Printing Requisition/Graphic Services

54232

1. Please complete unshaded areas on form as applicable.
2. Distribute copies as follows: White and Yellow to Graphic Services. Retain Pink Copies for your records.
3. On completion of order the Yellow copy will be returned with the printed material.
4. Please direct enquiries, quoting requisition number and account number, to extension 3451.

TITLE OR DESCRIPTION: SEPARATION TM in d Dimensions or Strip Mining in Asteroid Fields CS-89-05

DATE REQUISITIONED: Feb. 1/89
DATE REQUIRED: ASAP
ACCOUNT NO.: 1 2 6 6 1 7 6 4 1

REQUISITIONER- PRINT: D. Wood
PHONE: 4456
SIGNING AUTHORITY: D. Wood

MAILING INFO – NAME: Sue DeAngelis
DEPT.: C.S.
BLDG. & ROOM NO.: DC 2314
[X] DELIVER
[ ] PICK-UP

Copyright: I hereby agree to assume all responsibility and liability for any infringement of copyrights and/or patent rights which may arise from the processing of, and reproduction of, any of the materials herein requested. I further agree to indemnify and hold blameless the University of Waterloo from any liability which may arise from said processing or reproducing. I also acknowledge that materials processed as a result of this requisition are for educational use only.

NUMBER OF PAGES: 22
NUMBER OF COPIES: 150

TYPE OF PAPER STOCK: [ ] BOND [ ] NCR ____ PT. [ ] COVER [ ] BRISTOL [ ] SUPPLIED [ ] Alpac Ivory 140M

PAPER SIZE: [ ] 8½ x 11 [ ] 8½ x 14 [ ] 11 x 17 [ ] 10x14 Glosscoat 10 pt Rolland Tint

PAPER COLOUR: [ ] WHITE [X] ______
INK: [X] BLACK [ ] ______

PRINTING: [ ] 1 SIDE ____ PGS. [X] 2 SIDES ____ PGS.
NUMBERING: FROM ____ TO ____

BINDING/FINISHING: [X] COLLATING [ ] STAPLING [ ] ____ HOLE PUNCHED [ ] PLASTIC RING

FOLDING/PADDING: 7x10 saddle stitched
CUTTING SIZE:

Special Instructions

Beaver Cover

Both cover and inside in black ink please

COPY CENTRE — OPER. NO. / BLDG. / MACH. NO.

DESIGN & PASTE-UP — OPER. NO. / TIME / LABOUR CODE

OPER. NO.	TIME	LABOUR CODE
		D01
		D01
		D01

TYPESETTING — QUANTITY

	QUANTITY	OPER. NO.	TIME	LABOUR CODE
PAP000000				T01
PAP000000				T01
PAP000000				T01

PROOF

PRF				
PRF				
PRF				

NEGATIVES

	QUANTITY	OPER. NO.	TIME	LABOUR CODE
FLM				C01
FLM				C01
FLM				C01
FLM				C01
FLM				C01

PMT

PMT				C01
PMT				C01
PMT				C01

PLATES

PLT				P01
PLT				P01
PLT				P01

STOCK

				001
				001
				001
				001

BINDERY

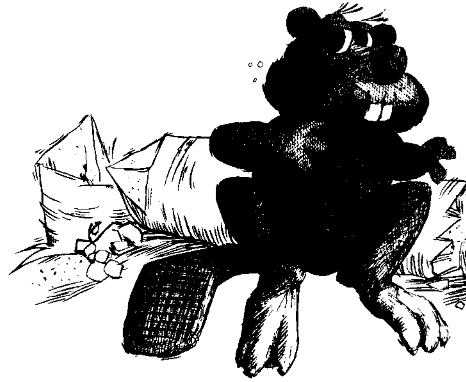
RNG				B01
RNG				B01
RNG				B01
MIS000000				B01

OUTSIDE SERVICES

$ ______ COST

TAXES – PROVINCIAL [ ] FEDERAL [ ]

UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT



SEPARATION™ in d Dimensions

Michel P. Devine
Derick Wood

Data Structuring Group
Research Report CS-89-05

January, 1989

$SEPARATION^{TM}$ in d Dimensions or Strip Mining in Asteroid Fields *

Michel P. Devine† Derick Wood†

Abstract

The $SEPARATION^{TM}$ problem for two and higher dimensions is to compute a separating sequence for a collection of disjoint orthogonal d-dimensional polyhedra. We give an algorithm for the 3-dimensional problem that requires $O(N\sqrt{N}\log^3 N)$ time and $O(N\sqrt{N}\log N)$ space, where N is the total number of vertices. The same algorithm when suitably restricted solves the 2-dimensional problem in $O(N\log^3 N)$ time using $O(N\log N)$ space. The extension to $d \geq 4$ dimensions yields an algorithm that requires $O(dN^{d/2}\log^3 N)$ time and $O(dN^{d/2}\log N)$ space.

Keywords: rectangles, finite orientation geometry, orthogonal partition tree, priority search tree, space sweep, asteroid mining.

1 Introduction

Chazelle, Ottmann, Soisalon-Soininen and Wood [COSSW84] introduced the family of puzzles known collectively as $SEPARATION^{TM}$: Given a set of disjoint simple orthogonal polygons determine a sequence of orthogonal translations that separate all polygons, if one exists. The aim of this paper is to present a solution to the corresponding d-dimensional problem, where $d \geq 3$.

Our intent is to develop an algorithm for one of the crucial applications of the next century: strip mining of mineral-rich asteroids. Consider the following scenario. We wish to use our latest planetoid-swallowing mining machine to extract minerals and water from the asteroids orbiting the sun

*This work was supported under a Natural Sciences and Engineering Research Council of Canada Grant No. A-5692 and under a grant from the Information Technology Research Centre.

†Data Structuring Group, Department of Computer Science, University of Waterloo, WATERLOO, Ontario N2L 3G1, CANADA.

between Mars and Saturn. Unfortunately, asteroids tend to cluster into fields (or "gaggles")[1]. A small thermonuclear device is installed on each asteroid such that the impulse from the explosion will propel the asteroid away from the field, where it can be captured and processed.

