A DATA STRUCTURE FOR MANIPULATING THREE-DIMENSIONAL SUBDIVISIONS

Michael Jay Laszlo

(Thesis)

CS-TR-125-87

August 1987

A DATA STRUCTURE FOR MANIPULATING THREE-DIMENSIONAL SUBDIVISIONS

Michael Jay Laszlo

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

August 1987

for my parents Maurice and Phyllis

Acknowledgements

I wish foremost to thank my thesis advisor David Dobkin. He introduced me to computational geometry and computer graphics. He has been consistently supportive and encouraging, and generous with his time and thoughts. He has involved himself with this work, making himself available for discussion, criticizing my ideas and offering ideas of his own. I thank David profusely.

Bernard Chazelle and Andrea LaPaugh were the readers for this dissertation. I thank them for their valuable comments and suggestions. I also wish to thank Bernard Chazelle, David Dobkin, Richard Lipton, Kenneth Steiglitz, Kenneth Suppowit and Mihalis Yannakakis with the help of whose excellent courses I learned the foundations of computer science. I thank Gerree Pecht, Sharon Rodgers and Grace Kehoe for always being so helpful. I am grateful to the Department of Computer Science generally for giving me this wonderful opportunity to study at Princeton.

I wish to thank colleagues and friends with whom I shared the joys and frustrations of studying computer science (and summer baseball): Rob Abbott, K. Balasubramanian, Heather Booth, Matt Clark, Tim Corica, Mordecai Golin, Luen Heng, Boris Kogan, Bill Lin, E. S. Panduranga, Arvin Park, Ken Salem, Deborah Silver, and Annemarie Spauster. I wish blessings to other friends at Princeton who lent light to my days and nights, in particular to Kiki van Raalte, Andy Lanouette, Susannah Wolfson and Karen Weissman. Blessings also to my friends beyond the pale of Princeton whose friendship is much valued: Jeremy Geller, Henny Regnier, Andy Strauss, Daniel Williams, Sue Gordon, Ronald Kamm and of course the Laszlo and Worton families.

Some of the work presented here has appeared in the paper "Primitives for the Manipulation of Three-Dimensional Subdivisions" [DL].

The work was supported in part by NSF grants MCS83-03926 and CCR85-05517.

Table of Contents

Chapter 1

Introduction

1.1 Overview

One principal concern of computational geometry is the design of algorithms for solving inherently geometric problems. Many such problems involve computing properties of geometric objects. Computing the number of intersections among n line segments in the plane, or determining the volume of some polyhedron, are examples of this type of problem. Another sort of geometric problem constructs new geometric objects. Constructing the intersection of two polyhedra, or the convex hull of n points in the plane, are problems of this type. An algorithm that solves either type of problem needs to be able to represent and manipulate the geometric objects to which it is applied. An algorithm for solving the latter type of problem requires the capacity to represent and construct the geometric object that it is expected to produce. Geometric objects are represented and handled by the use of data structures. The data structures vary enormously according to the nature of the geometric object to be represented and the manipulations to be supported. The design of data structures is an important part of computational geometry since algorithms depend so heavily upon them.

The *polygonal subdivision* is one type of geometric object which many disparate applications and problems must represent and handle. This is a partition of a bounded or unbounded region of $\Re^2$ into a collection of polygons. A polygon may be decomposed into convex polygons because its properties (such as non-convexity) make it hard to work with, but its convex parts are more amenable to computation. The plane may be embellished by a map to reflect geographical regions or non-overlapping 'spheres of influence.' The plane may be divided into convex polygons by an arrangement of lines, a partition of interest in its own right. The boundary of a polyhedron can be treated

as a polygonal subdivision, to be used for rendering the polyhedron with computer graphics. Several data structures exist for handling polygonal subdivisions, among them the quad-edge structure [GS], the winged-edge structure [Ba], and the doubly connected edge list [MP].

A *polyhedral subdivision* — the 3-dimensional analogue of the polygonal subdivision — is a partition of a bounded or unbounded region of $\Re^3$ into a collection of polyhedra. The need to manipulate polyhedral subdivisions arises from problems involving arrangements of planes in $\Re^3$, decomposition of polyhedra, 3-dimensional visibility graphs (often used in robotics), questions involving spheres of influence, and computer graphics, to name a few.

In the past, there has existed no single cohesive data structure for representing and manipulating polyhedral subdivisions. When a researcher needed such a data structure, he or she typically concocted an ***ad hoc*** structure to suit his or her immediate purpose. The current dissertation fills this gap. We present a data structure, known as the *facet-edge data structure*, for representing polyhedral subdivisions. We develop a notation for formulating queries of the data structure, operators for creating and modifying the structure, and a concise implementation. We also give applications of the facet-edge structure.

Our design of the facet-edge structure is guided in large part by an examination of data structures for *polygonal* subdivisions. The quad-edge structure in particular proves a rich mine of resources for our purposes. The quad-edge structure treats the edge as its atom. An edge e connects two vertices (its endpoints), and two polygons (in whose boundaries e lies). Edge e is adjacent to four edges, two of which belong to the polygon to the left of e, and two to the polygon to the right of e. The quad-edge structure permits each of the edges adjacent to e to be determined in constant time. The facet-edge structure treats as ***its*** atom the *facet-edge pair*. This is a pair a consisting of a facet f and an edge e incident to f. The facet-edge pair connects two

vertices (the endpoints of e) and two polyhedra (in whose boundaries facet f lies). Facet-edge pair a is adjacent to four facet-edge pairs. In the ring of facets incident to edge e, two facets are adjacent to facet f — these are the facets of two of the facet-edge pairs adjacent to a. In the ring of edges that bound f, there are two edges adjacent e — these are the edges of the other two facet-edge pairs adjacent to a. The facet-edge structure makes the four facet-edge pairs adjacent to a available in constant time. The role played by the edge in the quad-edge structure is analogous to the role played by the facet-edge pair in the facet-edge structure.

1.2 Previous Work

We review the data structures used by researchers to represent polyhedral subdivisions. We emphasize the essentials of each data structure, by and large neglecting the sundry embellishments researchers have added to accommodate particular applications. We assume the reader has knowledge of basic notions of combinatorial topology such as incidence and adjacency. The reader is welcome to first scan Chapter 2 of the thesis, or the books [Ag, Gi, He], for definitions as needed.

In the literature, polyhedral subdivisions are represented by several different means. One way is to represent the graph known as the *incidence lattice*. Each cell of the subdivision — that is, each vertex, edge, facet and polyhedron — is represented by a node. Nodes n and n' are connected by an edge iff the $k+1$-dimensional cell (or $k+1$-cell) corresponding to n is incident to the k-cell corresponding to n', where $0 \leq k < 3$. The incidence lattice is typically represented so that connected nodes n and n' each possesses a pointer to the other, so pointers can be followed to higher- and lower-dimensional cells. The incidence lattice is used to represent the convex hull of a finite point set in $\Re^m$ in both Kallay's beneath-beyond method [Ka] and Seidel's shelling method [Se]. (Note that the incidence lattice is suitable for subdivisions of arbitrary dimension.) Aurenhammer and Edelsbrunner also use the data structure to represent a polyhedral subdivision [AE]. (Their application is discussed further in

Section 8.4.) They enhance the data structure — specifically for handling *polyhedral* subdivisions — as follows. Pointers to the edges that bound a facet are stored (at that facet's node) in the order in which they occur. Similarly, the facets incident to a common edge are stored at that edge in order. Furthermore, 'pointers are established such that given a facet f and incident edge e, the adjacent two edges incident with f and the adjacent two facets incident with e are available in constant time [AE].' The structure they propose is spelled out in no greater detail.

A second way of representing a polyhedral subdivision is by storing the graph we may call the *k-adjacency graph*. For fixed $k \in \{0,1,2,3\}$, each k-cell is represented by a node. Two nodes are connected by an edge iff their corresponding cells are adjacent in the subdivision. The 0-adjacency graph represents the vertex-edge graph embedded in the subdivision. Bowyer uses a 3-adjacency graph to represent a tetrahedral subdivision [Bo]. The node corresponding to each tetrahedron possesses a pointer to each of the four adjacent tetrahedra. Seidel's horizon graph is an $m-2$-adjacency graph of the current $m-1$-dimensional horizon of the convex hull of a finite point set in $\Re^m$ [Se].

A third technique for representing polyhedral subdivisions is to represent each k-cell independently, for fixed k. The relations between k-cells are not explicitly represented; however, since distinct k-cells share (lower-dimensional) cells in common, typically these relations are implicitly captured by the data structure. Chazelle represents each polyhedron of a polyhedral subdivision independently [Ch]. Each polyhedron is modelled using a boundary representation scheme [Re]. Watson represents a tetrahedral subdivision by storing the tetrahedra that comprise it [Wa]. Each tetrahedron is represented by a list of its four vertices. Bhattacharya models a tetrahedral subdivision by representing each of its triangular facets [Bh]. Each facet node is stored in a dictionary in which look-up is performed by keying on the three vertices that determine a facet. Use of the dictionary enables efficient queries regarding the relations that hold between facets.

Data structures also exist for handling *polygonal* subdivisions. The winged-edge [Ba] and quad-edge [GS] structures are presented and examined in Chapter 3.

1.3 Organization of the Dissertation

In Chapter 2 we introduce notation and definitions concerning polygonal and polyhedral subdivisions.

Chapter 3 focuses upon polygonal subdivisions. We give examples of problems and applications which give rise to, and require the manipulation of, these subdivisions. We present notation for formulating queries concerning polygonal subdivisions, and we present two common data structures — the winged-edge and the quad-edge structures — for representing them. Our interest in polygonal subdivisions is several-fold. First, we wish to set a context for the work of later chapters. Second, the notation used to query polygonal subdivisions serves as a model for developing notation for querying *polyhedral* subdivisions. Third, as we have already indicated, examination of data structures for handling polygonal subdivisions provides insight into the design of a data structure for handling polyhedral subdivisions.

In Chapter 4 we develop notation, and in particular a small set of functions, in terms of which we may formulate queries concerning polyhedral subdivisions. Queries involve such things as determining the two endpoints of an edge, or the cycle of facets incident to a common edge.

Chapter 5 presents an implementation of the facet-edge structure, that is, a characterization of the data structure in terms of simpler, more familiar data structures.

In Chapter 6 we introduce a set of construction primitives. With these primitives, facet-edge representations of polyhedral subdivisions can be created and modified.

In Chapter 7 we present higher-level construction operators — defined in terms of the primitives of Chapter 6 — which make the construction of non-trivial polyhedral subdivisions manageable.

In Chapter 8 we develop several applications that benefit from use of the facet-edge structure. In particular, an algorithm for constructing a 3-dimensional Voronoi diagram is described in considerable detail. Working code for this application, as well as pictures of 3-dimensional Voronoi diagrams created using the code, are presented in the Appendices.

In Chapter 9 we review the contributions of this dissertation and pose some open problems.

Chapter 2

Some Notation, Definitions and Observations

2.1 Cell Complexes, Subdivisions and Related Concepts

Let p be a simple vertex-edge path embedded in the plane. Assume p divides the plane into two regions — either p is a cycle, or p is a chain whose first and last edges are rays. A polygon in the plane is either of the two closed regions bounded by such a path p. A subdivision of the plane is a collection of polygons which forms a partition of the plane. Two examples of such subdivisions are pictured in Figure 1.

We give more formal definitions of these ideas. Given Euclidean space $\Re^m$, a *k-cell* (where $0 \leq k \leq m$) is a closed subspace of $\Re^m$ whose relative interior is homeomorphic to $\Re^k$, and whose boundary is non-null. We call a 0-cell a *vertex*, a 1-cell an *edge*, a 2-cell a *polygon* or a *facet*, and a 3-cell a *polyhedron*. (Note that a polyhedron in this sense has neither handles nor cavities. We treat only such polyhedra in this thesis.) A k-cell is said to have *dimension* k. Note that a cell may be unbounded; for instance, an edge can be a closed segment (bounded by two vertices) or a ray (bounded by one vertex). It is sometimes intuitively convenient to regard k-cell c as lying in the k-dimensional flat or affine space *aff* c. However, we do not insist upon this restriction. Edges may be regarded as curves, and polygons as curved sheets.

A *(closed) cell complex* of $\Re^m$ is a finite collection C of cells of $\Re^m$ such that

(i) the relative interiors of cells of C are pairwise disjoint,

(ii) for each cell $c \in C$, the boundary *bd* c of cell c is the union of elements of C, and

(iii) if $c, d \in C$ and $c \cap d \neq \emptyset$, then $c \cap d$ is the union of elements of C.

A complex is n-dimensional if it contains an n-cell but no cell of greater dimension. An n-dimensional complex for which every k-cell is contained in (the boundary of) some

n-cell is called an *n-complex*. We let $\mathcal{U}(C)$ be the union of the cells of the n-complex C, and consider C a *subdivision* of $\mathcal{U}(C)$.

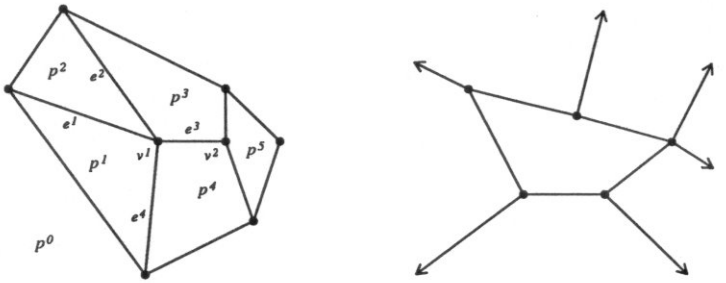


Fig. 1. This diagram depicts cell subdivisions C (on the left) and C' (on the right).

Figure 1 depicts two 2-complexes. The complex C includes the five polygons p^1 through p^5, but not p^0. The union of its cells $\mathcal{U}(C)$ is homeomorphic to a closed disk or 2-ball. The unbounded region labelled p^0 is *not* a 2-cell — being an annulus it is not homeomorphic to $\Re^2$ — and so is not part of C. However, we may add to the plane a point p^∞ at infinity , thereby forming the *extended plane* or sphere. The unbounded region p^0 contains point p^∞, and is now a legitimate 2-cell. By adding polygon p^0 to C, complex C becomes a subdivision of the sphere.

Cell complex C' *is* a subdivision of the plane. By adding to the plane the point p^∞ — where the rays meet at infinity — the complex is viewed as a subdivision of the sphere.

The 2-dimensional subdivisions with which we shall work are generally assumed to be subdivisions of the sphere. The advantage in working in the sphere rather than the plane is that it allows all k-cells to be treated in a uniform manner. For instance, in the plane, some edges are bounded while other edges are rays. In the sphere, "rays"

are in fact bounded edges one of whose endpoints is p^∞. We shall use the phrases '2-dimensional subdivision', 'subdivision of the sphere' and 'polygonal subdivision' interchangably.

The 3-dimensional subdivisions with which we shall work are generally assumed to lie in $\Re^3$ augmented by a point p^∞ at infinity, called *extended space*. This permits edges, facets and polyhedra each to be treated in a uniform manner, analogous to the 2-dimensional case. We use the phrases 'polyhdedral subdivision' and '3-dimensional subdivision' interchangably. We talk about a 'ball-complex' when referring to a polyhedral subdivision of a 3-ball.

A cell $d \subset c$ is said to be a *face* of c; if in addition $c \neq d$, then d is a *proper face* of c. Note that any proper face of cell c lies in the boundary of c. If one of c or d is a proper face of the other, c and d are said to be *incident*. The *combinatorial boundary* of a cell c, denoted ∂c, is the set of proper faces of c. Note that $\mathcal{U}(\partial c) = bd\, c$. The combinatorial boundary of cell complex C is $\partial C = \{c \in C \,|\, c \subset bd\, \mathcal{U}(C)\}$. The *combinatorial closure* of a cell c, denoted $\partial^* c$, is defined by $\partial^* c = \partial c \cup \{c\} = \{d \in C \,|\, d \subset c\}$. For instance, in complex C of Figure 1, vertex v^1 is a face of both edge e^2 and polygon p^2, and is in addition incident with each of these cells. The combinatorial boundary of e^3 is $\partial e^3 = \{v^1, v^2\}$.

Given n-complex C, by convention there exists one (null) $n+1$-cell of which every n-cell of C is a proper face; likewise there exists one (null) (-1)-cell which is a proper face of every vertex. Distinct k-cells c and d (for $0 \leq k \leq n$) are then said to be adjacent if (i) there exists some $k-1$-cell of C that is a face of both c and d, and (ii) there exists some $k+1$-cell of C of which each of c and d is a face. In complex C of Figure 1, edges e^1 and e^2 are adjacent since they contain a common face (vertex v^1) and are contained by a common cell (polygon p^2). Vertices v^1 and v^2 are adjacent since they contain a common face (the null (-1)-cell) and are contained by a common cell (edge e^3). On the other hand, edges e^1 and e^3 are not adjacent since they are the face of no common

polygon. Note that two cells that are incident are of different dimensions, whereas two cells that are adjacent are of the same dimension.

The relationships of incidence and adjacency that hold among the cells of a subdivision are called *combinatorial relationships*. In this dissertation we are principally concerned with the means of representing such relationships in 2- and 3-dimensional subdivisions. A query concerning some combinatorial relationship is called a *combinatorial query* or a *traversal query*. The term *traversal* arises from viewing a sequence of queries as a means of moving from cell to adjacent cell within a complex, that is, of traversing the complex.

Two subdivisions C and C' are said to be *combinatorially equivalent* if there exists a bijection ϕ from C onto C' such that

(i) the image of a k-cell under ϕ is a k-cell, and

(ii) cells c and d are adjacent in C iff cells $\phi(c)$ and $\phi(d)$ are adjacent in C'.

Let C and C' be two combinatorially equivalent subdivisions of the sphere. C' may be obtained from C by revolving C in the sphere, and possibly changing the relative locations of its vertices and the shapes of its edges. It may also be necessary to turn the sphere inside-out. This last operation amounts to reflecting the sphere across a plane that cuts through its center.

2.2 Duality

The dual of polygonal subdivision C is a second polygonal subdivision C^* of the sphere. There is associated with each vertex (edge, polygon) of *primal* subdivision C, a polygon (edge, vertex) of *dual* subdivision C^*, such that two cells of C are adjacent in C if and only if their corresponding cells are adjacent in C^*. We denote by c^* the dual of cell c, that is, the cell associated with c under duality. Figure 2 illustrates a primal subdivision and one of its dual subdivisions.

More generally, the dual of a complex C of space $\Re^m$ is a second complex C^* of $\Re^m$ for which there exists a one-to-one mapping * from C onto C^* such that

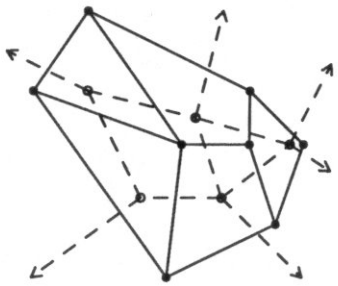


Fig. 2. This diagram illustrates a subdivision (bullets are vertices, solid lines are edges) and one of its dual subdivisions (hollow bullets are vertices and dashed lines are edges).

(i) the image of a k-cell under * is an $m-k$-cell, and

(ii) cells c and d are adjacent in C iff cells c^* and d^* are adjacent in C^*.

The complex C^* dual to C is not unique; however, up to combinatorial equivalence it is unique. Furthermore, $(C^*)^* = C$.

2.3 Orientation, Direction and Spin

A disk (or 2-ball) together with a sense of rotation is said to be *oriented*. If the oriented disk lies in $\Re^3$ and is viewed from one of its sides, its orientation will appear to have either clockwise or counter-clockwise sense of rotation. A sphere is said to be oriented if all disks that lie in it have the same orientation. That is, if viewed from say the outside of the sphere, all disks have clockwise, or all disks have counter-clockwise, sense of rotation. Where C is a subdivision of the oriented sphere, each cell $c \in C$ is said to be oriented, and has the sphere's orientation.

The points of an edge may be assigned either of two linear orderings. Each is called a *direction*, and an edge together with a direction is said to be *directed*. A directed edge has an *origin* vertex (the first point of its initial segment) and a *destination* vertex (the last point of its terminating segment).

A directed edge in $\Re^3$ may be assigned either of two senses of rotation. Each is called a *spin*, and a directed edge with spin is said to be *spun*. When viewed from its

destination to origin along the length of the edge, a directed edge's sense of rotation appears clockwise (counter-clockwise) if its spin is left-handed (right-handed). This can be remembered if one extends the thumb and curls the finger of one hand. The thumb represents a directed edge, its base the edge's origin, its tip the edge's destination. The fingers curl in agreement with the sense of rotation about the edge. For instance, if the right hand is used, the fingers curl counter-clockwise as the thumb is viewed from tip to base, indicating that right-handed spin implies a counter-clockwise sense of rotation.

2.4 Observations Concerning Closed Subdivisions

We make some observations concerning arbitrary (closed) subdivision C of the sphere. The paragraph that follows each observation outlines a proof.

Observation 1: Any edge $e \in C$ has two distinct endpoints (vertices to which it is incident).

Since every edge is bounded, e is incident to a vertex at both its initial and terminating segment. If these were the same vertex, e would be a loop; its relative interior would then *not* be homeomorphic to a line $\Re^1$, so e would not be an edge.

Observation 2: Any edge $e \in C$ is incident to two distinct polygons.

Since edge e is embedded in the sphere, it is incident to no more than two polygons. Suppose edge e were incident on both of its sides to the same polygon p. Then for points x of the relative interior of e, there would exist neighborhoods of x contained in p. Thus the relative interiors of p and e would intersect at x, violating condition (i) of a cell complex.

Observation 3: Any edge $e \in C$ is adjacent to no more than four distinct edges.

Each edge adjacent to e shares a vertex and a polygon with e. By the previous observations, there are only four combinations of vertices and polygons.

We also make observations concerning any polyhedral subdivision C of extended space.

Observation 4: Any edge $e \in C$ has two distinct endpoints.

See the justification for Observation 1.

Observation 5: Any facet $f \in C$ is incident to two distinct polyhedra.

Similar to the justification for Observation 2, but one dimension higher.

Observation 6: Given facet $f \in C$ and edge $e \in \partial f$, there exist one or two facets of C adjacent to f, which are also incident to edge e.

Consider one of the polyhedra p incident to f. The boundary ∂p is (combinatorially) a subdivision of the sphere. Edge $e \in \partial p$ is incident to two distinct facets, one of which is f, the second of which is adjacent to f in C. Similarly, the other polyhedron incident to f in C contributes one facet adjacent to f and containing edge e. The facet that each of the two polyhedra contributes may be one and the same.

Observation 7: Given facet $f \in C$ and edge $e \in \partial f$, there exist one or two edges of C adjacent to e, which also lie in the boundary of f.

There can be only one or two edges of ∂f adjacent to $e \in \partial f$ since ∂f is a simple vertex-edge path.

2.5 Open Cell Complexes and Open Subdivisions

The notion of an open subdivision of the sphere is essentially a generalization of the notion of a (closed) subdivision of the sphere. Open subdivisions permit edges that are loops, for which one vertex serves as both endpoints. It also permits edges which belong twice to the boundary of the same polygon. An open polygonal subdivision, and its dual, is depicted in Figure 3.

More formally, we define an *open k-cell* of Euclidean space $\Re^m$ to be an open subspace homeomorphic to $\Re^k$. An open 0-cell is a vertex, an open 1-cell an *open edge* (an edge without its endpoints), an open 2-cell an *open polygon* (a polygon without its boundary). An open edge may have two limit points (its endpoints, in which case it

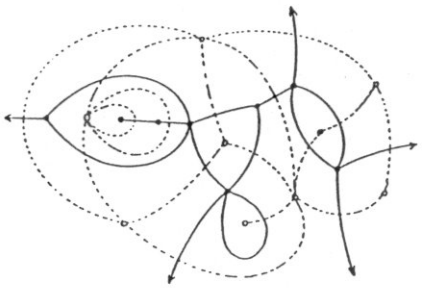


Fig. 3. This diagram, taken from [GS], depicts a subdivision of the sphere and one of its duals.

may be referred to as a *segment*), or just one limit point (a *loop*). For brevity, we will use the terms 'edge' and 'polygon' when the cells are known by context to be open.

An *open cell complex* of $\Re^m$ is a finite collection C of open cells of such that

(i) the cells of C are pairwise disjoint,

(ii) for each cell $c \in C$, $bd\, c$ is the union of elements of C, and

(iii) if $c, d \in C$ and $cl\, c \cap cl\, d \neq \emptyset$, then $cl\, c \cap cl\, d$ is the union of elements of C.

Here $cl\, c$ denotes the closure of cell c. Each of the concepts related to cell complexes (such as incidence, adjacency and duality) apply to open cell complexes in a natural way, and we shall not try the reader's patience by defining these anew.

We observe that each (closed) subdivision C coincides with a unique open subdivision C' readily obtained from C. By stripping away the boundary of each cell of C, we produce C'. Open complexes are important to us for one reason only. The construction of a (closed) subdivision of the sphere, to be presented in Chapter 7, is an incremental process in which a sequence of open subdivisions are built until one is obtained which coincides with the target (closed) subdivision.

Chapter 3

Subdivisions of the Sphere

3.1 Introduction

In this chapter we examine (2-dimensional) subdivisions of the sphere. The examination is motivated by several concerns. First, we wish to set a context for the work of later chapters. Second, we wish to familiarize the reader with some problems and applications that involve 2-dimensional subdivisions, with the hope that the reader does not inadvertantly presume that subdivisions are useless, contrived constructs not worth representing in the first place. Third, by studying what sort of queries are supported by data structures for handling 2-dimensional subdivisions, as well as the implementation of such structures, we gain insight into the design goals and implementation of a data structure for handling 3-dimensional subdivisions.

In Section 3.2 we mention some problems and applications that involve 2-dimensional subdivisions. In particular, we define and discuss the *Voronoi diagram*, the planar version of which is a subdivision of the plane. (The discussion paves the way to Chapter 8, in which the facet-edge structure is used to build a 3-dimensional Voronoi diagram.) In Section 3.3, we present the edge functions of the quad-edge structure, in terms of which queries of 2-dimensional open subdivisions may be formulated. In Section 3.4, we present two data structures for representing open subdivisions, the winged-edge and the quad-edge structures.

The current chapter focuses upon *open* subdivisions of the *sphere*. We choose to treat subdivisions of the *sphere*, rather than of other surfaces, for a couple of reasons. First, we will later (in Chapter 7) be interested in manipulating the boundaries of polyhedra — since each such boundary is combinatorially a subdivision of the sphere (the polyhedron having genus 0 and no cavities), the notation and concepts developed

here will apply. Second, the methods for handling subdivisions of the sphere proves a sufficiently rich source of inspiration for our subsequent work with 3-dimensional subdivisions. Treating subdivisions of other surfaces, particularly non-orientable ones, introduces unnecessary complications. We choose to work with *open* subdivisions since we will later (in Chapter 7) be concerned with the process of constructing a (closed) subdivision of the sphere — this involves building a succession of open subdivisions. The concepts presented here will apply.

3.2 Voronoi Diagrams

Diverse problems and applications involve 2-dimensional subdivisions. One class of problems involves the manipulation of polyhedra. One way to represent a polyhedron is by its boundary, which is a subdivision of the sphere. Many of the operations we might wish to perform upon polyhedra so represented require the capacity to manipulate their boundaries. One such operation is that of rendering a scene of polyhedra, which requires traversal of their boundaries to perform hidden surface removal [Ro]. Another operation is that of constructing the polyhedron formed by the intersection, union or difference of two polyhedra — this requires the capacity to traverse, build and modify boundaries [MP, HMMN].

A second type of problem involving polygonal subdivisions is that of decomposing a polygon into convex parts. The decomposition process requires the capacity to manipulate the partial decompositions, and the final decomposition is useful only if its representation admits queries.

Another problem, to which we now briefly turn, is that of building a Voronoi diagram. Let $S = \{s_1, \cdots, s_n\}$ be a finite set of points of the plane $\Re^2$ with Euclidean metric δ. The points s_i are called *sites*. Assume $n \geq 3$, and that the sites of S are in general position: no three are colinear and no four are cocircular. The Voronoi region of site s_i is $VR(s_i) = \{x \in \Re^2 \mid \delta(x, s_i) \leq \delta(x, s_j) \text{ for all } j \in \{1, \cdots, n\}\}$. Each Voronoi region is a convex, possibly unbounded polygon. The edges and vertices

of a Voronoi region are called *Voronoi edges* and *Voronoi vertices*, respectively. The collection of Voronoi regions $VR(s_i)$ as s_i ranges over S, together with all Voronoi edges and vertices, is called the *Voronoi diagram* of S, denoted $V(S)$. The Voronoi diagram is a subdivision of the plane.

