interior ABOVE e */
                                lower_fence = 0;
                        else    /* P's interior BELOW e */
                                upper_fence = 0;
        }
        if ((k = upper_fence - lower_fence) == 0 && upper_fence == 0)
                return(2);
        return(k);
}



void    collect_segments(e, up_low_fence, above, below)
int             up_low_fence;
Edge            *e;
Table           *above, *below;
{
        int             f_ind, j;
        Point_3d        *ep1, *ep2;
        Vertex          *v1, *v2, p;
        Facet           *f, *f1, *f2;
        LineSegment     seg;

        f1 = e->facet[0];       f2 = e->facet[1];
        v1 = e->vertex[0];      v2 = e->vertex[1];

        /* go through all facets of polyhedron, and compute intersections   */
        /*      of the edge e and those that are vertically above/below e   */
        for (f= P.facets, f_ind= 0;  f_ind < P.facet_num;  f_ind++, f++){
                /* ignore facet, if garbage, vertical, incident upon e,     */
                /* leave polyhedron below when we do not need upper fence,  */
                /* or leave it above when we do not need a lower fence, and */
                /* when they do not intersect the vertical strip through e  */
                if ((j = f->sign) == 0 || E_COMPARE(f->c, 0.0) == 0
                                       || f == f1 || f == f2 || j * up_low_fence > 0
                                       || (j = compute_intersection(f, e, &seg)) < 0)
                        continue;

                if (j > 0)              /* intersection on an edge; mark it */
                        seg.facet_no = -f_ind - 1;
```

```
                else
                        seg.facet_no = f_ind;
                ep2 = (ep1 = seg.endpoint) + 1;
                MAKE_MIDPOINT_3(p, *ep1, *ep2);
                switch (vert_rel_pos(&p, v1, v2)){
                case  1:
                        if (up_low_fence >= 0){ /* we need an upper fence */
                                INS_TABLE(seg, LineSegment, above);
#ifdef DEBUG
        fprintf(stderr, "entry from facet %d inserted in ABOVE\n", f_ind);
#endif
                        }
                        break;
                case  0: break;
                case -1:
                        if (up_low_fence <= 0){ /* we need a lower fence */
                                INS_TABLE(seg, LineSegment, below);
#ifdef DEBUG
        fprintf(stderr, "entry from facet %d inserted in BELOW\n", f_ind);
#endif
                        }
                        break;
                }
        }
}



void    report(p1, p2, seg_parallel_yzplane)    /* report a piece of fence */
int             seg_parallel_yzplane;
Endpoint        p1, p2;
{
        LineSegment     seg;
        Table           *t;

        if (seg_parallel_yzplane){
                if (E_COMPARE(p1.y, p2.y) == 0)
                        return;  /* "zero"-length segment: no need to report */
        }
        else{
                if (E_COMPARE(p1.x, p2.x) == 0)
                        return;  /* "zero"-length segment: no need to report */
        }
#ifdef DEBUG
        fprintf(stderr, "facet no %d:  ", p1.facet_no);
        fprintf(stderr, "(%.3f, %.3f, %.3f)", p1.x, p1.y, p1.z);
        fprintf(stderr, " - (%.3f, %.3f, %.3f)\n", p2.x, p2.y, p2.z);
#endif

        /* insert it, if needed, in the array of segments */
        if ((seg.facet_no = p1.facet_no) >= 0){
                        /* trace of fence NOT on an edge */
                        /* note that seg.facet_no assignment is OPTIONAL */
                COPY_POINT_3(seg.endpoint[0], p1);
                COPY_POINT_3(seg.endpoint[1], p2);
                t = segs + p1.facet_no;
                INS_TABLE(seg, LineSegment, t);
        }
}



void    compute_fence(a_b, list_head, segs_parallel_yzplane)
int             a_b, segs_parallel_yzplane;
Table           *list_head;
{
        int             i, k, total, list_size = list_head->n, cur_seg,
                        xz_ep_compare(), yz_ep_compare(), segheap_compare();
        double          from_x_comp_z(), from_y_comp_z(), (*comp_z)();
        Endpoint        *points, *cur_p, start, current, new_p;
        LineSegment     *list;

        /* allocate space for endpoints of intersections of facets and the  */
        total = 2 * list_size;                  /* vertical strip through e */
        MY_ALLOC(points, Endpoint, total);

        list = (LineSegment *)list_head->to_elem;
        /* collect the endpoints and put them in the 'points' array */
        for (cur_seg = 0, cur_p = points;  cur_seg < list_size;  cur_seg++)
                for (i = 2, k = list[cur_seg].facet_no;  --i >= 0;  cur_p++){
                        COPY_POINT_3(*cur_p, list[cur_seg].endpoint[i]);
                        cur_p->kind = i;        /* appropriate endpoint */
                        cur_p->seg_no = cur_seg;
                        cur_p->facet_no = k;
                }

        /* sort endpoints by x or y-coordinate.  If ties, use z-coordinate,  */
        /* and in case of ties again, right endpoints precede left endpoints */
        qsort((char *)points, total, sizeof(*points),
                (segs_parallel_yzplane) ? yz_ep_compare : xz_ep_compare);
#ifdef DEBUG
        printtable(points, total);  /*debugging purposes */
#endif

        if (segs_parallel_yzplane)
                comp_z = from_y_comp_z;
        else
                comp_z = from_x_comp_z;
        /* sweep them in sorted order (in a heap), and determine fence */
        init_segheap(list, list_size, a_b, &current, segheap_compare, comp_z);

        start = points[0];
#ifdef DIAGNOSTICS
        if (start.kind != 0)    /* right endpoint met */
                ERROR("compute_fence> first point met is a right endpoint");
#endif

        heap_insert(start.seg_no);
        for (i = 1;  i < total;  i++){
                current = points[i];
                cur_seg = current.seg_no;
                if (current.kind == 1){         /* right endpoint */
                        if (cur_seg != heap_top())
                                heap_delete(cur_seg);
                        else{
                                report(start, current, segs_parallel_yzplane);
                                heap_delete(cur_seg);

                                if (heap_size() > 0){
                                        start = current;
                                        k = heap_top();
                                        start.z = comp_z(start, list[k]);
                                        start.seg_no = k;
                                        start.facet_no = list[k].facet_no;
                                }
                        }
                }
                else{   /* left endpoint */
                        k = heap_top();
                        heap_insert(cur_seg);
```

