A Dual Space Representation for Geometric Data

Oliver Gunther and Eugene Wong
EECS Department, 231 Cory Hall
University of California
Berkeley CA 94720

Abstract

This paper presents a representation scheme for polyhedral objects in arbitrary dimensions. Each object is represented as the algebraic sum of convex polyhedra (*cells*). Each cell in turn is represented as the intersection of halfspaces and encoded in a vector. The notion of vertices is abandoned completely as it is not needed for the set and search operators we intend to support. We show how this approach allows us to decompose set operations (such as intersection) on polyhedral objects into two steps. The first step consists of a collection of vector operations; the second step is a garbage collection where vectors that represent empty cells are eliminated.

1. Introduction

Modern database systems are no longer limited to business applications. Non-standard applications such as computer-aided design, computer vision, or geographic data processing are becoming increasingly important, and geometric data play a crucial role in many of these new applications. For efficiency reasons it is essential that the special properties of geometric data be fully utilized in the data base management system. It is important to view geometric objects (such as points, lines, or polygons) as integral entities and not as tuples of numbers that may be used to represent them.

Furthermore, the special operators that are defined on these objects need to be supported. Common examples include set operators such as union or intersection or search operators such as range search or point location. These operators are substantially different from the operators defined on numerical data. They are often harder to compute, and it is not trivial to determine the smallest domain on which they are closed. Even the regularized* set operators, for example, are not closed on the set of simple polyhedra; see figure 1.1.

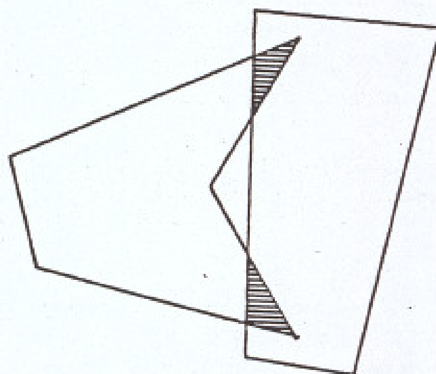

Fig. 1.1: The intersection of two simple polyhedra is not necessarily a simple polyhedron.

In short, to deal with geometric data effectively requires some recognition of geometry, and nowhere is this more important than in the *representation* of geometric objects, which can be interpreted as the mapping of the original data objects into a set of objects that facilitates the computation of a particular class of operators. The significance of representation schemes for efficient data management has been discussed by Requicha [Requ80]. A survey of various representation schemes for two- and three-dimensional geometric data can be found in [Besl85]. In this paper we develop this theme in connection with a particular representation scheme for an important class of geometric objects, viz.,

This research was sponsored under research contract DAAG29-85-0223 and, in the case of the first author, a scholarship from the German National Scholarship Foundation.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

* The regularized set operators, as defined by Tilove [Tilo80] , include intersection, union, and difference. They differ from the corresponding simple set operators by an additional step making the result *regular*, i.e. the closure of its interior. This way, the dimension of the result is equal to the lowest dimension of any of the operands. In this paper, all set operators that are defined on point sets are assumed to be regularized.

polyhedra.

In section 2, we describe the concept of polyhedral chains where a polyhedral object is represented as an algebraic sum of simple polyhedra (*cells*). Section 3 introduces a representation scheme for convex cells; each cell is represented as an intersection of halfspaces and encoded in a vector. Section 4 shows how set operations are carried out using this representation scheme, and section 5 contains our conclusions.

2. Polyhedral Chains

We extend the notion of polyhedron in the following way. A d-dimensional *polyhedral chain* in Euclidean space $\mathbf{E}^d$ [Whit57] is an expression of the form

$$x_P = \sum_{i=1}^{m} p_i$$

Here, the p_i are d-dimensional regular polyhedra in $\mathbf{E}^d$ that are not necessarily bounded. We consider a point $t \in \mathbf{E}^d$ inside the polyhedral chain P if and only if it is inside any of the polyhedra p_i, i.e.

$$t \in P \iff t \in p_i \text{ for some } i = 1 \ldots m$$

This way, each polyhedral chain represents a polyhedral point set. Two polyhedral chains P and Q are *equivalent* if they represent the same point set, i.e. if

$$t \in P \iff t \in Q$$

Polyhedral chains are a simple and powerful tool to describe various kinds of polyhedral objects. They may be used to describe any simple (i.e. non self-intersecting) polyhedral point set in $\mathbf{E}^d$ (fig. 2.1), as well as self-intersecting polyhedra of any shape (fig. 2.2, 2.3).