An important dual to $V(S)$ is called the *Delaunay triangulation*, denoted $DT(S)$. The vertices of $DT(S)$ are the sites of S. Two sites s_i and s_j are connected in $DT(S)$ by a straight edge iff their respective Voronoi regions $VR(s_i)$ and $VR(s_j)$ are adjacent in $V(S)$. Since S is in general position, the polygons of $DT(S)$ are triangles, called *Delaunay triangles*. One interesting property of $DT(S)$ is that the circle determined by the vertices of any Delaunay triangle contains no sites in its interior. Figure 4 illustrates a Voronoi diagram and its dual Delaunay triangulation. See [PS] for other properties of Voronoi diagrams and Delaunay triangulations.

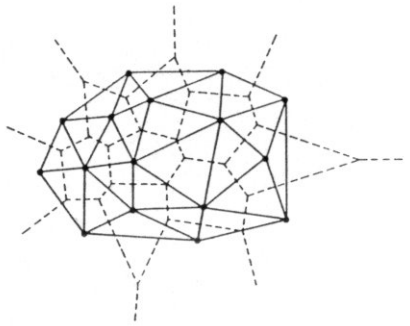


Fig. 4. This diagram, from [PS], depicts a Voronoi diagram (dashed edges) and its Delaunay triangulation (solid edges). The sites are represented by solid bullets.

There exist several algorithms for constructing $V(S)$. An algorithm of the divide-and-conquer variety partitions S into two sets S_1 and S_2 of approximately equal size, forms $V(S_1)$ and $V(S_2)$ by applying itself recursively to S_1 and S_2, and merges these two Voronoi diagrams to form $V(S)$ [PS, SH]. The merge step employs adjacency queries each of which is applied to some edge e and returns some edge adjacent to e. Which of the four edges adjacent to e is returned by the query is determined by the (possibly implicit) parameters with which the query is called. The linear-time worst-case behavior of the merge step, and the $\Theta(n \log n)$ worst-case time complexity of the overall algorithm, depends upon the ability to perform $O(m)$ adjacency queries in time $O(m)$. The winged-edge and quad-edge data structures support this. In fact, each of these data structures supports a single adjacency query in constant-time.

Algorithms for constructing the Voronoi diagram for a set of sites S adopt either of two approaches. Certain of the algorithms work in the primal space, constructing $V(S)$ directly [Ki1, SH]. Others work in the dual space, constructing $DT(S)$ directly, from whose representation $V(S)$ is derived [GS, LS]. The latter approach is sometimes preferable since it is easier (for some people) to conceptually manipulate the Delaunay triangles of the dual space. In addition, no Voronoi vertices need to be computed. However, this approach requires a data structure that readily yields its dual subdivision. The quad-edge data structure is ideal in this respect, and is used in [GS] for just this purpose.

$V(S)$ may be used to solve the following problem: Given a query point $q \in \Re^2$, determine which site of S is closest to q. The problem reduces to that of determining the region of $V(S)$ to which q belongs — the site that owns the region is the solution. This type of query is called a *point location query*. There are sundry data structures that support repeated point location queries efficiently. The process of building many of these structures from $V(S)$ require the use of traversal queries of $V(S)$ [Ki2, Pr].

The Voronoi diagram example suggests certain features that a data structure for handling 2-dimensional subdivisions should possess. First, it should provide notation for formulating traversal queries with which the four edges adjacent to any edge can be distinguished. Second, it should perform adjacency queries efficiently (ideally in constant time). Third, it should represent both a subdivision and its dual, and permit manipulation of both with equal ease. The quad-edge structure, which we present later in the current chapter, possesses these features.

3.3 Functions for Manipulating Open Subdivisions of the Sphere

In this section we present the edge functions of [GS], in terms of which queries about 2-dimensional open subdivisions may be formulated. Each edge function is applied to an edge, and returns a second edge. Our presentation simplifies some of the notions of [GS] since we are concerned solely with open subdivisions of the *sphere*, while Guibas and Stolfi treat open subdivisions of n-tori and of non-orientable surfaces as well. In addition, working in an orientable surface permits us to define the effect of applying an edge function to a vertex or polygon as well as an edge. To argue (as we do at the end of the current section) that the edge functions support all combinatorial queries concerning a 2-dimensional subdivision, we also introduce the new edge functions *Vertexedge* and *Polygonedge*.

Let C be an open subdivision of the sphere, and let $e \in C$ be any directed and oriented edge. Since directions and orientations can be selected independently, there exist four directed, oriented versions of e. Henceforth by 'edge' we mean just such a version; we use 'basic edge' to refer to an edge whose direction and orientation is unspecified or irrelevant to the matter at hand.

Where e is any edge, the *origin* vertex of e, denoted $e\mathit{Org}$, and the (possibly same) *destination vertex* of e, denoted $e\mathit{Dest}$, are determined by e's direction. They are defined such that e is directed from $e\mathit{Org}$ towards $e\mathit{Dest}$. The *left polygon* of e, denoted $e\mathit{Left}$, is that polygon incident to e whose orientation agrees with the direction of e, while the *right polygon* of e, denoted $e\mathit{Right}$, is the other (possibly same) polygon incident to e. The subdivision to which edge e belongs inherits e's orientation. The cells $e\mathit{Org}$, $e\mathit{Dest}$, $e\mathit{Left}$ and $e\mathit{Right}$ are taken to have the same orientation as edge e.

We define the flipped version $e\mathit{Flip}$ of edge e to be the same basic edge as e, with e's direction but taken (in a sphere) with opposite orientation. We also define the symmetric version $e\mathit{Sym}$ of edge e to be the same basic edge as e, with direction opposite that of e but taken with the same orientation. To grasp the effect of edge

functions *Flip* and *Sym* intuitively, assume edge e has an arrow on it, and we are viewing the open subdivision to which e belongs from inside the sphere. Operation $e\mathit{Flip}$ corresponds to viewing the open subdivision from *outside* the sphere, while $e\mathit{Sym}$ corresponds to reversing the direction of the arrow on e.

The flipped version of an oriented vertex or polygon can also be defined. Given oriented vertex v, we define $v\mathit{Flip}$ to be the same basic vertex but with orientation opposite that of v. Oriented polygon $p\mathit{Flip}$ is defined similarly. For completeness, *Sym* when applied to a vertex or a polygon is identity.

To define edge function *Onext*, consider the cycle of edges incident to vertex $e\mathit{Org}$. Edges that are loops each occur twice in this cycle, while segments (non-loop edges) each occur once. The sense of rotation in this cycle is determined by the orientation of the sphere. We call this cycle the *edge-cycle* $e\mathit{Org}$. Where edge e belongs to this cycle, we define edge $e\mathit{Onext}$ to be the edge in this cycle that follows the initial segment of e. Note if e is a loop, the edge's direction determines which of its ends is its initial segment. Edge $e\mathit{Onext}$ is directed so that its initial segment follows the initial segment of e in the cycle. (Note that this implies that $e\mathit{Onext}\,\mathit{Org} = e\mathit{Org}$.) Furthermore, the orientation of edges e and $e\mathit{Onext}$ is the same. Edge-cycle $e\mathit{Org}$ is simply $(e\mathit{Onext}^0\ e\mathit{Onext}^1 \cdots e\mathit{Onext}^{n-1})$ where vertex $e\mathit{Org}$ has degree n.

Let *edge-cycle* $e\mathit{Left}$ denote the cycle of edges that bound polygon $e\mathit{Left}$. The sense of rotation of the cycle is determined by the orientation of edge e. Where edge e belongs to this cycle, we define $e\mathit{Lnext}$ to be the edge that follows e in this cycle. Edge $e\mathit{Lnext}$ is directed so that its initial segment is coterminous with the terminal segment of e, along $e\mathit{Left}$. In addition, edges e and $e\mathit{Lnext}$ have the same orientation.

The dual version $e\mathit{Dual}$ of edge e is that version of edge e^* of the dual subdivision for which the following properties hold:

(A1′) $e\mathit{Dual}^2 = e$.

(A2′) $e\mathit{Sym}\mathit{Dual} = e\mathit{Dual}\mathit{Sym}$.

(A3′) $eFlip\,Dual = eDual\,Flip\,Sym$.

(A4′) $eDual\,Onext = e\,Onext\,Sym\,Dual$.

Where e is any directed, oriented edge, $eDual$ is directed from vertex $(eLeft)^*$ towards $(eRight)^*$. The orientation of $eDual$ is opposite that of e.

Dual can also be applied to an oriented vertex or polygon. Given oriented vertex v, $v\,Dual$ is the polygon v^* with orientation opposite that of v. Similarly, where p is an oriented polygon, vertex $pDual = p^*$ has orientation opposite that of p.

It is intuitively appealing to obtain a version of edge e^* whose orientation is the same as that of e. We would then not have to mentally reverse the orientation of the sphere in jumping from one open subdivision to its dual. To this end, we define the edge function *Rot* by

(A5′) $eRot = eFlip\,Dual = eDual\,Flip\,Sym$.

The edge $eRot$, called the rotated version of e, is directed from $(eRight)^*$ towards $(eLeft)^*$, and is oriented the same as e. Observe the following relations:

$$eRot^2 = eSym,$$

$$eRot^3 = eSym\,Rot = eRot^{-1},$$

$$eRot^4 = e.$$

In addition, vertex v and polygon $v\,Rot$ have the same orientation, as do polygon p and vertex $pRot$.

The edge functions are pictured in Figure 5.

For arbitrary edge e, edge-cycle $eLeft$ corresponds under duality to edge-cycle $eRot^{-1}\,Org$. The i^{th} edge of edge-cycle $eLeft$ is identical to the i^{th} edge of edge-cycle $eRot^{-1}\,Org$ to which Rot is applied: $eLnext^i = eRot^{-1}\,Onext^i\,Rot$. This is depicted in Figure 6.

The edge functions satisfy the following properties (as noted in [GS]):

(A1) $eRot^4 = e$.

(A2) $eRot\,Onext\,Rot\,Onext = e$.

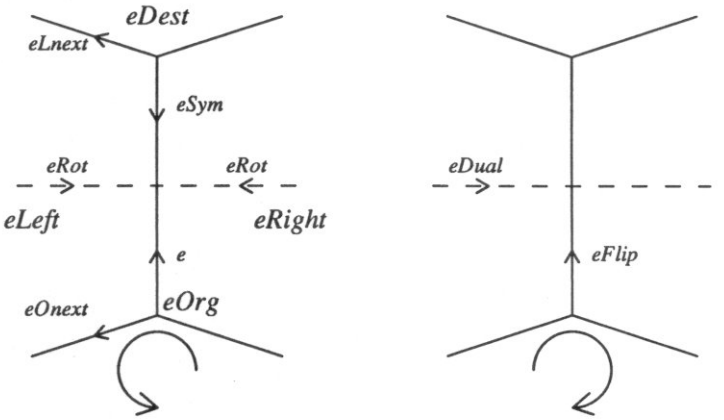


Fig. 5. The edge functions. The directed circular arc beneath each figure indicates orientation.

(A3) $eRot^2 \neq e$.

(A4) $e \in C$ iff $eRot \in C^*$.

(A5) $e \in C$ iff $eOnext \in C$.

(A6) $eFlip^2 = e$.

(A7) $eFlip\,Onext\,Flip\,Onext = e$.

(A8) $eFlip\,Onext^n \neq e$ for any n.

(A9) $eFlip\,Rot\,Flip\,Rot = e$.

(A10) $e \in C$ iff $eFlip \in C$.

It is worthwhile noting that the following properties are also satisfied:

(A11) $eDest = eSym\,Org$.

(A12) $eLeft = eRot^{-1}\,Org$.

(A13) $eRight = eRot\,Org$.

(A14) $eOnext^{-1} = eRot\,Onext\,Rot$.

(A15) $eLnext = eRot^{-1}\,Onext\,Rot = eSym\,Onext^{-1}$.

(A16) $eLnext^{-1} = eRot^{-1}\,Onext^{-1}\,Rot = eOnext\,Sym$.

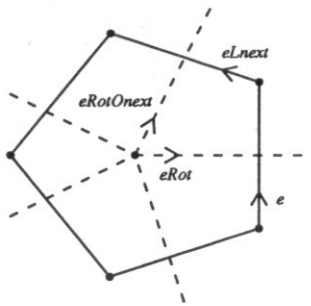


Fig. 6. The figure illustrates the relation between the edge-cycles $eLeft$ and $eRot^{-1}\,Org$.

We next define the edge functions *Vertexedge* and *Polygonedge*. Where v is an oriented vertex, $v\,Vertexedge$ denotes some edge e having the same orientation as v, and for which $v = e\,Org$. Where p is an oriented polygon, $p\,Polygonedge$ denotes some edge e having the same orientation as p, and for which $p = e\,Left$. Function *Polygonedge* can be defined in terms of *Vertexedge*:

(A17) $pPolygonedge = pRot\,VertexedgeRot$.

We observe that all traversal queries can be answered with use of the edge functions. Consider first queries that involve an oriented vertex v. To answer such queries, we first obtain some edge $e = v\,Vertexedge$ whose origin is v. Edge-cycle $eOrg = (e\ eOnext^1 \cdots eOnext^{n-1})$ can then be obtained via iterative use of *Onext*. The vertices adjacent to v are precisely the $eOnext^i\,Dest$, for $i = 0, \cdots, n-1$. (Vertex v itself occurs in the list if any of the edges $eOnext^i$ is a loop.) The cells incident to v are the edges $eOnext^i$ and the polygons $eOnext^i\,Left$, for $i = 0, \cdots, n-1$.

Traversal queries involving an oriented polygon are answered in an analogous manner. Edge functions *Polygonedge* and *Lnext* can be used to obtain the cycle of edges that bound the polygon, from which traversal queries can be answered.

We consider finally traversal queries that involve an edge e. The cells incident to e are its endpoints $eOrg$ and $eDest$, and its left and right polygons $eLeft$ and $eRight$. The basic edges adjacent to e are $eOnext$, $eFlip\,Onext$, $eSym\,Onext$ and $eFlip\,Sym\,Onext$.

3.4 Two Data Structures for Handling Open Subdivisions

3.4.1 The Winged-Edge Data Structure

The winged-edge data structure was introduced by Baumgart in [Ba]. It is designed to handle 2-dimensional subdivisions of orientable surfaces — spheres and n-tori. It is not intended to permit the handling of a subdivision's dual as well. The scheme for representing a subdivision described in the current section is that of Baumgart's. However, the set of queries whose implementation we present are not Baumgart's, but rather the edge functions of the previous section.

The edges of an open subdivision may be partitioned into groups of four. Each group consists of the four directed and oriented versions of a basic edge. We select a canonical representative in each group. The canonical representatives are constrained to all have the same orientation, say counter-clockwise. Any edge $e \in C$ can then be written as $\overline{e}\,Flip^f\,Sym^s$ where $f, s \in \{0, 1\}$, and $\overline{e}$ is the canonical representative of the group to which edge e belongs. We write $group(e)$ to denote this group.

Each group is represented in the data structure by an *edge-node*, each (unoriented) vertex by a *vertex-node*, and each (unoriented) polygon by a *polygon-node*. The edge-node is a structure consisting of fields e, v and p:

```
struct edge_node {
  struct edge_node *e[2, 2];
  struct vertex_node *v[2];
  struct polygon_node *p[2];
  };
```

Let edge-node n correspond to the group whose canonical representative is edge $\overline{e}$. Then edge $e = \overline{e}\,Flip^f\,Sym^s$ is represented by the triplet (n, f, s), called an *edge reference*. Component f of the edge reference has value 0 (1) iff e has counter-clockwise (clockwise) orientation. This is because $\overline{e}$ (like every canonical representative) was selected with

counter-clockwise orientation. Component s has value 0 (1) iff the direction of e agrees (disagrees) with the direction of $\overline{e}$.

Edge-node n stores values as follows. Element $n.e[f,s]$, which corresponds to edge $\overline{e}Flip^f Sym^s$, holds the address of the edge node which corresponds to $group(\overline{e}Flip^f Sym^s Onext)$. Element $n.v[s]$ holds the address of the vertex-node which corresponds to $\overline{e}Sym^s Org$. Element $n.p[f]$ holds the address of the polygon-node which corresponds to $\overline{e}Flip^f Left$. Edge-node n is depicted in Figure 7.

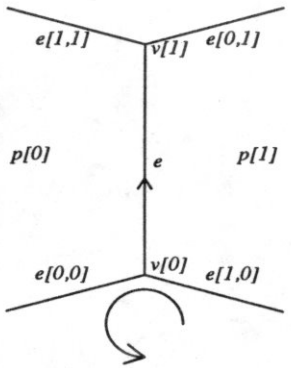


Fig. 7. This diagram indicates the edge groups, vertices and polygons referenced by the edge-node n, where n corresponds to the edge group with canonical representative $\overline{e}$. The circular arc indicates the orientation of $\overline{e}$.

Each vertex v of an open subdivision is represented in the data structure by a vertex-node n_v. The vertex-node n_v holds the address of some edge-node whose canonical representative is incident to v. Let $\overline{v}$ denote the version of v having counter-clockwise orientation. Then *oriented* vertex v is of the form $v = \overline{v}Flip^f$ where $f \in \{0,1\}$. Vertex v is represented by a pair (n_v, f), called a *vertex reference*. Component f has value 0 (1) iff v has counter-clockwise (clockwise) orientation.

Similarly, each polygon p is represented by a polygon-node n_p. The polygon-node n_p holds the address of an edge-node whose canonical representative is incident

to polygon p. Oriented polygon $p = \overline{p}Flip^f$ is represented by the *polygon reference* (n_p, f).

Each edge function is applied to an edge reference, vertex reference or polygon reference, and returns a new edge, vertex or polygon reference as appropriate. The functions are implemented in Figure 8a and Figure 8b.

We argue for the correctness of this implementation. *Sym* is correctly implemented since

$$\begin{aligned}(n,f,s)Sym &= \overline{e}Flip^f Sym^s Sym \\ &= \overline{e}Flip^f Sym^{s+1} \\ &= (n,f,s+1).\end{aligned}$$

Flip is similarly shown to be correctly implemented.

To treat *Onext*, we observe that

(i) element $n.v[s]$ addresses $\overline{e}Flip^f Sym^s Org$, and

(ii) element $n.p[f+s]$ addresses $\overline{e}Flip^f Sym^s Left$.

Observation (i) follows since $n.v[s]$ addresses $\overline{e}Sym^s Org$ by definition, and applying *Flip* to an edge does not change its origin. Observation (ii) follows since $n.p[f]$ addresses $\overline{e}Flip^f Left$ by definition, and applying *Sym* to an edge swaps its left and right polygons.

Let $\varepsilon = eOnext = \overline{e}Flip^f Sym^s Onext$ be given by edge reference (N', F', S'). Line 1 of procedure *Onext* sets $N = N'$ since $n.e[f,s]$ holds the address of the edge-node which corresponds to $group(\overline{e}Flip^f Sym^s Onext)$, by definition. Line 2 sets $F = F'$ since e and ε have the same orientation. Line 4 is performed iff ε is a segment. Its selection of S ensures that the relation $eOrg = \varepsilon Org$ is satisfied, by (i). Since ε is a segment, $\varepsilon Org \neq \varepsilon Dest$, so S is determined. Line 6 is performed iff ε is a loop. Its selection of S ensures that $eLeft \neq \varepsilon Left$ is satisfied, by (ii). Since ε is a loop, $\varepsilon Left \neq \varepsilon Right$, so S is determined. Hence *Onext* is correctly implemented. A similar argument is used to

$(n,f,s)Sym = (n,f,s+1)$

$(n_v,f)Sym = (n_v,f)$

$(n_p,f)Sym = (n_p,f)$

$(n,f,s)Flip = (n,f+1,s)$

$(n_v,f)Flip = (n_v,f+1)$

$(n_p,f)Flip = (n_p,f+1)$

```
Onext( (n,f,s) )
  {
  N ← n.e[f,s];                                          1
  F ← f;                                                 2
  if (N.v[0] ≠ N.v[1])                                   3
    select S ∈ {0,1} such that N.v[S] = n.v[s];          4
  else                                                   5
    select S ∈ {0,1} such that N.p[F+S] ≠ n.p[f+s];      6
  return( (N,F,S) );                                     7
  }

Onext^-1( (n,f,s) )
  {
  N ← n.e[f+1,s];
  F ← f;
  if (N.v[0] ≠ N.v[1])
    select S ∈ {0,1} such that N.v[S] = n.v[s];
  else
    select S ∈ {0,1} such that N.p[F+S+1] ≠ n.p[f+s+1];
  return( (N,F,S) );
  }
```

Fig. 8a. Implementation of some of the edge functions for the winged-edge data structure. Addition is computed modulo 2.

show correctness of $Onext^{-1}$.

Procedures *Lnext* and $Lnext^{-1}$ are correct by each of their definitions.

To treat procedure *Org*, consider the vertex reference (n_v,f) returned by $(n,f,s)Org$. The vertex-node component n_v is correct since element $n.v[s]$ of edge-

$(n,f,s)Lnext = (n,f,s)Sym\,Onext^{-1}$

$(n,f,s)Lnext^{-1} = (n,f,s)Onext\,Sym$

```
Org( (n,f,s) )
  {
  n_v ← n.v[s];
  return( (n_v,f) );
  }

Left( (n,f,s) )
  {
  n_p ← n.p[f];
  return( (n_p,f) );
  }

Vertexedge( (n_v,f) )
  {
  n ← *n_v;
  select s ∈ {0,1} such that n.v[s] = n_v;
  return( (n,f,s) );
  }

Polygonedge( (n_p,f) )
  {
  n ← *n_p;
  select s ∈ {0,1} such that n.p[s+f] = n_p;
  return( (n,f,s) );
  }
```

Fig. 8b. Implementation of some of the edge functions for the winged-edge data structure. The contents of address n_v is denoted $*n_v$.

node n addresses the vertex-node that corresponds to $\overline{e}Flip^f\,Sym^s\,Org$ for $f \in \{0,1\}$. Component f is correct since an edge and its origin have the same orientation. Procedure *Left* is shown to be correct in a similar fashion.

To show correctness of procedure *Vertexedge*, consider the components of edge reference (n,f,s) returned by the operation $v\,Vertexedge$. Edge-node n corresponds to a

canonical representative $\overline{e}$ one of whose endpoints is vertex v. Component s determines the direction of $\overline{e}$ so that its *origin* is v. Component f follows since the edge returned by the procedure should have the same orientation as v. Note that component s is selected non-deterministically if $\overline{e}$ is a loop — this is fine since *Vertexedge* should return *any* edge incident to v. Procedure *Polygonedge* is treated similarly.

3.4.2 The Quad-Edge Data Structure

The winged-edge structure is not ideal. Although the edge functions *Onext* and $Onext^{-1}$ are constant-time, they are not terribly efficient. Having obtained the address of an edge-node, each of the functions must examine vertices and/or polygons to select the correct edge from the group represented by the edge-node. The quad-edge structure avoids this by storing edge references, rather than edge-node addresses, in each of its edge-nodes. Its edge-nodes reference specific directed and oriented edges instead of groups of edges. (Another advantage in this is that it permits the quad-edge structure to handle subdivisions of non-orientable surfaces. But that is a different story.)

The winged-edge structure does in effect represent both an open subdivision *and* its dual. This is because it completely represents the primal open subdivision, so the dual is determined. However, the dual subdivision is represented by a different scheme than the primal. To extend each of the edge functions to accommodate both primal and dual subdivisions, two distinct procedures must be written — one to handle the primal and the other to handle the dual subdivision. The quad-edge structure, on the other hand, represents both subdivisions simultaneously by the same scheme. The implementation of the edge functions under this data structure is the same whether the primal or the dual subdivision is being queried.

The quad-edge structure was introduced by Guibas and Stolfi in [GS]. We present their structure, enhanced by the introduction of vertex nodes and vertex references to accommodate the functions *Org* and *Vertexedge*.

The edges of open subdivisions C and C^* may be partitioned into groups of eight. Each group consists of the four directed and oriented versions of a basic edge e, plus the four versions of dual edge e^*. A canonical representative $\overline{e}$ is selected arbitrarily from each group. Any edge $e \in C \cup C^*$ can then be written as $\overline{e}Rot^r Flip^f$ where $r \in \{0,1,2,3\}$, $f \in \{0,1\}$ and $\overline{e}$ is the canonical representative of the group to which edge e belongs.

Each group is represented in the data structure by an *edge record*. Let the group with canonical representative $\overline{e}$ be represented by edge record n. Edge $e = \overline{e}Rot^r Flip^f$ of the group is referenced by the triplet (n, r, f), called an *edge reference*. Edge record n is a four element array. Each element $n[r]$ of the array has two fields, *next* and *org*. Element $n[r]$ corresponds to edge $\overline{e}Rot^r$. Field $n[r].next$ holds the edge reference to $\overline{e}Rot^r Onext$. Four groups correspond to the four basic edges adjacent to e; each element of edge record n references one of these groups. This is illustrated by Figure 9. (Field *org* will be described shortly.)

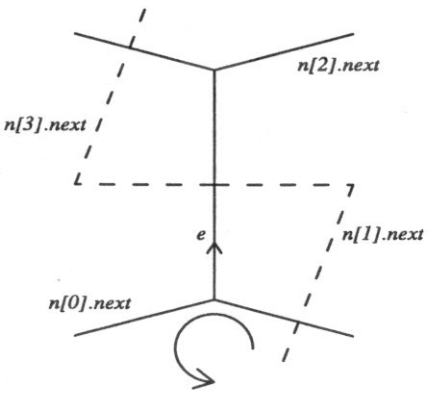


Fig. 9. The figure shows the edge groups referenced by edge record n, where n corresponds to the group with canonical representative $\overline{e}$.

Given an arbitrary edge reference (n, r, f), the edge functions *Rot*, *Flip* and *Onext*

are implemented as follows:

$$(n,r,f)Rot = (n,r+1+2f,f),$$

$$(n,r,f)Flip = (n,r,f+1),$$

$$(n,r,f)Onext = (n[r+f].next)Rot^f Flip^f,$$

where the r and f components are computed modulo 4 and 2, respectively. We observe that

$$(n,r,0)Onext = n[r].next,$$

and

$$\begin{aligned}(n,r,1)Onext &= (n[r+1].next)RotFlip \\ &= (n,r+1,0)OnextRotFlip \\ &= (n,r,0)RotOnextRotFlip \\ &= (n,r,0)Onext^{-1}Flip.\end{aligned}$$

The last equation indicates that moving forward around a vertex under one orientation is the same as moving backwards around the vertex under the opposite orientation.

Each vertex v of C or C^* is represented by a vertex-node n_v. Node n_v holds an edge reference to some edge having origin v and counter-clockwise orientation. An oriented vertex or polygon is of the form $\overline{v}Rot^r Flip^f$ where $r,f \in \{0,1\}$. Let the vertex or polygon be represented by the tuple (n_v,r,f), called a vertex reference. Component r has value 0 (1) iff the cell being referenced is a vertex (polygon). Component f has value 0 (1) iff the cell being referenced has counter-clockwise (clockwise) orientation.

In edge-node n (which corresponds to the edge $\overline{e}Rot^r$), field $n[r].org$ holds the vertex reference to $\overline{e}Rot^r Org$. Edge functions *Org* and *Vertexedge*, and *Rot* and *Flip* applied to vertex references, are implemented as follows:

$$(n,r,f)Org = (n[r].org)Flip^f,$$

$$(n_v,0,f)Vertexedge = (*n_v)Flip^f,$$

$$(n_v,r,f)Rot = (n_v,r+1,f),$$

$$(n_v,r,f)Flip = (n_v,r,f+1),$$

where addition is computed modulo 2. By the (A) relations of Section 3.3, which characterize the properties of edge functions, the remaining edge functions are easily implemented in terms of *Rot*, *Flip*, *Onext*, *Org* and *Vertexedge*.