```
                        if (cur_seg == heap_top()){     /* topmost heap-element */
                                if (heap_size() > 1){
                                        new_p = current;
                                        new_p.z = comp_z(new_p, list[k]);
                                        report(start, new_p, segs_parallel_yzplane);
                                }
                                start = current;
                        }
                }
        }
        cleanup_heap();
        (void) free((char *)points);
}
```

```
   /*********                  heap.c                          *********/
   /*              Routines for handling indirect heaps                */


#include "include.h"


static int              heap_a_b,      /* 1 if interested in lowermost*/
                                        /*                    -1 else */
                        n_heap,         /* number of elements in heap */
                        *heap,          /* actual heap */
                        *to_heap,       /* array of reverse pointers */
                        (*heap_compare)();
static double           (*heap_compute_z)();
static LineSegment      *heap_array;
static Endpoint         *ref_point_ptr;



/***                                             ***/
/***    General Purpose INDIRECT HEAP routines    ***/
/***                                             ***/

int     heap_size()             /* returns the size of heap */
{
        return(n_heap);
}



int     heap_top()
        /* returns content of topmost heap-element, or -1 if heap is empty */
{
        return( (n_heap <= 0) ? -1 : heap[1] );
}



int     upheap(ind, elem_ind)     /* move elem_ind from position ind upwards */
int     ind, elem_ind;                          /* return its final position */
{
        int     h, k = ind / 2;

        while (ind != 1 && heap_compare(heap[k], elem_ind) == 1){
                h = heap[ind] = heap[k];
                to_heap[h] = ind;
                ind = k;
                k /= 2;
        }
        return(ind);
}



int     downheap(ind, elem_ind)  /* move elem_ind from position ind downwards */
int     ind, elem_ind;                          /* return its final position */
{
        int     h, k, half_size = n_heap / 2;

        while ((k = ind) <= half_size) {
                ind = 2 * k;
                if (ind < n_heap && heap_compare(heap[ind], heap[ind+1]) == 1)
                        ind++;
                if (heap_compare(elem_ind, heap[ind]) <= 0)
                        break;
                h = heap[k] = heap[ind];
                to_heap[h] = k;
        }
        return(k);
}



void    heap_insert(seg_ind)    /* insert at appropriate place in heap */
int     seg_ind;
{
        int     k;

        k = upheap(++n_heap, seg_ind);
        heap[k] = seg_ind;
        to_heap[seg_ind] = k;
}



void    heap_delete(seg_no)     /* delete from heap, entry containing seg_no */
int     seg_no;
{
        int     elem_ind, k;

        k = to_heap[seg_no];
#ifdef DIAGNOSTICS
        if (k <= 0 || k > n_heap)
                ERROR("heap_delete> illegal heap reference");
        if (n_heap < 0)
                ERROR("heap_delete> heap underflow");
#endif
        if (n_heap == 1 || k == n_heap){
                /* heap contains one element, or we delete last element */
                n_heap--;
                return;
        }

        /* heap contains more elements, apart from the one to be deleted */
        elem_ind = heap[n_heap--];
        k = upheap(k, elem_ind);        /* up-heap, if necessary */
        k = downheap(k, elem_ind);      /* down-heap, if necessary */

        heap[k] = elem_ind;
        to_heap[elem_ind] = k;
}



void    cleanup_heap()          /* return space allocated for the heap */
{
#ifdef DIAGNOSTICS
        if (n_heap != 0)
                ERROR("cleanup_heap> heap not empty");
#endif
        (void) free((char *)heap);
        (void) free((char *)to_heap);
}



/***                                     ***/
/***    Special Purpose Heap Routines    ***/
/***                                     ***/
```

```
/***            SEGMENTS  HEAP          ***/

void    init_segheap(array, array_size, a_b, ref_ptr, compare_f, compute_z)
        /* initialize everything needed to handle a heap of line segments */
int             array_size,
                a_b,        /* 1 if interested in lowermost, -1 else */
                (*compare_f)();
double          (*compute_z)();
Endpoint        *ref_ptr;
LineSegment     *array;
{
        /* allocate space for the heap, and the reverse pointers */
        MY_ALLOC(heap, int, array_size + 1);
        MY_ALLOC(to_heap, int, array_size);

        heap_a_b = a_b;
        heap_array = array;
        ref_point_ptr = ref_ptr;
        heap_compare = compare_f;
        heap_compute_z = compute_z;
        n_heap = 0;
}



int     segheap_compare(seg1_ind, seg2_ind)
        /* return  -1/0/1, if seg1_ind is closer/equidistant/farther    */
        /*      than seg2_ind from edge whose fence we compute.         */
        /* NOTE: from the way the heap-elements are collected, the two  */
        /*      segments are not expected to touch.                     */
int     seg1_ind, seg2_ind;
{
        int             cond;
        double          z, zprime;
        Endpoint        ref_point;

        z = heap_compute_z((ref_point = *ref_point_ptr), heap_array[seg1_ind]);
        zprime = heap_compute_z(ref_point, heap_array[seg2_ind]);

        if ((cond = E_COMPARE(z, zprime)) == 0)
                return(0);      /* two edges on same vertical plane */
        /* seg1 closer to edge than seg2 if (cond == -heap_a_b) */
        return((cond == -heap_a_b) ? -1 : 1);
}
```

```
   /*********                  regions.c                        *********/
   /*  Routines for the regions handling in the Hertel & Mehlhorn's     */
   /*                  triangulation algorithm.                         */
   /*  Each region is represented as a doubly linked list of vertices   */
   /*  forming a weakly concave chain (with respect to the region).     */
   /*  The regions are stored in a red-black tree, and they are linked  */
   /*  in a linked list in y-order.  A sentinel reg_sentnl is included  */
   /*  as the topmost region.                                           */


#include "include.h"



#define RED     0
#define BLACK   1



extern Table    triangles;



static int              reg_count, node_count, tree_count;
static Region           *reg_table, *reg_sentnl;
static ChainNode        *node_table;
static rbTreeNode       *tree_table, *tree_root, *tree_sentnl;