Each asteroid is modeled by a simple three-dimensional orthogonal polyhedron and the explosion vectors are constrained to lie parallel to the implied orthogonal directions. Only one asteroid is allowed to move at any given time, and, in the interest of safety, no rotations or collisions are permitted. A moving asteroid will continue to move until it is sufficiently far away from the remaining ones to be processed by the mining machine (for practical purposes, it is at infinity). Determining the location of the devices as well as the firing sequence required to separate all asteroids in a given field is equivalent to solving the *SEPARATION*^{TM} problem.

While the topic of asteroid mining has seldom appeared in the computational geometry literature, a number of researchers have investigated separation questions in more down-to-earth domains. The asteroid mining problem is related to admittedly less pragmatic concerns such as motion planning, puzzle solving, and the assembly of composite objects. As an illustration of the use of *SEPARATION*^{TM} in assembly, notice that a separating sequence can be inverted to yield an assembling sequence: Given a description of the parts $P_1, \ldots, P_p$ in their desired configuration and a separating sequence s, the composite object can be assembled by applying the moves given by the sequence s^{-1}, moving each P_i to its destination.

1.1 Previous Work

Guibas and Yao [GY80] investigate translating rectangles in the plane and present an optimal $O(N \log N)$ time algorithm that is applicable to sets of convex polygons. They show that any set of convex polygons can always separated in a single direction, while the same is not true for sets of convex polyhedra.

Dehne and Sack [DS86] separate arbitrary simple polygons (when possible) in $O(N^2 \log N)$ time. Nurmi [Nur87] separates simple polygons in $O(N \log N)$ time and simple polyhedra in $O(k \log N)$ time, where $N \leq k \leq N^2$ is the number of intersections between the objects when projected onto a hyperplane perpendicular to the direction of travel. Chazelle et al [COSSW84] outline an algorithm for solving *SEPARATION*^{TM} for orthogonal polygons using multiple orthogonal directions that requires $O(N \log^2 N)$ time. Toussaint [Tou85] presents an excellent survey of separability results.

[1]While it is true that the asteroid fields are actually quite sparse, our algorithm has other applications such as removing debris in Earth orbit and exploiting Saturn's dense rings.

1.2 Outline of the paper

The following section covers the definitions necessary for our discussion of d-$SEPARATION^{TM}$. Section 3 contains a discussion of the general problem and in section 4 we give an overview of the methodology used in the paper. In section 5, a solution for the three dimensional variant of $SEPARATION^{TM}$ is presented and in §6 we discuss extensions to arbitrary dimensions. In the final section, we present our conclusions and outline areas for further research.

2 Preliminaries

In this section, we introduce the basic definitions and terminology needed in our discussion of the $SEPARATION^{TM}$ problem. An important assumption is that a complete description of the scene is given. The objects under consideration are pairwise disjoint polyhedra that are simple in the sense that no two non-adjacent faces share a point. We do not insist that all d-polyhedra be homeomorphic to a d-sphere, so holes are allowed. The polyhedra are also orthogonal — all edges are parallel to one of the coordinate axes — and movement is restricted to a single orthogonal translation for each object. The set of allowable translation directions $\{\pm x_0, \pm x_1, \ldots, \pm x_d\}$ is denoted by Δ.

Now, when is a polyhedron free to move without risking any collisions? Intuitively, if we illuminate an object, the shadow cast by the object represents the space through which it must travel during translation. If the shadow is clear of obstacles, the object may be translated.

Definition 2.1 *Given a point p and a direction δ, the ray starting at p and going to infinity in direction δ is the* δ-shadow *of p.*

Definition 2.2 *The* δ-shadow *of a polyhedron is the union of the δ-shadows of all points in the polyhedron.*

We can now state formally the condition ensuring that a polyhedron can be separated in direction δ.

Definition 2.3 *Given a direction δ, a polyhedron P is* δ-blocked *by polyhedron Q if the δ-shadow of P intersects the interior of Q. A polyhedron P that is not δ-blocked by any other polyhedron is* δ-free.

Our definition of blocking permits polyhedra to make contact during translation, provided no interiors overlap. A *separating sequence* for a set $\mathcal{P}$ of polyhedra is a sequence of ordered pairs $(\mathcal{P}_i, \delta_i), 1 \leq i \leq |\mathcal{P}|$, such that $\mathcal{P}_i$ is δ_i-free with respect to the set $\mathcal{P} - \{\mathcal{P}_j : 1 \leq j < i\}$.

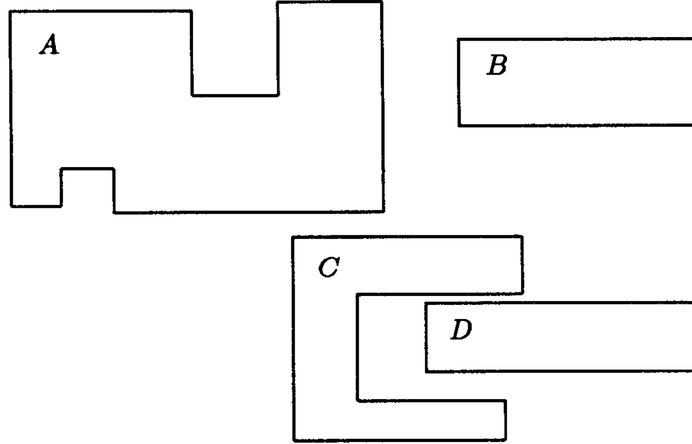


Figure 1: Separation in a set of 2-polyhedra

Definition 2.4 *A set of d-polyhedra $\mathcal{P}$ is:*

iso-separable	*with respect to a given direction σ if there is a separating sequence in which $\delta_i = \sigma$, for all $1 \le i \le \lvert\mathcal{P}\rvert$,*
one-separable	*if there exists a separating sequence,*
all-separable	*if it is iso-separable with respect to all $2d$ directions in Δ.*

Consider the set of two dimensional polyhedra in figure 1. The subset $\{A, B, C\}$ is iso-separable with respect to the positive y direction, subsets $\{A, C\}$ and $\{A, D\}$ are all-separable, and $\{D, C\}$ is not iso-separable for directions $\pm y$. The set $\{A, B, C, D\}$ is one-separable; one separating sequence is $\langle (A; +y), (B; +y), (D; +x), (C; -x) \rangle$.