The correctness of the above implementation is confirmed by the following derivations, where edge record n corresponds to $group(\overline{e})$, and vertex-node n_v to vertex v.

$$\begin{aligned}(n,r,0)Rot &= \overline{e}Rot^r Rot \\ &= \overline{e}Rot^{r+1} \\ &= (n,r+1,0). \\ (n,r,1)Rot &= \overline{e}Rot^r FlipRot \\ &= \overline{e}Rot^r Rot^{-1}Flip && (A9) \\ &= \overline{e}Rot^{r+3}Flip && (A1) \\ &= (n,r+3,1).\end{aligned}$$

$$\begin{aligned}(n,r,f)Flip &= \overline{e}Rot^r Flip^f Flip \\ &= \overline{e}Rot^r Flip^{f+1} \\ &= (n,r,f+1).\end{aligned}$$

$$\begin{aligned}(n,r,0)Onext &= n[r].next. \\ (n,r,1)Onext &= \overline{e}Rot^r FlipOnext \\ &= \overline{e}Rot^r RotOnextRotFlip && (A2,7) \\ &= \overline{e}Rot^{r+1}OnextRotFlip \\ &= (n[r+1].next)RotFlip.\end{aligned}$$

$$\begin{aligned}(n,r,f)Org &= \overline{e}Rot^r Flip^f Org \\ &= \overline{e}Rot^r OrgFlip^f\end{aligned}$$

$$
\begin{aligned}
&= (n[r].org)\,Flip^f.\\
(n_v,0,f)\,Vertexedge &= (n_v,0,0)\,Vertexedge\,Flip^f\\
&= (*n_v)\,Flip^f.\\
(n_v,r,0)Rot &= vRot^r\,Rot\\
&= vRot^{r+1}\\
&= (n_v,r+1,0).\\
(n_v,r,1)Rot &= vRot^r\,Flip\,Rot\\
&= vRot^r\,Rot^{-1}\,Flip\\
&= vRot^{r-1}\,Flip\\
&= (n_v,r+1,1).\\
(n_v,r,f)Flip &= vRot^r\,Flip^f\,Flip\\
&= vRot^r\,Flip^{f+1}\\
&= (n_v,r,f+1).
\end{aligned}
\qquad (A9)
$$

Chapter 4

Functions for Manipulating Polyhedral Subdivisions

4.1 Introduction

We now turn to a discussion of the facet-edge structure for handling polyhedral subdivisions. The facet-edge structure is similar to the quad-edge structure, one dimension higher. The quad-edge structure considers an edge as its atom. The edge connects two vertices and two polygons, and is adjacent to four edges. The atom upon which the facet-edge structure operates is the facet-edge pair, which consists of a facet f and an edge e of the facet's boundary. The facet-edge pair connects two vertices (the endpoints of edge e) and two polyhedra (incident to facet f). The facet-edge pair is adjacent to four facet-edge pairs: two consist of e and one of the two facets adjacent to f along e, and two others consist of f and one of the two edges adjacent to e along f.

In the current section, we describe the facet-edge functions. These are used to formulate queries concerning polyhedral subdivisions, and are analogous to the edge functions of the previous chapter. There are two types of facet-edge functions: traversal functions and vertex functions.

In Sections 4.2 through 4.4, we present the five *traversal functions* ***Fnext***, ***Enext***, ***Spin***, ***Clock*** and ***Sdual***. We call these *traversal* functions because they provide the means of traversing or moving about the cells of a polyhedral subdivision. The first two traversal functions are used to move from cell to adjacent cell. ***Spin*** and ***Clock*** are used to change a local sense of direction, so ***Fnext*** and ***Enext*** know the direction in which each is to traverse. The function ***Sdual*** is used to move between a subdivision and its dual.

In Section 4.5, we present the *vertex functions* ***Org***, ***Dest***, ***Ppos*** and ***Pneg***, with which one can determine the two vertices incident to an edge (its endpoints), and the

two polyhedra incident to a facet. They are called *vertex* functions since each returns a reference either to a vertex, or to a polyhedron whose dual is a vertex.

4.2 Basic Traversal Functions

Let f be a facet of polyhedral subdivision C. Facet f is bounded by a ring of edges $e^0 \ldots e^{n-1}$ where edges e^i and e^{i+1} are adjacent (addition modulo n). We call this ring, denoted $\mathcal{E}_f$, the *edge-ring* of facet f. $\mathcal{E}_f$ can be assigned either of two senses of rotation whereby we can distinguish between the two edges belonging to the ring that are adjacent to edge e^i. We write $\mathcal{E}_f = (e^0 \ldots e^{n-1})$ to indicate the edge-ring whose sense of rotation proceeds from edge e^0 to e^1 to e^2, and so forth.

Similarly we define the *facet-ring* of edge e, denoted $\mathcal{F}_e$, to be the ring of facets $f^0 \cdots f^{m-1}$ incident to e. The facet-ring too can have either of two senses of rotation. We write $\mathcal{F}_e = (f^0 \cdots f^{m-1})$ to indicate facet-ring $\mathcal{F}_e$ whose sense of rotation proceeds from facet f^0 to f^1 to f^2, and so on.

The atomic unit on which queries are formulated is called a *facet-edge pair*. This is a pair a consisting of a facet f and an edge e, such that f and e are incident. The *edge component* of a is denoted $edge(a)$, and the *facet component* of a is denoted $facet(a)$. The facet-edge pair a determines two rings in C, these being edge-ring $\mathcal{E}_{facet(a)}$ and facet-ring $\mathcal{F}_{edge(a)}$. There are four versions of a which derive from the two senses of rotation that each of its two rings can assume. By the phrase 'facet-edge pair' we mean one such version — each of the two rings determined by the facet-edge pair has a specified sense of rotation. We will use the phrase 'basic facet-edge pair' to refer to a facet-edge pair whose rings have unspecified sense of rotation. Edge-ring $\mathcal{E}_a$ denotes the edge-ring $\mathcal{E}_{facet(a)}$ whose sense of rotation is determined by a; the facet-ring $\mathcal{F}_a$ is similarly defined.

Facet-edge pair a is said to be *oriented* if its facet component $facet(a)$ is oriented. Where a is oriented, the sense of rotation in the edge-ring $\mathcal{E}_a$ is always taken to agree with the orientation of a.

Given an oriented facet-edge pair a, the edge component $edge(a)$ is directed to agree with the orientation of a. That is, $edge(a)$ is directed so that the sense of rotation in $facet(a)$ passes $edge(a)$ from origin towards destination. We define the *origin vertex* of a, denoted $aOrg$, to be the origin of directed $edge(a)$. Similarly the *destination vertex* of a, denoted $aDest$, is defined to be the destination of directed $edge(a)$. In this manner we are able to distinguish between the two vertices incident with $edge(a)$.

We also wish to distinguish between the two polyhedra incident to $facet(a)$. In order to do this, we say that oriented facet-edge pair a has *spin* if its (directed) edge component $edge(a)$ has spin. The facet-edge pair's specific spin (that is, left-handed or right-handed) is simply its edge component's spin. Let a be an oriented, spun facet-edge pair. Where the spin of a is right-handed (left-handed), we define H_a^+ to be that open half-space determined by the plane $aff\ facet(a)$ from which the orientation of a appears counter-clockwise (clockwise). We then define the *positive polyhedron* of a, denoted $aPpos$, to be that polyhedron p of C incident to $facet(a)$ for which points of the interior of p arbitrarily close to the relative interior of $facet(a)$ lie in H_a^+. The *negative polyhedron* $aPneg$ of a is the other polyhedron of C incident to $facet(a)$. The sense of rotation in the facet-ring $\mathcal{F}_a$ passes from $aPneg$ towards $aPpos$. Figure 10 illustrates some of the notions presented so far in this section. (For convenience, we have assumed that $facet(a)$ lies in a plane. This is an unnecessary assumption that we later drop.)

We are now able to define the traversal functions *Fnext*, *Enext*, *Spin* and *Clock*. Each is applied to some facet-edge pair and returns a new facet-edge pair.

Fnext is defined by $a' = aFnext$ where $edge(a') = edge(a)$ and $facet(a')$ follows $facet(a)$ in the facet-ring $\mathcal{F}_a$. The sense of rotation in the rings of a' are such that $\mathcal{F}_{a'} = \mathcal{F}_a$ and $a'Org = aOrg$. In particular, a and a' have the same spin.

Enext is defined by $a' = aEnext$ where $facet(a') = facet(a)$ and $edge(a')$ follows $edge(a)$ in the edge-ring $\mathcal{E}_a$. The rings of a' are directed so that $\mathcal{E}_{a'} = \mathcal{E}_a$ and $a'Ppos = aPpos$. Facet-edge pairs a and a' have the same orientation.

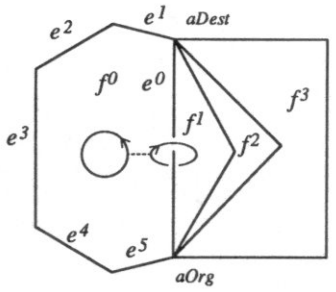


Fig. 10. This "handcuff diagram" depicts a region of a subdivision. The handcuff represents facet-edge pair a. Its circular loop indicates the orientation of facet component $facet(a) = f^0$, while its elliptical loop indicates the spin of edge component $edge(a) = e^0$. Here $\mathcal{F}_a = (f^0 \ldots f^3)$ and $\mathcal{E}_a = (e^0 \ldots e^5)$. Polyhedron $aPpos$ lies above the page and contains facets f^0 and f^1, while $aPneg$ lies behind the page and contains f^0 and f^3.

Spin is defined by $a' = aSpin$ where a' and a are different versions of the same basic facet-edge pair for which the sense of rotation in $\mathcal{F}_{a'}$ is opposite that of $\mathcal{F}_a$, and the senses of rotation in $\mathcal{E}_{a'}$ and $\mathcal{E}_a$ are the same.

Clock is defined by $a' = aClock$ where a' and a are different versions of the same basic facet-edge pair, for which the senses of rotation in $\mathcal{E}_{a'}$ and $\mathcal{F}_{a'}$ are opposite those in $\mathcal{E}_a$ and $\mathcal{F}_a$, respectively.

Figure 11 illustrates these various traversal functions. Traversal functions *Spin* and *Clock* can be viewed as follows. Let a be an oriented, spun facet-edge pair. The effect of *Spin* is to reverse spin. This reverses the sense of rotation in the facet-ring. The effect of *Clock* is to reverse orientation. This reverses the direction of $edge(a)$, resulting in reversal of the sense of rotation in both facet- and edge-rings. Each of the four versions of a facet-edge pair has unique orientation and spin. Its orientation and spin are used as handles to manipulate the sense of rotation in its two rings.

Traversal functions *Fnext* and *Enext* enable us to move from facet-edge pair to adjacent facet-edge pair. Facet-edge pairs a and b are said to be adjacent if either

(*i*) $facet(a)$ and $facet(b)$ are adjacent in the facet-ring $\mathcal{F}_a (= \mathcal{F}_b)$, or

(*ii*) $edge(a)$ and $edge(b)$ are adjacent in the edge-ring $\mathcal{E}_a (= \mathcal{E}_b)$.

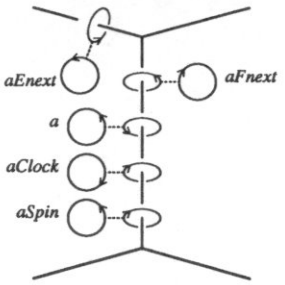


Fig. 11. This handcuff diagram depicts the four functions *Clock*, *Spin*, *Fnext* and *Enext*. The region pictured is a winged-edge, consisting of five edges and (part of) two facets (to the left and right of the vertical edge). We assume these two facets to belong to a common polyhedron that lies behind the plane of the page, this being a$Ppos$.

The facet-edge pair a is adjacent to the facet-edge pairs $aFnext$, $aFnext^{-1}$, $aEnext$ and $aEnext^{-1}$.

The following relations hold among the traversal functions.

(B1) $aSpin^2 = a$.

(B2) $aClock^2 = a$.

(B3) $a(Spin\,Clock)^2 = a$.

(B4) $aFnext^{-1} = aClock\,Fnext\,Clock$.

(B5) $aFnext^{-1} = aSpin\,Fnext\,Spin$.

(B6) $aEnext^{-1} = aClock\,Enext\,Clock$.

(B7) $aEnext^{-1} = aClock\,Spin\,Enext\,Clock\,Spin$.

(B8) $aClock\,Fnext^i \neq a$ for any i.

(B9) $aSpin\,Fnext^i \neq a$ for any i.

(B10) $aSpin\,Enext^i \neq a$ for any i.

(B11) $aClock\,Enext^i \neq a$ for any i.

(B12) $a \in C$ iff $aFnext \in C$.

(B13) $a \in C$ iff $aClock \in C$.

(B14) $a \in C$ iff $aSpin \in C$.

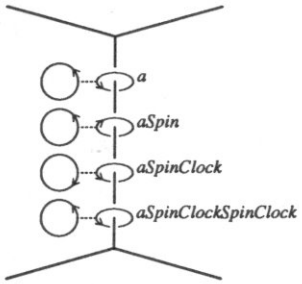


Fig 12. This handcuff diagram confirms relation (B3), that $a(Spin\,Clock)^2 = a$.

Relations (B1–11) can be verified with the use of handcuff diagrams. This is done for relation (B3) in Figure 12. Relations (B12–14) forbid the basic traversal functions from jumping between a polyhedral subdivision and its dual. We attempt no formal derivation of the (B) relations, but adopt them based upon their intuitive appeal.

Observe that edge-rings $\mathcal{E}_{aSpin}$ and $\mathcal{E}_a$ have the same sense of rotation, and that facet-rings $\mathcal{F}_{aSpin}$ and $\mathcal{F}_a$ have opposite senses of rotation. The former assertion follows from

$$\begin{aligned} aEnextSpin &= aClockSpinEnext^{-1}ClockSpinSpin && (B1,2,7)\\ &= aSpinClockEnext^{-1}Clock && (B1,3)\\ &= aSpinEnext, && (B2,6) \end{aligned}$$

while the latter assertion follows from

$$aFnextSpin = aSpinFnext^{-1}. \qquad (B1,5)$$

Symmetrically, facet-rings $\mathcal{F}_{aClockSpin}$ and $\mathcal{F}_a$ have the same sense of rotation, while edge-rings $\mathcal{E}_{aClockSpin}$ and $\mathcal{E}_a$ have opposite senses of rotation. The former statement follows from

$$aFnextClockSpin = aClockSpinFnext$$

which is easily derived, the latter assertion from

$$aEnextClockSpin = aClockSpinEnext^{-1}.$$

Thus *Spin* and *ClockSpin* play symmetric roles: the former reverses the sense of rotation in a facet-ring, while the latter reverses the sense of rotation in an edge-ring. Note in addition that $aSpinClock = aClockSpin$, since

$$\begin{aligned} aSpinClockSpin^{-1}Clock^{-1} &= aSpinClockSpinClock && (B1,2)\\ &= a(SpinClock)^2\\ &= a. && (B3) \end{aligned}$$

4.3 Duality

The traversal function *Sdual* (which stands for Space DUAL) is applied to a facet-edge pair a of polyhedral subdivision C, and returns a second facet-edge pair $aSdual$ belonging to C^*. The edge component of $aSdual$ is $edge(aSdual) = (facet(a))^*$, and its facet component is $facet(aSdual) = (edge(a))^*$. In order to define the particular version of $aSdual$ — that is, the sense of rotation in its two rings — we first extend the notion of duality to facet- and edge-rings.

Let $\mathcal{E}_a = (e^0 \ldots e^{n-1})$ be an edge-ring of C. The dual of $\mathcal{E}_a$ is the facet-ring $(\mathcal{E}_a)^* = (e^{0^*} \ldots e^{n-1^*})$ of C^*. The dual of a facet-ring is similarly defined. The sense of rotation in the facet- and edge-ring of $aSdual$ are then defined so that

$\mathcal{E}_{aSdual} = (\mathcal{F}_a)^*$, and

$\mathcal{F}_{aSdual} = (\mathcal{E}_a)^*$.

The first formula implies that to each facet-ring there corresponds an edge ring under duality, and both have the same sense of rotation. Symmetrically, the second formula implies that to each edge-ring there corresponds a dual facet-ring having the same sense of rotation.

Facet-edge pairs a and $aSdual$ have the same spin. More precisely, if $aSdual$ were transformed in R^3 so that $edge(a)$ and $edge(aSdual)$ were to coïncide in direction as well as location, then the two edges would have the same spin.

The relation between a and $aSdual$ can be grasped by imagining the two facet-edge pairs superimposed, $edge(a)$ piercing $facet(aSdual)$ orthogonally, and $facet(a)$ pierced by $edge(aSdual)$ orthogonally. Edge $edge(aSdual)$ is directed from $aPneg$ towards $aPpos$. Facet-ring $\mathcal{F}_{aSdual}$ moves from $aOrg$ towards $aDest$, so $aDest$ is the space-dual of $aSdualPpos$. This is depicted in Figure 13.

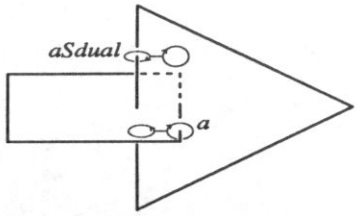


Fig. 13. This diagram depicts the relation between facet-edge pairs a and $aSdual$. Facet $facet(a)$ is a square which protrudes from the page, and so appears foreshortened; $facet(aSdual)$ is the triangle flush with the page.

The following relations hold between $Sdual$ and the other traversal functions:

(B15) $aSdual^2 = a$.

(B16) $aClockSdual = aSdualClock$.

(B17) $aSpinSdual = aSdualClockSpin$.

(B18) $aEnext = aSdualFnextSdual$.

(B19) $a \in C$ iff $aSdual \in C^*$.

Relation (B17) indicates that reversal of the sense of rotation in $\mathcal{F}_a$ corresponds to reversal in $\mathcal{E}_{aSdual}$. The symmetric relation

$$aSpinClockSdual = aSdualSpin$$

is easily derived, implying that reversal of the sense of rotation in $\mathcal{E}_a$ corresponds to reversal of the sense of rotation in $\mathcal{F}_{aSdual}$. Note furthermore that the relation $aFnext = aSdualEnextSdual$ holds, since

$$\begin{aligned} aFnext &= aSdualSdualFnextSdualSdual && (B15) \\ &= aSdualEnextSdual. && (B18) \end{aligned}$$

Observe finally that relation (B18) defines $Enext$ in terms of $Fnext$ and $Sdual$, so maintaining the facet-rings in both C and C^*, together with the ability to move between C and C^* in a well defined manner, effectively represents the $edge$-rings of both subdivisions. The implementation of the facet-edge structure, presented in the next chapter, exploits this insight.

4.4 A Combinatorial Formulation of the Traversal Functions

In this section we briefly redefine the traversal functions solely in combinatorial terms. We altogether ignore geometric assumptions (such as assuming each facet to lie in some plane). This reformulation will help us understand the data structure to be presented in the next chapter. We first introduce notation for describing adjacent cells.

Let c^k represent a k-cell of a (closed) polyhedral subdivision. We then define $adj(c^k, c^{k+1}, c^{k-1})$ to denote the unique k-cell adjacent to c^k, which share cells c^{k+1} and c^{k-1} with c^k. For shorthand, we drop argument c^{k+1} if $k = 3$; similarly we drop argument c^{k-1} if $k = 0$. We assume the arguments to adj are reasonable. For example, $adj(e, v, f)$ denotes the edge e' adjacent to edge e, where e and e' share the common vertex v, and belong to the common facet f. The expression is not defined if v is not an endpoint of e, or if e does not lie in the boundary of f. The expression $adj(v, e)$ denotes the endpoint of edge e other than v.

Let a be a facet-edge pair. We will represent a by a triplet $[a, v, p]$. Component v is one of the endpoints of $edge(a)$, and component p one of the polyhedra incident to $facet(a)$. (Intuitively, component $v = aDest$ and component $p = aPpos$.) Where a is so represented, $Spin$ and $Clock$ are defined as follows:

$$[a, v, p]Spin = [a, v, adj(p, facet(a))], \text{ and}$$

$$[a, v, p]Clock = [a, adj(v, edge(a)), adj(p, facet(a))].$$

We introduce a new function Dir that is symmetric to $Spin$:

$[a, v, p]Dir = [a, adj(v, edge(a)), p]$.

Note that $aDir = aClockSpin = aSpinClock$.

The function *Sdual* takes the dual of a, and swaps the roles of the vertex and polyhedron components of $[a, v, p]$:

$[a, v, p]Sdual = [a^*, p^*, v^*]$.

To present functions *Fnext*, $Fnext^{-1}$, *Enext* and $Enext^{-1}$, we first introduce functions $Next_f$ and $Next_e$. $Next_f$ replaces component a of $[a, v, p]$ by a' where $facet(a)$ and $facet(a')$ are adjacent. $Next_e$ replaces a by a' where $edge(a)$ and $edge(a')$ are adjacent.

$[a, v, p]Next_f = [a', v, p]$ where

$facet(a') = adj(facet(a), edge(a), p)$, and

$edge(a') = edge(a)$.

$[a, v, p]Next_e = [a'', v, p]$ where

$facet(a'') = facet(a)$, and

$edge(a'') = adj(edge(a), v, facet(a))$.

Note that $[a, v, f]Next_f$ denotes the facet-edge pair whose facet component is adjacent to $facet(a)$ along $edge(a)$ and polyhedron p.

The remaining traversal functions are then defined as follows:

$[a, v, p]Fnext = [a, v, p]Next_f Spin$.

$[a, v, p]Fnext^{-1} = [a, v, p]SpinNext_f$.

$[a, v, p]Enext = [a, v, p]Next_e Dir$.

$[a, v, p]Enext^{-1} = [a, v, p]DirNext_e$.

4.5 The Vertex Functions

It is worthwhile to represent every vertex of C and C^* by the class of facet-edge pairs for which the vertex serves as origin. Where facet-edge pair a belongs to C, the origin and destination of a are represented by two such classes corresponding to vertices of C. The positive and negative polyhedra of a are represented by two classes corresponding to vertices of C^*.

We define an *origin partition* to be a partition of the set of facet-edge pairs comprising C and C^*. The vertex $aOrg$ is represented by the equivalence class consisting of all facet-edge pairs whose origin is $aOrg$. Each class is named by the name of the vertex it represents. Each class generally has numerous names since each vertex goes by many names; for instance, vertex $v = aOrg$ is named by $bOrg$ for each facet-edge pair b whose origin is v.

Dest, *Ppos* and *Pneg* are intended to provide even more names for the classes of the origin partition. Formally, these additional names are superfluous; however, they coincide with our notions of destination vertex, and positive and negative polyhedra, and so are useful. For instance, $aDest$ corresponds to the destination vertex of the directed $edge(a)$, and is in some contexts more suggestive than $aClockOrg$ (another name for the same class).

The classes of the origin partition are related as follows.

(B20) $aOrg$, $aDest$, $aPpos$ and $aPneg$ are all distinct.

(B21) $aSpinOrg = aOrg$.

(B22) $aClockOrg = aDest$.

(B23) $aSpinClockOrg = aDest$.

(B24) $aFnext^i Org = aOrg$ for all integer i.

(B25) $aSpinPpos = aPneg$.

(B26) $aClockPpos = aPneg$.

(B27) $aSpinClockPpos = aPpos$.

(B28) $aEnext^i Ppos = aPpos$ for all integer i.

(B29) $aSdualOrg = aPneg$.

(B30) $aSdualDest = aPpos$.

Chapter 5

The Facet-Edge Data Structure

5.1 Introduction

In the current chapter we present the facet-edge data structure. The scheme for representing a polyhedral subdivision and simultaneously its dual is described. The implementation of the facet-edge functions is presented.

The atomic unit upon which the facet-edge structure is based is the facet-edge pair. The role it plays is similar to the role played by edges in the quad-edge structure for handling 2-dimensional open subdivisions. The similarity is sufficiently deep that 3-dimensional subdivisions can be represented by a data structure that closely resembles the quad-edge structure.

Under the quad-edge scheme, the edges of a 2-dimensional subdivision C_2 and its dual C_2^* may be partitioned into groups of eight of the form $eRot^r Flip^f$ where $r \in \{0,1,2,3\}$ and $f \in \{0,1\}$. Similarly, the facet-edge pairs comprising the polyhedral subdivisions C_3 and its dual C_3^* may be partitioned into groups of eight. Where a is any facet-edge pair of C_3 or C_3^*, a belongs to the group $\{aSdual^d Clock^c Spin^s \mid d,c,s \in \{0,1\}\}$. These groups partition the collection of facet-edge pairs that comprise C_3 and C_3^*. The similarity suggests the use of *facet-edge nodes*, one per group, in our data structure.

Each edge e of C_2 or C_2^* is adjacent to four other edges, these being $eOnext$, $eOnext^{-1}$, $eSymOnext$ and $eSymOnext^{-1}$. Similarly, each facet-edge pair of C_3 or C_3^* is adjacent to the facet-edge pairs $aFnext$, $aFnext^{-1}$, $aEnext$ and $aEnext^{-1}$. This suggests that, in the facet-edge node associated with a, we store references to (some version of) each of the four facet-edge pairs adjacent to a.

Since the facet-edge structure resembles the quad-edge structure, it is helpful to define a facet-edge function analogous to the edge function *Rot*. In Section 5.2, we present facet-edge function *Srot*, which plays this role. In Section 5.3, we present the scheme for representing a polyhedral subdivision and its dual, and an implementation for the facet-edge traversal functions. In Section 5.4 we prove correctness of implementation of the traversal functions. In Section 5.5, we discuss the implementation of the vertex functions.

5.2 Traversal Function Srot

In this section, we introduce the traversal function *Srot* (for Space ROTation), which is defined in terms of the facet-edge functions that have been presented so far. *Srot* is defined by

$$aSrot = aSdualSpin = aSpinClockSdual.$$

Facet-edge pair $aSrot$ is called the rotated version of a. Its edge component is directed from $aPneg$ towards $aPpos$, and its spin is opposite the spin of a. Observe the following relations:

$$\begin{aligned} aSrot^2 &= aSdualSpinSpinClockSdual \\ &= aClock, && (B1,15,16) \\ aSrot^3 &= aClockSrot \\ &= aClockSpinClockSdual \\ &= aSpinSdual && (B1,2,3) \\ &= aSrot^{-1}, \\ aSrot^4 &= a. \end{aligned}$$

Srot plays a significant role in the facet-edge structure. Given facet-edge pair a, the two facet-edge pairs of $\mathcal{F}_a$ adjacent to a are of the form

$$aSrot^0 Fnext,$$

$$aSrot^2\,Fnext = aClock\,Fnext = aFnext^{-1}\,Clock.$$

The two facet-edge pairs of $\mathcal{E}_a$ adjacent to a are of the form

$$aSrot^1\,Fnext Srot = aClock\,Enext = aEnext^{-1}\,Clock,$$

$$aSrot^3\,Fnext Srot = aEnext.$$

By associating with a the facet-edge pairs $aSrot^r\,Fnext$ for $r = 0, 1, 2, 3$, we can obtain the four facet-edge pairs adjacent to a (assuming we can move easily between a subdivision and its dual). By storing the $aSrot^r\,Fnext$ in the facet-edge node associated with a, the four facet-edge pairs adjacent to a are available in constant time.

5.3 Implementation of the Traversal Functions

Polyhedral subdivision C (and simultaneously subdivision C^*) are represented by the facet-edge data structure. The directed, spun facet-edge pairs comprising C and C^* may be partitioned into groups of eight. Where facet-edge pair a is an arbitrary member of some group, the facet-edge pairs of the group are of the form $aSrot^r\,Spin^s$ where $r \in \{0, 1, 2, 3\}$ and $s \in \{0, 1\}$. An arbitrary member $\overline{a}$ of each group is designated the *canonical representative* of the group.

A group is represented by a *facet-edge node* n, an array consisting of elements $n[0]$ through $n[3]$. Element $n[r]$ corresponds to the facet-edge pair $\overline{a}Srot^r$. The facet-edge pair $\overline{a}Srot^r\,Spin^s$ is represented by the triplet (n, r, s), where $r \in \{0, 1, 2, 3\}$ and $s \in \{0, 1\}$. Such a triplet is called a *facet-edge reference*. The facet-edge reference can be viewed as a pointer to the array element $n[r]$, plus a bit s indicating whether *Spin* is to be applied to the facet-edge pair $\overline{a}Srot^r$ which corresponds to $n[r]$.

Each element $n[r]$ of the facet-edge node contains two fields, *data* and *next*. Field *data* is used to hold application-dependent information corresponding to $\overline{a}Srot^r$ such as geometry, and need not concern us. Field *next* contains a facet-edge reference to $\overline{a}Srot^r\,Fnext$. Given arbitrary facet-edge reference (n, r, s), the functions *Srot*, *Spin* and *Fnext* are given by the formulas

$$(n, r, s)Srot = (n, r + 1 + 2s, s),$$

$$(n, r, s)Spin = (n, r, s + 1),$$

$$(n, r, s)Fnext = (n[r + 2s].next)Srot^{2s}\,Spin^s,$$

where the r and s components are computed modulo 4 and 2, respectively.