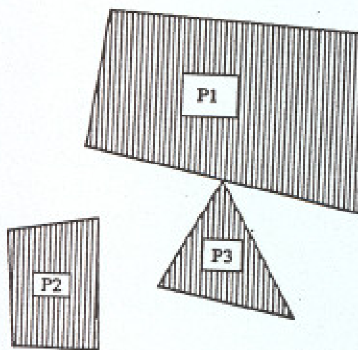

Fig. 2.1: $p_1+p_2+p_3$

As pointed out in [Newe80] , applications for non-simple polyhedra are becoming increasingly important in areas like computer-aided design or geographic data processing. Also, there are numerous applications for higher-dimensional geometric objects, such as linear programming [Dant63] or logic databases where

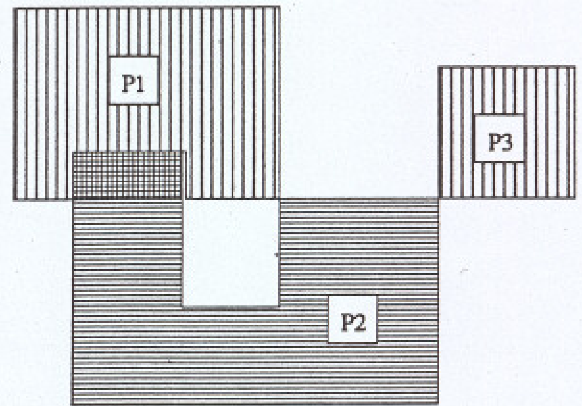

Fig. 2.2: $p_1+p_2+p_3$

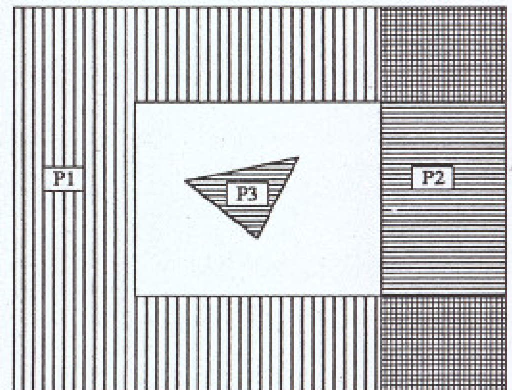

Fig. 2.3: $p_1+p_2+p_3$

geometric objects are used to represent predicates [Ston86].

Unlike simple polyhedra, polyhedral chains are closed under all regularized set operators. Furthermore, the boundary of a convex polyhedron of dimension d is a polyhedral chain of dimension $(d-1)$. Hence, the complete set of polyhedral chains of dimensions 0 through d in $\mathbf{E}^d$ is closed under the boundary operator ∂. For these reasons, polyhedral chains form an appropriate set for embedding polyhedra.

Now consider a database consisting of a collection of (possibly self-intersecting) d-dimensional polyhedra in Euclidean space $\mathbf{E}^d$. The restriction to polyhedra, rather than general subsets of $\mathbf{E}^d$, is justified by the fact that those are commonly used to approximate general shapes in practice [Faux79].

To support search and set operators, we represent the polyhedra in the database as *convex* polyhedral chains, i.e. as sums of *convex* polyhedra p_i (*cells*). Each cell in turn will be represented as the intersection of halfspaces and encoded in a vector. Our scheme is conceptually simple, provides support for set and search operators, and seems well suited for parallel processing.