We now briefly make some observations about the separability characteristics of sets of polyhedra.

Lemma 2.1 *Given a set of polyhedra $\mathcal{P}$,*

1. *Any subset of $\mathcal{P}$ that is all-separable is necessarily one- and iso-separable (for any direction).*
2. *A subset of $\mathcal{P}$ that is iso-separable for a given direction is one-separable.*

Proof: Straighforward from the definitions. □

3 Problem Definition

The definition of the $SEPARATION^{TM}$ problem for arbitrary dimension d is as follows:

Problem: Given a set $\mathcal{P}$ of disjoint simple orthogonal d-polyhedra with a total of N vertices, determine whether the set is one-separable and compute a separating sequence, if one exists.

The algorithm we present actually solves a slightly different problem: we compute a separating sequence s for a maximal subset of $\mathcal{P}$ that is one-separable. Thus, if $|s| = |\mathcal{P}|$, then $\mathcal{P}$ is one-separable.

Remark: From now on, the qualifiers "simple" and "orthogonal" are dropped for brevity.

3.1 A First Attempt

The "blind human" approach involves picking a polyhedron P, computing its δ-shadows, for all $\delta \in \Delta$, and determining whether some σ-shadow is free of intersections with other polyhedra, in which case we output the pair (P, σ), delete P from $\mathcal{P}$ and iterate. Otherwise, we simply pick a different polyhedron and continue. The algorithm terminates when either no polyhedron is free to move in any of the allowable directions or no polyhedra remain.

Due to the arbitrary nature of the objects, $O(N^2)$ choices are required in the worst case. For each choice, building the shadow of an object (which is related to computing its ortho-convex hull [MF82,NLLW82,OSSW84]) and finding intersections between polyhedra are relatively expensive processes. We now outline a strategy which yields a better algorithm.

4 Overview of the Algorithm

Recall that a polyhedron P is δ-free if it is not δ-blocked by any other polyhedra. In other words, P is δ-free if all the points in its interior are visible from infinity in direction δ. Our approach is reminiscent of hidden-line elimination: a projection hyperplane perpendicular to direction δ is placed at infinity and the visible contribution of each polyhedron is computed. Whenever all the points in the interior of some polyhedron become visible, the polyhedron is δ-free.

Definition 4.1 *Given a set of polyhedra $\mathcal{P}$ and a direction δ, the set of points visible from infinity in direction δ is called the* δ-view *of $\mathcal{P}$ and is denoted by V_δ.*

All $2d$ different views of the set $\mathcal{P}$ are maintained simultaneously. If some polyhedron P_i is entirely visible in some view V_σ, we output (P_i, σ) and delete P_i from all views. The algorithm terminates when no polyhedron can be moved or no polyhedra remain.

4.1 Building the View of a Set of Polyhedra

For each direction $\delta \in \Delta$, a hyperplane perpendicular to δ is swept through $\mathcal{P}$, from $-\infty$ to $+\infty$. The intersection of the sweeping hyperplane with $\mathcal{P}$ is a collection of $(d-1)$-dimensional polyhedra (for example, in three dimensions the intersection is a set of polygons). At each stopping point in the sweep, the intersection is inserted into a data structure representing the view V_δ of $\mathcal{P}$. When the insertion process is complete, the view structure must enable us to decide quickly which polyhedra (if any) are visible. Finally, efficient deletion of polyhedra must be supported.

Since the set of polyhedra is known in advance (no insertions or deletions of polyhedra are permitted), we preprocess the set and use a relatively simple semi-dynamic structure to hold the visibility information collected during the space sweep.

5 Three-Dimensional *SEPARATION*^{TM}

In order to illustrate the general technique, we solve the *SEPARATION*^{TM} problem for a set of 3-polyhedra. In the first subsection, an efficient representation for the visible portions of one view is investigated; the process involved in producing the views for the five other directions is similar. Then, we discuss augmenting the structure to hold information related to the freedom of the polyhedra. Finally, we consider the effects of deleting free polyhedra with the resulting update of freedom for the remaining objects.

5.1 Sweeping 3-Space

The space sweep algorithm for direction $+z$ follows:

1. Sort the facets of the polyhedra in non-decreasing order of z coordinates;
2. Build a skeletal hierarchical view structure V_{+z} for the collection of polygonal facets;
3. **for** $z = -\infty$ **to** $+\infty$ **do**
4. Insert the polygonal facets into V_{+z}

Step 1 set ups the event schedule required for the space sweep and the semi-dynamic skeletal view structure V_{+z} is built in step 2. As indicated above, the intersection of the sweep plane with the set of polyhedra is a set of polygons. The polygons are inserted into V_{+z} during steps 3 and 4.

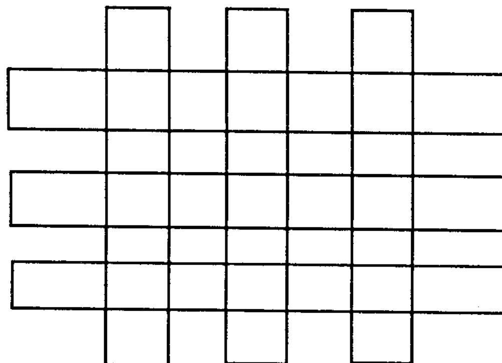


Figure 2: A trellis of $O(N)$ rectangles

The task of maintaining a set of polygons can be simplified by representing each polygon by a collection of elementary objects. Given a simple orthogonal polygon (possibly with holes) having a total of n vertices, a rectangular partition can easily be computed in $O(n \log n)$ time and space, yielding $O(n)$ rectangles. It is not necessary to insist on a minimal rectangular decomposition.

