Masaryk University, Brno

Faculty of Informatics

Bounding Volumes in Computer Graphics

Master thesis

April 1998, Brno

Petr Konečný

Zadání

Prostudujte publikované metody, které používají obalová tělesa a jejich hierarchie v počítačové grafice. Zaměřte se na úlohy řešení globální iluminace a detekce kolizí. V písemné části vypracujte porovnání vybraných metod a teoreticky zpracujte nově navženou metodu.

V praktické části práce vyzkoušejte některé z těchto metod a porovnejte výsledky získané při vyhodnocení experimentálních scén.

Task

Investigate published methods that use bounding volumes and their hierarchies in the computer graphics. Focus on the problems of global illumination and collision detection. In written part compare selected methods and theoretically examine new method.

In practical part implement some of these methods and compare results obtained on experimental scenes.

Prohlášení

Prohlašuji, že tato práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal a z nichž jsem čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

Declaration

I declare that this thesis is my original work and that I have written it independently. All sources and literature that I have used during elaboration of the thesis are cited with complete reference to the corresponding source.

Acknowledgement

First of all I would like to thank Jiří Sochor, the supervisor of the thesis, for numerous advices and invaluable discussions during my work on it.

This work would not be possible without Karel Zikan who has drawn my attention to the subject and provided support during the research as well as enormous amount of help in many mathematical topics. My thanks go also to Henry Sowizral of Sun Microsystems for support during first implementation of these ideas.

I would also like to thank researchers from SUNY at Stony Brook who kindly provided their implementation for comparison purposes, namely Martin Held, James Klosowski and Joseph Mitchell.

The work on this thesis would be much harder without existence of Debian GNU/Linux distribution. Each letter in this thesis (and then some more) was typed on a keyboard attach to this system.

Abstract

In this thesis we show how the techniques based on bounding volume hierarchies can be used to solve various problems in computer graphics. We concentrate mainly on the problem of collision detection and a more general task of proximity computation. We employ a special class of bounding volumes – fixed directions hulls – to implement time and memory efficient algorithms for these problems. The fixed directions hulls are analyzed in a detail and some rarely used methods are utilized in implementation of basic operations. A new approach to traversal of the hierarchy is proposed together with possible applications in raytracing.

Keywords: Spatial hierarchies, bounding volumes, proximity computation, collision detection, fixed directions hulls, discrete orientation polytopes.

Contents

Chapter 1

Introduction

Major development of computer graphics applications in the last few decades led to the need of processing large amounts of graphical data. It became evident that "brute-force" approaches are very inefficient for most of the problems. Plenty of methods for storage of data and access to stored data were developed during this period to overcome inefficiency of brute-force methods. These methods range from simple database techniques to employing complicated data-structures in modern computational geometry. It soon became evident that maintaining some basic consistency properties of large volumes of data will be a very hard task.

Though very efficient methods for data management of geometrical data were developed in the theory, their practical implementations are often subject to errors caused by degeneracies of input sets. This is probably caused by the fact that handling singular cases in theoretical work is usually very tedious task and such cases are often discarded in theorems that are bases of algorithms. In this work I shall analyze methods that lay between straightforward approaches of the past and intricate methods of computational geometry.

Here we concentrate mainly on the problem of *collision detection* or more general *proximity computation*. By proximity computation I mean an algorithm for finding a distance between pairs of possibly moving objects or its lower bound. The collision detection is then a method to check whether this distance is zero. For practical use the distance itself is often insufficient and an information about closest or intersecting parts of object is usually required. These problems often arise in many areas of computer graphics and many real-world problems can be efficiently transformed to these tasks.

Examples of application of proximity computation can be found in many CAD systems. One of the tasks these systems have to cope with is checking if the given model satisfies some engineering constraints, such as required separation of heat sources from flammable material inside the model of a building or ensuring minimum distance of the hydraulic systems of an aircraft from the aircraft's moving parts. Another large area of use arises in virtual reality where it can be used to provide the feeling of obstacles in the virtual environment to the user.

Problem of *global illumination* is another topic addressed in this thesis, though not in much detail. Methods for solving these tasks often rely on collision detection in some form. For example experiments in raytracing show that the most expensive part of the process is finding intersections (i.e. collisions) of rays with rendered objects. Some of the ideas from raytracing are actually used even in this work, namely the class of bounding volumes used in hierarchies [Ka-86].

In the thesis I focus on using bounding volumes and their hierarchies for solving these problems.