Formally, each data object P is represented as a convex polyhedral chain in $\mathbf{E}^d$,

$$x_P = \sum_{i=1}^{m} p_i$$

with all p_i being convex. Obviously, for any polyhedral chain in $\mathbf{E}^d$ there is an equivalent convex polyhedral chain in $\mathbf{E}^d$. For simplicity (see section 3) we require that for each face f of any convex cell p_i there be a face g of P, such that f and g are both subsets of the same $(d-1)$-dimensional hyperplane. Note that we do not require the p_i to be mutually disjoint. Disjointness is hard to maintain and provides no particular advantages for the operators we intend to support.

3. The h-Vector

The next question is how to represent the convex cells p_i. It is well known that any convex polyhedron in $\mathbf{E}^d$ can be represented as the intersection of halfspaces in $\mathbf{E}^d$. Each halfspace in turn can be represented as a product $h \cdot H$ where H is an oriented $(d-1)$-dimensional hyperplane and h is an integer number. In particular, we define $1 \cdot H$ as the closed halfspace to the right of H, $-1 \cdot H$ as the closed halfspace to the left of H, and for completeness $0 \cdot H$ as $\mathbf{E}^d$.

Let $\mathbf{H} = H_1 H_2 \ldots H_{|\mathbf{H}|}$ denote a list of $(d-1)$-dimensional oriented hyperplanes such that for each face f of any data object in the database there is a hyperplane in $\mathbf{H}$ that embeds f. Now each cell p can be represented as a ternary vector $\mathbf{h}_p = \{0,1,-1\}^{|\mathbf{H}|}$, such that $p = \bigcap_{i=1}^{|\mathbf{H}|} (\mathbf{h}_p)_i \cdot H_i$.

We note that for a given cell p, $\mathbf{h}_p$ is by no means unique. For example, suppose that hyperplane H_i and cell p are disjoint and p is a subset of the halfspace $1 \cdot H_i$. Then whether $(\mathbf{h}_p)_i$ is 0 or 1 makes no difference. For a given p, the set of all possible $\mathbf{h}_p$-vectors is an equivalence class which contains a unique vector with the minimum number of nonzero components. For this unique minimum $\mathbf{h}_p$ every nonzero component corresponds to a supporting hyperplane of p. Note that there is no unique minimum vector to represent the empty set. On the other hand, there is a unique minimum vector to represent the whole space $\mathbf{E}^d$, viz., the vector $0^{|\mathbf{H}|}$.

The insertion of new data objects is performed by adding new hyperplanes to $\mathbf{H}$, if necessary. For simplicity we assume that the components of the ternary vectors $\mathbf{h}_p$ default to zero if they are not explicitly specified. Under this assumption an insertion does not change the representation of existing cells.

The deletion of data objects may cause some hyperplanes in $\mathbf{H}$ to become redundant. The deletion of such a hyperplane from $\mathbf{H}$ corresponds to a compression of each vector $\mathbf{h}_p$ by one component. Although it may not be efficient to perform this update after each single deletion, it might be worthwhile to do such a clean-up after a certain number of deletions. Otherwise a large number of redundant hyperplanes will inflate the representations unnecessarily.

It $|\mathbf{H}|$ is large, as it may well be, the explicit storage representation of $\mathbf{h}_p$ is not feasible. However, the simple structure of $\mathbf{h}_p$ allows many alternative data structures to be used. As one example, $\mathbf{h}_p$ can be represented by a set of (signed) pointers, pointing to those hyperplanes that correspond to the nonzero elements. In this paper we do not explore the relative computational efficiencies of such alternatives.

Note that this approach to represent polyhedral data objects abandons the notion of vertex completely. Representation of cells by $\mathbf{h}$-vectors has both conceptual and computational advantages. To represent cells in terms of supporting hyperplanes rather than in terms of vertices is usually the most space-efficient way because no adjacency relations need to be stored. This becomes especially important in higher dimensions as the number of adjacencies may grow exponentially in the dimension [Prep85]. Furthermore, it seems that vertices are not necessary for the search and set operators we intend to support. Search operators such as point location or range search can be supported efficiently by search structures that are based on supporting hyperplanes rather than vertices; an example for such a structure is the binary space partitioning tree [Fuch80]. All set operations on cells can be computed efficiently without using vertices by decomposing them into two parts: (a) an operation on the $\mathbf{h}$-vectors without references to the geometric coordinates of the hyperplanes, and (b) a generic operation that tests whether a vector $\mathbf{h}_p$ is *null*, i.e. whether the intersection of the halfspaces specified by $\mathbf{h}_p$ is empty. This decomposition will be described in detail in the following section.