The *quadtree* [FB74] is a well-known structure that has been extended by [vLW81] to support the insertion and deletion of rectangles. The quadtree requires $O(N^2)$ space and $O(N)$ time for the insertion or deletion of a rectangle in the worst case, because of the possibility of "trellises": Given $O(N)$ rectangles, $O(N^2)$ rectangular regions are formed in the worst case (see Figure 2). Therefore, any algorithm based on quadtrees requires $O(N^2)$ time in the worst case. Recently, Overmars and Yap [OY88] introduced a new data structure suitable for sets of rectangles: *the orthogonal partition tree*, a generalization of the *k-d* tree [Ben75] exploiting the regularity of trellises. Their main observation is that all rectangular regions need not be represented explicitly; each trellis can be fully described by two sets of overlapping rectangles (the horizontal and vertical ones) at the boundaries of the bounding box of the trellis. As a consequence, the description of a scene of quadratic complexity requires only linear space.

We use the two-dimensional version of the orthogonal partition tree as a skeleton for the V_{+z} view structure. Two-dimensional space is partitioned and represented by binary tree T where:

1. The root of T represents the whole space.

2. Each internal node $v \in T$ has a *left* and *right* son and the region represented by v (denoted by R_v) is a (possibly unbounded) rectangle

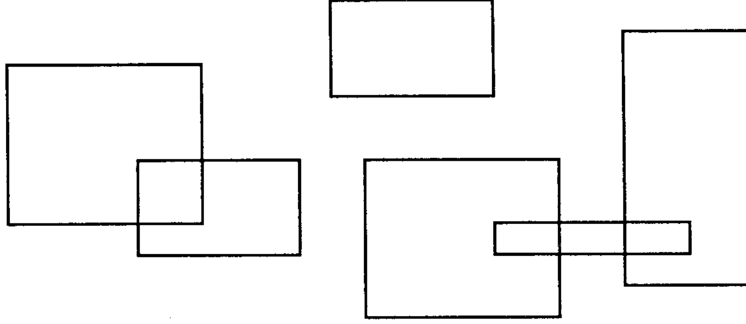


Figure 3: A set of rectangles

satisfying the following constraints:

(a) $R_v = R_{left(v)} \cup R_{right(v)}$; and

(b) $R_{left(v)} \cap R_{right(v)} = \emptyset$.

3. The region R_v of each leaf $v \in T$ is a rectangular *cell.*

We shall now describe the construction of the orthogonal partition tree for a set of rectangles (for example, see Figure 3). Let X be the set of distinct x-coordinates of the vertical boundaries of the rectangles ($|X| = O(N)$). The x axis is divided into $\sqrt{N}$ *strips* such that each strip contains at most $2\sqrt{N}$ coordinates from X. Each strip is divided into $O(\sqrt{N})$ cells by carefully chosen horizontal segments. For a given strip s, let V be the set of rectangles having a vertical boundary inside s, and H the set of rectangles having only horizontal boundaries in s (a rectangle in H traverses the strip completely). Thus, $|V| \leq 2\sqrt{N}$, and $|H| = O(N)$. Now, we partition s into cells by drawing horizontal line segments through both horizontal boundaries for each rectangle in V, and through each $\sqrt{N}$-th horizontal boundary of the rectangles in H (refer to Figure 4). As a result, each strip is partitioned into at most $6\sqrt{N}$ cells.

The tree is built from the partition by merging neighbouring cells in a strip, and then merging neighbouring strips. The upper levels of the tree split the space according to the x-coordinates, the lower levels according to the y-coordinates, and the leaves store the cells.

Lemma 5.1 *An orthogonal partition tree T for N rectangles has the following properties.*

1. *There are $O(N)$ nodes in T.*
2. *Each rectangle appears in $O(\sqrt{N})$ leaves.*
3. *Each vertex appears in no cell.*

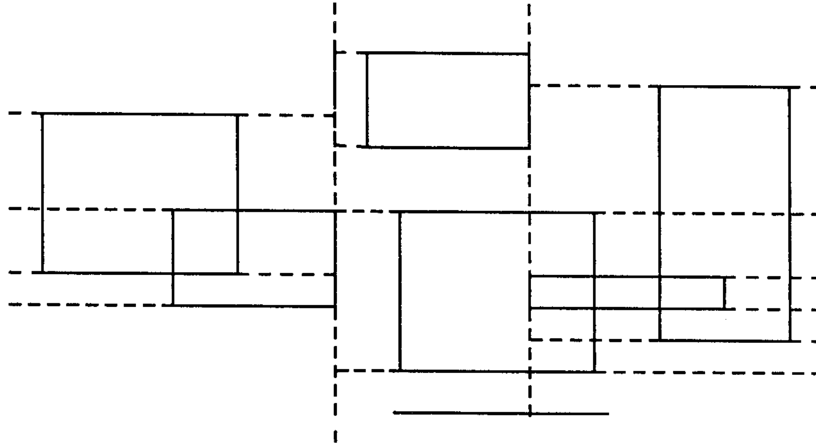


Figure 4: The partition resulting from a set of rectangles

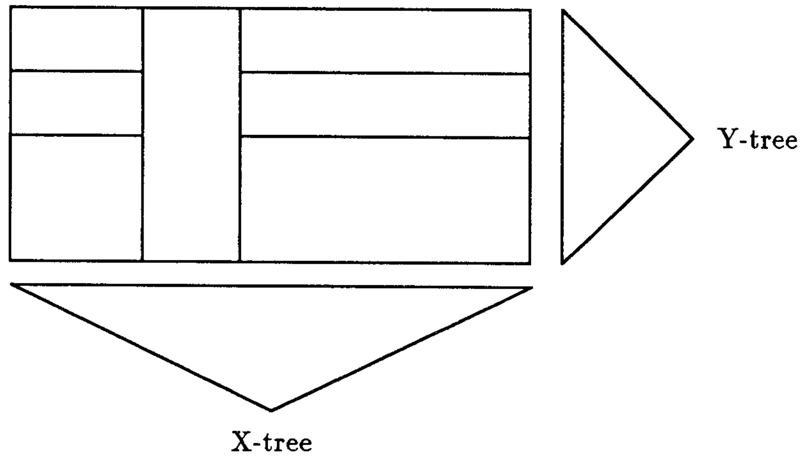


Figure 5: A cell and its associated trees

4. *Each leaf or cell contains* $O(\sqrt{N})$ *rectangles.*
5. *The height of* T *is* $O(\log N)$.