1.1 Previous work

Naturally the problems mentioned in the previous section were addressed in the past. The methods related to those in this thesis can be divided into two major classes. They either decompose the space occupied by objects or approximate the objects with bounding volumes of some kind. The aim is to decrease the number of pairs of primitive objects that need to be considered.

The former class includes octrees, k-d trees, BSP-trees, brep-indices, tetrahedral meshes and regular grids. The core of these methods is dividing the space into cells and checking for collision only for pairs of objects that share the same cell. To further decrease running time some of these methods use hierarchical data structures (octrees, BSP-trees, etc.).

The idea behind the methods of the latter class is that if we approximate objects by simpler bounding volumes we can often discard pairs of objects from further considerations by checking that their bounding volumes are sufficiently separated. Moreover by employing hierarchies we can make these approximations more and more accurate, reaching the geometry of actual objects at some point. The bounding volumes most often used in these methods are spheres and axis-aligned bounding boxes, since it is extremely easy to check for intersection of pairs of such objects.

We chose the problem of proximity computation as a target application of the methods of this thesis. There are two variants of it that can be recognized, static and dynamic. While the these problems have been analyzed thoroughly in computational geometry and efficient algorithms are known, the majority of them is limited to only static environments or remains unimplemented.

The methods based on bounding volume hierarchies became very popular in the last decade. They usually differ in the choice of bounding volumes. Axis aligned bounding boxes are probably the first that were used for this purpose. They are often employed in GIS systems and their multidimensional extension is quiet easy, for instance see [Be-90, He-95]. Other bounding volumes include spheres, ellipsoids, cubes, etc.

One of publicly available implementations of these methods is the system RAPID from [Go-96]. The bounding volume used in it is oriented bounding box, i.e. bounding box with arbitrary orientation. Its authors reimplemented previously known method of collision test and used statistical methods for building the hierarchy. Nowadays their implementation is often used for comparison of different methods. The discrete orientation polytopes introduced probably in [Ka-86] are utilized for instance in [He-96, Za-95, Za-98] and we also use the same class of bounding volumes.

1.2 Overview

This thesis consists of four chapters. The first of them – Introduction – is finished by this overview. In Chapter 2 we define the class of fixed directions hulls, slightly more general term than discrete orientation polytopes used in [Ka-86]. We show how to represent these bounding volumes in memory efficient manner and describe some basic algorithms that are used later in Chapter 3. We introduce notion of hierarchical approximation of the object by the means of tree of bounding volumes. Different approaches to building such data structure are discussed and then the hierarchy is applied on the problem of proximity computation and as a special case on ray-tracing. In the final chapter we summarize pros and cons of our method and show ways for possible improvements and extensions of the work.

Chapter 2

Fixed Directions Hulls

To design efficient methods for proximity computation we can start by analyzing the bottlenecks of brute-force approaches. Clearly the most expensive part is the all-pairs collision test. Its super-quadratic complexity is an overkill when it comes to objects with the number of subparts in the order of thousands. The first crucial but simple observation is that if we substitute complicated object by larger one we can often discard further computations just by testing for collision of these *bounding volumes* and if they do not intersect we can skip further testing. However there are some pitfalls that have to be avoided.

The bounding volume must be simpler than the bounded object. It would be really useless if we would bound a sphere by some non-convex star-shaped polyhedron. On the contrary sphere can be often used as a bounding volume to simplify computations. Another important property of practically useful bounding volumes is their tightness.

Keeping issues from previous paragraphs in mind we will define a class of bounding volumes useful for solving problems discussed in this thesis, show some of its properties, some basic ideas and algorithms that are based on them. The same class of bounding volumes was probably used for the first time in [Ka-86] to implement efficient raytracing algorithm. It is direct generalization of the axis-aligned bounding boxes. Actually AABBs are just an instance of this approach. AABB of a given object is defined as the smallest box with edges parallel to coordinate system axes that contains it. For the purpose of generalization we can also view it as an intersection of three (in three dimensions) slabs orthogonal to coordinate system axes.

However the only reason to limit ourselves to only three slabs is that it is evident how to find an intersection of two such objects.

AABBs intersect iff all three pairs of parallel slabs overlap. This condition can be easily checked. To extend the idea of AABB we can use more than three slabs with fixed normals. Clearly each slab is given by intersection of halfspaces with opposite normals, thus to go even further we can use a general set of bounding halfspaces instead of the set of slabs.