4. Set Operations

Let P and Q be two general polyhedral objects. We now show that any set operation on P and Q can be decomposed into: (a) operations on the $\mathbf{h}$-vectors, and (b) deleting the null vectors from the set of resulting $\mathbf{h}$-vectors. The following propositions are easily verified with the definitions of set operations and of polyhedral chains.

Proposition 4.1: Let P and Q be represented by convex polyhedral chains $x_P = \sum_{i=1}^{m} p_i$ and $x_Q = \sum_{j=1}^{l} q_j$. Then

$$x_{P \cup Q} = x_P + x_Q$$
$$x_{P \cap Q} = \sum_{i,j} (p_i \cap q_j)$$
$$x_{\bar{P}} = x_{\bar{p}_1 \cap \cdots \cap \bar{p}_m}$$
$$x_{P-Q} = x_{P \cap \bar{Q}}$$

Proposition 4.2: Let $\mathbf{h}_p$ denote a $\mathbf{h}$-vector of a cell p. Then $x_{\bar{p}} = -\mathbf{h}_p \cdot \mathbf{H}$.

For an example see figure 4.1.

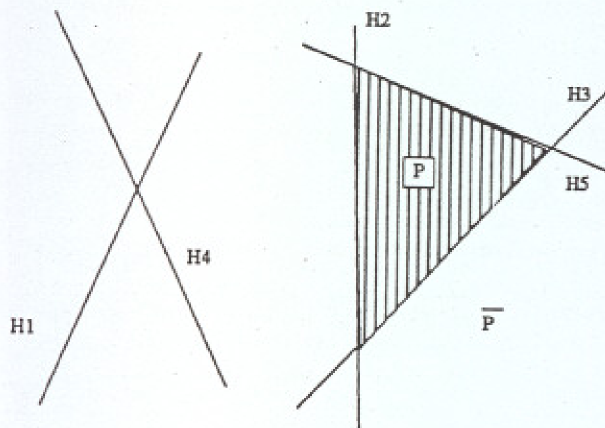

Fig. 4.1: $\mathbf{h}_p=(0,1,-1,0,-1)$, $x_{\overline{p}}=-1\cdot H_2+1\cdot H_3+1\cdot H_5$

Note that the length of this chain equals the number of nonzero components of the vector $\mathbf{h}_p$. It is therefore desirable to keep this number low, possibly at its minimum.

Proposition 4.3: Let $\mathbf{h}_p$ and $\mathbf{h}_q$ denote the **h**-vectors for two cells p and q respectively. Then $\mathbf{h}_{p\cap q}$ can be computed using the following table for each component $(\mathbf{h}_{p\cap q})_i$.

$(\mathbf{h}_{p\cap q})_i$		$(\mathbf{h}_q)_i$		
		0	1	-1
$(\mathbf{h}_p)_i$	0	0	1	-1
	1	1	1	*
	-1	-1	*	-1

Table 4.1: In those cases denoted by *, the hyperplane H_i separates p and q, i.e. $p\cap q=\varphi$.

Note that both the intersection and the complementation operator are defined on the components of the **h**-vector. The components are independent of each other and can therefore be processed in parallel. In particular, a systolic array [Kung79] or a connection machine [Hill85] with one processor per hyperplane seem to be promising for an efficient implementation.

It follows from propositions 4.1-4.3 that for any set operation &, the **h**-vector representation of $P\&Q$ can be computed from the **h**-vector representations of P and Q. However, the **h**-vectors in the resulting representation may not be minimal. Also, some vectors may define empty sets, due to the fact that condition * is a sufficient, but not a necessary condition for non-intersection. Two cells p and q may not intersect, but there is no component $(\mathbf{h}_{p\cap q})_i$ where condition * occurs. In that case, the resulting vector $\mathbf{h}_{p\cap q}$ defines an empty set. Although that case is consistent with our data model, it is not desirable. A large number of empty cells p_i in the convex polyhedral chains $x_P=\sum_{i=1}^{m} p_i$ representing the data objects may slow down the system performance considerably. We therefore need an efficient means for detecting empty cells.