Proof: See [OY88] for proofs of these statements. □

Rectangles do not appear at internal partition nodes, since they are stored in the leaves[2]. By property 3, any rectangle stored in a leaf traverses the associated cell completely. The portion of rectangle r that is inside the cell R_v of leaf v is called a *fragment* of r. For a rectangle r and leaf v, if $r \cap R_v \neq \emptyset$ three situations arise:

[2]The situation is quite different in the quadtree, where all levels of the tree can hold rectangles.

1. r contains or covers R_v;

2. $r \cap R_v$ is an x-slab in v's cell; or

3. $r \cap R_v$ is a y-slab in v's cell.

An *x-slab* is a fragment which has at least one vertical boundary inside cell R_v, and a *y-slab* is a fragment which has only horizontal boundaries inside R_v. A cell is represented by two segment trees [Ben77,PS85]; the x-slabs are stored in an X-tree, the y-slabs in a Y-tree, and the covering fragments in either tree (see Figure 5).

Inserting a rectangle r entails identifying the leaves $v \in T$ whose regions intersect r. Properties 2 and 5 of the partition tree imply that $O(\sqrt{N})$ time is required to find the resting places of the fragments of r. By property 4, each leaf holds $O(\sqrt{N})$ fragments and, therefore, the heights of its two segment trees are $O(\log \sqrt{N}) = O(\log N)$.

5.2 Augmenting the View Structure and Detecting Visible Objects

The focus of this subsection is the computation of the visible portions of the set of polyhedra $\mathcal{P}$ stored at the leaves of the orthogonal partition tree.

During the space sweep, each polygonal facet is partitioned into rectangles and inserted into the view structure yielding a number of fragments (if the facet has $O(n)$ vertices, it is decomposed into $O(n\sqrt{N})$ fragments). When a fragment is inserted into a cell, it is further partitioned into several non-overlapping intervals represented by nodes in the appropriate segment tree; each such interval is called a *sliver* of the fragment. For each leaf v in the partition tree, the number of visible slivers of each polyhedron $\mathcal{P}_i, 1 \leq i \leq |\mathcal{P}|$, is maintained in the set $VISIBLE(v)$; the member of $VISIBLE(v)$ corresponding to polyhedron $\mathcal{P}_i$ is called the *representative* of $\mathcal{P}_i$. Naturally, even though there is a representative for polyhedron $\mathcal{P}_i$ in the *VISIBLE* set, it is possible that some part of $\mathcal{P}_i$ is obscured by other polyhedra. The value $count\,(i), 1 \leq i \leq |\mathcal{P}|$, is the total number of slivers inserted for each polyhedron. When the number of visible slivers for $\mathcal{P}_i$ equals $count\,(i)$, the entire polyhedron is $+z$-free.

Note that the segment trees of a cell do not represent disjoint space, so care must be exercised in determining visibility: a single y-slab at height z is sufficient to block any number of x-slabs at heights $< z$, and vice-versa. To solve this problem, a fragment in inserted into *both* trees: each slab completely covers the cell in one direction and, therefore, is stored at the root of one of the segment trees. If a fragment covers the entire cell region, it is represented by two covering intervals.

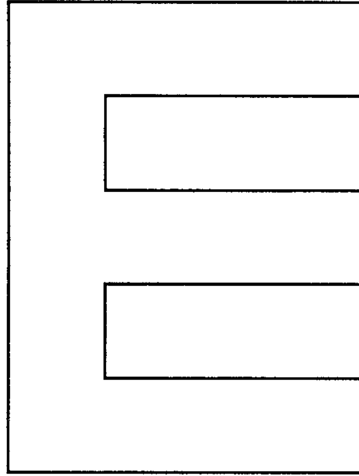


Figure 6: A comb in x-z cross section.

For the remainder of this discussion, we focus our attention on the X-tree, the Y-tree being analogous.

5.2.1 Inserting fragments into the X-tree

With each fragment f we associate the value $f.z$ of the z coordinate of its facet, the name of the originating polyhedron $f.poly$, and the interval $[f.x_1, f.x_2]$. Two fragments f and g are *siblings* if $f.poly = g.poly$. Each node u in the X-tree has a left and right son (denoted by $left(u)$ and $right(u)$) and represents the x interval $[u.x_1, u.x_2]$; the x-interval of the root of the X-tree is the x-extent (or "width") of the cell.

During insertion, if fragment f covers node u, a sliver of f is added to the *COVER* set of u (the insertion routine is given below). The *COVER* set is implemented as a doubly-linked list and the element with maximum z value is found using $TOP(COVER(u))$. The data structure used to implement the slivers has the following fields:

poly	the identity of the originating polyhedron,
z_{min}, z_{max}	$[z_{min}, z_{max}]$ is the z-interval covered by the sliver,
weight	the number of sibling slivers that are $+z$-free in $[z_{min}, z_{max}]$.