#ifdef DEBUG
/*****************************************************************************/

void    reglist_print()
        /* print out the contents of the regions data-structure */
{
        ChainNode       *q;
        Region          *r;
        Endpoint        *p;

        fprintf(stderr, "\n    List of Regions  (top to bottom)\n\n");
        for (r = reg_sentnl;  (r = r->next_reg) != NULL;  ){
                fprintf(stderr, "region R%d:  pointed by T%d\n",
                                r - reg_table, r->tnode_ptr - tree_table);
                p = r->up_p;
                fprintf(stderr, "\tup_p= (%.3f, %.3f, %.3f)\n",p->x,p->y,p->z);
                p = r->low_p;
                fprintf(stderr, "\tlow_p= (%.3f, %.3f, %.3f)\n",p->x,p->y,p->z);
                fprintf(stderr, "\tforward chain:");
                q = r->top;
                do{
                        p = q->p;
                        fprintf(stderr, "  (%.3f, %.3f, %.3f),",p->x,p->y,p->z);
                        q = q->next;
                }while (q != NULL);
                fprintf(stderr, "\n\tbackward chain:");
                q = r->bottom;
                do{
                        p = q->p;
                        fprintf(stderr, "  (%.3f, %.3f, %.3f),",p->x,p->y,p->z);
                        q = q->previous;
                }while (q != NULL);
                p = r->rightmost->p;
                fprintf(stderr, "\n\trightmost: (%.3f, %.3f, %.3f)\n",p->x,p->y,p->z);
        }
}



void    print(t, ident)
int             ident;
rbTreeNode      *t;
{
        int             i;
        rbTreeNode      *new_t;

        if ((new_t = t->left) != tree_sentnl)
                print(new_t, ident + 1);

        for (i = ident;  --i >= 0;  )
                fprintf(stderr, "     ");
        fprintf(stderr, "T%d: (R%d, ", t - tree_table, t->reg_ptr - reg_table);
        if (t->color == RED)
                fprintf(stderr, "RED),");
        else
                fprintf(stderr, "BLACK),");
        if ((new_t = t->parent) == NULL)
                fprintf(stderr, " ROOT\n");
        else
                fprintf(stderr, " parent = T%d\n", new_t - tree_table);

        if ((new_t = t->right) != tree_sentnl)
                print(new_t, ident + 1);
}



void    regtree_print()
{
        fprintf(stderr, "\n    Red-Black Tree of Regions\n\n");
        if (tree_root == tree_sentnl)
                fprintf(stderr, "The red-black tree is EMPTY.\n");
        else
                print(tree_root, 0);
        fprintf(stderr, "\n");
}

/*****************************************************************************/
#endif



void    init_regions(seg_num, point_num, reg_sentnl_ptr)
        /* prepare the regions table and the tree that stores them */
int             seg_num, point_num;
Region          **reg_sentnl_ptr;
{
        Region          *r;
        rbTreeNode      *t;

        MY_ALLOC(r, Region, reg_count = seg_num);
        MY_ALLOC(node_table, ChainNode, node_count = 2 * point_num);
        MY_ALLOC(t, rbTreeNode, tree_count = seg_num);
        /* 0th element of reg_table is a sentinel (topmost region) */
        reg_table = reg_sentnl = *reg_sentnl_ptr = r;
        r->top = r->bottom = r->rightmost = NULL;
        r->up_p = r->low_p = NULL;
        r->next_reg = r->prev_reg = NULL;
        r->tnode_ptr = NULL;
```

```
        /* 0th element of tree_table allocated to sentinel (= NULL) */
        tree_table = tree_sentnl = t;
        t->color = BLACK;
        t->reg_ptr = reg_sentnl;
        t->parent = NULL;
        t->left = t->right = t;         /* point to sentinel */
        tree_root = NULL;               /* indicates empty tree */
}



Region  *get_reg_rec()
        /* return a pointer to a record of a new region */
{
        if (--reg_count <= 0)
                ERROR("get_reg_rec> not enough region records");
        return(reg_table + reg_count);
}



ChainNode       *get_list_rec()
        /* return a pointer to a new node record for the chain of a region */
{
        if (--node_count < 0)
                ERROR("get_list_rec> not enough list records");
        return(node_table + node_count);
}



Region  *new_region(p, q1, q2)
Endpoint        *p, *q1, *q2;
{
        ChainNode       *t;
        Region          *reg;

        t = get_list_rec();
        t->next = t->previous = NULL;
        t->p = p;
        reg = get_reg_rec();
        reg->top = reg->bottom = reg->rightmost = t;
        reg->up_p = q1;
        reg->low_p = q2;
        reg->prev_reg = reg->next_reg = NULL;
        return(reg);
}



int     wrt_boundr(sel, regptr, p)
   /* sel is 1 if we are interested in upper boundary, -1 else  */
   /* returns:  -1, if point p is below boundary of region,     */
   /*            0, if point p is on the boundary of region,    */
   /*            1, if point p is above boundary of region.     */
Region          *regptr;
Endpoint        *p;
{
        double          dx, temp;
        Endpoint        *q, *r;

        if (sel == 1){
                q = regptr->up_p;
                r = regptr->top->p;
        }
        else{
                q = regptr->low_p;
                r = regptr->bottom->p;
        }
        dx = q->x - r->x;
        switch (E_COMPARE(dx, 0.0)){
        case  0:                /* vertical boundary */
#ifdef DIAGNOSTICS
                if (E_COMPARE(q->x, p->x) != 0)
                        ERROR("wrt_boundr> region is not swept symmetrically");
#endif
                return((E_COMPARE(p->y, q->y) + E_COMPARE(p->y, r->y)) / 2 );
                break;
        case  1:                /* non-vertical boundary */
                temp = (r->y - q->y) * (p->x - q->x) + dx * (p->y - q->y);
                return( E_COMPARE(temp, 0.0) );
                break;
        }
#ifdef DIAGNOSTICS
        ERROR("wrt_boundr> region is not x-monotone");
#endif
        return(-5);
}



int     wrt_region(regptr, point)
   /* returns:   3, if point is on upper & lower boundary of region,    */
   /*            2, if point is above the region,                       */
   /*            1, if point is on the upper boundary of the region,    */
   /*            0, if point is inside the region,                      */
   /*           -1, if point is on the lower boundary of the region,    */
   /*           -2, if point is below the lower boundary of region      */
Region          *regptr;
Endpoint        *point;
{
        int                     c_up, c_low;

        c_low = wrt_boundr(-1, regptr, point);
        if ((c_up = wrt_boundr(1, regptr, point)) == 0 && c_low == 0)
                return(3);
        else{
#ifdef DIAGNOSTICS
                if (c_low < c_up)
                        ERROR("wrt_region> inconsistent region");
#endif
                return(c_up + c_low);
        }
}



void    left_rotate(t)
                /* rotate left around t */
                /* ASSUMPTION: t's right child is not NULL (= sentinel) */
rbTreeNode      *t;
{
        rbTreeNode      *x, *p;

        x = t->right;
        if ((t->right = x->left) != tree_sentnl)        /* update t->right */
                t->right->parent = t;           /* update reverse pointer */
        if ((x->parent = t->parent) == NULL)            /* NULL x->parent */
                tree_root = x;                  /* update root of tree */
        else
```