Observe that in the third formula, we have

$$(n, r, 0)Fnext = (n[r].next),$$

and

$$\begin{aligned}(n, r, 1)Fnext &= (n[r + 2].next)Srot^2\,Spin \\ &= (n, r + 2, 0)Fnext Srot^2\,Spin \\ &= (n, r + 2, 0)Fnext\,Clock\,Spin \\ &= (n, r, 0)Clock\,Fnext\,Clock\,Spin \\ &= (n, r, 0)Fnext^{-1}\,Spin.\end{aligned}$$

The last equation implies that moving forward around a facet-ring is the same as moving backward around the same ring given with reverse sense of rotation. (We shall see shortly that our use of the relation $(n, r + 2, 0) = (n, r, 0)Clock$ is justified.)

The remaining traversal functions are defined in terms of *Srot*, *Spin* and *Fnext* as follows. Observe that all traversal queries take constant time.

$$\begin{aligned}(n, r, s)Clock &= (n, r, s)Srot^2 \\ &= (n, r + 1 + 2s, s)Srot \\ &= (n, (r + 1 + 2s) + 1 + 2s, s) \\ &= (n, r + 2, s).\end{aligned}$$

$$(n, r, s)Fnext^{-1} = (n, r, s)Clock\,Fnext\,Clock.$$

$$\begin{aligned}(n, r, s)Sdual &= (n, r, s)Srot\,Spin \\ &= (n, r + 1 + 2s, s + 1).\end{aligned}$$

$$(n, r, s)Enext = (n, r, s)Sdual\,Fnext\,Sdual.$$

$$(n, r, s)Enext^{-1} = (n, r, s)Clock\,Enext\,Clock.$$

5.4 Proof of Implementation Correctness

In this section we show that relations (B1-7) and (B15-18) of Chapter 4 are satisfied by the implementation described in the previous section. The remaining traversal relations are not ensured by this implementation. It is up to the application, and its use of the facet-edge construction operators, to guarantee that *these* relations are met.

$(B1)$ $$\begin{aligned}(n,r,s)Spin^2 &= (n,r,s+1)Spin\\ &= (n,r,s).\end{aligned}$$

$(B2)$ $$\begin{aligned}(n,r,s)Clock^2 &= (n,r+2,s)Clock\\ &= (n,r,s).\end{aligned}$$

$(B3)$ $$\begin{aligned}(n,r,s)(Spin\,Clock)^2 &= (n,r,s+1)Clock\,Spin\,Clock\\ &= (n,r+2,s+1)Spin\,Clock\\ &= (n,r+2,s)Clock\\ &= (n,r,s).\end{aligned}$$

$(B4)$ $$(n,r,s)Fnext^{-1} = (n,r,s)Clock\,Fnext\,Clock. \qquad \text{(by definition)}$$

$(B5)$ $$\begin{aligned}(n,r,s)Fnext^{-1} &= (n,r,s)Clock\,Fnext\,Clock\\ &= (n,r+2,s)Fnext\,Clock\\ &= (n[r+2+2s].next)Clock^s\,Spin^s\,Clock\\ &= (n[r+2(s+1)].next)Clock^{s+1}\,Spin^{s+1}\,Spin\\ &= (n,r,s+1)Fnext\,Spin\\ &= (n,r,s)Spin\,Fnext\,Spin.\end{aligned}$$

$(B6)$ $$(n,r,s)Enext^{-1} = (n,r,s)Clock\,Enext\,Clock. \qquad \text{(by definition)}$$

$(B7)$ $$\begin{aligned}(n,r,s)Enext^{-1} &= (n,r,s)Clock\,Enext\,Clock\\ &= (n,r,s)Clock\,Sdual\,Fnext\,Sdual\,Clock\\ &= (n,r+2,s)Sdual\,Fnext\,Sdual\,Clock\\ &= (n,r+3+2s,s+1)Fnext\,Sdual\,Clock\\ &= (n[r+3+2s+2(s+1)].next)\\ &\qquad Clock^{s+1}\,Spin^{s+1}\,Sdual\,Clock\\ &= (n[r+1].next)Clock^{s+1}\,Spin^{s+1}\,Sdual\,Clock\\ &= (n[r+1].next)Clock^s\,Spin^s\,Sdual\,Spin\,Clock\\ &= (n,r+2s+1,s)Fnext\,Sdual\,Clock\,Spin\\ &= (n,r+3+2(s+1),s)Fnext\,Sdual\,Clock\,Spin\\ &= (n,r+2,s+1)Sdual\,Fnext\,Sdual\,Clock\,Spin\\ &= (n,r,s)Clock\,Spin\,Sdual\,Fnext\,Sdual\,Clock\,Spin\\ &= (n,r,s)Clock\,Spin\,Enext\,Clock\,Spin.\end{aligned}$$

$(B15)$ $$\begin{aligned}(n,r,s)Sdual^2 &= (n,r+1+2s,s+1)Sdual\\ &= (n,r+1+2s+1+2(s+1),s)\\ &= (n,r,s).\end{aligned}$$

$(B16)$ $$\begin{aligned}(n,r,s)Clock\,Sdual &= (n,r+2,s)Sdual\\ &= (n,r+3+2s,s+1)\\ &= (n,r+1+2s,s+1)Clock\\ &= (n,r,s)Sdual\,Clock.\end{aligned}$$

$(B17)$ $$\begin{aligned}(n,r,s)Spin\,Sdual &= (n,r,s+1)Sdual\\ &= (n,r+1+2(s+1),s)\\ &= (n,r+3+2s,s+1)Spin\\ &= (n,r+1+2s,s+1)Clock\,Spin\\ &= (n,r,s)Sdual\,Clock\,Spin.\end{aligned}$$

$(B18)$ $$(n,r,s)Enext = (n,r,s)Sdual\,Fnext\,Sdual. \qquad \text{(by definition)}$$

5.5 Implementation of the Vertex Functions

The origin partition over the set of facet-edge pairs comprising polyhedral complexes C and C^* was defined in Section 4.5. Applying function *Org* to facet-edge pair a reduces to finding a in the origin partition. Specifically, the query $(n,r,s)Org$ reduces to a *find* operation on $\overline{a}Rot^r$. Component s of the facet-edge reference is not used by *find* since $\overline{a}Rot^r\,Org = \overline{a}Rot^r\,Spin\,Org$. The vertex functions *Dest*, *Left* and *Right* reduce to *Org* by the (B) relations of Chapter 4.

The origin partition may be maintained using any data structure suitable for handling an arbitrary partition of an unordered set. The data structure should support

operations *find* (to which *Org* reduces), and *insert* and *delete* (to which the construction operator *transfer* of Chapter 6 reduces). One possible data structure uses 2-3 trees [AHU, Sec. 4.10]. Each tree corresponds to a class of the partition, that is, to a vertex of C or C^*. Since the elements of each 2-3 tree should be ordered, we associate with facet-edge reference $(n, r, 0)$ the integer $n + r$, thereby ordering the elements in the natural way. (Facet-edge references of the form $(n, r, 1)$ need not be stored in the trees since $(n, r, 0)Org = (n, r, 1)Org$ — that is, $\overline{a}Srot^r Org = \overline{a}Srot^r Spin\, Org$.) A second possible data structure is the hash table [AHU, Sec. 4.2]. The hash value of reference $(n, r, 0)$ depends upon the values of n and r. A third possible structure maintains a linked-list of facet-edge references for each class of the partition [AHU, Sec. 4.6].

Let facet-edge node n correspond to the group of facet-edge pairs whose canonical representative is $\overline{a}$. Element $n[r]$ of the node corresponds to $\overline{a}Rot^r$ (where $r \in \{0, 1, 2, 3\}$), and belongs to the equivalence class corresponding to vertex $\overline{a}Rot^r Org$. The array element $n[r]$ is endowed with an additional field(s) to accommodate the data structure of choice for representing the origin partition.

Chapter 6

Primitive Construction Operators

6.1 Introduction

In this chapter we address the problem of constructing polyhedral subdivisions. Polyhedral subdivisions are built with the primitive construction operators *make_facet_edge*, *splice_facets*, *splice_edges* and *transfer*. The first operator obtains and initializes a new facet-edge node, and returns a facet-edge reference to the node's canonical facet-edge pair. Operators *splice_facets* and *splice_edges* are used to modify the facet- and edge-rings of a subdivision. Operator *transfer* is used to change incidence relations involving vertices and polyhedra, by modifiying the origin partition.

Two caveats accompany these primitive operators. First, no class of subdivisions is closed with respect to these operators: their use does not guarantee that subdivisions are produced. Operator *make_facet_edge* does not, in fact, create a polyhedral subdivision at all — $edge(a)$ of the facet-edge pair a it returns is incident to $facet(a)$ and to no other facet, and so does not belong to the boundary of a polyhedron. Furthermore, little imagination is needed in using *splice_facets* or *splice_edges* to create the most exquisite garbage. Second, these primitives are not easy to use in constructing subdivisions of even moderate complexity. The reader need not be vexed. In Chapter 7, we define higher-level operators in terms of these primitives which make the task of construction quite feasible (if not also easy).

To help describe these primitives, we introduce some notation for manipulating rings. The notation permits us to describe the manipulation of subdivisions in terms of the essentially one dimensional manipulation of facet- and edge-rings. Let $\Phi = (a_1 \ldots a_m)$ and $\Phi' = (a_{m+1} \ldots a_n)$ be two rings with all a_i distinct. Then $concat(\Phi, \Phi')$ represents the ring

$$concat(\Phi, \Phi') = (a_1 \cdots a_m \; a_{m+1} \cdots a_n).$$

The operation $split(\Phi, a_p)$ represents the pair of rings

$$split(\Phi, a_p) = \big((a_1 \ldots a_{p-1}), (a_p \ldots a_m) \big)$$

where $0 < p \leq m+1$. Operations *first* and *second* are used to access the first and second rings of the pair $split(\Phi, a_p)$, respectively. Rings Φ and Φ' are *equivalent*, denoted $\Phi \equiv \Phi'$, if they represent the same cycle of elements — that is, where $|\Phi| = |\Phi'| = n$, there exists an integer j such that, for each $1 \leq i \leq n$, the i^{th} element of Φ is identical to the $(i+j)^{th}$ element of Φ', modulo n. By convention, facet-ring $\mathcal{F}_a$ denotes the ring $\mathcal{F}_a = (aFnext^0 \; aFnext^1 \ldots aFnext^{n-1})$ where $|\mathcal{F}_a| = n$. Similarly, edge-ring $\mathcal{E}_a = (aEnext^0 \; aEnext^1 \ldots aEnext^{m-1})$ where $|\mathcal{E}_a| = m$.

6.2 Operator Make_facet_edge

Construction primitive *make_facet_edge* constructs a new facet-edge node, and returns a facet-edge reference to the node's canonical facet-edge pair $\overline{a}$. The relations (B) of Chapter 4 hold among the eight facet-edge pairs $\overline{a}Srot^r Spin^s$ (where $r \in \{0,1,2,3\}$ and $s \in \{0,1\}$) of the group associated with the node.

The operator is implemented as follows.

```
make_facet_edge()
  {
  n ← a free facet-edge node;
  for r = 0,1,2,3
    n[r].next ← (n,r,0);
  return( (n,r,0) );
  }
```

Where $a = \overline{a}Sdual^d \, Clock^c \, Spin^s$ is any of the eight facet-edge pairs associated with the node, the relations

(*i*) $aFnext = a$, and

(*ii*) $aEnext = a$

are established by the procedure. To justify this assertion, assume that a is represented by the facet-edge reference (n,r,s). We then have

(*i*)
$$\begin{aligned}(n,r,0)Fnext &= n[r].next \\ &= (n,r,0).\end{aligned}$$

(*i*)
$$\begin{aligned}(n,r,1)Fnext &= (n[r+2].next)\,ClockSpin \\ &= (n,r+2,0)\,ClockSpin \\ &= (n,r,0)Spin \\ &= (n,r,1).\end{aligned}$$

(*ii*)
$$\begin{aligned}(n,r,0)Enext &= (n,r,0)SdualFnextSdual \\ &= (n,r+1,1)FnextSdual \\ &= (n[r+3].next)\,ClockSpinSdual \\ &= (n,r+3,0)\,ClockSpinSdual \\ &= (n,r+1,0)SpinSdual \\ &= (n,r+1,1)Sdual \\ &= (n,r,0).\end{aligned}$$

(*ii*)
$$\begin{aligned}(n,r,1)Enext &= (n,r,1)SdualFnextSdual \\ &= (n,r+3,0)FnextSdual \\ &= (n[r+3].next)Sdual \\ &= (n,r+3,0)Sdual \\ &= (n,r,1).\end{aligned}$$

6.3 Operator Splice_facets

The operation $splice\_facets(a,b)$ takes as arguments two facet-edge pairs, and returns no value. The operation affects the facet-rings $\mathcal{F}_a$ and $\mathcal{F}_b$ as follows:

(a) if the two rings are distinct, it combines them into one ring;

(b) if the rings are identical, it breaks the ring into two distinct rings.

The arguments determine where the facet-rings are to be cut and joined. In rings $\mathcal{F}_a$ and $\mathcal{F}_b$, the cuts occur immediately after facets $facet(a)$ and $facet(b)$, respectively. If

the two rings are distinct, the distinct edges $edge(a)$ and $edge(b)$ are coalesced into one edge, and the two rings combined at the cuts. If the two rings are identical, the edge $edge(a)$ $(= edge(b))$ is cleaved lengthwise into two new edges, and each serves as pivot to one of the two new facet-rings resulting from the cuts. The operator, which is illustrated in Figure 14, is similar to the *splice* operator of the quad-edge structure.

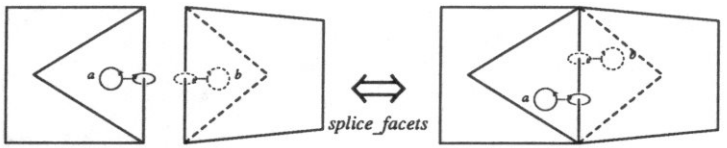


Fig. 14. This diagram illustrates the effect operator *splice_facets* has upon facet-rings.

The operation can be viewed as a way of replacing certain facet-rings with others.

```
splice_facets(a, b)
  {
  if (F_a ≡ F_b)
    replace F_a by the two rings of split(F_aFnext, bFnext);
  else
    replace F_a and F_b by concat(F_aFnext, F_bFnext);
  }
```

Operation $splice\_facets(a, b)$ is accomplished by interchanging the value of $aFnext$ with $bFnext$. The operation affects the $Fnext$ relation in subdivisions C_a and C_b (where by C_a we mean the subdivision to which cells $facet(a)$ and $edge(a)$ belong). Let $\overline{Fnext}$ denote the $Fnext$ relation immediately after the operation is performed. Where $\alpha = aFnext\,Clock$ and $\beta = bFnext\,Clock$, relations $Fnext$ (immediately before the operation) and $\overline{Fnext}$ are related as follows.

(C1) $a\overline{Fnext} = bFnext$.

(C2) $b\overline{Fnext} = aFnext$.

(C3) $\alpha\overline{Fnext} = \beta Fnext$.

(C4) $\beta\overline{Fnext} = \alpha Fnext$.

(C5) $a\,ClockSpin\overline{Fnext} = \beta Spin$.

(C6) $b\,ClockSpin\overline{Fnext} = \alpha Spin$.

(C7) $\alpha\,ClockSpin\overline{Fnext} = bSpin$.

(C8) $\beta\,ClockSpin\overline{Fnext} = aSpin$.

(C9) $\gamma\overline{Fnext} = \gamma Fnext$ for all other facet-edge pairs γ.

To appreciate the significance of the above relations, assume first that operation $splice\_facets(a, b)$ combines distinct rings $\mathcal{F}_a$ and $\mathcal{F}_b$ into a single ring $\mathcal{F}_c$. Consider facets $facet(a)$ and $facet(bFnext)$, which are adjacent in $\mathcal{F}_c$. Viewing $\mathcal{F}_c$ with sense of rotation where $facet(a)$ precedes $facet(bFnext)$, we require relations (C1,5). Viewing $\mathcal{F}_c$ with reversed sense of rotation (so that $facet(a)$ *follows* $facet(bFnext)$), we require relations (C4,8). If we next assume that *splice_facets* is used to split ring $\mathcal{F}_a$ $(= \mathcal{F}_b)$ into two distinct rings, relations (C1,4,5,8) are similarly accounted for by considering facets $facet(a)$ and $facet(bFnext)$ which are adjacent in one of the two resulting facet-rings. Relations (C2,3,6,7) are explained by considering in like fashion the affect of *splice_facets* upon the facets $facet(b)$ and $facet(aFnext)$.

The implementation for $splice\_facets(a, b)$ is quite simple. In the following, we assume the last four assignments are performed simultaneously; in practice, some temporary variables would be used as is customary when swapping values.

```
splice_facets(a, b)
  {
  assume (n, r, s) = a;
  assume (n', r', s') = b;
  (ν, ρ, σ) ← a Fnext Clock;
  (ν', ρ', σ') ← b Fnext Clock;
  n[r + 2s].next ← b Fnext Clock^s Spin^s;
  n'[r' + 2s'].next ← a Fnext Clock^s' Spin^s';
  ν[ρ + 2σ].next ← b Clock^(σ+1) Spin^σ;
  ν'[ρ' + 2σ'].next ← a Clock^(σ'+1) Spin^σ';
  }
```

Correctness of implementation is shown by proving that the procedure *splice_facets* results in the (C) relations. In Figure 15 we show that the *splice_facets* procedure results in (C1,3,5,7). The remaining relations (except (C9)) are shown similarly, while (C9) follows because the procedure involves only four facet-edge node assignments, so *Fnext* of no more than eight facet-edge pairs are affected.

6.4 Operator Splice_edges

The operation *splice_edges*(a, b) takes as arguments two facet-edge pairs, and returns no value. The operation modifies the edge-rings $\mathcal{E}_a$ and $\mathcal{E}_b$ as follows:

(a) if the two rings are distinct, it combines them into one ring;

(b) if the rings are equivalent, it breaks the ring into two rings;

As with *splice_facets*, the arguments to *splice_edges* determine where edge-rings are to be cut and joined. In rings $\mathcal{E}_a$ and $\mathcal{E}_b$, cuts occur immediately after edges *edge*(a) and *edge*(b), respectively. Figure 16 illustrates the effect of *splice_edges*.

The operator can be seen as replacing certain edge-rings with others.

$$
\begin{aligned}
a\overline{Fnext} &= (n,r,0)\overline{Fnext} \\
&= n[r].next \\
&= bFnext.
\end{aligned}
\qquad
\begin{aligned}
a\overline{Fnext} &= (n,r,1)\overline{Fnext} \\
&= (n[r+2].next)\,ClockSpin \\
&= bFnext\,ClockSpin\,ClockSpin \\
&= bFnext.
\end{aligned}
$$

$$
\begin{aligned}
\alpha\overline{Fnext} &= (\nu,\rho,0)\overline{Fnext} \\
&= \nu[\rho].next \\
&= bClock \\
&= \beta Fnext.
\end{aligned}
\qquad
\begin{aligned}
\alpha\overline{Fnext} &= (\nu,\rho,1)\overline{Fnext} \\
&= (\nu[\rho+2].next)\,ClockSpin \\
&= bSpin\,ClockSpin \\
&= \beta Fnext.
\end{aligned}
$$

$$
\begin{aligned}
ClockSpin\overline{Fnext} &= (n,r,0)\,ClockSpin\overline{Fnext} \\
&= (n,r+2,0)Spin\overline{Fnext} \\
&= (n,r+2,1)\overline{Fnext} \\
&= (n[r].next)\,ClockSpin \\
&= bFnext\,ClockSpin \\
&= \beta Spin.
\end{aligned}
\qquad
\begin{aligned}
a\,ClockSpin\overline{Fnext} &= (n,r,1)\,ClockSpin\overline{Fnext} \\
&= (n,r+2,0)\overline{Fnext} \\
&= n[r+2].next \\
&= bFnext\,ClockSpin \\
&= \beta Spin.
\end{aligned}
$$

$$
\begin{aligned}
\alpha\,ClockSpin\overline{Fnext} &= (\nu,\rho,0)\,ClockSpin\overline{Fnext} \\
&= (\nu,\rho+2,1)\overline{Fnext} \\
&= (\nu[\rho].next)\,ClockSpin \\
&= bClock\,ClockSpin \\
&= bSpin.
\end{aligned}
\qquad
\begin{aligned}
\alpha\,ClockSpin\overline{Fnext} &= (\nu,\rho,1)\,ClockSpin\overline{Fnext} \\
&= (\nu,\rho+2,0)\overline{Fnext} \\
&= \nu[\rho+2].next \\
&= bSpin
\end{aligned}
$$

Fig. 15. Proof that procedure *splice_facets* results in relations (C1,3,5,7).

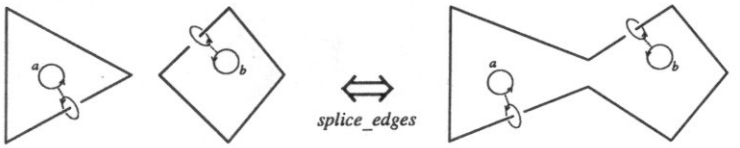


Fig. 16. This diagram illustrates the effect operator *splice_edges* has upon edge-rings.

splice_edges(a, b)
{
if $(\mathcal{E}_a \equiv \mathcal{E}_b)$
replace $\mathcal{E}_a$ by the two rings of $split(\mathcal{E}_{aEnext}, bEnext)$;
else
replace $\mathcal{E}_a$ and $\mathcal{E}_b$ by $concat(\mathcal{E}_{aEnext}, \mathcal{E}_{bEnext})$;
}

Operation *splice_edges*(a, b) affects the *Enext* relation in subdivisions C_a and C_b (or equivalently, the *Fnext* relation in C_a^* and C_b^*). Let $\overline{Enext}$ denote the *Enext* relation immediately after the operation. Where $\alpha = aEnext\,Clock$ and $\beta = bEnext\,Clock$, *Enext* (before the operation) and $\overline{Enext}$ are related as follows.

(D1) $a\overline{Enext} = bEnext$.

(D2) $b\overline{Enext} = aEnext$.

(D3) $\alpha\overline{Enext} = \beta Enext$.

(D4) $\beta\overline{Enext} = \alpha Enext$.

(D5) $aSpin\overline{Enext} = \beta Clock Spin$.

(D6) $bSpin\overline{Enext} = \alpha Clock Spin$.

(D7) $\alpha Spin\overline{Enext} = bClock Spin$.

(D8) $\beta Spin\overline{Enext} = aClock Spin$.

(D9) $\gamma\overline{Enext} = \gamma Enext$ for all other facet-edge pairs γ.

The relations (D) are dual to relations (C) of Section 6.3. Operations *Enext*, $\overline{Enext}$, *Spin* and *ClockSpin* of (D) replace operations *Fnext*, $\overline{Fnext}$, *ClockSpin* and *Spin* of (C), respectively. The reader is invited to use this observation to account for the relations (D), much as we have done for relations (C) in Section 6.3.

As we might expect, an edge-ring of one subdivision can be modified by operating on the corresponding facet-ring of the dual subdivision. Indeed, *splice_edges* is implemented in terms of *splice_facets* as follows.

splice_edges(a, b)
{
splice_facets$(aSdual, bSdual)$;
}

To show correctness of this implementation, it suffices to show that the (D) relations are satisfied by *splice_facets*$(aSdual, bSdual)$. Below we show this for (D2,4,6,8); remaining relations (except for (D9)) are shown similarly, whereas (D9) holds since *splice_facets* affects no more than four facet-edge node elements.

Operation *splice_facets*$(aSdual, bSdual)$ establishes the following relations, where $\overline{Fnext}$ denotes the *Fnext* relation immediately after the operation.

(i) $bSdual\overline{Fnext} = aSdual Fnext$

(ii) $bSdual Fnext Clock\overline{Fnext} = aSdual Clock$

(iii) $bSdual Spin Clock\overline{Fnext} = aSpin Sdual Fnext$

(iv) $bSdual Fnext Spin\overline{Fnext} = aSdual Spin$

Relations (i), (ii), (iii) and (iv) follow easily from relations (C2), (C4), (C6) and (C8), respectively, resulting from operation *splice_facets*$(aSdual, bSdual)$. They are used in showing relations (D) below.

(D2)
$$\begin{aligned} b\overline{Enext} &= bSdual\overline{Fnext}Sdual \\ &= aSdual Fnext Sdual && (i) \\ &= aEnext \end{aligned}$$

(D4)
$$\begin{aligned} \beta\overline{Enext} &= bEnext\,Clock\overline{Enext} \\ &= bSdual Fnext Sdual\, Clock Sdual\overline{Fnext}Sdual \\ &= bSdual Fnext Clock\overline{Fnext}Sdual \\ &= aSdual Clock Sdual && (ii) \\ &= aClock \\ &= \alpha Enext \end{aligned}$$

(D6)
$$bSpin\overline{Enext} = bSpin Sdual\overline{Fnext}Sdual$$

$$= bSdual\,Spin\,Clock\,\overline{Fnext}\,Sdual$$
$$= aSpin\,Sdual\,Fnext\,Sdual \qquad (iii)$$
$$= aSpin\,Enext$$
$$= \alpha\,Enext\,Spin$$

(D8) $$\beta Spin\,\overline{Enext} = bEnext\,Clock\,Spin\,\overline{Enext}$$
$$= bSdual\,Fnext\,Sdual\,Clock\,Spin\,Sdual\,\overline{Fnext}\,Sdual$$
$$= bSdual\,Fnext\,Spin\,\overline{Fnext}\,Sdual$$
$$= aSdual\,Spin\,Sdual \qquad (iv)$$
$$= aClock\,Spin$$

6.5 Modifying Vertex Incidence Relations

By each operator's definition, neither *splice_facets* nor *splice_edges* affects incidence relations involving vertices and polyhedra. For instance, when *splice_facets* is used to split a single facet-ring $\mathcal{F}_e$, thereby cleaving edge e length-wise, the two resulting edges share a common origin and destination vertex. Higher-level construction operators generally need to modify incidence relations involving vertices and polyhedra. To permit this, we introduce the operator *transfer*.

Let P be a partition of some universe U, let A be an equivalence class of P, and let $B \subset U$. The operation $transfer(A, B)$ modifies partition P by transfering each element $b \in B$ from its respective equivalence class $class(b)$ into A.

```
transfer(A, B)
  {
  for each b ∈ B {
    class(b) ← class(b) − b;
    A ← A ∪ b
    }
  }
```

If $A = \emptyset$, then B becomes an equivalence class in the new partition. If B is an equivalence class of P, then $transfer(A, B)$ simply forms the union of the two classes.

The usefulness of *transfer* becomes evident if we recall that each vertex and polyhedron of a polyhedral subdivision is represented by an equivalence class of the origin partition (over the facet-edge pairs that comprise a subdivision and its dual). Incidence relations are implied by the partition. For instance, vertex v is an endpoint of edge e iff there exists some facet-edge pair a with $edge(a) = e$, for which $a\,Org = v$. If this is the case, it follows that $aSpin\,Org = v$ also, and that none of the six remaining oriented, spun versions of a have origin v (assuming $edge(a)$ is not illegally a loop). Edge e can then be given a new endpoint $v' = bOrg$ by the operation $transfer(bOrg, \{a, aSpin\})$.

The implementation of *transfer* depends upon the scheme chosen for handling the origin partition.

Chapter 7

Manipulating Individual Polyhedra

7.1 Introduction

In this chapter we are concerned with how individual polyhedra are manipulated. By treating individual polyhedra, the task of handling and constructing non-trivial polyhedral subdivisions is made manageable.

In Section 7.2, we show how, for any polyhedron of a polyhedral subdivision, queries regarding the polyhedron's boundary can be answered. The boundary is a (closed) subdivision of the sphere, queries concerning which may be expressed in terms of the edge functions. Each edge function is reduced to queries of the underlying facet-edge structure.

In Section 7.3, we treat the construction of an individual polyhedron, whose representation is to be given by the facet-edge structure. (A polyhedron may be regarded as a trivial polyhedral complex.) Since the polyhedron's boundary is combinatorially a subdivision of the sphere, construction operators appropriate for the construction of such objects are introduced. We show how each such operator creates and modifies the underlying facet-edge structure. By reducing each such operator to the facet-edge construction primitives of the previous chapter, the edge operator's effect upon the underlying facet-edge structure is well defined.

In Section 7.4, we show how polyhedra may be glued together along facets. The tools presented in Sections 7.3 and 7.4 together enable us to form non-trivial polyhedral subdivisions. Use of the tools will be illustrated in the applications of Chapters 8.