To see why our data structure is reasonable for slivers, consider the "comb" drawn in x-z cross-section in Figure 6. The teeth of the comb give rise to a sequence of consecutive sibling slivers stored at the same node. The situation is simplified by replacing consecutive sibling slivers by a single one representing the interval $[z_{min}, z_{max}]$, where z_{min} and z_{max} are the minimum

and maximum z coordinates of the slivers. The *weight* field is adjusted accordingly. A single sliver of some fragment f corresponds to the interval $[f.z, f.z]$, and has $weight = 1$. The insertion routine used to insert fragments into cell trees is given below.

```
procedure Insert (f, u);
/* Insert slivers corresponding to fragment f into the tree rooted at node u. */
begin Insert
    if f covers u then
        if COVER(u) = ∅ orelse f.poly ≠ TOP(COVER(u)).poly then
            COVER(u) := ⟨ [f.poly, f.z, f.z, 1], COVER(u) ⟩
        else
            /* Combine sibling slivers */

            let Candidate = TOP(COVER(u));
            Candidate.z_max := f.z;
            Candidate.weight := Candidate.weight + 1;
        fi
    else    /* Insert into the subtrees, if necessary */

        if f.x_1 < ⌊(u.x_1 + u.x_2)/2⌋ then
            Insert (f, left(u))
        fi;

        /* Similarly for the right son of u. */
end Insert;
```

5.2.2 Determination of $+z$-freedom

Since the space is swept from $z = -\infty$ to $+\infty$, a sliver appearing at the head of a *COVER* list has locally maximal z-value and is a candidate for freedom. Sliver s blocks sliver t if the following conditions hold:

1. the intersection of the interiors of s and t is non-empty;

2. s and t are not siblings; and

3. s is above t.

By condition 1, any sliver that blocks a candidate s stored at some node u must overlap the interval defined by u. A larger blocking sliver is necessarily in the *COVER* set of a proper ancestor of u, while any smaller blocking sliver must appear in the subtrees rooted at u.

The key idea is to propagate candidates up until either they reach the root, or they are blocked. When a candidate reaches the root and it is not blocked by any slivers in *COVER*(root), then it is truly $+z$-free and we update the visibility information at the corresponding leaf of the partition tree.

Actually, rather than moving the slivers bodily up the tree, whenever a sliver s from node u would migrate to its parent v, we send a copy of s to the parent of u in the guise of representative r, and add r to the set *VISIBLE*(v). Slivers are distinguished from representatives in that a sliver is inserted at a particular node in a segment tree while a representative is a mobile agent working on the behalf of a polyhedron. Further differences are elucidated below. The migration continues until either r is blocked by some sliver or it reaches *VISIBLE*(X-tree). Thus, each X-tree node u is augmented with the set *VISIBLE*(u) of representatives corresponding to the candidate slivers stored in the subtrees rooted at u that are blocked by some member of *COVER*(u). If node u is a leaf of the X-tree, *VISIBLE*(u) $= \emptyset$.

The initial construction of the *VISIBLE* set for each segment tree node is done during a post-order traversal; routine BuildVisible (given below) computes the contribution of a node to the *VISIBLE* set of its parent. When the process is complete, the set *VISIBLE*(root) consists of all representatives which are blocked, at worst, by slivers in *COVER*(root). Then, the visible fraction of each polyhedron is found by applying the routine BuildVisible a final time to the root of both segment trees. The *weight* fields of the representatives in the *VISIBLE* set of the partition leaf reflect the visible portions of the different polyhedra.

5.2.3 Implementing the *VISIBLE* set

The *VISIBLE* set V associated with a node must support several different operations:

Insert (r, V)	adds the representative r to V;
FindMax (V)	returns a reference to the representative with maximal z_{max} value;
DeleteMax (V)	returns the representative with maximal z_{max} and deletes it from V;
MaxNonSibling (r, V)	returns the representative with maximal z_{max} value of all non-siblings of r, if there are any in V; and
FoundSibling (r, s, V)	returns **true** if there is a sibling of r in V (s is the sibling) and **false**otherwise.

Each *VISIBLE* set is implemented by a priority search tree [McC85], a structure combining searching and priority queue operations in $O(\log n)$ time and $O(n)$ space, where n is the size of the set. The primary key is the polyhedron name and the priority key is the z_{max} field of the representatives. Since the number of fragments is a linear function of N, all operations on *VISIBLE* sets can be performed in $O(\log N)$ time. The routine AddVisible(Rep, Visible) adds the representatives $r \in$ Rep to the set Visible.

procedureAddVisible (Rep, Visible);
beginAddVisible
    ifVisible = $\emptyset$ **then**
        Visible := Rep;
    else
        for$r \in$ Rep **do**
            ifFoundSibling (r, s, Visible) **then**
                /* Combine siblings */

                $weight(r) := weight(r) + weight(s)$;
                $z_{max}(r) := \max\ (z_{max}(r), z_{max}(s))$;
                $z_{min}(r) := \min\ (z_{min}(r), z_{min}(s))$;
            fi;
            Insert (r, Visible)
        od
endAddVisible;

If *VISIBLE*(u) has a representative r for some polyhedron, then r absorbs all subsequent sibling representatives by updating the z-interval and the weight. The result is that if all slivers of a polyhedron are free below u, then no representatives of that polyhedron remain in the subtrees rooted at u.

Routine BuildVisible (child,parent) computes the contribution of the child node to the *VISIBLE* set of the parent. The process of determining which slivers should be propagated deserves explanation. Let sliver s be the candidate for freedom at node u. Two cases must be considered:

s is free,	all non-siblings $r \in$ *VISIBLE*(u) are blocked, so dispatch a representative for s to the parent.
s is not free,	some non-sibling representative blocks s, propagate all free $r \in$ *VISIBLE*(u).