One approach would be to abandon the concept of minimality and to increase the number of nonzero components in the **h**-vector, possibly to its maximum, i.e.

$$(\mathbf{h}_p)_i = \begin{cases} 1 \text{ if } p\subseteq\ 1\cdot H_i \\ -1 \text{ if } p\subseteq -1\cdot H_i \\ 0 \text{ otherwise} \end{cases}$$

Each nonzero component increases the chance that a separating hyperplane is found, i.e. that condition * is met if two polyhedra do not intersect. If each **h**-vector had a maximum number of nonzero components then a separating hyperplane would be detected immediately; i.e. condition * would be a necessary and sufficient condition for non-intersection. On the other hand, this approach makes the identification of supporting hyperplanes and therefore the cell complementation and boundary retrieval operations much more difficult. Also, computing the above function for each cell p in the database requires an immense amount of computation and produces a lot of data that is probably never needed.

A garbage collector seems to be a better solution. Each time a new cell is computed as the intersection of two cells, the new cell is tagged. A background process (the garbage collector) keeps checking the tagged cells in the database for emptiness. If a cell is found nonempty, it is untagged. Otherwise, it is deleted from storage and from the chains that contain that cell. Unfortunately, the representation of cells by means of their **h**-vectors does not lead to an efficient algorithm to check cells for emptiness. A better approach to this problem, based on geometric duality, is presented in a separate paper [Gunt86]. In that paper, we show that the time complexity to check two cells for intersection is polylogarithmic and therefore sublinear in the number of vertices of any of the cells.

In order to avoid duplicating computational effort and loosing information, we propose to cache the results obtained by the garbage collector. Whenever a cell intersection $p\cap q$ is computed a second time, it should be immediately clear from the vectors $\mathbf{h}_p$ and $\mathbf{h}_q$ if the intersection $p\cap q$ is empty or not. Whenever the garbage collector checks a new cell $r=p\cap q$, it either discovers a separating hyperplane (if p and q are disjoint) or it discovers that there are no separating hyperplanes (if p and q intersect). This result can be cached by extending the notion of the **h**-vector to capture more information in the following way.

Each cell p is represented as a vector $\mathbf{h}_p^+$ with the following semantics.

$(\mathbf{h}_p^+)_i$	Meaning
(1,Y)	$p \subseteq 1 \cdot H_i$, H_i may be a supporting hyperplane of p
(−1,Y)	$p \subseteq -1 \cdot H_i$, H_i may be a supporting hyperplane of p
(1,N)	$p \subseteq 1 \cdot H_i$, H_i is not a supporting hyperplane of p
(−1,N)	$p \subseteq -1 \cdot H_i$, H_i is not a supporting hyperplane of p
I	H_i intersects the interior of p (hence, it is not a supporting hyperplane)
0	H_i is not a supporting hyperplane of p

Table 4.2

Components that are not explicitly specified default to 0. Now $(\mathbf{h}^+_{p \cap q})_i$ is given by the following tables.

$(\mathbf{h}^+_{p \cap q})_i$		$(\mathbf{h}_q^+)_i$		
		(1,Y)	(−1,Y)	(1,N)
$(\mathbf{h}_p^+)_i$	(1,Y)	(1,Y)	*	(1,N)
	(−1,Y)	*	(−1,Y)	*
	(1,N)	(1,N)	*	(1,N)
	(−1,N)	*	(−1,N)	*
	I	(1,Y)	(−1,Y)	(1,N)
	0	(1,Y)†	(−1,Y)†	(1,N)