7.2 Traversal in the Boundary of a Polyhedron

Given a polyhedral subdivision, some applications require the capacity to select any

polyhedron of the subdivision, and to make combinatorial queries of that polyhedron's boundary while neglecting the rest of the subdivision. One such application is that of creating pictures of a polyhedral subdivision using computer graphics. This is a difficult problem since many polyhedra tend to be "buried" in a large subdivision, and so are occluded by other polyhedra. One solution is to create pictures of specific regions of interest which ignore the rest of the subdivision. Each such picture can be created by selecting a subset of polyhedra and neglecting the rest. The capacity to make combinatorial queries of these select polyhedra facilitates their rendering.

In the current section, we show how a polyhedron p of subdivision C or C^* is selected, and how edge function queries are made of its boundary. Where polyhedral subdivisions C and C^* are represented by the facet-edge structure, each edge function is reduced to queries of the underlying facet-edge structure. In the next subsection, we establish a correspondence between the directed, oriented edges of ∂p and the facet-edge pairs of C and C^*, a correspondence we call the *boundary representation scheme*. In the subsequent subsection we use this correspondence to reduce the edge functions to the facet-edge functions.

7.2.1 The Boundary Representation Scheme

Let p be any polyhedron of one of the subdivisions C or C^*, where C and simultaneously C^* are represented by the facet-edge structure. Let Q be the polygonal subdivision ∂p, and let Q^* be the polygonal subdivision dual to Q. Let e be any directed and oriented edge of Q or Q^*.

Edge e is represented by the pair $\langle a, d \rangle$, called a *boundary reference*. The first component of the pair is a facet-edge reference to facet-edge pair a, while the second component is a *duality bit* $d \in \{0, 1\}$. These components are determined as follows. Let edge e' be given by

$$e' = \begin{cases} e & \text{if } e \in Q \\ eDual & \text{if } e \in Q^* \end{cases}$$

Facet-edge pair a is determined so that

(i) basic edge $edge(a)$ and basic edge e' are the same,

(ii) basic facet $facet(a)$ and basic facet $e'Left$ are the same,

(iii) the orientation of a agrees with the orientation of e' in Q, and

(iv) $aPpos = p$.

Duality bit d is determined so that

$$d = \begin{cases} 0 & \text{if } e \in Q \\ 1 & \text{if } e \in Q^* \end{cases}$$

The correspondence between edge and boundary reference is depicted in Figure 17.

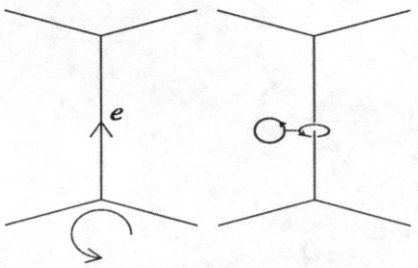


Fig. 17. The boundary representation scheme is depicted. Both figures correspond to the same region of $Q = \partial p$, where polyhedron p lies behind the page. Edge e' is represented by $\langle a, 0\rangle$, where facet-edge pair a is depicted by the handcuff. Edge $e'Dual$ (not shown) is then represented by $\langle a, 1\rangle$.

We show that the boundary representation scheme is unambiguous and unique. We show unambiguity first. Observing that boundary reference $\langle a, d\rangle$ can refer to only one polyhedron $p = aPpos$ by (iv), it suffices to show that $\langle a, d\rangle$ represents a unique edge of Q or Q^*, where $Q = \partial p$. Consider boundary reference $\langle a, 0\rangle$ first. Under the scheme, $\langle a, 0\rangle$ represents basic $edge(a) = e' \in Q$ by (i). The orientation of e' is fixed by (iii). The direction of e' is then determined since under fixed orientation, e' can assume only one direction and still satisfy (ii). Hence $\langle a, 0\rangle$ represents the unique

directed, oriented edge $e' \in Q$. Finally, we note that $\langle a, 1\rangle$ must then represent edge $e'Dual \in Q^*$, which is well-defined. Unambiguity of the scheme is shown.

To argue the uniqueness of the scheme, we show that any edge of any polyhedron is represented by a unique boundary reference $\langle a, d\rangle$. Let edge e belong to Q or Q^*, where $Q = \partial p$, and p is any polyhedron. Let edge e' be given by

$$e' = \begin{cases} e & \text{if } e \in Q \\ eDual & \text{if } e \in Q^* \end{cases}$$

The value of the duality bit d is determined by whether e belongs to Q or to Q^*. The edge and facet components of facet-edge pair a are determined by (i) and (ii) respectively, so the basic facet-edge pair a is determined. The orientation of e' determines the orientation of a by (iii). Finally, the orientation of a together with condition (iv) determines the spin of a. Hence components a and d of boundary reference $\langle a, d\rangle$ are determined, and uniqueness of the scheme is shown.

7.2.2 Implementation of the Edge Functions

The boundary representation scheme allows us to reduce edge function queries in the boundary of polyhedron p to traversal queries in the polyhedral subdivision to which p belongs. Each edge function can be defined in terms of its effect upon a boundary reference. More precisely, for each edge primitive Op and duality bit d, there exists a sequence of facet-edge functions Op' and duality bit d', for which

$$\langle a, d\rangle\, Op = \langle a\, Op', d'\rangle.$$

The following characterizes each edge function in this fashion.

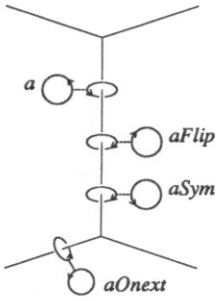


Fig. 18. This diagram depicts the facet-edge pairs used to represent directed, oriented edges in the boundary of polyhedron p, which lies behind the page.

(E1) $\langle a, 0\rangle Flip = \langle a Fnext Spin, 0\rangle$.

(E2) $\langle a, 0\rangle Sym = \langle a Fnext Clock, 0\rangle$.

(E3) $\langle a, 0\rangle Onext = \langle a Enext^{-1} Fnext Clock, 0\rangle$.

(E4) $\langle a, d\rangle Dual = \langle a, 1-d\rangle$.

(E5) $\langle a, 1\rangle Flip = \langle a Spin Clock, 1\rangle$.

(E6) $\langle a, 1\rangle Sym = \langle a Fnext Clock, 1\rangle$.

(E7) $\langle a, 1\rangle Onext = \langle a Enext^{-1}, 1\rangle$.

The remaining edge functions are defined in terms of these. Figure 18 motivates formulae (E1–3). Formula (E4) follows directly from the significance of duality bit d in $\langle a, d\rangle$. Formulae (E5–7) are derived (below) from (E1–4), together with the relations that hold among edge functions, and among facet-edge traversal functions.

(E5)
$$\begin{aligned}
\langle a,1\rangle Flip &= \langle a,0\rangle Dual Flip && (E4)\\
&= \langle a,0\rangle Flip Sym Dual && (A3')\\
&= \langle a Fnext Spin,0\rangle Sym Dual && (E1)\\
&= \langle a Fnext Spin Fnext Clock,0\rangle Dual && (E2)\\
&= \langle a Spin Clock,0\rangle Dual && (B1,5)\\
&= \langle a Spin Clock,1\rangle. && (E4)
\end{aligned}$$

(E6)
$$\begin{aligned}
\langle a,1\rangle Sym &= \langle a,0\rangle Dual Sym && (E4)\\
&= \langle a,0\rangle Sym Dual && (A2')
\end{aligned}$$

$$\begin{aligned}
&= \langle a Fnext Clock,0\rangle Dual && (E2)\\
&= \langle a Fnext Clock,1\rangle. && (E4)
\end{aligned}$$

(E7)
$$\begin{aligned}
\langle a,1\rangle Onext &= \langle a,0\rangle Dual Onext && (E4)\\
&= \langle a,0\rangle Onext Sym Dual && (A4')\\
&= \langle a Enext^{-1} Fnext Clock,0\rangle Sym Dual && (E3)\\
&= \langle a Enext^{-1} Fnext Clock Fnext Clock,0\rangle Dual && (E2)\\
&= \langle a Enext^{-1},0\rangle Dual && (B2,4)\\
&= \langle a Enext^{-1},1\rangle && (E4).
\end{aligned}$$

The correctness of (E1–7) can be formally verified by showing that the edge primitives so defined satisfy the properties of Section 3.3, or [GS, section 2.3]. For instance, $eFlip^2 = e$ is shown by

$$\begin{aligned}
eFlip^2 &= \langle a,0\rangle Flip^2\\
&= \langle a Fnext Spin Fnext Spin,0\rangle\\
&= \langle a,0\rangle\\
&= e.
\end{aligned}$$

$$\begin{aligned}
eFlip^2 &= \langle a,1\rangle Flip^2\\
&= \langle a Spin Clock Spin Clock,1\rangle\\
&= \langle a,1\rangle\\
&= e.
\end{aligned}$$

We do not give the proof here since it singularly tedious and unenlightening.

7.3 Constructing a Polyhedron

A polyhedron can be characterized by its boundary, which is a (closed) subdivision of the sphere. In the current section, we introduce the three edge construction operators *make_segment*, *make_loop* and *splice*, for building the boundary of a polyhedron. The operators are used iteratively to produce *open* subdivisions of the sphere, until the desired one is produced which corresponds to the boundary of the target polyhedron.

In this section, the edge operators are also reduced to the facet-edge construction primitives; in this manner, the effect each edge operator has on the underlying facet-edge structure is described.

The polygonal subdivision Q built with the use of the edge operators is the boundary of a polyhedron p. The polyhedron is explicitly represented by the facet-edge structure. The 2-dimensional subdivision Q^* which is dual to Q, is also formed by use of the edge operators. Q^* is implicitly represented by the facet-edge structure, under the boundary representation scheme. It should not be regarded as the boundary of a polyhedron, but merely as the 2-dimensional dual to Q. We call Q the *primal*, and Q^* the *dual*, polygonal subdivision of p.

With use of the edge construction operators, we maintain a current collection of open subdivisions of the sphere, and their duals. In each sphere S of the collection, we distinguish between the primal and the dual open subdivisions. Primal open subdivisions are combined with other primal open subdivisions to eventually form the primal polygonal subdivision Q. Dual open subdivisions combine to form the dual subdivision Q^*. The boundary representation scheme requires that this strict division between primal and dual be observed, and forbids a primal and dual from being combined during the construction of an open subdivision.

The edge operators *make_segment* and *make_loop* create elementary open subdivisions of the sphere. They are based on the *make_edge* operator of [GS]. The edge operator *splice*, identical to the operator *splice* of [GS], is used to modify open subdivisions. Using the tools of the current section, computer code for building a quad-edge representation of a polyhedron via calls to *make_edge* and *splice* can be easily modified to produce a facet-edge representation of the same polyhedron.

7.3.1 Two Elementary Open Subdivisions of the Sphere

There are two elementary open subdivisions of the sphere. One of these consists of a single edge e that is a segment, the edge's two endpoint vertices, and a single polygon

which is incident to the edge. We denote this open subdivision by C_s (in which subscript s stands for 'segment'). Where edge e has arbitrary orientation and direction, we have $e\,Org \neq e\,Dest$ and $e\,Left = e\,Right$. The following properties hold in C_s:

(F1) $e\,Onext = e$.

(F2) $e\,Sym\,Onext = e\,Sym$.

(F3) $e\,Flip\,Onext = e\,Flip$.

(F4) $e\,Flip\,Sym\,Onext = e\,Flip\,Sym$.

The other elementary open subdivision of the sphere consists of a single edge e' that is a loop, a single vertex which serves as the edge's sole endpoint, and two polygons incident to the edge. We denote this open subdivision by C_ℓ (in which subscript ℓ stands for 'loop'). Where edge e' has arbitrary orientation and direction, we have $e'\,Org = e'\,Dest$ and $e'\,Left \neq e'\,Right$. The following properties hold in C_ℓ:

(F5) $e'\,Onext = e'\,Sym$.

(F6) $e'\,Sym\,Onext = e'$.

(F7) $e'\,Flip\,Onext = e'\,Flip\,Sym$.

(F8) $e'\,Flip\,Sym\,Onext = e'\,Flip$.

Open subdivisions C_s and C_ℓ are dual to one another. This assertion is illustrated in Figure 19. Since we wish to simultaneously construct an open subdivision and its dual, creation of one of these open subdivisions should result in the creation of the other. The edge operator *make_segment* builds a representation for both C_s and C_ℓ, in which C_s is primal and C_ℓ dual. The edge operator *make_loop* builds a representation for the two open subdivisions in which C_ℓ is primal and C_s dual. The effect of each of the two operators is to build a facet-edge data structure which, under the boundary representation scheme of the previous section, simultaneously represents C_s and C_ℓ.

Operator *make_segment* builds the primal open subdivision C_s and the dual C_ℓ. It returns a boundary reference to one version of the segment $e \in C_s$. It is implemented as follows.

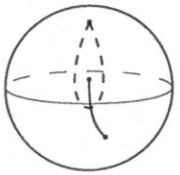


Fig. 19. The diagram shows open subdivisions C_s and C_ℓ, superimposed on the same sphere to reflect their relation to each other.

```
make_segment()
  {
  a ← make_facet_edge();
  b ← make_facet_edge();
  splice_facets(a, b);
  splice_edges(a, bClock);
  return( ⟨a, 0⟩ );
  }
```

The operation $e \leftarrow$ *make_segment*() first obtains two new facet-edge nodes whose canonical facet-edge pairs are a and b. The operation *splice_facets*(a, b) results in

(i) $aFnext = b$,

(ii) $bFnext = a$,

while operation *splice_edges*$(a, bClock)$ results in

(iii) $aEnext = bClock$,

(iv) $aEnext^{-1} = bClock$.

The reader can easily verify this claim by appealing to the definitions of operators *splice_facets* and *splice_edges* given in Chapter 6. The boundary reference $\langle a, 0\rangle$ represents the edge e, where e is returned by *make_segment*(). The boundary reference $\langle bSpin, 0\rangle$ represents edge $eFlip$. This is illustrated in Figure 20.

The correctness of the implementation is shown by verifying that relations (F1–4) hold in C_s, and that (F5–8) hold in C_ℓ where $e' = eDual$. Here edge $e = \langle a, 0\rangle$, so edge $e' = eDual = \langle a, 0\rangle Dual = \langle a, 1\rangle$ under the boundary representation scheme. The

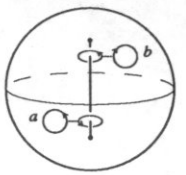


Fig. 20. This figure depicts the roles of facet-edge pairs a and b in the representation of the open subdivision C_s.

relations $(i$–$iv)$ established by the *make_segment* operation are used below.

(F1)
$$\begin{aligned} eOnext &= \langle a, 0\rangle Onext \\ &= \langle aEnext^{-1} Fnext Clock, 0\rangle && (E3) \\ &= \langle bClock Fnext Clock, 0\rangle && (iv) \\ &= \langle bFnext^{-1}, 0\rangle && (B4) \\ &= \langle a, 0\rangle && (i) \\ &= e. \end{aligned}$$

(F2)
$$\begin{aligned} eSym Onext &= \langle a, 0\rangle Sym Onext \\ &= \langle aFnext Clock Enext^{-1} Fnext Clock, 0\rangle && (B2,3) \\ &= \langle bClock Enext^{-1} Fnext Clock, 0\rangle && (i) \\ &= \langle aEnext Enext^{-1} Fnext Clock, 0\rangle && (iii) \\ &= \langle a, 0\rangle Sym && (E2) \\ &= eSym. \end{aligned}$$

(F3)
$$\begin{aligned} eFlip Onext &= \langle a, 0\rangle Flip Onext \\ &= \langle aFnext Spin Enext^{-1} Fnext Clock, 0\rangle && (E1,3) \\ &= \langle bSpin Enext^{-1} Fnext Clock, 0\rangle && (i) \\ &= \langle aSpin Clock Enext Enext^{-1} Fnext Clock, 0\rangle && (iv) \\ &= \langle aSpin Clock Enext Enext^{-1} Fnext Clock, 0\rangle && (B7) \\ &= \langle aSpin Fnext^{-1}, 0\rangle && (B4) \\ &= \langle aFnext Spin, 0\rangle && (B1,5) \\ &= \langle a, 0\rangle Flip && (E1) \\ &= eFlip. \end{aligned}$$

(F4) $eFlip Sym Onext = \langle a, 0\rangle Flip Sym Onext$

$$= \langle a\,Fnext\,Spin\,Fnext\,Clock\,Enext^{-1}\,Fnext\,Clock, 0\rangle \quad (E1,2,3)$$
$$= \langle a\,Spin\,Clock\,Enext^{-1}\,Fnext\,Clock, 0\rangle \quad (B1,5)$$
$$= \langle a\,Enext\,Fnext\,Spin, 0\rangle \quad (B7)$$
$$= \langle b\,Clock\,Fnext\,Spin, 0\rangle \quad (iii)$$
$$= \langle b\,Fnext^{-1}\,Clock\,Spin, 0\rangle \quad (B4)$$
$$= \langle a\,Clock\,Spin, 0\rangle \quad (i)$$
$$= \langle a, 0\rangle\,Flip\,Sym \quad (B5; E1,2)$$
$$= e\,Flip\,Sym.$$

(F5) $$e'\,Onext = \langle a, 1\rangle\,Onext$$
$$= \langle a\,Enext^{-1}, 1\rangle \quad (E7)$$
$$= \langle b\,Clock, 1\rangle \quad (iv)$$
$$= \langle a\,Fnext\,Clock, 1\rangle \quad (i)$$
$$= \langle a, 1\rangle\,Sym \quad (E6)$$
$$= e'\,Sym.$$

(F6) $$e'\,Sym\,Onext = \langle a, 1\rangle\,Sym\,Onext$$
$$= \langle a\,Fnext\,Clock\,Enext^{-1}, 1\rangle \quad (E6,7)$$
$$= \langle b\,Clock\,Enext^{-1}, 1\rangle \quad (i)$$
$$= \langle a\,Enext\,Enext^{-1}, 1\rangle \quad (iii)$$
$$= \langle a, 1\rangle$$
$$= e'.$$

(F7) $$e'\,Flip\,Onext = \langle a, 1\rangle\,Flip\,Onext$$
$$= \langle a\,Spin\,Clock\,Enext^{-1}, 1\rangle \quad (E5,7)$$
$$= \langle b\,Clock\,Enext^{-1}\,Spin\,Clock\,Enext^{-1}, 1\rangle \quad (iii)$$
$$= \langle b\,Clock\,Spin\,Clock\,Enext\,Enext^{-1}, 1\rangle \quad (B7)$$
$$= \langle b\,Spin, 1\rangle \quad (B3)$$
$$= \langle a\,Fnext\,Spin, 1\rangle \quad (i)$$
$$= \langle a, 1\rangle\,Flip\,Sym \quad (B4,5; E5,6)$$
$$= e'.$$

(F8) $$e'\,Flip\,Sym\,Onext = \langle a, 1\rangle\,Flip\,Sym\,Onext$$
$$= \langle a\,Fnext\,Spin\,Enext^{-1}, 1\rangle \quad (B5; E5,6)$$
$$= \langle b\,Spin\,Enext^{-1}, 1\rangle \quad (i)$$

$$= \langle b\,Enext^{-1}\,Spin, 1\rangle \quad (B1,2,6,7)$$
$$= \langle a\,Clock\,Enext\,Enext^{-1}\,Spin, 1\rangle \quad (iv)$$
$$= \langle a\,Spin\,Clock, 1\rangle \quad (B3)$$
$$= \langle a, 1\rangle\,Flip \quad (E5)$$
$$= e'\,Flip.$$

Operator *make_loop* builds the primal open subdivision C_ℓ and the dual C_s. It returns the boundary reference to one version of the loop edge e' belonging to C_ℓ. Edge e' serves as dual to the edge returned by *make_segment*. The operator is implemented as follows.

```
make_loop()
  {
  a ← make_facet_edge();
  b ← make_facet_edge();
  splice_facets(a, b);
  return( ⟨a, 0⟩ );
  }
```

The operation $e' \leftarrow$ *make_loop*() obtains two new facet-edge nodes whose canonical facet-edge pairs are a and b. Operation *splice_facets*(a, b) results in

(i') $a\,Fnext = b$,

(ii') $b\,Fnext = a$,

while the absence of a call to *splice_edges* results in

(iii') $a\,Enext = a$,

(iv') $b\,Enext = b$.

The boundary reference $\langle a, 0\rangle$ represents e', where e' is returned by *make_loop*. Boundary reference $\langle b\,Spin, 0\rangle$ represents edge $e'\,Flip$. This is illustrated in Figure 21.

Correctness of implementation is shown by verifying that relations (F1–4) hold in open subdivision C_s and that (F5–8) hold in C_ℓ, where edge e' is represented by the boundary reference $\langle a, 0\rangle$, and edge $e = e'\,Dual = \langle a, 0\rangle\,Dual = \langle a, 1\rangle$. Relations $(i'$–$iv')$

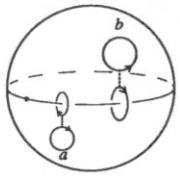


Fig. 21. This figure depicts the roles of facet-edge pairs a and b in the representation of the open subdivision C_t.

established by the *make_loop* operation are used below.

(F1)
$$\begin{aligned} e\,Onext &= \langle a,1\rangle\, Onext \\ &= \langle a\,Enext^{-1},1\rangle && (E7) \\ &= \langle a,1\rangle && (iii') \\ &= e. \end{aligned}$$

(F2)
$$\begin{aligned} e\,Sym\,Onext &= \langle a,1\rangle\, Sym\,Onext \\ &= \langle a\,Fnext\,Clock\,Enext^{-1},1\rangle && (E6,7) \\ &= \langle b\,Enext\,Clock,1\rangle && (i') \\ &= \langle b\,Clock,1\rangle && (iv') \\ &= \langle a\,Fnext\,Clock,1\rangle && (i') \\ &= \langle a,1\rangle\, Sym && (E6) \\ &= e. \end{aligned}$$

(F3)
$$\begin{aligned} e\,Flip\,Onext &= \langle a,1\rangle\, Flip\,Onext \\ &= \langle a\,Spin\,Clock\,Enext^{-1},1\rangle && (E5,7) \\ &= \langle a\,Enext\,Spin\,Clock,1\rangle && (B7) \\ &= \langle a\,Spin\,Clock,1\rangle && (iv') \\ &= \langle a,1\rangle\, Flip && (E5) \\ &= e\,Flip. \end{aligned}$$

(F4)
$$\begin{aligned} e\,Flip\,Sym\,Onext &= \langle a,1\rangle\, Flip\,Sym\,Onext \\ &= \langle a\,Fnext\,Spin\,Enext^{-1},1\rangle && (B4,7;E5,6,7) \\ &= \langle b\,Enext^{-1}\,Spin,1\rangle && (i') \\ &= \langle b\,Spin,1\rangle && (iv') \\ &= \langle a\,Fnext\,Spin,1\rangle && (i') \end{aligned}$$

$$\begin{aligned} &= \langle a,1\rangle\, Flip\,Sym && (B4;E5,6) \\ &= e\,Flip\,Sym. \end{aligned}$$

(F5)
$$\begin{aligned} e'\,Onext &= \langle a,0\rangle\, Onext \\ &= \langle a\,Enext^{-1}\,Fnext\,Clock,0\rangle && (E3) \\ &= \langle a\,Fnext\,Clock,0\rangle && (iii') \\ &= \langle a,0\rangle\, Sym && (E2) \\ &= e'\,Sym. \end{aligned}$$

(F6)
$$\begin{aligned} e'\,Sym\,Onext &= \langle a,0\rangle\, Sym\,Onext \\ &= \langle a\,Fnext\,Clock\,Enext^{-1}\,Fnext\,Clock,0\rangle && (E2,3) \\ &= \langle b\,Clock\,Enext^{-1}\,Fnext\,Clock,0\rangle && (i') \\ &= \langle b\,Enext\,Clock\,Fnext\,Clock,0\rangle && (B6) \\ &= \langle b\,Clock\,Fnext\,Clock,0\rangle && (iv') \\ &= \langle b\,Fnext^{-1},0\rangle && (B4) \\ &= \langle a,0\rangle && (i') \\ &= e'. \end{aligned}$$

(F7)
$$\begin{aligned} e'\,Flip\,Onext &= \langle a\,Fnext\,Spin\,Enext^{-1}\,Fnext\,Clock,0\rangle && (E1,3) \\ &= \langle b\,Spin\,Enext^{-1}\,Fnext\,Clock,0\rangle && (i') \\ &= \langle b\,Enext^{-1}\,Fnext^{-1}\,Spin\,Clock,0\rangle && (B2,3,6,7) \\ &= \langle b\,Fnext^{-1}\,Spin\,Clock,0\rangle && (iv') \\ &= \langle a\,Spin\,Clock,0\rangle && (i') \\ &= \langle a,0\rangle\, Flip\,Sym && (B5;E1,2) \\ &= e'\,Flip\,Sym. \end{aligned}$$

(F8)
$$\begin{aligned} e'\,Flip\,Sym\,Onext &= \langle a\,Spin\,Clock\,Enext^{-1}\,Fnext\,Clock,0\rangle && (E1,2,3) \\ &= \langle a\,Enext\,Spin\,Clock\,Fnext\,Clock,0\rangle && (B7) \\ &= \langle a\,Spin\,Clock\,Fnext\,Clock,0\rangle && (iii') \\ &= \langle a\,Fnext\,Spin,0\rangle && (B4,5) \\ &= \langle a,0\rangle\, Flip\,(E1) \\ &= e'\,Flip. \end{aligned}$$

7.3.2 Modifying Open Subdivisions of the Sphere

The operator *splice* is used to modify open subdivisions of the sphere. The operator is introduced and described in full in [GS]. Operation $splice(e_a, e_b)$ takes as arguments two directed, oriented edges e_a and e_b, and returns no value. The operation affects the two edge-cycles $e_a Org$ and $e_b Org$ and, independently, the two edge-cycles $e_a Left$ and $e_b Left$. In each case, if the two cycles are distinct, *splice* combines them into one cycle; and if the two cycles are identical, *splice* breaks it into two distinct cycles. The arguments e_a and e_b determine where the cycles will be cut and joined. For cycles $e_a Org$ and $e_b Org$, the cuts occur immediately after e_a and e_b; for cycles $e_a Left$ and $e_b Left$, the cuts occur immediately after $e_a Onext Rot$ and $e_b Onext Rot$.

We recall that the edge-cycle $e_a Left$ corresponds under duality to the edge-cycle $e_a Onext Rot Org$. To describe *splice*, it is convenient to work solely with edge-cycles incident to vertices (and not with cycles that bound polygons). Hence we define the two edges $\varepsilon_a = e_a Onext Rot$ and $\varepsilon_b = e_b Onext Rot$ for describing the affect of *splice* on $e_a Left$ and $e_b Right$.

The operator *splice* can be viewed as replacing some cycles by other cycles. Where edge-cycle $e_a Org$ is $(e_a Onext^0\ e_a Onext^1 \cdots e_a Onext^{n-1})$, we can use the ring manipulation notation to describe the effect of $splice(e_a, e_b)$:

```
splice(e_a, e_b)
  {
  if (e_a Org ≡ e_b Org)
    replace e_a Org by the two rings of split(e_a Onext Org, e_b Onext);
  else
    replace e_a Org and e_b Org by concat(e_a Onext Org, e_b Onext Org);
  if (ε_a Org ≡ ε_b Org)
    replace ε_a Org by the two rings of split(ε_a Onext Org, ε_b Onext);
  else
    replace ε_a Org and ε_b Org by concat(ε_a Onext Org, ε_b Onext Org);
  }
```

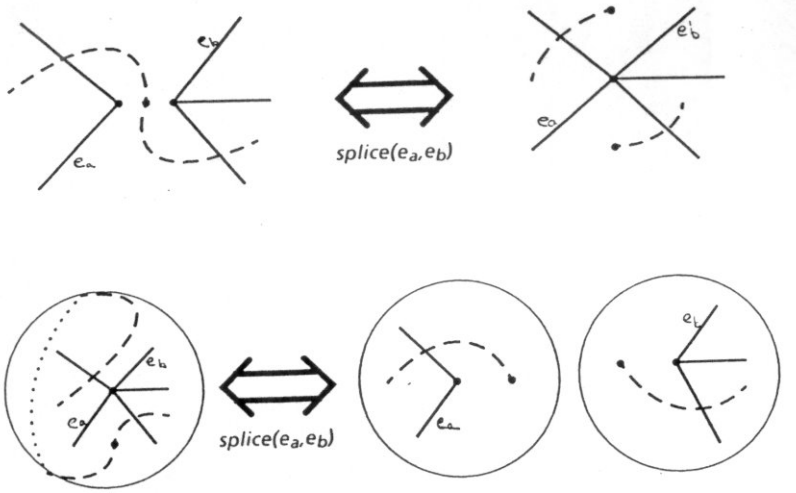



Fig. 22. This diagram shows the effect of *splice* on edge-cycles. In the top figure, $e_a Org \neq e_b Org$, $e_a Left = e_b Left$ on the left; $e_a Org = e_b Org$, $e_a Left \neq e_b Left$ on the right. In the bottom figure, $e_a Org = e_b Org$, $e_a Left = e_b Left$ on the left; $e_a Org \neq e_b Org$, $e_a Left \neq e_b Left$ on the right.