What happens to $t \in$ *VISIBLE*(u), a sibling of s? It may or may not be blocked by some *other* member of *COVER*(u). Since all adjacent sibling slivers are coalesced, the freedom of t can be ascertained by considering the

second member of $COVER(u)$ (if any). The detailed algorithm for Build-Visible appears below.

```
procedureBuildVisible (child, parent);
beginBuildVisible
1.  ifCOVER(child)= ∅ then
2.      AddVisible ( VISIBLE(child), VISIBLE(parent))
3.  elsifVISIBLE(child) = ∅ then
4.      AddVisible (TOP(COVER(child)), VISIBLE(parent))
5.  elsif
6.      letCand = TOP(COVER(child));
7.      letReps = VISIBLE(child);

        /* Check the sibling of Cand for freedom */

8.      SiblingIsNotFree := false;
9.      ifFoundSibling (Cand, Sibling, Reps) then
10.         ifTOP(COVER(parent) − Cand) = ∅ orelse
11.                 z_min (Sibling) > z_max (TOP(COVER(parent) − Cand)) then
12.             AddVisible ({ Sibling }, parent)
13.         else
14.             SiblingIsNotFree := true
15.         fi
16.     fi;

        /* Is the candidate Cand free to move? */

17.     ifweight (Cand) > 0 andthen
18.                 z_min (Cand) > z_max (MaxNonSibling (Cand, Reps)) then

19.         AddVisible ({ Cand }, VISIBLE(parent));
20.         weight (Cand) := 0
21.     else
            /* Cand is blocked, look for free representatives */

22.         Free := ∅;
23.         whileReps ≠ ∅ andz_max (Cand) < z_min (FindMax (Reps)) do
24.             Free := Free ∪ DeleteMax (Reps)
25.         od;
26.         AddVisible (Free, VISIBLE(parent));

            /* Sibling may have to be re-inserted into VISIBLE(child) */
```

```
27.         ifSiblingIsNotFree then
28.             Insert (Sibling, VISIBLE(child))
29.         fi
30.     fi
31. fi
endBuildFree;
```

Steps 1 through 4 handle the boundary conditions for the *COVER* and *VISIBLE* sets. Steps 8 through 14 concern the sibling of $TOP(COVER(\text{child}))$, the current candidate for freedom. If there is a sibling representative, it is removed from the *VISIBLE* set in $O(\log N)$ time and processed separately. Then, the freedom of the candidate and the remaining representatives is decided according to the rules outlined above. Finally, if the sibling representative is not free (the second member of *COVER*(child) blocks it), then we re-insert it into *VISIBLE*(child).

5.3 Deleting a free polyhedron

When a free polyhedron is found, its slivers must be removed from all partition trees. To accomplish this the view structure is modified one last time by threading the slivers of each polyhedron into the singly-linked list *slivers* $(i), 1 \leq i \leq |\mathcal{P}|$, as they are inserted into the partition tree. To delete a polyhedron, the list of its slivers is traversed and each one is deleted from its *COVER* set in constant time. Note that the deleted slivers are not necessarily the maximal ones, since the polyhedron may be free in some other view.

As noted above, a nice side-effect of the construction of the *VISIBLE* set is that if a polyhedron is free, none of its representatives will appear in any *VISIBLE* sets in the segment trees, thereby avoiding having to find and delete them. When a sliver is deleted from the *COVER* set of a node u, we must compute the freedom of the new candidate at u and potentially all of its ancestors on the path from the root to u. To find the newly freed slivers, the contribution of u to its parent is re-computed by invoking BuildVisible (u, parent (u)) and rebuilding the *VISIBLE* sets until either the leaf of the partition tree is reached or no new representatives are generated.

Each sliver must not be counted more than once otherwise spurious representatives will be generated. Fortunately, the weight field of a free sliver is set to 0 in BuildVisible (step 20), so it is never reconsidered for freedom.

6 Analysis of 3D Separation

Theorem 6.1 *The three dimensional Separation problem can be solved in $O(N\sqrt{N}\ \log^3 N)$ time and $O(N\sqrt{N}\log N)$ space.*

Proof: In three dimensions, there are $2d = 6$ allowable directions and the dimensionality d is independent of the complexity of the scene N.

The preparation of the event schedule for the space sweep requires $O(N\log N)$ time to sort the z coordinates of the facets. The decomposition of the polygonal facets yields $O(N)$ rectangles, and each rectangle is broken up into $O(\sqrt{N})$ fragments. Inserting a fragment involves updating $O(\log N)$ nodes in each of two segment trees in constant time for each node. Thus, inserting all polyhedra takes $O(N\sqrt{N}\log N)$ time.

Each representative contributes to at most $O(\log N)$ *VISIBLE* sets (since this is the height of the segment trees). The time required to generate a representative and to migrate from a child node to its parent is $O(\log N)$ (each *VISIBLE* set holds $O(N)$ representatives). Therefore, for each representative $O(\log^2 N)$ steps suffice to reach the *VISIBLE* set of the partition leaf. The number of representatives is bounded from above by the number of slivers so the total time to detect the freedom of all polyhedra is $O(N\sqrt{N}\log^3 N)$.

Once the *VISIBLE* sets for the partition leaves are constructed, a simple $O(N)$ time scan of all leaves is required to detect whether any polyhedron is free to move. If a polyhedron is free, deleting its slivers takes constant time for each sliver, thus a total of $O(N\sqrt{N}\log N)$ time. When a sliver is deleted from node u, BuildVisible is re-applied to (at worst) every node on the path from the root to u. Again, $O(\log N)$ time is spent at each node and the length of the path is $O(\log N)$. In summary, we expend $O(\log^2 N)$ time during the deletion of each sliver, for a total of $O(N\sqrt{N}\log^3 N)$.

Thus, the three dimensional algorithm requires $O(N\sqrt{N}\log^3 N)$ time.

The space required by the algorithm is easy to compute: $O(N\sqrt{N}\log N)$ slivers and representatives are constructed, each of which requires a constant amount of space. □

7 *SEPARATION*^{TM} in d-dimensions

As outlined in [OY88] the orthogonal partition tree can be generalized to d dimensions ($d \geq 2$) relatively easily, and it is possible to extend our algorithm to higher dimensions.

Lemma 7.1 *A d-dimensional orthogonal partition tree for N d-dimensional intervals has the following properties:*

1. *the tree has* $O(N^{d/2})$ *nodes;*
2. *each d-interval is stored in* $O(N^{(d-1)/2})$ *leaves;*
3. *no cell of a leaf contains any vertices; and*
4. *each leaf holds no more than* $O(\sqrt{N})$ *d-intervals.*

Proof: See [OY88]. □

We build a $(d-1)$ dimensional partition tree whose leaves represent $(d-1)$-dimensional cells by splitting first on the x_1 coordinate, then the x_2, and so on. Each cell is described by $(d-1)$ augmented segment trees identical to those used in the three-dimensional variant and the same algorithm will work correctly.