```
                if (t == (p = x->parent)->left) /* left child of its parent */
                        p->left = x;
                else                            /* right child of its parent */
                        p->right = x;
        x->left = t;
        t->parent = x;
}



void    right_rotate(t)
                /* rotate right around t */
                /* ASSUMPTION: t's left child is not NULL (=sentinel) */
rbTreeNode      *t;
{
        rbTreeNode      *x, *p;

        x = t->left;
        if ((t->left = x->right) != tree_sentnl)        /* update t->left */
                t->left->parent = t;            /* update reverse pointer */
        if ((x->parent = t->parent) == NULL)            /* NULL x->parent */
                tree_root = x;                  /* update root of tree */
        else
                if (t == (p = x->parent)->left) /* left child of its parent */
                        p->left = x;
                else                            /* right child of its parent */
                        p->right = x;
        x->right = t;
        t->parent = x;
}



Region  *reg_locate(point, place)
   /* locate in red-black tree a region containing point in its closure */
   /* return NULL if such an element was not found;  else a pointer to  */
   /* that region is returned, and place is 3/1/0/-1 iff the point lies */
   /* on both the upper AND lower boundaries/on the upper boundary only */
   /* /in the interior/only on the lower boundary of the region         */
int             *place;
Endpoint        *point;
{
        int             c;
        rbTreeNode      *t;
        Region          *regptr;

        for (t = tree_root;  t != tree_sentnl;  ){
                regptr = t->reg_ptr;
#ifdef DIAGNOSTICS
                if (regptr == reg_sentnl)
                        ERROR("reg_locate> point in complement of facet");
#endif
                switch (c = wrt_region(regptr, point)){
                        case  2: /* point above region */
                                 t = t->left;
                                 break;
                        case  1: /* point on upper boundary of *regptr */
                        case  0: /* point in the interior of region */
                        case -1: /* point on lower boundary of region */
                        case  3: /* point on upper & lower boundary of region */
                                 *place = c;
                                 return(regptr);
                                 break;
                        case -2: /* point below region */
                                 t = t->right;
                                 break;
                }
        }
#ifdef DIAGNOSTICS
        ERROR("reg_locate> tree traversal ended without locating region");
#endif
        return(NULL);
}



void    rb_ins_fix(t, p)
   /* ensure red-black property after insertion of new tree-node */
   /* called  ONLY IF  (p = t->parent) != NULL && p->color = RED */
rbTreeNode      *t, *p;
{
        rbTreeNode      *pp, *y;

        do{
                if (p == (pp = p->parent)->left){
                        if ((y = pp->right)->color == RED){     /* y not NULL */
                                p->color = y->color = BLACK;
                                pp->color = RED;
                                t = pp;
                        }
                        else{
                                if (t == p->right){
                                        left_rotate(p);
                                        p = t;  /* but pp remains unchanged */
                                }
                                p->color = BLACK;
                                pp->color = RED;
                                right_rotate(p->parent);
                                break;
                        }
                }
                else{
                        if ((y = pp->left)->color == RED){      /* y not NULL */
                                p->color = y->color = BLACK;
                                pp->color = RED;
                                t = pp;
                        }
                        else{
                                if (t == p->left){
                                        right_rotate(p);
                                        p = t;  /* but pp remains unchanged */
                                }
                                p->color = BLACK;
                                pp->color = RED;
                                left_rotate(p->parent);
                                break;
                        }
                }
        } while ((p = t->parent) != NULL && p->color == RED);
        tree_root->color = BLACK;
}



void    ins_region(new_reg, regptr)
        /* insert the new element AFTER the entry pointed by "regptr" */
Region          *new_reg, *regptr;
{
        Region          *nextr;
        rbTreeNode      *t, *tp;
```

```
        /* insert new element in the ordered list of regions */
        new_reg->prev_reg = regptr;
        if ((new_reg->next_reg = nextr = regptr->next_reg) != NULL)
                nextr->prev_reg = new_reg;
        regptr->next_reg = new_reg;

        /* insert new element in the red-black tree */
        if (--tree_count <= 0)
                ERROR("ins_region> not enough tree-node records");
        (t = tree_table + tree_count)->color = RED;
        t->left = t->right = tree_sentnl;
        t->reg_ptr = new_reg;
        new_reg->tnode_ptr = t;

        if (tree_root == NULL){
                tree_root = t;
                t->parent = NULL;
                t->color = BLACK;
        }
        else{           /* non-empty red-black tree */
                if ((tp = regptr->tnode_ptr) == NULL){
                        /* regptr = reg_sentnl; insert smallest entry in tree */
                        for (tp = tree_root;  tp->left != tree_sentnl;  )
                                tp = tp->left;
                        tp->left = t;
                }
                else
                        if (tp->right == tree_sentnl)
                                tp->right = t;
                        else{
#ifdef DIAGNOSTICS
                                if (nextr == NULL ||
                                    nextr->tnode_ptr->left != tree_sentnl)
                                        ERROR("ins_region> inconsistent regions");
#endif
                                (tp = nextr->tnode_ptr)->left = t;
                        }
                t->parent = tp;
                if (tp->color == RED)
                        rb_ins_fix(t, tp);
        }
}



void    rb_del_fix(t, p)
   /* ensure red-black property after the deletion of a tree-node */
   /* called  ONLY IF  (p which is equal to t->parent) != NULL    */
rbTreeNode      *t, *p;
{
        rbTreeNode      *w;