Table 4.3a

$(\mathbf{h}^+_{p \cap q})_i$		$(\mathbf{h}_q^+)_i$		
		(−1,N)	I	0
$(\mathbf{h}_p^+)_i$	(1,Y)	*	(1,Y)	(1,Y)†
	(−1,Y)	(−1,N)	(−1,Y)	(−1,Y)†
	(1,N)	*	(1,N)	(1,N)
	(−1,N)	(−1,N)	(−1,N)	(−1,N)
	I	(−1,N)	0	0
	0	(−1,N)	0	0

Table 4.3b

If p and q do not intersect there will be at least one separating hyperplane H_i that supports p or q. In this case $(\mathbf{b}_{p \cap q})_i$ corresponds to one of the cases denoted by * or by †. Therefore, a new cell $r = p \cap q$ is certainly empty if any component $(\mathbf{b}_{p \cap q})_i$ corresponds to one of the cases denoted by *. Otherwise, it needs to be tagged if and only if there is at least one component $(\mathbf{b}_{p \cap q})_i$ that corresponds to one of the cases with the †.

If a tagged cell $r = p \cap q$ is found empty, this result can be cached by the following updates. Let H_i be a separating hyperplane and, w.l.o.g. let $p \subseteq 1 \cdot H_i$ and $q \subseteq -1 \cdot H_i$.

IF $(\mathbf{h}_p^+)_i = 0$
THEN $(\mathbf{h}_p^+)_i := (1,N)$

IF $(\mathbf{h}_q^+)_i = 0$
THEN $(\mathbf{h}_q^+)_i := (-1,N)$

If, on the other hand, a tagged cell $r = p \cap q$ is found non-empty, we know that there are no separating hyperplanes between p and q. For any hyperplane H_i that supports p, either (a) q lies on the same side of H_i as p, or (b) H_i intersects the interior of q. A similar condition holds for any hyperplane H_i that supports q. This result can be cached by performing the following updates.

IF $(\mathbf{h}_p^+)_i = (\pm 1,Y)$ AND $(\mathbf{h}_q^+)_i = 0$ AND $H_i \cap q = \varphi$
THEN $(\mathbf{h}_q^+)_i := (\pm 1,N)$

IF $(\mathbf{h}_p^+)_i = (\pm 1,Y)$ AND $(\mathbf{h}_q^+)_i = 0$ AND $H_i \cap q \neq \varphi$
THEN $(\mathbf{h}_q^+)_i := I$

IF $(\mathbf{h}_q^+)_i = (\pm 1,Y)$ AND $(\mathbf{h}_p^+)_i = 0$ AND $H_i \cap p = \varphi$
THEN $(\mathbf{h}_p^+)_i := (\pm 1,N)$

IF $(\mathbf{h}_q^+)_i = (\pm 1,Y)$ AND $(\mathbf{h}_p^+)_i = 0$ AND $H_i \cap p \neq \varphi$
THEN $(\mathbf{h}_p^+)_i := I$

Whenever $p \cap q$ is computed again, it follows from the vectors $\mathbf{h}_p^+$ and $\mathbf{h}_q^+$ if p and q intersect or not. If they do intersect, the resulting cell will not have to be tagged again.

When a new cell is inserted into the database, most of the components of its **h**-vector are zero. As set operations are performed on the data objects, the database evolves. More and more zero components of the **h**-vectors are replaced, and the vectors carry more and more information. Therefore, it will happen less and less frequently that a new cell has to be tagged and checked for emptiness. Also, at some point it may be more efficient to test a new cell $r = p \cap q$ for emptiness by checking the hyperplanes that may be separating ones (i.e. the ones that correspond to components with a †) one by one if they are actually separating. If they are few enough components with a †, this may be simpler and faster than using the dual approach proposed in [Gunt86].

Problems such as complementation, point location or boundary retrieval may be solved by looking at only those hyperplanes that may be supporting, i.e. the hyperplanes H_i where $(\mathbf{h}_p^+)_i$ is (1,Y) or (−1,Y).