Theorem 7.2 SEPARATION^{TM} *in d dimensions for polyhedra with a total of N vertices can be solved in* $O(dN^{d/2}\log^3 N)$ *time and* $O(dN^{d/2}\log N)$ *space.*

8 Conclusions and Open Questions

We have presented a general solution to *SEPARATION*^{TM} in arbitrary dimensions. In this section, we explore possible extensions and mention some open problems.

8.1 Two-Dimensional *SEPARATION*^{TM}

In the original paper on *SEPARATION*^{TM} , [COSSW84] give an outline of an algorithm for the two dimensional version of the problem with a run time of $O(N\log^2 N)$. The original algorithm is unfortunately incorrect, because the edges of the polygons are not labeled with the polygon identity. The algorithm concludes that a single polygon is unable to move since the leading facets block the trailing ones.

The process involved in determining the visible slivers in a cell is identical to solving the two dimensional variant of *SEPARATION*^{TM} . We can state:

Theorem 8.1 *Solving two-dimensional* SEPARATION^{TM} *for a set of disjoint simple orthogonal polygons having a total of N vertices can be done in* $O(N\log^3 N)$ *time and* $O(N\log N)$ *space.*

Proof: We use a single augmented segment tree. The $O(N)$ line segments (for a given direction) defining the polygons are broken up into $O(N\log N)$ slivers by the cell tree. From this observation, the arguments used in the analysis of 3D *SEPARATION*^{TM} carry through. □

8.2 Efficiency Considerations

For a given orthogonal direction δ, a polyhedron P is said to be *δ-ortho-convex* if the intersection of P with each δ-oriented line is empty or connected. The *δ-ortho-convex* hull of a polyhedron P is the smallest δ-ortho-convex polyhedron containing it. In order to improve the practical efficiency of our algorithm, we can initially replace each polyhedron by its δ-ortho-convex hull to ensure that each polyhedron's projection is covered by exactly two faces of the hull — the leading and trailing faces — thus potentially reducing the number of appearances of each polyhedron in the structure, without incurring significant time cost since the ortho-convex hull of a polyhedron can be computed in $O(N \log N)$ time [KS86].

8.3 Further Research

Several avenues for further research remain open. An obvious question concerns the optimality of our solutions. The only lower bound we have is $O(N \log N)$ and it seems difficult to improve it. Is there an algorithm solving *SEPARATION*^{TM} (even in two dimensions) in $O(N \log N)$ time? If so, all of our upper bounds can be tightened. Concerning the space cost, Overmars and Yap [OY88] mention the use of *streaming* to reduce the storage required for the orthogonal partition tree to $O(N \log N)$. Is streaming applicable in our case?

Allowing more than one move per polyhedron would give different problems of the form: can we separate a polyhedron in two consecutive moves? in 3 moves? in k? Note that whether we insist that all moves affecting a polyhedron be consecutive or we allow the moves to be interleaved with the moves of other polyhedra is significant.

What if we replace each asteroid warhead by a small engine with limited fuel capacity and ask: compute a sequence separating the asteroids (in the sense that each asteroid escapes from the convex hull of the initial collection) and minimizes fuel consumption?

References

[Ben75] J.L. Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM*, 18:509–517, 1975.

[Ben77] J.L. Bentley. Algorithms for Klee's Rectangle Problems. Technical report, Department of Computer Science, Carnegie-Mellon University, 1977. Unpublished notes.

[COSSW84] B. Chazelle, T. Ottmann, E. Soisalon-Soininen, and D. Wood. The Complexity and Decidability of *SEPARATION*^{TM} . In J. Paradaens, editor, *ICALP'84*, pages 119–127. Springer-Verlag Lecture Notes in Computer Science 172, 1984.

[DS86] F. Dehne and J.R. Sack. Separability of Sets of Polygons. Technical Report SCS-TR-82, School of Computer Science, Carleton University, 1986.

[FB74] R.A. Finkel and J. L. Bentley. Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta Informatica*, 4:1–9, 1974.

[GY80] L. J. Guibas and F. F. Yao. On Translating a Set of Rectangles. In *SIGACT Symposium on Theory of Computing*, pages 154–160, 1980.

[KS86] D. G. Kirkpatrick and R. Seidel. The Ultimate Planar Convex Hull Algorithm. *SIAM Journal on Computing*, 15(1):287–299, 1986.

[McC85] E. M. McCreight. Priority Search Trees. *SIAM Journal on Computing*, 14:257–276, 1985.

[MF82] D.Y. Montuno and A. Fournier. Finding the $x-y$ Convex Hull of a Set of $x-y$ Polygons. Technical Report 182, University of Toronto, CSRG, 1982.

[NLLW82] T.M Nicholl, D.T. Lee, Y.Z. Liao, and C.K. Wong. Constructing the X-Y Convex Hull of a Set of X-Y Polygons. Technical report, IBM Research Center, 1982.

[Nur87] O. Nurmi. *Algorithms for Computational Geometry.* PhD thesis, Karlsruhe, 1987.

[OSSW84] T. Ottmann, E. Soisalon-Soininen, and D. Wood. On the definition and computation of rectilinear convex hulls. *Information Sciences*, 33:157–171, 1984.

[OY88] M. H. Overmars and C.K. Yap. New upper bounds in Klee's measure problem (extended abstract). In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pages 550–556, 1988.

[PS85] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, 1985.

[Tou85] G. T. Toussaint. Movable Separability of Sets. In G.T. Toussaint, editor, *Computational Geometry*, pages 335–375. Elsevier Science Publishers, North-Holland, 1985.

[vLW81] J. van Leeuwen and D. Wood. The Measure Problem for Rectangular Regions in d-Space. *Journal of Algorithms*, 2:282–300, 1981.