        while (t->color != RED){
                if (t == p->left){
                        /* p->right not NULL, as color(t) = BLACK */
                        if ((w = p->right)->color == RED){
                                /* color(t)=BLACK => left(w), right(w) != NULL*/
                                w->color = BLACK;
                                p->color = RED;
                                left_rotate(p);
                                w = p->right;
                        }
                        switch (w->right->color) {
                        case BLACK:
                                if (w->left->color != RED){
                                        w->color = RED;
                                        t = p;
                                        break;
                                }
                                w->left->color = BLACK;
                                w->color = RED;
                                right_rotate(w);
                                w = p->right;
                                /* fall through to next case */
                        case RED:
                                w->color = p->color;
                                p->color = w->right->color = BLACK;
                                left_rotate(p);
                                return; /* note that color(t) = BLACK */
                        }
                }
                else{           /* p->left not NULL, as color(t) = BLACK */
                        if ((w = p->left)->color == RED){
                                w->color = BLACK; /* left(w), right(w) != NULL*/
                                p->color = RED;
                                right_rotate(p);
                                w = p->left;
                        }
                        switch (w->left->color) {
                        case BLACK:
                                if (w->right->color != RED){
                                        w->color = RED;
                                        t = p;
                                        break;
                                }
                                w->right->color = BLACK;
                                w->color = RED;
                                left_rotate(w);
                                w = p->left;
                                /* fall through to next case */
                        case RED:
                                w->color = p->color;
                                p->color = w->left->color = BLACK;
                                right_rotate(p);
                                return; /* note that color(t) = BLACK */
                        }
                }
                if ((p = t->parent) == NULL)
                        break;
        }
        t->color = BLACK;
}



void    del_region(regptr)
        /* delete the region pointed by "regptr" */
Region          *regptr;
{
        Region          *r;
        rbTreeNode      *t, *x, *p;

        /* delete region from the ordered list of regions */
        if ((r = regptr->prev_reg) == NULL)     /* delete sentinel ? */
                ERROR("del_region> region sentinel threatened with deletion");
        if ((r->next_reg = regptr->next_reg) != NULL)
                r->next_reg->prev_reg = r;

        /* delete entry from the red-black tree */
```

```
        t = regptr->tnode_ptr;
        if (t->right == tree_sentnl)
                x = t->left;
        else{
                t->reg_ptr = regptr = regptr->next_reg;
#ifdef DIAGNOSTICS
                if (regptr == NULL || regptr->tnode_ptr->left != tree_sentnl)
                        ERROR("del_region> inconsistent regions");
#endif
                p = regptr->tnode_ptr;          /* swap t and p */
                regptr->tnode_ptr = t;
                t = p;          /* new victim has its left tree-pointer NULL */
                x = t->right;
        }
        if ((p = t->parent) == NULL)
                if (x == tree_sentnl)
                        tree_root = NULL;       /* tree is empty */
                else{           /* t was root of tree of size <= 2 */
                        tree_root = x;
                        x->parent = NULL;
                        x->color = BLACK;
                }
        else{           /* update parent of t */
                if (t == (x->parent = p)->left)
                        p->left = x;
                else
                        p->right = x;
                if (t->color != RED)
                        rb_del_fix(x, p);
        }
}



void    upd_topchain(regptr, pnt)
Region          *regptr;
Endpoint        *pnt;
{
        ChainNode       *q, *r;

        /* traverse backwards the chain to report triangles, if needed */
        for (q = regptr->top;  ;  q = r){
                if ((r = q->next) == NULL || xy_ep_angle(r->p, q->p, pnt) <= 0)
                        break;          /* collinear points or concave angle */

                /* convex angle */
                triangle_report(q->p, r->p, pnt);       /* report a triangle */
                regptr->top = r;
                r->previous = NULL;
        }
}



void    chaintop_insert(regptr, point, nextpoint)
Region          *regptr;
Endpoint        *point, *nextpoint;
{
        ChainNode       *q;

        /* traverse backwards the chain to report triangles, if needed */
        upd_topchain(regptr, point);

        /* insert point in the point-chain associated with region */
        q = get_list_rec();
        q->p = point;
        q->previous = NULL;
        q->next = regptr->top;
        (regptr->top)->previous = q;
        regptr->rightmost = regptr->top = q;
        regptr->up_p = nextpoint;
}



void    upd_bottomchain(regptr, pnt)
Region          *regptr;
Endpoint        *pnt;
{
        ChainNode       *q, *r;

        /* traverse backwards the chain to report triangles, if needed */
        for (q = regptr->bottom;  ;  q = r){
                if ((r = q->previous) == NULL || xy_ep_angle(r->p, q->p, pnt) >= 0)
                        break;          /* collinear points or concave angle */

                /* convex angle */
                triangle_report(q->p, r->p, pnt);       /* report a triangle */
                regptr->bottom = r;
                r->next = NULL;
        }
}



void    chainbottom_insert(regptr, point, nextpoint)
Region          *regptr;
Endpoint        *point, *nextpoint;
{
        ChainNode       *q;

        /* traverse backwards the chain to report triangles, if needed */
        upd_bottomchain(regptr, point);

        /* insert the points[point_ind] in the chain */
        q = get_list_rec();
        q->p = point;
        q->next = NULL;
        q->previous = regptr->bottom;
        (regptr->bottom)->next = q;
        regptr->rightmost = regptr->bottom = q;
        regptr->low_p = nextpoint;
}



void    chain_merge(regptr, point)
Region          *regptr;
Endpoint        *point;
{
        ChainNode       *q;
        Region          *nextr;

#ifdef DIAGNOSTICS
        if (regptr->next_reg == NULL)
                ERROR("chain_merge> nonexistent region to be merged");
#endif
        /* update the chains as if the point of merging was to be inserted */
        upd_bottomchain(regptr, point);
        upd_topchain((nextr = regptr->next_reg), point);
```

```
        /* insert the point of merging, and connect the two chains */
        q = get_list_rec();
        q->p = point;
        q->previous = regptr->bottom;
        q->next = nextr->top;
        (regptr->bottom)->next = (nextr->top)->previous = regptr->rightmost = q;
        regptr->low_p = nextr->low_p;
        regptr->bottom = nextr->bottom;
        /* the fields prev_reg & next_reg of *regptr and *nextr are OK */

        /* delete the lowermost of the two regions to be merged */
        del_region(nextr);
}



void    chain_split(regptr, point)
   /* splits the chain of points bounding region pointed by regptr,     */
   /* introducing the line segment that connects "point" to the         */
   /* rightmost point of the chain                                      */
Region          *regptr;
Endpoint        *point;
{
        ChainNode       *q, *cutpoint;
        Region          *r;

        cutpoint = regptr->rightmost;
        q = get_list_rec();             /* duplicate splitting point */
        *q = *cutpoint;