We proceed be defining some auxiliary symbols. Let $d \in \mathbb{R}^k$ be a vector of unit length ($\|d\| = 1$) and $X \subseteq \mathbb{R}^k$ be a set of points. We define

$$b_d(X) = \begin{cases} -\infty & \text{if } X = \emptyset, \\ \sup\{d^\top x | x \in X\} & \text{otherwise.} \end{cases}$$

This value clearly becomes $+\infty$ for sets unbounded in the direction d. Using this value we define *bounding halfspace* of X in the direction d

$$H_d(X) = \{x | d^\top x \le b_d(X)\}$$

This set is empty or whole space if X is empty or unbounded in the direction d respectively, otherwise it is really a closed halfspace. However this is the only case we want to solve. Points in the set

$$\{x | d^\top x = b_d(X)\}$$

then form *supporting hyperplane* of X in the direction d.

From the previous definition of $H_d(X)$ we see that this set is always superset of the corresponding set X. Moreover it is also the smallest closed halfspace with the normal d that contains X. Now we have come to the point of the main definition. Let $D = \{d_1, \dots, d_n\}$ be a fixed set of n vectors (directions) of unit length. Given bounded and non-empty set X we say that the collection of halfspaces $(H_d(X))_{d \in D}$ is (D-induced) *fixed directions hull*[1] of X. We will use the notation

$$\mathrm{FDH}_D(X) = (H_d(X))_{d \in D}$$

Since we know that X is a subset of $H_d(X)$ we immediately see that X is also a subset of the intersection of these bounding halfspaces. Because every bounding halfspace is minimal it follows that this interestion is also minimal such that it contains X. The question is why we did not define FDH simply as this intersection.

[1] An alternate name for what we call an “FDH” is the term discrete orientation polytop (“DOP”) [He-96] - an easier to pronounce abbreviation, but perhaps slightly less precise.

To answer this question we first show that these notions are not the same. Let us consider an example in two dimensions. As a set of directions we take the set D_8 of normals of edges of regular octagon i.e.

$$D_8 = \{\pm(1,0), \pm(0,1), \pm(1,1)/\sqrt{2}, \pm(1,-1)/\sqrt{2}\}$$

and for the set X we choose the triangle with vertices $(0,0), (0,1)$ and $(1,0)$. Then $\text{FDH}_D(X)$ is the collection of halfspaces defined by the following eight inequalities:

$$0 \le x \le 1, 0 \le y \le 1, 0 \le x+y \le 1, -1 \le x-y \le 1 \tag{2.1}$$

while the corresponding intersection can be most easily described by only three inequalities:

$$0 \le x, 0 \le y, x+y \le 1$$

However it is obvious that this set can be described in many different ways. We can for instance change up to five inequalities in (2.1) to get the same set. Figure 2.1 shows graphical presentation of these simple ideas.

Because many of the methods presented in this chapter rely on the representation of FDH as a set of inequalities this ambiguity would certainly lead to the poor performance and maybe even incorrect behavior of these algorithms.

Although we have shown just that intersections and collections of halfspaces are not the same, we can (and will) use these concepts interchangeably most of the time. Such expediency, however, can cause the mathematics behind the computations to break down.

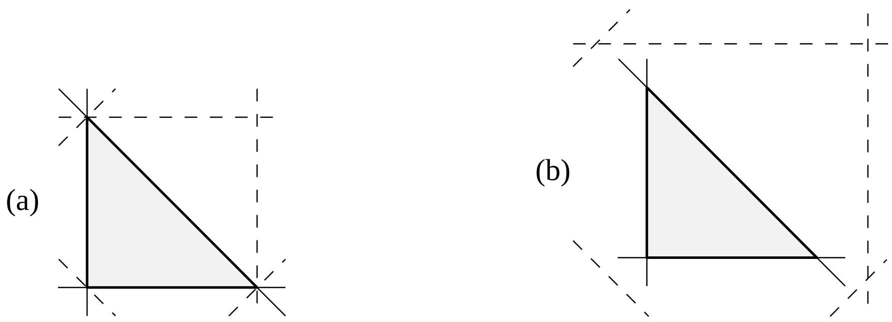

Figure 2.1: FDH_8 versus intersection of halfspaces

The basic assumption about the representation of FDH is expressed by the following characterization of FDHs.