The effect of *splice* upon edge-cycles is depicted in Figure 22. We note that since $e_a Onext Rot Onext Rot = \varepsilon_a Onext Rot = e_a$, the operations $splice(e_a, e_b)$ and $splice(\varepsilon_a, \varepsilon_b)$ are equivalent. Operator *splice* is quite similar to *splice_facets* and *splice_edges*, and served as the model for the latter two operators.

Operation $splice(e_a, e_b)$ is performed by interchanging the values of $e_a Onext$ with $e_b Onext$, and $\varepsilon_a Onext$ with $\varepsilon_b Onext$. More formally, where $\overline{Onext}$ denotes the *Onext* relation immediately after the operation, $splice(e_a, e_b)$ establishes the following relations between $\overline{Onext}$ and *Onext*:

(G1) $e_a \overline{Onext} = e_b Onext$.

(G2) $e_b \overline{Onext} = e_a Onext$.

(G3) $\varepsilon_a \overline{Onext} = \varepsilon_b Onext$.

(G4) $\varepsilon_b \overline{Onext} = \varepsilon_a Onext$.

(G5) $e_a Onext Flip \overline{Onext} = e_b Flip$.

(G6) $e_b Onext Flip \overline{Onext} = e_a Flip$.

(G7) $\varepsilon_a Onext Flip \overline{Onext} = \varepsilon_b Flip$.

(G8) $\varepsilon_b Onext Flip \overline{Onext} = \varepsilon_a Flip$.

(G9) $e_\gamma \overline{Onext} = e_\gamma Onext$ for all other edges e_γ in Q or Q^*.

The (G) relations are taken from [GS, section 5]. The reader is referred to [GS] for a more thorough treatment of *splice*.

Assume edges e_a and e_b are represented by boundary references $\langle a, d\rangle$ and $\langle b, d\rangle$, respectively. Operation $splice(e_a, e_b)$ is implemented in terms of the facet-edge operator *splice_edges* as follows.

```
splice( ⟨a,d⟩, ⟨b,d⟩ )
  {
  if (d = 0)
    splice_edges(aClockSpin, bClockSpin);
  else
    splice_edges(aEnext^-1, bEnext^-1);
  }
```

The duality bit d of the two arguments to *splice* are assumed to be identical — $splice(e_a, e_b)$ is defined only if e_a and e_b are both primal, or both dual.

To show the correctness of the implementation, let $\overline{Onext}$ ($\overline{Enext}$) denote the $Onext$ ($Enext$) relation immediately after $splice(e_a, e_b)$. Assume first that edges e_a and e_b, represented by the boundary references $\langle a, 0\rangle$ and $\langle b, 0\rangle$ respectively, are primal. Operation $splice\_edges(aClockSpin, bClockSpin)$ establishes relations (i) and (ii):

(i) $a\overline{Enext}^{-1} = bEnext^{-1}$

(ii) $aEnext^{-1} ClockSpin \overline{Enext}^{-1} = bClockSpin$

Relations (i) and (ii) follow readily from the (D) relations of Section 6.4, which characterize the affect of operator *splice_edges*. We use them to show that (G1,3,5,7) hold below. Relations (G2,4,6,8) are shown analogously, while (G9) follows from the limited effect of *splice_edges*.

(G1)
$$\begin{aligned} e_a \overline{Onext} &= \langle a, 0\rangle \overline{Onext} \\ &= \langle a\overline{Enext}^{-1} Fnext Clock, 0\rangle && (E3) \\ &= \langle bEnext^{-1} Fnext Clock, 0\rangle && (i) \\ &= \langle b, 0\rangle Onext && (E3) \\ &= e_b Onext. \end{aligned}$$

(G3)
$$\begin{aligned} \varepsilon_a \overline{Onext} &= \langle a, 0\rangle Onext Rot \overline{Onext} \\ &= \langle aEnext^{-1} Fnext Clock Fnext Spin, 1\rangle \overline{Onext} && (E1,2,4) \\ &= \langle aEnext^{-1} ClockSpin, 1\rangle \overline{Onext} && (B4) \\ &= \langle aEnext^{-1} ClockSpin \overline{Enext}^{-1}, 1\rangle && (E7) \\ &= \langle bClockSpin, 1\rangle && (ii) \\ &= \langle bEnext^{-1} ClockSpin Enext^{-1}, 1\rangle && (B7) \\ &= \langle bEnext^{-1} ClockSpin, 0\rangle Dual Onext && (E4) \\ &= \langle bEnext^{-1} Fnext Clock Fnext Spin, 0\rangle Dual Onext && (B4) \\ &= \langle b, 0\rangle Onext Flip Dual Onext && (E1,3) \\ &= e_b Onext Rot Onext \\ &= \varepsilon_b Onext. \end{aligned}$$

(G5)
$$\begin{aligned} e_a Onext Flip \overline{Onext} &= \langle a, 0\rangle Onext Flip \overline{Onext} \\ &= \langle aEnext^{-1} Fnext Clock Fnext Spin, 0\rangle \overline{Onext} && (E1,3) \\ &= \langle aEnext^{-1} Fnext Clock Fnext Spin \overline{Enext}^{-1} Fnext Clock, 0\rangle && (E3) \\ &= \langle aEnext^{-1} ClockSpin \overline{Enext}^{-1} Fnext Clock, 0\rangle && (B4) \\ &= \langle bClockSpin Fnext Clock, 0\rangle && (ii) \\ &= \langle bFnext Spin, 0\rangle && (B3,4,5) \\ &= \langle b, 0\rangle Flip && (E1) \\ &= bFlip. \end{aligned}$$

(G7)
$$\begin{aligned} \varepsilon_a Onext Flip \overline{Onext} &= a Onext Rot Onext Flip \overline{Onext} \\ &= aRot^{-1} Flip \overline{Onext} && (A3', 4', 5') \end{aligned}$$

$$
\begin{aligned}
&= aDual\,Flip\,Flip\,\overline{Onext} && (A1', 5', 6)\\
&= \langle a, 1\rangle \overline{Onext} && (E5)\\
&= \langle a\overline{Enext}^{-1}, 1\rangle && (E7)\\
&= \langle bEnext^{-1}, 1\rangle && (i)\\
&= \langle b, 1\rangle\, Onext && (E7)\\
&= \langle b, 0\rangle\, Dual\, Onext && (E4)\\
&= bOnext\, Sym\, Dual && (A4')\\
&= bOnext\, Sym\, Sym\, Rot\, Flip && (A5')\\
&= bOnext\, Rot\, Flip && (A1)\\
&= \varepsilon_b\, Flip.
\end{aligned}
$$

Notice that the *splice_edges* operation (in the above implementation of *splice*) only modifies facet-rings of the subdivision C_a^* and C_b^*, and *not* the facet-rings of C_a and C_b to which a and b belong respectively. Since each occurance of *Fnext* in the derivations above apply only to facet-rings of C_a, we have been free to assume that *Fnext* has not been changed by *splice_edges*; that is, $Fnext = \overline{Fnext}$.

We have shown the correctness of an implementation for *splice* when its arguments are primal edges. Assume now that *splice* is passed two *dual* edges e_a and e_b given by $\langle a, 1\rangle$ and $\langle b, 1\rangle$, respectively. Let $\varepsilon_a = e_a\, OnextRot$ and $\varepsilon_b = e_b\, OnextRot$ be represented by $\langle \alpha, 0\rangle$ and $\langle \beta, 0\rangle$, respectively. Observe that

$$
\begin{aligned}
\langle \alpha, 0\rangle &= \varepsilon_a\\
&= e_a\, Onext\, Rot\\
&= \langle a, 1\rangle\, Onext\, Rot\\
&= \langle aEnext^{-1}, 1\rangle\, Flip\, Sdual && (A5'; E7)\\
&= \langle aEnext^{-1}\, Spin\, Clock, 1\rangle\, Sdual && (E5)\\
&= \langle aEnext^{-1}\, Spin\, Clock, 0\rangle, && (E4)
\end{aligned}
$$

whence $\alpha = aEnext^{-1}\, Spin\, Clock$; similarly, $\beta = bEnext^{-1}\, Spin\, Clock$.

Since operations $splice(e_a, e_b)$ and $splice(\varepsilon_a, \varepsilon_b)$ are equivalent, and ε_a and ε_b are primal, we know $splice(e_a, e_b)$ to be correctly implemented by

$splice\_edges(\alpha\, ClockSpin, \beta\, ClockSpin)$.

But this is equivalent to

$splice\_edges(aEnext^{-1}, bEnext^{-1})$, since

$\alpha\, ClockSpin = aEnext^{-1}\, Spin\, Clock\, ClockSpin = aEnext^{-1}$, and similarly $\beta\, ClockSpin = bEnext^{-1}$. Hence $splice(e_a, e_b)$ is implemented by $splice\_edges(aEnext^{-1}, bEnext^{-1})$ where edges e_a and e_b are dual.

7.4 Melding Polyhedra Together

Being able to build individual polyhedra is not enough to permit the construction of non-trivial polyhedral subdivisions. We would like to be able to combine polyhedra to form larger subdivisions. The operator *meld*, the subject of the current section, permits this.

Operator *meld* is applied to two polyhedra p_a and p_b. The operator is handed facet-edge pairs a and b for which $facet(a) \in \partial p_a$, $facet(b) \in \partial p_b$ and $|facet(a)| = |facet(b)| = n$ (that is, n edges bound each of the facets). The *meld* operation is used to meld (coalesce or fuse) the two facets together, thereby effectively glueing together the polyhedra along a facet of each. Each polyhedron may belong to a larger subdivision. If subdivisions C_a (to which polyhedron p_a belongs) and C_b (to which p_b belongs) are distinct, the *meld* operation serves to combine the subdivisions into a single one. If C_a and C_b are the same subdivision, the operation serves to modify that subdivision.

Let a and b be facet-edge pairs with facet components $facet(a)$ and $facet(b)$. Operation $meld(a, b)$ glues together the polyhedra *aPneg* and *bPpos* along facets $facet(a)$ and $facet(b)$. If C_a and C_b are distinct, C_a is regarded as a ball-complex whose unbounded polyhedron is *aPpos*. (Combinatorially, before C_a is assigned a geometry, any one of its polyhedra — including in particular *aPpos* — can be treated as the complex's unbounded polyhedron.) The operation then effectively locates C_a inside the polyhedron *bPneg*. If C_a and C_b are the same, we assume the polyhedra *aPpos* and *bPneg*

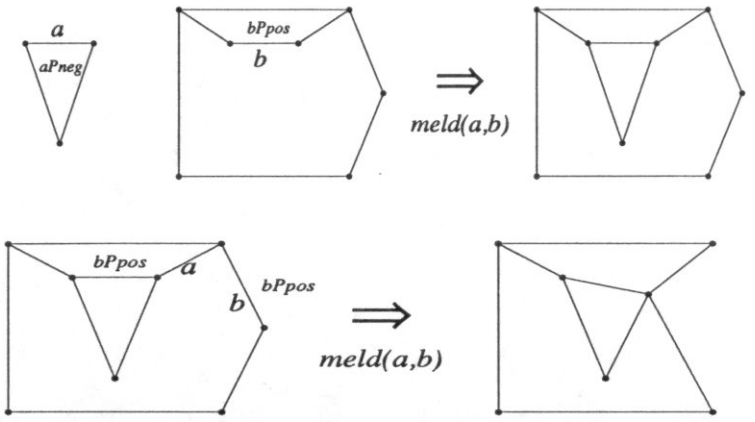


Fig. 23. These drawings illustrate a 2-dimensional analogue of the *meld* operation. The top drawing illustrates the case $C_a \neq C_b$, while the bottom drawing depicts $C_a = C_b$.

are the same. Figure 23 illustrates the 2-dimensional analogue of two applications of *meld*. In the first application $C_a \neq C_b$, while in the second $C_a = C_b$.

The edge components $edge(a)$ and $edge(b)$ of a and b are used to align the edges of facets for the *meld* operation. Facets $facet(a)$ and $facet(b)$ are fused so that edge $e_{a_i} = edge(a_i)$ fuses (lengthwise) with edge $e_{b_i} = edge(b_i)$, where $a_i = aEnext^i$ and $b_i = bEnext^i$ for $i = 0, 1, \cdots, n-1$.

The strategy for performing $meld(a, b)$ is as follows. Edges e_{a_i} and e_{b_i} are first fused (if they are distinct), for $i = 0, 1, \cdots, n-1$. This is accomplished by

for $i = 0, \cdots, n-1$
  if $(\mathcal{F}_{a_i} \not\equiv \mathcal{F}_{b_i})$
    replace $\mathcal{F}_{a_i}$ and $\mathcal{F}_{b_i}$ by $concat(\mathcal{F}_{a_i Fnext}, \mathcal{F}_{b_i})$;

This results in a "pillow" consisting of the edge-ring $\mathcal{E}_a \equiv \mathcal{E}_b$, together with the facets

$facet(a)$ and $facet(b)$. The next step is to remove $facet(a)$ from this pillow. This is done by

for $i = 0, \cdots, n-1$
  replace $\mathcal{F}_{a_i}$ by $first(split(\mathcal{F}_{a_i Fnext}, a_i))$;

In the implementation of *meld*, these facet-ring manipulations are performed with operator *splice_facets*.

The *meld* operation also changes some incidence relations involving vertices and polyhedra. In particular, in the boundary of polyhedron $aPneg$, facet $facet(a)$ is replaced by $facet(b)$ — call the resulting polyhedron p. Furthermore, if distinct, the polyhedra $aPpos$ and $bPneg$ are combined to produce a new polyhedron, which we shall call q. We then have

$$facets\_of(p) = facets\_of(aPneg) - \{facet(a)\} \cup \{facet(b)\}, \text{ and}$$
$$facets\_of(q) = facets\_of(bPneg) \cup facets\_of(aPpos) - \{facet(a), facet(b)\}.$$

where $facets\_of(p)$ denotes the set of facets in the boundary of polyhedron p. The 2-dimensional analogue of these changes in incidence are depicted in Figure 24.

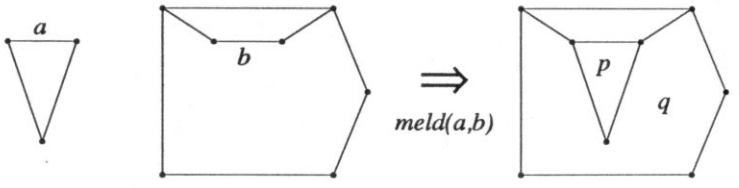


Fig. 24. This drawing depicts a 2-dimensional analogue of the polyhedra p and q formed by operation $meld(a, b)$.

Incidence relations involving vertices are also changed by $meld(a, b)$. Additional edges must be made incident to the vertices $v_i = b_i Org$ for $i = 0, 1, \cdots, n-1$. To

reflect the fusion of edges e_{a_i} and e_{b_i}, the edges incident to vertex $a_i Org$ must be made incident to v_i. However, the two edges e_{a_i} and $e_{a_{i-1}}$ must not be made incident to v_i since facet $facet(a)$ is being removed. Thus we have, for $0 \leq i \leq n-1$,

$$edges\_of(v_i) = edges\_of(b_i Org) \cup edges\_of(a_i Org) - \{e_{a_i}, e_{a_{i-1}}\}.$$

In the implementation of *meld*, these changes in incidence are performed with operator *transfer*.

The implementation of *meld* consists of a single loop in which construction of the pillow and the removal of $facet(a)$ are interleaved. The procedure is given in Figure 25.

```
meld(a, b)
  {
  firsta ← a;
  transfer(bPneg, aPpos);
  do {
    if(F_a ≢ F_b)
      splice_facets(a, bFnext^-1);
    splice_facets(a, aFnext^-1);
    transfer(bOrg, aOrg);
    transfer(aPneg, {bSdual, bSdualSpin});
    transfer(∅, {aSdual^d Clock^c Spin^r | d, c, r ∈ {0,1}});
    a ← aEnext
    b ← bEnext
    } until (a = firsta);
  }
```

Fig. 25. The procedure *meld*.

Chapter 8

Applications

8.1 Introduction

In the current chapter we consider applications of the facet-edge structure. Our purpose is two-fold. First, we wish to justify the claim that the data structure has a myriad of uses. Second, we wish to develop a couple of algorithms to a sufficient level of detail to persuade the reader that the data structure is practical.

Section 8.2 presents an algorithm for building a facet-edge representation of a 3-dimensional Voronoi diagram. In Section 8.3 we present an algorithm for decomposing a polyhedron which expresses the decomposition using the facet-edge structure. In Sections 8.4 and 8.5 we outline two additional applications which would benefit from use of the data structure. These involve the construction of *weighted* Voronoi diagrams in the plane and the manipulation of 4-polyhedra, respectively.

8.2 Constructing a 3-Dimensional Voronoi Diagram

8.2.1 The Algorithm

We describe how to build the 3-dimensional Delaunay triangulation $DT(S)$ of a finite set $S \subset \Re^3$ of $n \geq 4$ sites in general position. Since the facet-edge structure represents both a subdivision and its dual, the algorithm also serves to construct the Voronoi diagram $V(S)$. The strategy is to first construct some tetrahedron of $DT(S)$ — called a *D-tetrahedron* — to serve as an initial current cell complex C. Complex C is then grown by iteratively discovering, building and melding a new D-tetrahedron to one or more triangular facets of ∂C, until it is known that $C = DT(S)$. The algorithm is described in [AB], and under geometric inversion that maps S to a set of points S' on a

3-dimensional hypersphere in $\Re^4$ [Br], corresponds to the gift-wrapping method of [CK] for building the convex hull of S'. The process of finding an initial and subsequent D-tetrahedra is described in [AB], so we describe this only briefly in the next couple paragraphs, before presenting the entire algorithm.

Assume triangle f of $DT(S)$ is on the boundary of complex C (that is, $f \in \partial C$), and that t is the (sole) D-tetrahedron incident to f which is known. Operation *find_tetrahedron*(f,t) constructs the other D-tetrahedron t' incident to f (if it exists). Let $H_{f,t}$ denote that open half-space determined by *aff* f which does not contain t. The vertices that define t' are then the vertices of f together with site q, where $q \in H_{f,t}$ is that site of $H_{f,t}$ for which the sphere determined by q and the vertices of f is of minimal radius. It is shown in [Bh] that the interior of this sphere contains no sites, hence t' is indeed a D-tetrahedron. If $S \cap H_{f,t}$ is empty, then f lies on the convex hull of S and t' does not exist.

An initial D-tetrahedron is found by first finding some triangular facet f on the convex hull of S by the method of [CK]. The D-tetrahedron incident to f is discovered using the strategy given above, where candidate sites q range over all sites (except for the three that determine f).

We examine the algorithm *delaunay* of Figure 26, focusing upon its use of the facet-edge structure. The following definition facilitates our discussion. Let t be a D-tetrahedron and f a facet of t. Facet-edge pair a is said to *describe* t and f if

(*i*) $facet(a) = f$,

(*ii*) $aPpos = t$, and

(*iii*) the orientation of a appears counter-clockwise from beyond $facet(a)$ (that is, when viewed from half-space $H_{f,t}$).

If a describes t and f, then $aEnext$ and $aEnext^2$ do so as well.

Let F denote the set of facets for which a D-tetrahedron has been sought on exactly one side of the facet. F consists of those facets belonging to the boundary of the current

complex C, less those facets that have been determined to lie on the convex hull of S. We call each facet $f \in F$ a *candidate* facet. Dictionary $\mathcal{F}$ contains the facets of F. More precisely, for each candidate facet $f \in F$, $\mathcal{F}$ contains one facet-edge pair that describes t and f where t is the known D-tetrahedron incident to f. A look-up in the dictionary is performed by $b \leftarrow \mathcal{F}(a)$, which returns that element b of $\mathcal{F}$ whose facet component $facet(b)$ is determined by the same three vertices which determine $facet(a)$ (these being the vertices $aOrg$, $aEnext\,Org$ and $aEnext^2\,Org$). The look-up returns $\emptyset$ if no such element exists.

D-tetrahedron t is represented by any facet-edge pair which describes t and f, where f is any facet of t. The set $facet\_edges\_of(t)$ consists of four facet-edge pairs which describe t and f as f ranges over the four facets of t. The set, which is not unique, is easily obtained by applying facet-edge traversal functions to the facet-edge pair representing t. Procedure *align*(a,b), whose arguments should satisfy $facet(a) = facet(b)$, returns that facet-edge pair $a' = aEnext^i$ for which $a'Org = bOrg$. Procedures *facet_edges_of* and *align* are implemented in Appendix A.

The algorithm *delaunay* of Figure 26 constructs the Delaunay triangulation $DT(S)$ of a finite site set $S \subset \Re^3$, in general position. Lines 1 and 2 of procedure *delaunay* set the initial candidate facets to be those of some D-tetrahedron. While candidate facets exist (line 3 is true), one is retrieved (line 4) and a second D-tetrahedron t incident to the facet is sought (line 5). If t does not exist (line 6 is true), the facet belongs to the convex hull of S and is no longer considered a candidate (line 7). Otherwise each facet a of t is examined (lines 9-18). If there exists a candidate facet b for which $facet(a) = facet(b)$ (line 11 is true), then t is the second D-tetrahedron incident to $facet(b)$. In lines 12-14 facet b is no longer considered a candidate, and its two D-tetrahedra are melded together. Otherwise there exists no candidate facet b for which $facet(a) = facet(b)$ (line 11 is false), hence t is the sole *known* D-tetrahedron incident to $facet(a)$, so a becomes a candidate facet (line 17).

```
delaunay(S)
  {
  t ← an initial D-tetrahedron of S;                          1
  F ← facet_edges_of(t);                                      2
  while (F ≠ ∅) {                                             3
    a ← some element of F;                                    4
    t ← find_tetrahedron(facet(a), aPpos);                    5
    if (t does not exist)                                     6
      F ← F − a;                                              7
    else {                                                    8
      for each a ∈ facet_edges_of(t) {                        9
        b ← F(a);                                             10
        if (b ≠ ∅) {                                          11
          F ← F − b;                                          12
          a ← align(aClock, b);                               13
          meld(a, b);                                         14
          }                                                   15
        else                                                  16
          F ← F + a;                                          17
        }                                                     18
      }
    }
  }
```

Fig. 26. The procedure *delaunay*.

8.2.2 Implementation

The algorithm *delaunay* was implemented in the C programming language. The implementation of the facet-edge traversal and construction operators, quad-edge functions and operators and memory allocation routines amounts to some 1050 lines of code. Additional code required specifically for the *delaunay* application comes to about 1100 lines.

Each Delaunay tetrahedron is constructed by procedure *tetra*, which makes calls to the quad-edge operators *make_segment* and *splice*. Since it is desirable to keep track of the four sites that determine the tetrahedron, *tetra* is passed pointers to vertex nodes

v_0, v_1, v_2 and v_3 representing these four sites. It is also passed two pointers to vertices v_4 and v_5 dual to the interior and exterior of the tetrahedron. Care is taken to ensure that vectors v_0v_1, v_0v_2 and v_0v_3 form a right-handed system; if they do not, the roles of v_0 and v_1 are exchanged. This ensures that when two tetrahedra are melded, the sense of rotation between each pair of facet-rings to be spliced together (with *splice_facets*) are consistent. Procedure *tetra* is implemented in Appendix A.

Operator *make_segment* is passed four vertex pointers. Two represent the origin and destination of the segment s returned by the operator. The other two represent vertices dual to the interior and the exterior of the sphere or polyhedron to which segment s belongs.

Operator *make_segment* in turn builds a segment by obtaining two new facet-edge nodes and splicing together the facet-edge pairs represented by these nodes. The resulting construct represents a segment under the boundary representation scheme as described in Section 7.3. Each new facet-edge pair requires that the vertices and polyhedra to which each is incident are determined. Accordingly, procedure *make_facet_edge* is handed four vertex pointers, to serve as origin, destination, positive polyhedron and negative polyhedron to the facet-edge pair returned by the procedure. Under the (B) relations of Section 4.5, the vertices and polyhedra incident to each of the other facet-edge pairs represented by the node are determined.

The table of Figure 27 shows statistics regarding Delaunay triangulations built by *delaunay* for n sites chosen at random (uniformly) in the unit cube. Time refers to number of CPU seconds where the program was run on a VAX785. $\alpha_k(n)$ is the number of k-dimensional cells belonging to $DT(S)$. $\alpha_3(n)$ is the number of D-tetrahedra built, and $\alpha_2(n)$ is the number of facets. $\alpha_2^i(n)$ is the number of interior facets (incident to two distinct D-tetrahedra), while $\alpha_2^h(n)$ is the number of facets lying in the convex hull of the site set. Note that $\alpha_2^i(n)$ is equal to the number of calls made to *meld*, and that $\alpha_2(n) = 4\alpha_3(n) - \alpha_2^i(n) = \alpha_2^i(n) + \alpha_2^h(n)$. $\alpha_0^h(n)$ is the number of sites belonging to

the convex hull of S. Since the hull is a 2-dimensional triangulation over these sites, we have $\alpha_0^h(n) = (\alpha_2^h(n) + 4)/2$.

nbr sites n	secs	α_3	α_2	α_2^i	α_2^h	α_0^h
25	0.15	84.3	182.7	154.5	28.2	16.1
50	0.44	222.8	466.1	425.1	41.0	22.5
75	0.57	366.2	758.5	706.3	52.2	28.1
100	1.05	516.2	1061.7	1003.1	58.6	31.3
250	3.22	1457.0	2954.6	2873.4	81.2	42.6
500	15.32	3058.6	6172.6	6061.8	110.8	57.4
1000	23.56	6341.0	12753.4	12610.6	142.8	73.4

Fig. 27. Statistics of Delaunay triangulation produced by *delaunay*. The first five rows of the table indicate averages over 10 triangulations, the last two rows indicate averages over 5 triangulations.

The worst-case time complexity of *delaunay* is dominated by the $\alpha_2(n)$ calls to procedure *find_tetrahedron*, each call taking time $O(n)$. Since the number of facets $\alpha_2(n) = O(n^2)$, the overall time complexity is $O(n^3)$. However, in practice $\alpha_2(n)$ tends to be linear in n as indicated in Figure 27, so the expected-case time complexity appears no worse than quadratic. Details of the analysis are provided in [AB, Bh].

Appendix A presents the implementation of those procedures used to build the Voronoi diagram which directly involve the facet-edge data structure. Other functions independent of choice of data structure, such as for determining Delaunay tetrahedra, can be implemented based upon the references already cited. Appendix B presents steropsis diagrams of some 3-dimensional Voronoi diagrams.

8.3 Decomposing a Polyhedron

The process of partitioning a polyhedron into simpler constituent polyhedra is called *decomposition*. One reason for decomposing a polyhedron $\overline{p}$ is that $\overline{p}$ may possess

properties that preclude certain algorithms from being applied to it — for instance, it may be non-convex, or possess cavities or handles. Sometimes the difficulty may be overcome by decomposing $\overline{p}$ into more primitive pieces, and then applying the algorithm to *these*. Instances of this are cited in [Ch].

There are various strategies for performing decomposition. We will concern ourselves with an incremental strategy in which polyhedra are iteratively detached from the original polyhedron $\overline{p}$ until nothing remains of the original. Each piece split off from the original is not subject to further decomposition, and satisfies whatever "simplicity" criteria is required of the algorithm. Such an algorithm maintains a current polyhedron S (initially $\overline{p}$), and a current complex C (initially $\emptyset$). The algorithm iteratively detaches a polyhedron p_i (in the i^{th} iteration) from S and transfers it to C. The process stops when S represents a null polyhedron — complex C then represents the decomposition of $\overline{p}$. In the present section, we show how collection C assembled during the course of decomposition can be represented by the facet-edge structure. Each polyhedron detached from S is attached to C by *meld* operations. For simplicity, we assume that S is always polyhedral in the sense we have been using the word (in particular, having genus zero and no cavities), and that C always consists of zero or more ball-complexes.