        /* break the chain, update "r" which corresponds to its lower part */
        r = get_reg_rec();
        if (regptr->bottom == cutpoint)
                r->bottom = q;
        else{
                r->bottom = regptr->bottom;
                regptr->bottom = cutpoint;
                cutpoint->next->previous = q;
        }
        r->top = r->rightmost = q;
        cutpoint->next = q->previous = NULL;
        r->low_p = regptr->low_p;
        regptr->low_p = r->up_p = point;
        ins_region(r, regptr);
}



int     triangle_report(p1, p2, p3)
Endpoint        *p1, *p2, *p3;
{
        int             c12, c13, c23;
        Endpoint        *q;
        Triangle        t;

        if (xy_ep_angle(p1, p2, p3) != 0){
                if ((c12 = xy_ep_compare(p1, p2)) == 0 ||
                    (c13 = xy_ep_compare(p1, p3)) == 0 ||
                    (c23 = xy_ep_compare(p2, p3)) == 0)
                        ERROR("triangle_report> degenerate triangle");
                if (c12 > 0){
                        q = p1;  p1 = p2;  p2 = q;
                }
                if (c13 > 0){
                        q = p1;  p1 = p3;  p3 = q;
                }
                if (c23 > 0){
                        q = p2;  p2 = p3;  p3 = q;
                }
                COPY_POINT_3(t.p[0], *p1);
                COPY_POINT_3(t.p[1], *p2);
                COPY_POINT_3(t.p[2], *p3);
#ifdef DEBUG
        fprintf(stderr, ">>> (%.3f, %.3f)-(%.3f, %.3f)-(%.3f, %.3f)\n",
                                        p1->x, p1->y, p2->x, p2->y, p3->x, p3->y);
#endif
                INS_TABLE(t, Triangle, &triangles);
        }
}



void    clean_regions()
        /* free the space allocated for the regions and red-black tree */
{
#ifdef DIAGNOSTICS
        if (reg_sentnl->next_reg != NULL)
                ERROR("clean_regions> regions not exhausted");
#endif
        (void)free((char *)reg_table);
        (void)free((char *)node_table);
        (void)free((char *)tree_table);
}
```

```
   /*********                   shatter.c                         *********/
   /*    Routines for computing the constrained triangulation of a      */
   /*   triangular facet.  Constraints are associated line segments     */
   /*  The routines employ Hertel & Mehlhorn's triangulation algorithm  */
   /*  which is changed to handle ONLY facets that are simple polygons  */
   /*  whose boundary forms two y-monotone chains.                      */


#include "include.h"



typedef struct local_line_sgmnt {
                        struct endpoint_type    *endp[2];
} LocalLineSegm;

static int              n_points, n_segs, seg_ind;
static Region           *sentnl;
static Endpoint         *points;
static LocalLineSegm    *newseg_table, *s;



#ifdef DEBUG
/*****************************************************************************/

void    seg_print(seg_table, size)
LineSegment     *seg_table;
int                     size;
{
        LineSegment     *t;

        for (t = seg_table;  --size >= 0;  t++)
                fprintf(stderr, "e0 = (%.3f, %.3f),   e1 = (%.3f, %.3f)\n",
                                        t->endpoint[0].x, t->endpoint[0].y,
                                        t->endpoint[1].x, t->endpoint[1].y);
}



void    endp_print(point_table, size)
int             size;
Endpoint        *point_table;
{
        int             i;
        Endpoint        *p;

        for (i = 0, p = point_table;  i < size;  i++, p++)
                fprintf(stderr, "%d:  (%.3f, %.3f),  kind= %d,  seg_no= %d\n",
                                        i, p->x, p->y, p->kind, p->seg_no);
}

/*****************************************************************************/
#endif



void    preprocess(seg_table, n)
int             n;
LineSegment     *seg_table;
{
        int             i, j, seg_compare(), xy_ep_compare(), xy_p_compare();
        Endpoint        *cur, *prev;
        LineSegment     *prevs, *seg;
        LocalLineSegm   *newseg;

        /* sort the line_segments by left endpoint and slope */
        qsort((char *)seg_table, n, sizeof(*seg_table), seg_compare);

        /* go through the line segments, and among those with the "same" */
        /* left endpoint and the "same" slope keep the longer ones      */
        for (i = n, seg = (prevs = seg_table) + 1;  --i > 0;  seg++){
                if (seg_compare(prevs, seg) != 0){
                        prevs++;
                        *prevs = *seg;
                }
                else            /* select longer */
                        if (xy_p_compare(prevs->endpoint+1, seg->endpoint+1) < 0)
                                *prevs = *seg;
        }
        n = n_segs = prevs - seg_table + 1;

#ifdef DEBUG
        seg_print(seg_table, n);
#endif

        /* collect & sort endpoints of the line segments associated with f */
        n_points = 2 * n;
        MY_ALLOC(points, Endpoint, n_points);
        for (i = 0, cur = points, seg = seg_table;  i < n;  i++, seg++)
                for (j = 0;  j < 2;  j++, cur++){
                        COPY_POINT_3(*cur, seg->endpoint[j]);
                        cur->kind = j;
                        cur->seg_no = i;
                }
        qsort((char *)points, n_points, sizeof(*points), xy_ep_compare);

        /* go through the endpoints assigning indexes to the distinct ones */
        /* create a new table that stores pointers to the endpoints */
        MY_ALLOC(newseg_table, LocalLineSegm, n);
        cur = points;
        newseg = newseg_table + cur->seg_no;
        newseg->endp[cur->kind] = cur;
        for (i = n_points, prev = cur, cur++;  --i > 0;  cur++){
                if (xy_ep_compare(prev, cur) != 0){
                        prev++;
                        *prev = *cur;
                }
                newseg = newseg_table + cur->seg_no;
                newseg->endp[cur->kind] = prev;
        }
        n_points = prev - points + 1;
#ifdef DEBUG
        fprintf(stderr, "n_points = %d\n", n_points);
        endp_print(points, n_points);
#endif
}



void    mult_ins_regions(regptr, p)
        /* create & insert new regions, after region pointed by regptr  */
        /* the regions are defined by segments incident upon p          */
Region          *regptr;
Endpoint        *p;
{
        Region          *r, *new_region();
        Endpoint        *u, *v;
```

```
        for (u = s->endp[1], s++;  ++seg_ind < n_segs && p == s->endp[0];
                                                s++, u = v, regptr = r)
                ins_region(r = new_region(p, u, (v = s->endp[1])), regptr);
}



void    in_region(regptr, p)
Region          *regptr;
Endpoint        *p;
{
        Region          *rnext;