There are variations to this approach. First, one may prefer to have only minimal **h**-vectors, i.e. to identify the supporting hyperplanes of each cell explicitly. This can be achieved, for example, by extending the garbage collector as follows. Each time an intersection cell is found non-empty, its supporting hyperplanes are computed and the **h**-vector is updated accordingly. Second, one may decide to simplify the update procedure above by introducing symbols (1,NI) and (−1,NI) which represent (1,N) OR I and (−1,N) OR I, respectively. Then the set of updates for the case that p and q intersect can be simplified to

IF $((\mathbf{h}_p^+)_i = (\pm 1, Y)$ AND $(\mathbf{h}_q^+)_i = 0$
THEN $(\mathbf{h}_q^+)_i := (\pm 1, NI)$

IF $((\mathbf{h}_q^+)_i = (\pm 1, Y)$ AND $(\mathbf{h}_p^+)_i = 0$
THEN $(\mathbf{h}_p^+)_i := (\pm 1, NI)$

In particular, it is not necessary anymore to check any hyperplane H_i that supports p (q) if it intersects the interior of q (p), i.e. if $H_i \cap q$ ($H_i \cap p$) $= \varphi$. As proven in [Gunt86] , the time complexity to check this condition for a particular hyperplane H_i is logarithmic in the number of vertices of q (p).

5. Conclusions

We presented a representation scheme for polyhedral data objects, based on convex polyhedral chains. Each cell is represented as an intersection of halfspaces, encoded in a vector. The notion of vertices is abandoned completely as it is not needed for the set and search operators we intend to support.

Based on this representation, we described a scheme to decompose the execution of set operators into two steps. The first step consists of a set of vector operations; the second step is a garbage collection where those vectors are eliminated that represent empty cells. All results of the garbage collection are cached in the vectors in such a way that no computations have to be duplicated. As the database is learning more and more information through the garbage collector, it will be able to detect empty cells immediately such that no additional test for emptiness is required. Future work will focus on an experimental implementation of our scheme.

Also, we believe that this approach is more amenable to parallel processing than a vertex-based approach. In particular, the components of the **h**-vectors are processed independently from each other. Therefore, it seems possible to assign one processor to each hyperplane in **H** and to carry out a significant fraction of the necessary computations locally without interprocessor communication. We are currently working on the details of this approach and are planning an experimental implementation on a connection machine.

References

[Besl85] Besl, P. J. and R. C. Jain, Three-dimensional object recognition, *Computing Surveys* **17**, 1 (March 1985).

[Dant63] Dantzig, G. B., *Linear Programming and Its Extensions*, Princeton University Press, Princeton, NJ, 1963.

[Faux79] Faux, I. D. and M. J. Pratt, *Computational geometry for design and manufacture*, Ellis Horwood, Chichester, Great Britain, 1979.

[Fuch80] Fuchs, H., Z. Kedem, and B. Naylor, On visible surface generation by a priori tree structures, *Computer Graphics* **14**, 3 (June 1980).

[Gunt86] Gunther, O. and E. Wong, *A dual approach to detect polyhedral intersections in arbitrary dimensions*, U.C. Berkeley Memorandum No. UCB/ERL/M86/88 , December 1986.

[Hill85] Hillis, W. D., *The connection machine*, MIT Press, Cambridge, Ma., 1985.

[Kung79] Kung, H. T., Systolic arrays, *Computer* **11**, 4 (Dec. 1979), pages 397-409.

[Newe80] Newell, M. E. and C. H. Sequin, The inside story on self-intersecting polygons, *LAMBDA*, Second Quarter, 1980.

[Prep85] Preparata, F. P. and M. I. Shamos, *Computational geometry*, Springer-Verlag, New York, NY, 1985.

[Requ80] Requicha, A., Representations for rigid solids: theory, methods, and systems, *Computing Surveys* **12**, 4 (Dec. 1980).

[Ston86] Stonebraker, M., T. Sellis, and E. Hanson, An analysis of rule indexing implementations in data base systems, in *Proc. of the 1st International Conference on Expert Data Base Systems*, April 1986.

[Tilo80] Tilove, R. B., Set membership classification: A unified approach to geometric intersection problems, *IEEE Trans. on Computers* **C-29**, 10 (Oct. 1980), pages 874-883.

[Whit57] Whitney, H., *Geometric integration theory*, Princeton, NJ, 1957.