Wördenweber uses this incremental strategy in [Wö] to decompose a polyhedron into tetrahedra. He makes no attempt to assemble the pieces, but allows the sequence of operations by which they were detached to represent the resulting decomposition. We briefly describe Wördenweber's algorithm to locate this incremental strategy in a concrete setting. We refer the reader to [Wö] for a description of how the algorithm selects a tetrahedron to be detached from the current polyhedron S in each iteration. The actual removal of the tetrahedron from S is accomplished by one of the four operators *op0*, *op1*, *op2* or *op3*. Each *opk* modifies the polyhedron S to reflect the removal of the tetrahedron. The index k of *opk* indicates how the tetrahedron t that

opk is designed to detach is related to the rest of S: tetrahedron t has k "buried" facets (by which it is attached to the rest of S), and $4-k$ "exposed" facets (which belong to the boundary of S). To reflect S's loss of t, *opk* replaces the exposed facets by the buried facets in the representation of S. (We can assume that S is represented by a data structure suitable for handling subdivisions of the sphere, say by the quad-edge structure.) Some of the exposed facets correspond to facets of the boundary ∂C of the current complex. Where $f \in S$ is such a facet, $\tilde{f} \in \partial C$ denotes that facet to which f corresponds. Facets f and $\tilde{f}$ will have been created when some tetrahedron p_j was transferred from S to C in some earlier iteration j. More generally, cell $\tilde{c} \in \partial C$ corresponds to cell $c \in S$. The remaining exposed facets of t belong to the boundary of the original polyhedron $\overline{p}$, and corresponds to none of the facets of ∂C. Figure 28 illustrates the affect each *opk* has upon S.

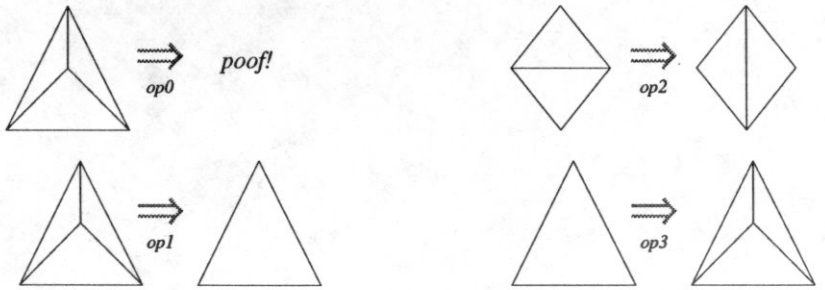


Fig. 28. This figure illustrates how each *opk* locally modifies S to produce S'. Each drawing depicts a region on the boundary of S (before the operation) and S' (after the operation).

A decomposition algorithm employing the incremental strategy requires the use of an operator *op*, analogous to Wördenweber's *opk* operators, for transfering a polyhedron p_i from S to C. The operator must build a facet-edge representation for p_i, attach p_i

to C (using calls to *meld*), and modify S (to reflect it's loss of p_i). The operator's most formidable task is in determining exactly how p_i is to be attached to C — that is, in determining the arguments to each of its calls to *meld*. To guide *op* in attaching p_i to C, each facet $f \in S$ possesses a *link pointer* $link(f)$ which references that facet $\tilde{f} \in \partial C$ to which f corresponds. The facets of S that belong to the boundary of the original polyhedron $\overline{p}$ do not correspond to any facet of C, and so have null link pointers. To summarize, in iteration i, *op* performs the following steps in succession:

(i) constructs a facet-edge representation for polyhedron p_i, to be transferred from S to C,

(ii) attaches p_i to C, thereby forming C' (to serve as C in the next iteration),

(iii) modifies S to reflect its loss of p_i, thereby forming S' (to serve as S in the next iteration), and

(iv) updates the link pointers of S'.

We do not elaborate on steps (i) and (iii). Presumably the description of p_i handed to *op* is adequate for these steps to be performed using the edge operators of the quad-edge structure. Steps (ii) and (iv) do require elaboration. Henceforth denoting by p the polyhedron p_i constructed in step (i), we discuss how we ascertain which facets of p are to be melded to C, how the link pointer is used to guide each *meld* operation, and how the link pointers are updated in S' to serve later iterations.

Consider the relationship between p and S. That patch of p to be glued to C coincides with subcomplex $S_p = \bigcup\{\partial^* f | f \in S \text{ is exposed in } p\}$. (Recall that the combinatorial closure $\partial^* f$ is the complex consisting of the faces of cell f; since f is here a facet, it consists of f and the vertices and edges that bound f.) Subcomplex S_p is generally a patch of S, homeomorphic to a closed disk. (In the final iteration in which S itself is transferred to C, we have $S_p = S$.) We denote by $\phi(c)$ that cell of p that coincides with cell $c \in S_p$. The mapping $\phi : S_p \rightarrow p$ is one-to-one, though not generally onto. Consider next the relationship between p and S'. That patch of p that lies in $\partial C'$ (after

op has attached p to C) coincides with subcomplex $S'_p = \bigcup\{\partial^* f | f \in S' \text{ is buried in } p\}$. (After the last iteration however, $S' = S'_p = \emptyset$). We denote by $\phi'(c)$ that cell of p that coincides with cell $c \in S'_p$. The mapping $\phi' : S'_p \rightarrow p$ is one-to-one, though not onto. The patches $\phi(S_p)$ and $\phi'(S'_p)$ cover polyhedron p. Their intersection $\phi(S_p) \cap \phi'(S'_p)$ is a vertex-edge cycle in p, called the *silhouette* of p. These notions are depicted in Figure 29.

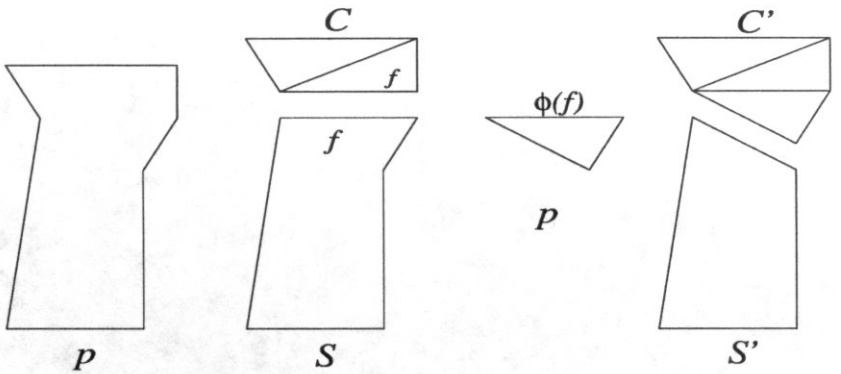


Fig. 29. This figure depicts a 2-dimensional analogue of the effect of *op*. Each edge of the figure corresponds to a facet, and each polygon to a polyhedron.

To attach p to C, for each facet $f \in S_p$, facet $\phi(f) \in p$ is melded to facet $\tilde{f} \in C$ using the link pointer associated with f. Pointer $link(f)$ consists of the two fields *edgeofS* and *edgeofC*, whose contents are as follows.

$link(f).edgeofS$: an edge reference to directed and oriented edge $e \in S$ such that

$$eLeft = f.$$

$link(f).edgeofC$: a boundary reference to edge $e' \in \partial C$ such that

(i) $e'Left = \tilde{f}$,

(ii) $e'Lnext^i = \tilde{e}Lnext^i$ for all i.

Facets $\phi(f) \in p$ and $\tilde{f} \in \partial C$ are coalesced by the following code:


```
if (link(f) ≠ ∅) {
  a ← facet-edge pair component of boundary reference link(f).edgeofC;
  e ← link(f).edgeofS;
  b ← facet-edge pair component of the boundary reference to φ(e);
  meld(a, b);
  }
```

Since $e' = \langle a, 0\rangle \in \partial C$, $aPneg$ is the sole polyhedron of C incident to facet $e'Left$. Since $\phi(e) = \langle b, 0\rangle$ is an edge in ∂p, $bPpos = p$. Hence the operation $meld(a, b)$ glues polyhedron p to the unique polyhedron of C that is incident to $\tilde{f}$. Regarding field $link.edgeofC$, condition (i) ensures that the polyhedra are glued together along the correct facets, while condition (ii) ensures that their edges are properly aligned.

Each facet of S_p is obtained by treating the 2-dimensional subdivision S^*_p as a graph, and performing an exhaustive search in S^*_p. Each vertex visited corresponds to a facet of S_p. The edges $\phi(e)$ required by the code above are obtained by performing an identical graph search in the patch $\phi(S^*_p) \subset p^*$, coincident with the search in S^*_p. It is necessary to restrict the respective searches to S^*_p and $\phi(S^*_p)$. The silhouette of p is used to do this. Specifically, the search algorithm considers two vertices adjacent in $\phi(S^*_p)$ iff the edge that connects them is not dual to a silhouette edge of p.

Having attached p to C and modified S to produce S', the link pointers of S' must be updated. This involves setting the link pointer of each facet created by *op* (these being the buried facets of p); the link fields of the other facets of S' are still correct. Much as before, we perform a graph search in S'^*_p and a coinciding search in $\phi(S'^*_p) \subset p^*$, using the silhouette of p to limit both searches. When we visit a vertex of S'^*_p, dual say to facet $f \in S'_p$, $link(f)$ is set by the following:

```
let e be an edge for which eLeft = f;
link(f).edgeofS ← e;
link(f).edgeofC ← boundary reference to φ'(e);
```

The isomorphisms ϕ and ϕ' are each computed on the fly by performing identical searches in two distinct graphs. Each pair of searches must start at coinciding cells for each isomorphism to be correctly computed. To do this, we select some edge $e \in S$ for which $\phi(e)$ is a silhouette edge. Since e belongs to both S_p and S'_p, it can be used to compute the starting point for both pairs of searches. Let $e \in S$ have arbitrary orientation and direction. We wish first to determine the direction and orientation of $\phi(e)$ so that $\phi(e\,Ops) = \phi(e)\,Ops$ for any sequence Ops of edge functions restricted to S_p. Suppose p were embedded in S so that $\phi(S_p)$ coincides with S_p, and the interior of p is contained by the interior of S. Choosing that version of $\phi(e)$ for which $\phi(e)\,Org = \phi(e\,Org)$ and $\phi(e)\,Left = \phi(e\,Left)$ ensures that the direction and sense of rotation (when viewed from beyond patch S_p of e and $\phi(e)$ agree.

It is also necessary to determine the direction and orientation of $\phi'(e)$ so that $\phi'(e\,Ops) = \phi'(e)\,Ops$ for any sequence Ops of edge functions restricted to S'_p. Where p is embedded in S as above, the interior of p lies beyond the interior of S'. Polygon $\phi(e)\,Right \subset p$ corresponds to the sole facet of S'_p incident to edge $e \in S'$. We choose that version of $\phi'(e)$ for which $\phi'(e)\,Org = \phi(e)\,Org$ and $\phi'(e)\,Left = \phi(e)\,Right$. Observe that we have $\phi'(e) = \phi(e)\,Flip$. Intuitively, traversal of the exposed patch $\phi(S_p) \subset p$ is begun at the exposed facet incident to silhouette edge $\phi(e)$, whereas traversal of the buried patch $\phi'(S'_p) \subset p$ is begun at the buried facet incident to $\phi(e)$.

8.4 Constructing Weighted Voronoi Diagrams in the Plane

Let S be a finite set of sites in the plane with metric δ. One generalization of the Voronoi diagram associates a positive weight $w(s)$ with each site $s \in S$, which expresses the site's power to influence its neighborhood. The weighted distance between an arbitrary point $x \in \Re^2$ and site s is $\delta_w(x,s) = \delta(x,s)/w(s)$. The *weighted Voronoi diagram* $WV(S)$ is the subdivision of the plane in which each site s is associated with the region consisting of all points $x \in \Re^2$ for which s is a nearest weighted site. The Voronoi regions are possibly disconnected regions bounded by circular arcs. The Voronoi diagram of

Chapter 3 is a special case of the weighted Voronoi diagram in which all sites are weighted equally. Other properties are sited in [AE].

Aurenhammer and Edelsbrunner give an algorithm in [AE] for constructing $WV(S)$. The algorithm employs a geometric transform to reduce the original problem to that of constructing a specific polyhedral subdivision. Each weighted site s corresponds to a polyhedron $p(s)$ of the subdivision. The subdivision is constructed by successively inserting the weighted sites. Under the geometric transform, the insertion of site s corresponds to fitting polyhedron $p(s)$ into the current polyhedral subdivision C. This can be accomplished by intersecting the cells of the subdivision C by each of the planes that determine $p(s)$, then eliminating those resulting cells that lie in the interior of $p(s)$.

Intersecting a plane $p\ell$ with C reduces to intersecting $p\ell$ with each of the polyhedra p of C. Detecting the edges of $p \cap p\ell$ is easy since ∂p can be treated as a 2-dimensional subdivision under the boundary representation scheme. Visiting all the polyhedra of C reduces to an exhaustive search in the graph C^* — each vertex encountered corresponds to a polyhedron of C. (In fact, we need only visit those polyhedra that $p\ell$ intersects.) Eliminating the cells of the interior of polyhedron $p(s)$ requires an exhaustive search of the interior restricted by the facets of $p(s)$.

8.5 Manipulating 4-Polyhedra

A (3-dimensional) convex polyhedron is commonly represented by its boundary. By projecting the boundary, which lies in $\Re^3$, onto a plane, the polyhedron can be drawn on paper and readily grasped. The appearance of the drawing varies with the choice of the center of projection and the image plane. One good choice (which precludes overlapping edges) locates the center of projection just beyond some facet, and designates the image plane to be the plane determined by the facet. The resulting projection is what we would see were we to remove the facet and look at the polyhedron through the hole.

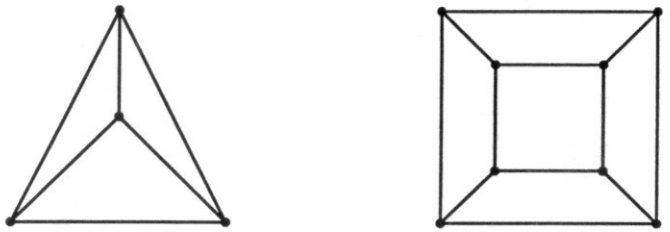


Fig. 30. Projection of the tetrahedron (left) and the cube (right).

Figure 30 depicts a projection of the tetrahedron and the cube. The unbounded region in which each figure lies corresponds to the facet which determines the image plane.

A 4-dimensional convex polyhedron (or 4-polyhedron) may be formed by the intersection of finitely many halfspaces of $\Re^4$. It too may be represented by its boundary, a subdivision of the 3-dimensional hypersphere. To envision the boundary, we may perform a projection into $\Re^3$ analogous to the one described above, one dimension higher. The center of projection is located just beyond a 3-cell of the boundary, and the image space is the hyperplane which contains that 3-cell. Figure 31 depicts the projection of three different regular 4-polyhedra. The unbounded space in which each of the figures lies corresponds to the 3-cell which determines the image space. The projection is a subdivision of $\Re^3$ and can be handled by the facet-edge structure. These notions are elucidated in the wonderful book [HC].

There exist numerous uses for the facet-edge structure's capacity to handle the boundaries of 4-polyhedra. One application is that of rendering an image of a 4-polyhedron. A second application involves the rendering of successive images of 3-polyhedra that move over time. Each polyhedron, over the duration of its existence, can be represented by a 4-polyhedron in x, y, z, t-space. The representation could be used to perform fast hidden surface removal of successive scenes since it captures the polyhedron's spatial and temporal coherence. It can be used for spatial and temporal

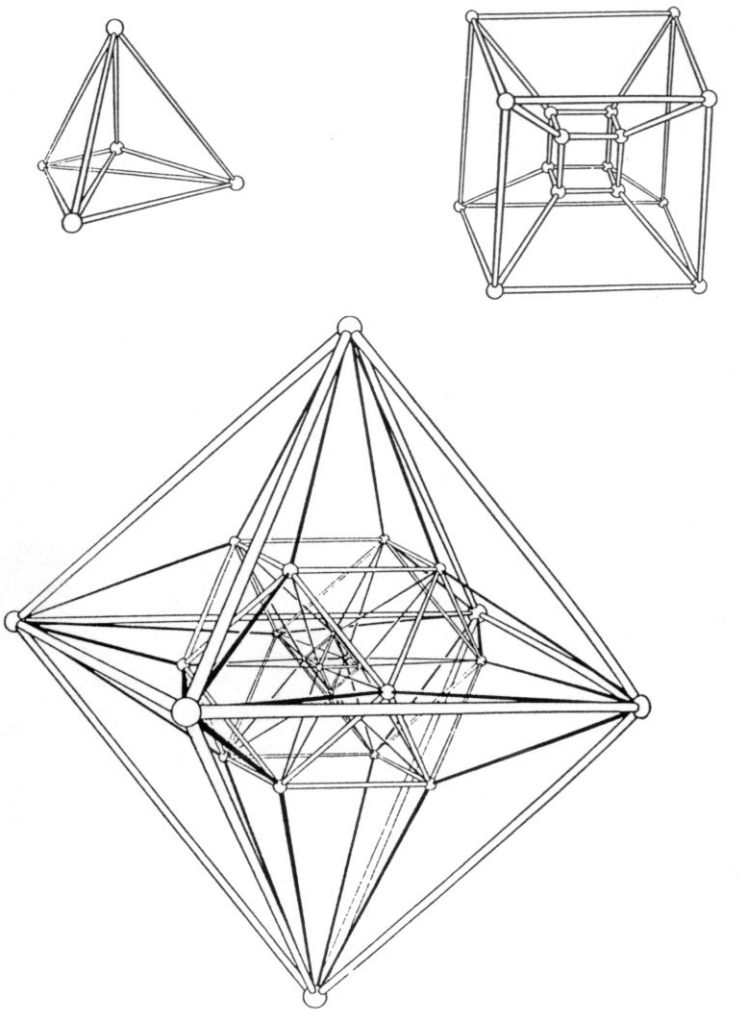


Fig. 31. Projections of the regular 4-polyhedra whose boundaries consist of five 3-cells (upper left), eight 3-cells (upper right) and twenty-four 3-cells (bottom), from [HC].

anti-aliasing [Gr].

The facet-edge structure can also be used in algorithms that mimic other algorithms, one dimension higher. Consider the problem of building the convex hull of a finite point set in $\Re^4$. A divide-and-conquer algorithm for doing the same in $\Re^3$ is given in [PH]. It represents a 3-polyhedron by its boundary, and relies crucially upon traversal queries of the sort supported by the quad-edge structure. The capacity of the facet-edge structure to support queries of 4-polyhedra suggests that it could be used in a similar algorithm one dimension higher.

Seidel describes a shelling method for constructing the convex hull of a point set in $\Re^d$ [Se]. A current partial hull is iteratively grown along its *horizon* until it coincides with the complete hull. The method employs a horizon graph (described in Section 1.2) to represent the boundary of the partial hull. For building a hull in $\Re^4$, use of the facet-edge structure obviates the need for the horizon graph. The horizon is simply the boundary of the partial hull. Traversal in this boundary yields information needed to grow the partial hull further. Polyhedra belonging to the convex hull can be *meld*ed to the boundary as each is discovered. The final hull produced by the algorithm is represented by the facet-edge structure.

Chapter 9

Conclusion

In this dissertation we have presented a new data structure for handling polyhedral subdivisions. We feel confident that this work is significant. Problems requiring the manipulation of polyhedral subdivisions abound, and the data structure is well suited for many of these problems. One of the purposes of this thesis has been to substantiate this claim.

The traversal functions for formulating queries concerning subdivisions provide a language or nomenclature for describing the manipulation of subdivisions. The nomenclature could gain concensus, thereby supplanting the more awkward descriptions of subdivision manipulation that occur in the literature. It also provides designers of alternative data structures with a set of query functions to shoot for.

We have presented construction primitives that permit one to build representations of polyhedral subdivisions. We have reduced construction operators for building individual polyhedra to these primitives. In addition, we have defined the *meld* operator for gluing together polyhedra. The task of construction is thereby reduced to an intuitively appealing scheme: that of assembling blocks each of which may be custom designed.

We have described an implementation of the data structure. This includes a scheme for representing a polyhedral subdivision, and concise implementation of each traversal and construction operator. Working code for the construction of a 3-dimensional Voronoi diagram is included in the thesis, and aspects of this application are discussed.

There are a number of questions and problems suggested by our work. One important problem is that of formulating conditions for the use of the facet-edge construction primitives, which ensure that a valid subdivision is produced. We have not treated this

issue, blithely pushing the burden onto the application using these primitives. Given a class of polyhedral subdivision which the facet-edge structure is capable of handling, can we define a set of construction operators with respect to which the class is spanned yet closed?

A second problem is that of characterizing the class of polyhedral subdivisions that the facet-edge structure is capable of handling. In this dissertation, we have restricted our interest to subdivisions of 3-balls and of Euclidean space. Can the data structure be used to manipulate subdivisions of other spaces, say subdivisions of projective space for instance? Is the structure suitable for handling subdivisions of non-orientable spaces? Were we to start at an edge e (with right-handed spin) and traverse an orientation-reversing loop of a non-orientable subdivision, we should return to edge e with left-handed spin. Can the construction operators be used to ensure that this occurs iff an orientation-reversing path is traversed? (Discussion of non-orientable spaces, and of 3-dimensional spaces generally, is presented in a remarkably intuitive fashion in [We].)

One direction for future work is that of designing a data structure for handling higher-dimensional subdivisions. Consider, for example, a 4-dimensional subdivision. The 3-cells incident to a common (2-dimensional) facet occur in a natural cyclic order about that facet, as do the edges that bound the facet. We define a triple to consist of a 3-cell p, facet f and edge e, where p and e are incident to f. To traverse to an adjacent triple, we may move around either of the two rings determined by f. Alternatively, we may fix 3-cell p and permit traversal among the facet-edge pairs that comprise the boundary of p. (This type of traversal corresponds under duality to fixing edge e and traversing about the facet–3-cell pairs incident to e.) The triple is represented by a node which is threaded into the two rings determined by facet f, and into the boundaries of the 3-cells p and e^*. The resulting data structure is clearly more complicated than the facet-edge structure, yet it embodies some of the ideas presented in this dissertation. Details have yet to be worked out.

Another question involves the general applicability of the data structure. The concern attends the design of most any new data structure. What existing algorithms become more elegant or efficient when formulated in terms of the data structure? What new algorithms are suggested by the data structure?

Appendix A

An Implementation of the Facet-Edge Structure

This appendix gives the implementation of some of the procedures described in the dissertation. The code presented here was used to build 3-dimensional Voronoi diagrams. Additional code specific to this particular application (discussed in Section 8.2 of the thesis) is also presented in this appendix. The sections of Appendix A are arranged as follows.

A.1 Structure definitions.

A.2 Facet-edge traversal functions.

A.3 Facet-edge construction operators.

A.4 Quad-edge traversal functions.

A.5 Quad-edge construction operators.

A.6 Procedures specific to the construction of 3-dimensional Voronoi diagrams.

Appendix A.1

Facet-Edge Structures

This section defines structures used by the facet-edge data structure. In the *deluanay* application, it is convenient to represent each class of the origin partition by a linked-list of the facet-edge pairs belonging to the class. Given facet-edge pair $(n, r, 0)$, fields *nnext* and *nprev* of array element *n*[*r*] reference the next and previous facet-edges in the linked-list associated with vertex $v = (n, r, 0)Org$. Field *origin* points to the node associated with vertex v. Field *head* of the vertex node of v references some facet-edge pair with origin v.

```
    /* facet-edge reference structure */
typedef struct facet_edge_reference {
    struct quarter_node *n;
    int r;    /* rotation */
    int s;    /* spin */
} FACET_EDGE_REFERENCE;


    /* quarter-node: an element of the facet-edge node */
typedef struct quarter_node {
    struct facet_edge_reference next;
    struct facet_edge_reference nnext;
    struct facet_edge_reference nprev;
    struct vertex *origin;
} QUARTER_NODE;


    /* facet-edge node: an array of four quarter-nodes */
typedef struct quarter_node *FACET_EDGE_NODE;


    /* vertex node */
typedef struct vertex {
    float x, y, z;    /* location of vertex */
    int type;    /* type of vertex: VORONOI or SITE */
    struct facet_edge_reference head;
} VERTEX;


    /* segment reference structure */
typedef struct segment_reference {
    struct facet_edge_ref r;
    int d;    /* duality */
} SEGMENT_REFERENCE;
```

Appendix A.2

Facet-Edge Traversal Functions

This section contains some facet-edge traversal functions. Each is applied to the first argument a. If the second argument is non-zero, it is the address of a *struct facet_edge_reference* into which the traversed-to facet-edge pair is to be placed. If the second argument is zero, the function allocates its own structure and puts the result there. It is okay for first and second arguments to address the same location. A pointer to the result is returned.

```
  static FACET_EDGE_REFERENCE *temp;
static FACET_EDGE_REFERENCE rtemp;


   FACET_EDGE_REFERENCE *
fnext(a, b)
FACET_EDGE_REFERENCE *a;
FACET_EDGE_REFERENCE *b;
{   /* fnext */
    s = a -> s;
    temp = b ? b :(FACET_EDGE_REFERENCE *)
        get_node(FACET_EDGE_REFERENCE_TYPE);
    *temp = (a -> n)[(a -> r + 2 * (a -> s))%4].next;
    if (s == 1) {
        temp -> r = (temp -> r + 2)%4;
        temp -> s = (temp -> s + 1)%2;
    }
    return (temp);
}   /* fnext */


   FACET_EDGE_REFERENCE *
spin(a, b)
FACET_EDGE_REFERENCE *a;
FACET_EDGE_REFERENCE *b;
{   /* spin */
    temp = b ? b :(FACET_EDGE_REFERENCE *)
        get_node(FACET_EDGE_REFERENCE_TYPE);
    *temp = *a;
    temp -> s = (temp -> s + 1)%2;
    return (temp);
}   /* spin */


   FACET_EDGE_REFERENCE *
srot(a, b)
FACET_EDGE_REFERENCE *a;
FACET_EDGE_REFERENCE *b;
{   /* srot */
```



```
    temp = b ? b :(FACET_EDGE_REFERENCE *)
        get_node(FACET_EDGE_REFERENCE_TYPE);
    *temp = *a;
    temp -> r = (a -> r + 1 + 2 * (a -> s))%4;
    return (temp);
}   /* srot */


   FACET_EDGE_REFERENCE *
clock(a, b)
FACET_EDGE_REFERENCE *a;
FACET_EDGE_REFERENCE *b;
{   /* clock */
    temp = b ? b :(FACET_EDGE_REFERENCE *)
        get_node(FACET_EDGE_REFERENCE_TYPE);
    *temp = *a;
    temp -> r = (temp -> r + 2)%4;
    return (temp);
}   /* clock */


   FACET_EDGE_REFERENCE *
enext(a, b)
FACET_EDGE_REFERENCE *a;
FACET_EDGE_REFERENCE *b;
{   /* enext */
    temp = b ? b :(FACET_EDGE_REFERENCE *)
        get_node(FACET_EDGE_REFERENCE_TYPE);
    *temp = *a;
    sdual(temp, temp);
    fnext(temp, temp);
    sdual(temp, temp);
    return (temp);
}   /* enext */


   FACET_EDGE_REFERENCE *
eprev(a, b)
FACET_EDGE_REFERENCE *a;
FACET_EDGE_REFERENCE *b;
{   /* eprev */
    temp = b ? b :(FACET_EDGE_REFERENCE *)
        get_node(FACET_EDGE_REFERENCE_TYPE);
    *temp = *a;
    clock(temp, temp);
    enext(temp, temp);
    clock(temp, temp);
    return (temp);
}   /* eprev */


   FACET_EDGE_REFERENCE *
```

```
fprev(a, b)
FACET_EDGE_REFERENCE *a;
FACET_EDGE_REFERENCE *b;
{   /* fprev */
    temp = b ? b :(FACET_EDGE_REFERENCE *)
        get_node(FACET_EDGE_REFERENCE_TYPE);
    *temp = *a;
    clock(temp, temp);
    fnext(temp, temp);
    clock(temp, temp);
    return (temp);
}   /* fprev */


   FACET_EDGE_REFERENCE *
sdual(a, b)
FACET_EDGE_REFERENCE *a;
FACET_EDGE_REFERENCE *b;
{   /* sdual */
    temp = b ? b :(FACET_EDGE_REFERENCE *)
        get_node(FACET_EDGE_REFERENCE_TYPE);
    *temp = *a;
    srot(temp, temp);
    spin(temp, temp);
    return (temp);
}   /* sdual */
```