#ifdef DIAGNOSTICS
        if (seg_ind >= n_segs || p != s->endp[0])
                ERROR("in_region> isolated point");
#endif

        chain_split(regptr, p);
        if ((rnext = regptr->next_reg) == NULL) /* save a ptr to lower part */
                ERROR("in_region> lower part of split region is lost");

        /* insert splitting point in the upper part of the splitted region */
        chainbottom_insert(regptr, p, s->endp[1]);

        /* create new regions */
        mult_ins_regions(regptr, p);

        /* insert splitting point in the lower part of the splitted region */
        chaintop_insert(rnext, p, (s-1)->endp[1]);
}



void    on_boundary(regptr, p)
        /* point p lies on the LOWER boundary of region pointed by regptr */
        /* there is at least one segment emanating from p, pointed by s */
Region          *regptr;
Endpoint        *p;
{
        int                     cond, xy_ep_compare();
        double                  slope, bound_slope;
        Endpoint        *q1, *q2, *e1, *e2;
        Region          *rnext, *new_region();

        /* set e1 & e2, the endpoints of boundary segment on which p lies */
        rnext = regptr->next_reg;
        if (regptr == sentnl){  /* on the uppermost boundary */
#ifdef DIAGNOSTICS
                if (rnext == NULL)
                        ERROR("on_boundary> regions exhausted");
#endif
                e1 = rnext->top->p;
                e2 = rnext->up_p;
        }
        else{
                e1 = regptr->bottom->p;
                e2 = regptr->low_p;
        }
        q1 = s->endp[1];

        if (E_COMPARE(e1->x, e2->x) == 0){ /* vertical region boundary */
#ifdef DIAGNOSTICS
                /* assert that current point is not behind or ahead */
                if (E_COMPARE(e1->x, p->x) != 0 || E_COMPARE(e1->y, p->y) > 0 ||
                                                        E_COMPARE(e2->y, p->y) < 0)
                        ERROR("on_boundary> sweep not proceeding as desired");
#endif
                /* cond = 0, if segment and boundary parallel */
                cond = E_COMPARE(p->x, q1->x);
        }
        else{           /* non-vertical region boundary */
#ifdef DIAGNOSTICS
                /* assert that current point is not behind or ahead */
                if (E_COMPARE(e1->x, p->x) > 0 || E_COMPARE(e2->x, p->x) < 0)
                        ERROR("on_boundary> sweep not proceeding as desired");
#endif

                bound_slope = (e1->y - e2->y) / (e1->x - e2->x);
                if (E_COMPARE(p->x, q1->x) == 0)        /* vertical segment */
                        cond = 1;
                else{
                        slope = (p->y - q1->y) / (p->x - q1->x);
                        /* cond =  1/0/-1  if slope >/=/< bound_slope */
                        cond = E_COMPARE(slope, bound_slope);
                }

                if (cond == 1){    /* update lower boundary of *regptr */
                        if (regptr != sentnl)    /*  if not topmost region */
                                chainbottom_insert(regptr, p, q1);

                        for (s++;  ;  s++, regptr = regptr->next_reg){
                                q2 = q1;
                                if (++seg_ind >= n_segs || p != s->endp[0]){
                                        ins_region(new_region(p, q2, e2), regptr);
                                        return;
                                }

                                q1 = s->endp[1];
                                slope = (q1->y - p->y) / (q1->x - p->x);
                                if ((cond = E_COMPARE(slope, bound_slope)) <= 0)
                                        break;
                                ins_region(new_region(p, q2, q1), regptr);
                        }
                        ins_region(new_region(p, q2, e2), regptr);
                        if ((regptr = regptr->next_reg) == NULL)
                                ERROR("on_boundary> error in region insertion");
                        rnext = regptr->next_reg;       /* maintain invariant */
                }
        }

        if (cond == 0){         /* parallel segment and boundary */
                /* pick longer segment, comparing e2 and q1 */
                if ((cond = E_COMPARE(e2->x, q1->x)) < 0
                                || cond == 0 && E_COMPARE(e2->y, q1->y) < 0){
                        /* use q1, not e2, in *regptr-*rnext boundary */
                        if (regptr != sentnl)
                                regptr->low_p = q1;
                        if (rnext != NULL)
                                rnext->up_p = q1;
                }
                /* advance segment pointer/index */
                if (++seg_ind >= n_segs || p != (++s)->endp[0])
                        return;
                q1 = s->endp[1];
        }
        /* insert new regions after *regptr */
        ins_region(new_region(p, e2, q1), regptr);
        mult_ins_regions(regptr->next_reg, p);
        if (rnext != NULL){
```

```
                q1 = (s-1)->endp[1];
                if (rnext->top->p != p)
                        chaintop_insert(rnext, p, q1);
                else
                        rnext->up_p = q1;
        }
}



void    shatter(seg_head)
Table           *seg_head;
{
        int             pos, past_p, p_ind;
        Endpoint        *p, *q;
        Region          *r, *rnext, *reg_locate();

        /* collect & sort the endpoints of line segments;  assign n_segs */
        preprocess((LineSegment *)seg_head->to_elem, seg_head->n);

        /* initialize the data structure for regions, inserting sentinel */
        init_regions(n_segs, n_points, &sentnl);

        /* process the leftmost point; create regions incident upon it */
        s = newseg_table;
        seg_ind = 0;
#ifdef DIAGNOSTICS
        if (s->endp[0] != points)
                ERROR("shatter> point & segment arrays inconsistent");
#endif
        mult_ins_regions(sentnl, p = points);

        /* sweep simultaneously the arrays of points and segments in order, */
        /*      updating the region map, and triangulating at the same time */
        for (p_ind = 0, p++;  ++p_ind < n_points;  p++){

#ifdef DEBUG
        reglist_print();
        regtree_print();
        fprintf(stderr, "point (%.3f, %.3f)\n", p->x, p->y);
#endif
                r = reg_locate(p, &pos);
#ifdef DIAGNOSTICS
                if (r == NULL || r == sentnl)
                        ERROR("shatter> point location faulted");
#endif

                if (pos == 0){  /* p in interior of region *r */
                        in_region(r, p);
                        continue;
                }
                if (pos == 1 || pos == 3)       /* p on upper boundary of *r */
                        r = r->prev_reg;

                /* p is on lower boundary of *r (possibly sentinel)  */
                /* find topmost such region with p not on its UPPER boundary */
                for (; r != sentnl && (p == r->up_p || wrt_boundr(1, r, p) == 0); )
                        r = r->prev_reg;