Appendix A.3

Facet-Edge Construction Operators

This section defines the primitive facet-edge construction operators.

```
    /* make_facet_edge obtains and initializes a new facet-edge node and places the reference to
    the node's canonical facet-edge pair a into location ref. If ref is zero, it allocates memory for
    a reference and places the reference there. Array vertices contains pointers to the vertices
    to which facet-edge pair a is incident: Vertices[0] ≡ aOrg, v[1] ≡ aPneg, v[2] ≡ aDest,
    v[3] ≡ aPpos. If v[i] ≡ 0, then a new vertex node is obtained and used as v[i]. A pointer to
    the facet-edge reference of a is returned. */
  FACET_EDGE_REFERENCE *
make_facet_edge(vertices, ref)
VERTEX *vertices[];
FACET_EDGE_REFERENCE *ref;
{   /* make_facet_edge */
    register int i;
    FACET_EDGE_NODE n;
    FACET_EDGE_REFERENCE *r, *tempref;
    VERTEX *v;
    n =(FACET_EDGE_NODE) get_node(FACET_EDGE_NODE_TYPE);
    for (i = 0; i < 4; i++) {
        r = &(n[i].next);
        r -> n = n;
        r -> r = i;
        r -> s = 0;
        v = vertices[i] ? vertices[i] :(VERTEX *) get_node(VERTEX_NODE_TYPE);
        transfer(v, r);
    }
    tempref = ref ? ref :(FACET_EDGE_REFERENCE *)
        get_node(FACET_EDGE_REFERENCE_TYPE);
    *tempref = n[0].next;
    return (tempref);
}   /* make_facet_edge */


    /* splice_facets splices together the facet-rings of the facet-edge pairs referenced by aa and bb.
    */
splice_facets(aa, bb)
FACET_EDGE_REFERENCE *aa;
FACET_EDGE_REFERENCE *bb;
{   /* splice_facets */
    FACET_EDGE_REFERENCE *a, *b, *alpha, *beta;
    FACET_EDGE_REFERENCE ap, bp, alphap, betap, t;
    int sigma, sigmaprime;
      /* setting a, b, alpha and beta */
    a = &((aa -> n)[(aa -> r + 2 * (aa -> s))%4].next);
    b = &((bb -> n)[(bb -> r + 2 * (bb -> s))%4].next);
    fnext(aa, &t);
```

```
    clock(&t, &t);
    sigma = t.s;
    alpha = &((t.n)[(t.r + 2 * sigma)%4].next);
    fnext(bb, &t);
    clock(&t, &t);
    sigmaprime = t.s;
    beta = &((t.n)[(t.r + 2 * sigmaprime)%4].next);
      /* setting ap, bp, alphap and betap */
    fnext(bb, &ap);
    if (aa → s ≡ 1) {
        clock(&ap, &ap);
        spin(&ap, &ap);
    }
    fnext(aa, &bp);
    if (bb → s ≡ 1) {
        clock(&bp, &bp);
        spin(&bp, &bp);
    }
    if (sigma ≡ 0)
        clock(bb, &alphap);
    else
        spin(bb, &alphap);
    if (sigmaprime ≡ 0)
        clock(aa, &betap);
    else
        spin(aa, &betap);
      /* resetting next fields of appropriate quarter-nodes */
    *a = ap;
    *b = bp;
    *alpha = alphap;
    *beta = betap;
}   /* splice_facets */


    /* splice_edges splices together the edge-rings of the facet-edge pairs referenced by aa and bb.
    */
splice_edges(aa, bb)
FACET_EDGE_REFERENCE *aa;
FACET_EDGE_REFERENCE *bb;
{   /* splice_edges */
    FACET_EDGE_REFERENCE a, b;
    sdual(aa, &a);
    sdual(bb, &b);
    splice_facets(&a, &b);
}   /* splice_edges */


    /* transfer makes vertex v the origin of facet-edge pair r. */
transfer(v, r)
VERTEX *v;
FACET_EDGE_REFERENCE *r;
{   /* transfer */
```



```
    register QNODE *n;
    register QNODE *tempn;
    n = r → n + r → r;
      /* nothing to be done perhaps */
    if (n → origin ≡ v)
        return;
      /* delete quarter-node n from its ring */
    tempn = n → nprev.n;
    if (n → origin ≡ 0)
        ;
    else if (tempn ≡ n)
            n → origin → head.n = 0;
    else {
        (n → nprev.n)[n → nprev.r].nnext = n → nnext;
        (n → nnext.n)[n → nnext.r].nprev = n → nprev;
        n → origin → head = n → nprev;
    }
      /* insert quarter-node n into the ring of v */
    if (v ≡ 0)    /* node drops out */
        return;
    if (v → head.n ≡ 0) {
        n → nnext = n → nprev = *r;
        v → head = *r;
    }
    else {
        tempn = v → head.n + v → head.r;
        n → nprev = v → head;
        n → nnext = tempn → nnext;
        tempn → nnext = *r;
        (n → nnext.n)[n → nnext.r].nprev = *r;
    }
    n → origin = v;
}   /* transfer */


    /* Transfer reassigns each facet-edge pair with origin v1 the new origin v0. */
Transfer(v0, v1)
VERTEX *v0;
VERTEX *v1;
{   /* Transfer */
    FEREF *r;
    if (v0 ≡ v1)
        return;
    r = &(v1 → head);
    while (r → n ≠ 0)
        transfer(v0, r);
}   /* Transfer */
```

Appendix A.4

Quad-edge Traversal Functions

This section contains the quad-edge edge traversal functions. Each function is applied to the edge a referenced by the first argument. The traversed-to edge is stored in the second argument b. If b is zero, the function obtains a *struct segment_reference* into which the result is placed. It is okay for first and second arguments to address the same location. A pointer to the result is returned.

```
    static FEREF *temp;
  static SREF *s;


    SEGMENT_REFERENCE *
flip(a, b)
SEGMENT_REFERENCE *a;
SEGMENT_REFERENCE *b;
{   /* flip */
    s = b ? b :(SEGMENT_REFERENCE *) get_node(SEGMENT_REFERENCE_TYPE);
    *s = *a;
    if (s → d ≡ 0) {
        fnext(&s → r, &s → r);
        spin(&s → r, &s → r);
    }
    else {
        spin(&s → r, &s → r);
        clock(&s → r, &s → r);
    }
    return (s);
}   /* flip */


  SEGMENT_REFERENCE *
onext(a, b)
SEGMENT_REFERENCE *a;
SEGMENT_REFERENCE *b;
{   /* onext */
    s = b ? b :(SEGMENT_REFERENCE *) get_node(SEGMENT_REFERENCE_TYPE);
    *s = *a;
    eprev(&s → r, &s → r);
    if (s → d ≡ 0) {
        fnext(&s → r, &s → r);
        clock(&s → r, &s → r);
    }
    return (s);
}   /* onext */


  SEGMENT_REFERENCE *
rot(a, b)
```



```
SEGMENT_REFERENCE *a;
SEGMENT_REFERENCE *b;
{   /* rot */
    s = b ? b :(SEGMENT_REFERENCE *) get_node(SEGMENT_REFERENCE_TYPE);
    *s = *a;
    flip(s, s);
    dual(s, s);
    return (s);
}   /* rot */
```

Appendix A.5

Quad-edge Construction Operators

This section defines some of the quad-edge construction operators.

```
    /* make_segment builds an elementary subdivision of the sphere. A reference to the segment s
       is placed into *s, else into allocated memory if s is zero. Array vertices contains pointers to
       two endpoints of the segment, and to (the duals to) the interior and exterior of the sphere to
       which the segment belongs: vertices[0] = sOrg, vertices[1] = sDest, vertices[2] = inside of
       sphere, vertices[3] = outside of sphere. A pointer to the reference of segment s is returned. */
  SEGMENT_REFERENCE *
make_segment(vertices, s)
VERTEX **vertices;
SEGMENT_REFERENCE *s;
{   /* make_segment */
    FACET_EDGE_REFERENCE a, b;
    SEGMENT_REFERENCE *stemp;
    VERTEX *v[4];
    v[0] = vertices[0]; v[1] = vertices[3];
    v[2] = vertices[1]; v[3] = vertices[2];
    make_facet_edge(v, &a);
    v[0] = vertices[0]; v[1] = vertices[2];
    v[2] = vertices[1]; v[3] = vertices[3];
    make_facet_edge(v, &b);
    splice_facets(&a, &b);
    splice_edges(&a, clock(&b, &b));
    stemp = s ? s :(SEGMENT_REFERENCE *) get_node(SEGMENT_REFERENCE_TYPE);
    stemp → r = a;
    stemp → d = 0;
    return (stemp);
}   /* make_segment */


    /* splice splices together the segments referenced by s0 and s1. */
splice(s0, s1)
SEGMENT_REFERENCE *s0;
SEGMENT_REFERENCE *s1;
{   /* splice */
    FACET_EDGE_REFERENCE a, b;
    VERTEX *v0, *v1;
    if (s0 → d ≡ 0)
        splice_edges(spin(clock(&s0 → r, &a), &a), spin(clock(&s1 → r, &b), &b));
    else
        splice_edges(eprev(&s0 → r, &a), eprev(&s1 → r, &b));
}   /* splice */
```


Appendix A.6

Delaunay Application Procedures

This section contains some of the procedures required by the procedure *delaunay*. Those procedures defined which directly use the facet-edge data structure (as opposed to performing computations and the like). The application as a whole is describe in Section 8.2 of the thesis.

```
    /* set of sites S */
  VERTEX S[];


    /* delaunay builds a facet-edge representation for the Delaunay triangulation induced by the
       sites of array S. The functions findsomeD, lookupD, insertD and deleteD operate upon the
       dictionary whose purpose is described in Section 8.2 of the thesis. */
delaunay()
{   /* delaunay */
    register int i;
    FACET_EDGE_REFERENCE a, b, t, temp;
    FACET_EDGE_REFERENCE fearray[4];
    initial_tetra(&t);
    facet_edges_of(&t, fearray);
    for (i = 0; i < 4; i++)
        insertD(&fearray[i]);
    while (findsomeD(&a)) {
        if (¬find_tetra(&a, &t))
            deleteD(&a);
        else {   /* else */
            facet_edges_of(&t, fearray);
            for (i = 0; i < 4; i++) {   /* for */
                a = fearray[i];
                if (lookupD(&a, &b)) {
                    deleteD(&b);
                    align(clock(&a, &a), &b);
                    meld(&a, &b);
                }
                else
                    insertD(&a);
            }
        }
    }
}   /* delaunay */


    /* align Enexts from facet-edge pair a until its vertices coincide with those of b. */
align(a, b)
FACET_EDGE_REFERENCE *a;
FACET_EDGE_REFERENCE *b;
{   /* align */
    int i;
```

```
    for (i = 0; i < 3; i++) {
        if (org(a) ≡ org(b))
            break;
        enext(a, a);
    }
    if ((org(a) ≠ org(b)) ∨ (dest(a) ≠ dest(b)))
        error(NON_ALIGNED, a, b);
    return;
}   /* align */
```

```
    /* facet_edges_of places four facet-edge pairs of tetrahedron t into the 4-element array f. Care
    is taken to ensure the orientation for each of the facet-edge pairs is the same. */
facet_edges_of(t, f)
FACET_EDGE_REFERENCE *t;
FACET_EDGE_REFERENCE *f;
{   /* facet_edges_of */
    FEREF temp;

    temp = *t;
    f[0] = *t;
    clock(fnext(&temp, &f[1]), &f[1]);
    enext(&temp, &temp);
    clock(fnext(&temp, &f[2]), &f[2]);
    enext(&temp, &temp);
    clock(fnext(&temp, &f[3]), &f[3]);
}   /* facet_edges_of */
```

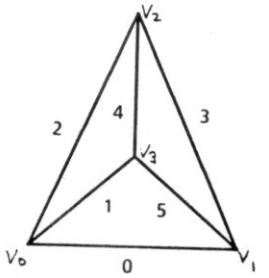
```
    /* meld(aa, bb) glues polyhedra P and P' together along facets f and f', where facet f
    belongs to P and facet-edge pair aa ≡ [f, e], and facet f' belongs to P' and bb ≡[f', e']. The
    edges of P' incident to the vertices of f' are made incident to the corresponding vertices of f.
    Facets f and f' should be bounded by the same number of edges. */
static SREF stemp, stemp1;
static SREF *sptr;

meld(a, b)
FEREF *a;
FEREF * b;
{   /* meld */
    FEREF cura, curb, temp, starta;

    cura = *a;
    curb = *b;
    starta = cura;
    Transfer(pneg(b), ppos(a));
    do {
        if (¬same_facet_ring(&cura, &curb)) {
            splice_facets(&cura, fprev(&curb, &temp));
        }
        splice_facets(&cura, fprev(&cura, &temp));
        Transfer(org(&curb), org(&cura));
        transfer(pneg(&cura), sdual(&curb, &temp));
        transfer(0, &cura);
        transfer(0, sdual(&cura, &temp));
```



```
        transfer(0, clock(&cura, &temp));
        transfer(0, sdual(clock(&cura, &temp), &temp));
        enext(&cura, &cura);
        enext(&curb, &curb);
    } while (cura.n ≠ starta.n) ;
}   /* meld */
```

```
  /* v[i]Org = vindices[i][0], v[i]Dest = vindices[i][1] */
static int vindices[6][2] = {
0, 1, 0, 3, 0, 2, 1, 2, 3, 2, 1, 3
};
```

/ *tetra* builds a tetrahedron, returning into argument S the segment reference to s where $sOrg = v[0]$, $sDest = v[1]$ and s has CCW orientation where *sLeft* is determined by vertices $v[0]$, $v[1]$, and $v[3]$. Vertices $v[0]$ thru $v[3]$ determine the tetrahedron, while $v[4]$ is inside the tetrahedron and $v[5]$ outside the tetrahedron. */




```
/* Edge 0 directed left to right.
All other edges directed from bottom
to top of picture. Vertices nbred:
0 lower left
1 lower right
2 top
3 center */
```

```
   SEGMENT_REFERENCE *
tetra(v, S)
VERTEX **v;    /*six element array of ptrs*/
SEGMENT_REFERENCE *S;
{   /* tetra */
SEGMENT_REFERENCE s[6];
int i;
VERTEX *vs[4];
for (i = 0; i < 6; i++)
    if (v[i] ≡ 0)
        v[i] =(VERTEX *) get_node(VERTEXTYPE);
vs[2] = v[4];    /*inside of sphere*/
vs[3] = v[5];    /*outside of sphere*/
for (i = 0; i < 6; i++) {
    vs[0] = v[vindices[i][0]];
    vs[1] = v[vindices[i][1]];
    make_segment(vs, &s[i]);
}
splice(&s[0], &s[1]);
splice(&s[1], &s[2]);
splice(&s[3], &s[5]);
splice(&s[5], sym(&s[0], &stemp));
splice(&s[4], sym(&s[1], &stemp));
splice(sym(&s[1], &stemp), sym(&s[5], &stemp1));
```

```
splice(sym(&s[2], &stemp), sym(&s[4], &stemp1));
splice(sym(&s[4], &stemp), sym(&s[3], &stemp1));
sptr = S ? S : ( SREF * ) get_node(SREFTYPE);
*sptr = s[0];
return (sptr);
}   /* tetra */
```

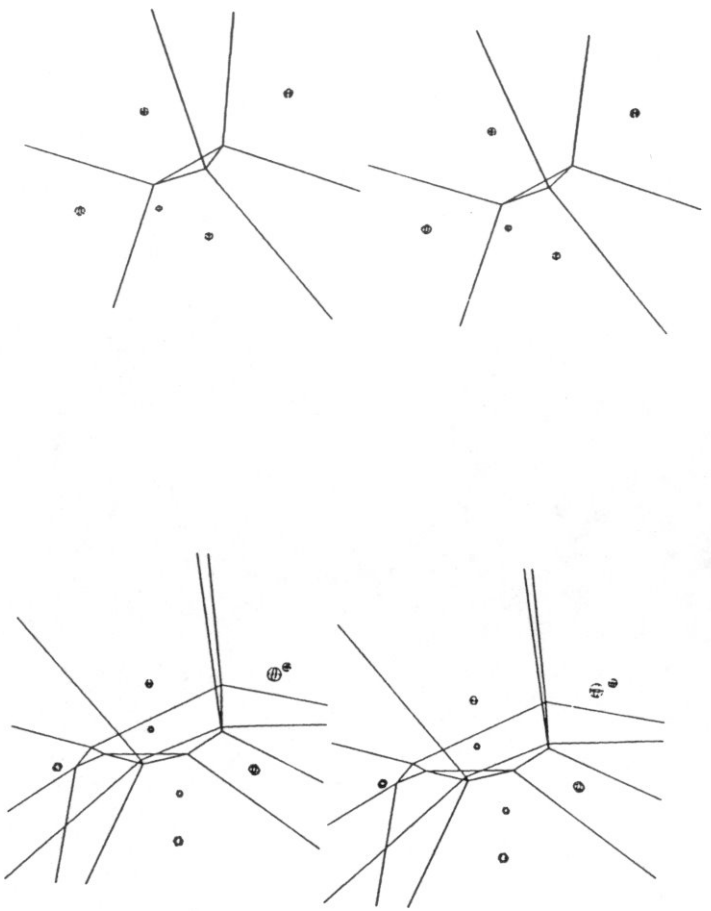
```
    /* same_facet_ring returns TRUE iff facet-edge pairs a and b belong to the same facet ring.
    The function may be implemented differently for different applications. One way is simply to
    test whether aOrg ≡ bOrg - if so, return TRUE. This may not work since facet-edge pairs to
    be eventually combined into the same facet-ring may be assigned (by the application) the
    same origins at an earlier time. Here we use a guaranteed, though less efficient, strategy of
    sweeping around the facet-ring of a looking for b. */
boolean
same_facet_ring(a, b)
FACET_EDGE_REFERENCE *a;
FACET_EDGE_REFERENCE *b;
{   /* same_facet_ring */
    FEREF cura;

    fnext(a, &cura);
    for ( ; cura.n ≠ a → n; fnext(&cura, &cura))
        if (cura.n ≡ b → n)
            return (1);
    return (0);
}   /* same_facet_ring */
```


Appendix B

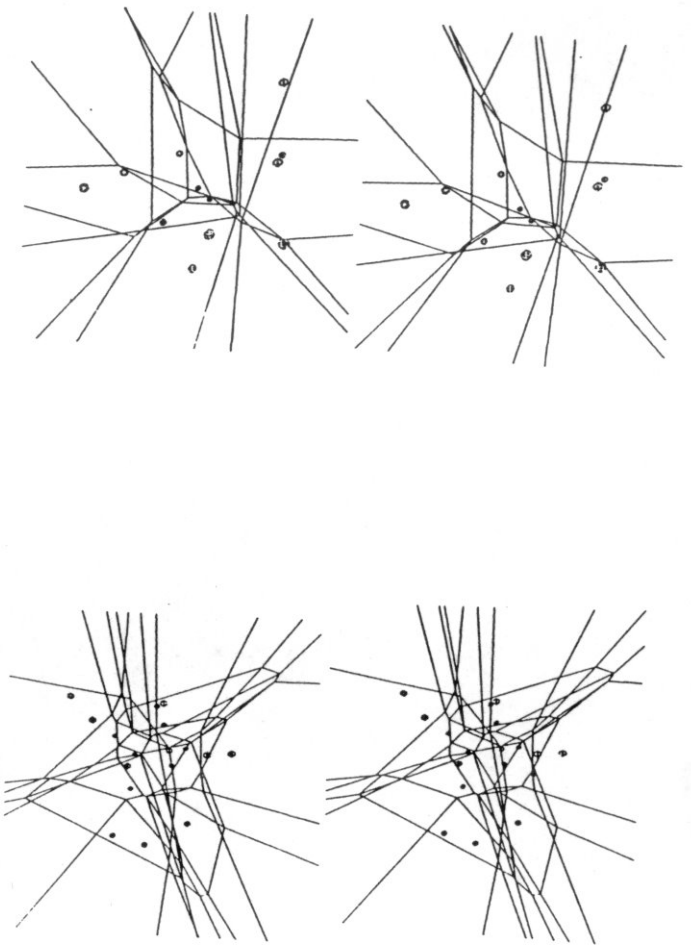
Stereopsis Diagrams of 3-Dimensional Voronoi Diagrams

A stereopsis diagram consists of two images of a 3-dimensional scene, one appropriate for the left eye, the other for the right. When the images are properly viewed, the mind tends to merge the images together to form the sort of 3D view to which we are accustomed by everyday life. The technique is described in [Ju].

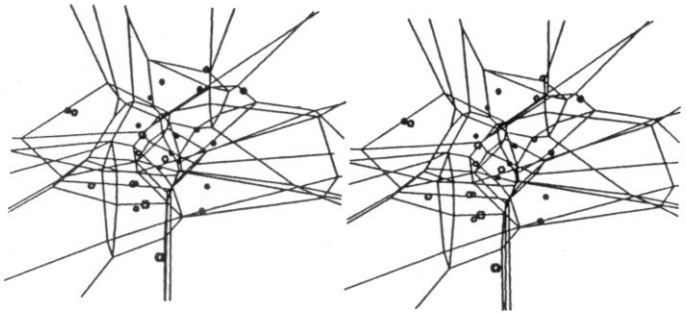
This appendix depicts stereopsis diagrams of several 3-dimensional Voronoi diagrams. Each diagram may be viewed as follows. Focus upon some fairly distant wall or object. Then lift the page at easy arms length (a foot or so in front of your face), continuing to focus "beyond" the page. Several images should appear which, with practice, can be fused into three images. The center image is the merged 3D view of interest. It may help to vary the distance at which the page is held. Successful viewing may take some time; however, once achieved, it becomes easier with practice. Another (much easier) viewing technique is to use stereo-optic glasses whose lenses magnify the images and simulate the distant point of focus.

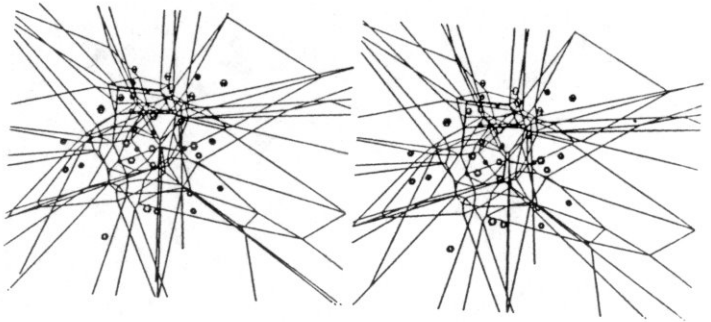


Voronoi diagrams determined by a set of 5 sites (above) and of 8 sites (below) of $\Re^3$.



Voronoi diagrams determined by 12 sites (above) and by 18 sites (below).





Voronoi diagrams determined by 24 sites (above) and by 36 sites (below).

References

[Ag] M. K. Agaston, *Algebraic Topology*, Marcel Dekker Inc., U.S.A., 1976.

[AB] D. Avis and B. K. Bhattacharya, "Algorithms for computing d-dimensional Voronoi diagrams and their duals", in *Advances in Computing Research.* Edited by F. P. Preparata, 1, JAI Press, 1983, pp. 159–180.

[AE] F. Aurenhammer and H. Edelsbrunner, "An Optimal Algorithm for Constructing the Weighted Voronoi Diagram in the Plane", Pattern Recognition, Vol. 17, No. 2, 1984, pp. 251–257.

[AHU] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Co., 1974.

[Ba] B. G. Baumgart, "A Polyhedron Representation for Computer Vision", in *1975 National Computer Conference, AFIPS Conference Proceedings*, vol. 44, AFIPS Press, 1975, pp. 589–596.

[Bh] B. K. Bhattacharya, "Application of Computational Geometry to Pattern Recognition Problems", Simon Fraser Univ. CS, Tech. Rep. 82-3 (1982).

[Bo] A. Bowyer, "Computing Dirichlet Tessellations", Computer Journal, Vol. 24, No. 2, May 1981, pp. 162–166.

[Br] K. Q. Brown, "Voronoi Diagrams from Convex Hulls", Info. Proc. Lett. 9, 1979, pp. 223–228.

[BHS] I. C. Braid, R. C. Hillyard, and I. A. Stroud, "Stepwise construction of polyhedra in geometric modelling", in *Mathematical Methods in Computer Graphics and Design*, K. W. Brodlie, Ed., Academic Press, London, 1980, pp. 123–141.

[Ch] B. Chazelle, "Convex Partitions of Polyhedra", SIAM J. Comput. 13(3), pp. 488–507.

[CK] D. R. Chand and S. S. Kapur, "An Algorithm for Convex Polytopes", JACM 17(1), Jan. 1970, pp. 77–86.

[DL] D. P. Dobkin and M. J. Laszlo, "Primitives for the Manipulation of Three-Dimensional Subdivisions", Proc. of the Third ACM Symp. on Computational Geometry, Waterloo, Canada, June 1987, pp. 86–99.

[EW] C. M. Eastman and K. Weiler, "Geometric Modeling using the Euler Operators", Inst. of Physical Planning, Carnegie-Mellon Univ., Research Rep. 78 (Feb. 1979).

[Gi] P. J. Giblin, *Graphs, Surfaces and Homology*, Chapman and Hall, London, 1977.

[Gr] C. W. Grant, "Integrated Analytic Spatial and Temporal Antialiasing for Polyhedra in 4-Space", Computer Graphics, Vol. 19, No. 3, 1985.

[GS] L. Guibas and J. Stolfi, "Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams", ACM Trans. on Graphics, Vol. 4, No. 2, Apr. 1985, pp. 75–123.

[He] M. Henle, *A Combinatorial Introduction to Topology*, W. H. Freeman & Co., San Francisco, 1979.

[HC] D. Hilbert and S. Cohn-Vossen, *Geometry and the Imagination*, Chelsea Publishing Co., U.S.A., 1983.

[HMMN] S. Hertel, M. Mäntylä, K. Melhorn and J. Nievergelt, "Space Sweep Solves Intersection of Convex Polyhedra", Acta Informatica, Vol. 21, No. 5, 1984.

[Ju] B. Julesz, "Cooperative Phenomena in Binocular Depth Perception", American Scientist, Vol. 62, Jan.–Feb. 1974, pp. 32–43.

[Ka] M. Kallay, "Convex Hull Algorithms in Higher Dimensions", Dept. of Mathematics, unpublished manuscript, Univ. of Oklahoma, Norman, Oklahoma, 1981.

[Ki1] D. G. Kirkpatrick, "Efficient Computation of Continuous Skeletons", Proc. 20th IEEE Annual Symp. on FOCS, Oct. 1979, pp. 18–27.

[Ki2] D. G. Kirkpatrick, "Optimal Search in Planar Subdivisions", SIAM J. Comput. 12(1), 1983, pp. 28–35.

[LS] D. T. Lee and B. Schachter, "Two Algorithms for Constructing Delaunay Triangulations", Int'l J. Comput. and Info. Sci. 9(3), 1980, pp. 219–242.

[MP] D. E. Mueller and F. P. Preparata, "Finding the Intersection of Two Convex Polyhedra", Theoretical Computer Science 7(2), Oct. 1978, pp. 217–236.

[Pr] F. P. Preparata, "A New Approach to Planar Point Location", SIAM J. Comput. 10(3), 1981, pp. 473–482.

[PH] F. P. Preparata and S. J. Hong, "Convex Hulls of Finite Sets of Points in Two and Three Dimensions", Comm. ACM, 2(20), Feb. 1977, pp. 87–93.

[PS] F. P. Preparata and M. I. Shamos, *Computational Geometry*, Springer-Verlag, New York, 1985.

[Re] A. A. G. Requicha, "Representations for Rigid Solids: Theory, Methods, and Systems", ACM Computing Surveys, Vol. 12, No. 4, Dec. 1980, pp. 437–464.

[Ro] D. F. Rogers, *Procedural Elements for Computer Graphics*, McGraw Hill Inc., 1985.

[Se] R. Seidel, "Constructing Higher-Dimensional Convex Hulls at Logarithmic Cost per Face", in *Proceedings of the Eighteenth Annual ACM Symp. on Theory of Computing*, Berkeley, 1986, pp. 404–413.

[SH] M. I. Shamos and D. Hoey, "Geometric Intersection Problems", Sixteenth Annual IEEE Symposium on FOCS, Oct. 1975, pp. 151–162.

[Wa] D. F. Watson, "Computing the n-Dimensional Delaunay Tessellation with Application to Voronoi Polytopes", Computer Journal, Vol. 24, No. 2, May 1981, pp. 167–172.

[We] J. R. Weeks, *The Shape of Space*, Marcel Dekker Inc., N.Y., 1985.

[Wö] B. Wördenweber, "Volume-triangulation", C. A. D. Group, University of Cambridge (1980).

[WS] D. F. Watson and F. G. Smith, "A Computer Simulation and Study of Grain Shape", Computers and Geosciences, Vol. 1, pp. 109–111.