                /* we assign q to the boundary's rightmost vertex */
                if (r == sentnl)                /* sentinel = topmost region */
                        q = r->next_reg->up_p;
                else
                        q = r->low_p;

                /* remove regions, if any, which end at p */
                for (past_p = (p != q);  (rnext = r->next_reg) != NULL;  ){
                        if (p != rnext->low_p){
                                if (wrt_boundr(-1, rnext, p) != 0)
                                        break;

#ifdef DIAGNOSTICS
                                if (past_p) /* 2 region borders extend past p */
                                        ERROR("shatter> crossing segments");
#endif
                                past_p = 1;
                                q = rnext->low_p;
                        }
                        /* there are regions to be terminated */
                        upd_topchain(rnext, p);
                        del_region(rnext);
                }

                if (past_p){    /* some region boundary extends past p */
                                /* update region above, if needed */
                        if (r != sentnl)        /* NOT topmost region */
                                /* include p on lower boundary of *r */
                                if (p == r->low_p)
                                        chainbottom_insert(r, p, q);
#ifdef DIAGNOSTICS
                                else
                                        if (q != r->low_p)
                                                ERROR("shatter> multiple qs");
#endif

                        /* update region below, if needed */
                        if ((rnext = r->next_reg) != NULL)
                                /* include p on up boundary of *rnext */
                                if (p == rnext->up_p)
                                        chaintop_insert(rnext, p, q);
#ifdef DIAGNOSTICS
                                else
                                        if (q != rnext->up_p)
                                                ERROR("shatter> multiple qs");
#endif

                        /* if segments emanating from p, process them */
                        if (seg_ind < n_segs && p == s->endp[0])
                                on_boundary(r, p);
                }
                else{
                        if (seg_ind >= n_segs && p_ind == n_points - 1)
                                break;  /* point, seg arrays exhausted */

                        if (seg_ind >= n_segs || p != s->endp[0])
                                chain_merge(r, p);
                        else{   /* there are segments to the right of p */
                                if (r != sentnl)
                                        chainbottom_insert(r, p, s->endp[1]);
                                rnext = r->next_reg;
                                mult_ins_regions(r, p);
                                if (rnext != NULL)
                                        chaintop_insert(rnext, p, (s-1)->endp[1]);
                        }
                }
        }
        clean_regions();
        (void) free((char *)points);
        (void) free((char *)newseg_table);
}
```

```
   /*********                   tetrahedra.c                    *********/
   /*   Routines that exploit the list of triangles to match them up    */
   /*         into triangular prisms, and to tetrahedralize them        */


#include "include.h"


extern Table            triangles, tetrhdr_table;



#ifdef DEBUG
/*****************************************************************************/

void    tri_dump(t_head)
Table           *t_head;
{
        int             i, k;
        Triangle        *t;

        fprintf(stderr, "\nTRIANGLES:  # = %d\n", t_head->n);
        for (t = (Triangle *)t_head->to_elem, k = t_head->n;  --k >= 0;  t++){
                for (i = 0;  i < 3;  i++)
                        fprintf(stderr, "  (%.3f, %.3f, %.3f)",
                                t->p[i].x, t->p[i].y, t->p[i].z);
                fprintf(stderr, "\n");
        }
        fprintf(stderr, "\n");
}

/*****************************************************************************/
#endif



#ifdef DIAGNOSTICS
/*****************************************************************************/

int     test_match(t1, t2)
                /* return 0 if t1 & t2 have same xy-projection, 1 else */
Triangle        *t1, *t2;
{
        int             xy_p_compare();
        Point_3d        *p1 = t1->p, *p2 = t2->p;

        return(xy_p_compare(p1, p2) != 0 || xy_p_compare(p1 + 1, p2 + 1) != 0
                                         || xy_p_compare(p1 + 2, p2 + 2) != 0);
}

/*****************************************************************************/
#endif



int     triang_compare(tr1, tr2)
Triangle        *tr1, *tr2;
{
        int             i, c, xy_p_compare();
        double          sum1, sum2;
        Point_3d        *p, *q;

        for (i = 0, sum1 = sum2 = 0.0, p = tr1->p, q = tr2->p;  i < 3;
                                        sum1 += p->z, sum2 += q->z, i++, p++, q++)
                if ((c = xy_p_compare(p, q)) != 0)
                        return(c);

        if ((c = E_COMPARE(sum1, sum2)) == 0)
                ERROR("triang_compare> double occurence of triangle");
        return(c);
}



void    extract_tetrahedra(tr1, tr2)
Triangle        *tr1, *tr2;
{
        int             i;
        Point_3d        *p, *q;
        Tetrahedron     t;

        for (i = 3;  --i >= 0; )
                t.p[i] = tr1->p[i];
        /* report the one/two/three tetrahedra */
        for (i = 3, q = tr2->p + 2;  --i >= 0;  t.p[i] = *q, q--)
                if (E_COMPARE(t.p[i].z, q->z) != 0){    /* remove vertex p[i] */
                        t.p[3] = *q;
                        INS_TABLE(t, Tetrahedron, &tetrhdr_table);
#ifdef DEBUG
                        fprintf(stderr, "tetrahedron: ");
                        p = t.p;
                        fprintf(stderr, " (%.2f, %.2f, %.2f),", p->x,p->y,p->z);
                        p++;
                        fprintf(stderr, " (%.2f, %.2f, %.2f),", p->x,p->y,p->z);
                        p++;
                        fprintf(stderr, " (%.2f, %.2f, %.2f),", p->x,p->y,p->z);
                        p++;
                        fprintf(stderr, " (%.2f, %.2f, %.2f)\n",p->x,p->y,p->z);
#endif
                }
}



void    tetrahedralize()
{
        int             i, n_triangles;
        Triangle        *t;

        if ((n_triangles = triangles.n) % 2 != 0)
                ERROR("tetrahedralize> unmatched triangle");

        /* sort the triangles */
        qsort(triangles.to_elem, n_triangles, sizeof(Triangle), triang_compare);
#ifdef DEBUG
        tri_dump(&triangles);
#endif

        /* match opposite triangles, and report tetrahedra */
        t = (Triangle *)triangles.to_elem;
        for (i = n_triangles;  (i -= 2) >= 0;  t += 2){
#ifdef DIAGNOSTICS
                if (test_match(t, t + 1) != 0)
                        ERROR("tetrahedralize> non-matching triangles");
#endif
                extract_tetrahedra(t, t + 1);
        }
}
```