Proposition 1. *Collection of halfspaces forms FDH if and only if the intersection of these halfspaces has common point with each of supporting hyperplanes.*

Proof. Assume that the collection of halfspaces forms FDH. For the sake of contradiction assume that there is a supporting hyperplane that does not intersect the FDH. Since FDH forms compact and convex set there is a hyperplane that separates the FDH and the supporting hyperplane and does not intersect any of them. The separating hyperplane is clearly parallel to the supporting hyperplane. The halfspace corresponding to the separating hyperplane is obviously strictly smaller than the one corresponding to the supporting hyperplane. Moreover it contains the FDH and thus the defining set X. This is a contradiction with the choice of the bounding halfspace as the smallest halfspace with given normal containing X.

On the other hand if the intersection of halfspaces has common point with each of supporting hyperplanes we can consider the set X of such points for all supporting hyperplanes. By definition the set X is subset of intersection of halfspaces, then it is also subset of each of halfspaces. Moreover for any supporting hyperplane there is point from X on it. This means that all halfspaces are minimal from those that contain X. Thus we have constructed the set X whose D-induced FDH is the collection of halfspaces from the claim of proposition. ☐

For reader versed in linear programming this means that the collection of halfspaces forms FDH iff every supporting hyperplane is in supporting position. This justifies the choice of its name. The example above shows that changing inequalities in (2.1) would invalidate the condition from the proposition. Figure 2.1b depicts what could happen.

2.1 Vector Representation

If we fix the order of n vectors in the set D we can put them in a matrix A with n rows and k columns[2]. The matrix is then fixed for the whole class of D-induced FDHs. An FDH is a collection of half-spaces, any of these half-spaces can be expressed by the inequality

$$d_i^\top x \leq b_{d_i}(X)$$

Since we have fixed the order of vectors in D we can write all these inequalities in a more symbolic form

$$Ax \leq b_D(X)$$

[2] I remind that we work in the space $\mathbb{R}^k$.

using common convention that $x \leq y$ means "$x_i \leq y_i$ for every i". Because we have fixed the order of the set of vectors we can use its matrix representation instead. From now on D will mean both the set of directions and the matrix of n rows and k columns. It should be clear from the context which of these meanings is used in any particular instance. The matrix D facilitates the translation from geometry to algebra and back. The *vector representation* of collection of halfspaces $(H_d)_{d \in D}$ is then b_D from above inequality.

The previous description of vector representation defines an isomorphism between collections of halfspaces and vectors in $\mathbb{R}^n$. However we have already seen that not every collection of halfspaces corresponds to an FDH. The set of vector representations of FDHs thus forms a proper subset of $\mathbb{R}^n$.

Although we develop the theory in the space $\mathbb{R}^k$, the case $k = 3$ is of course the most important. To show also example of FDHs in three dimensions we define:

$$D_{14} = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 & 0 & \nu & -\nu & -\nu & \nu & \nu & -\nu & \nu & -\nu \\ 0 & 0 & 1 & -1 & 0 & 0 & \nu & -\nu & \nu & -\nu & -\nu & \nu & \nu & -\nu \\ 0 & 0 & 0 & 0 & 1 & -1 & \nu & -\nu & \nu & -\nu & \nu & -\nu & -\nu & \nu \end{pmatrix}^\top$$

for $\nu = 1/\sqrt{3}$. This matrix defines class of FDHs denoted FDH_{14} in this thesis. The way this matrix was created is very simple. We take axis-aligned bounding box in 3D and cut off its corners. Figure 2.2 shows what kind of objects may arise.

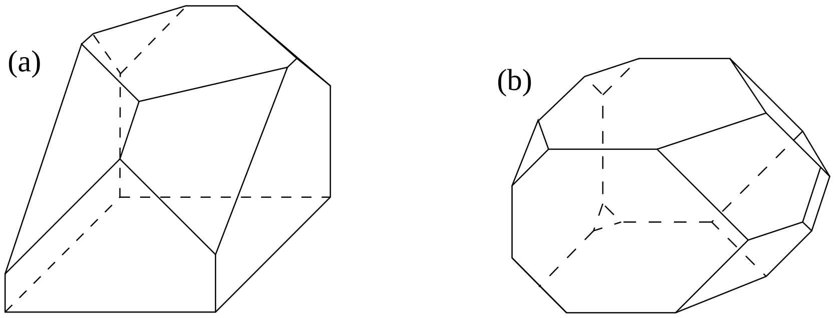

Figure 2.2: Examples of FDH_{14}

2.2 Properties of Vector Representations

In the this section we will explore the structure and properties of the set of vector representations of D-induced FDHs for arbitrary but fixed D. The first step we will do is expressing the basic property of FDHs from the Proposition 1 in the terms of vector representations.

Proposition 2. *Let $j_0, j_1, \dots, j_l$ be $l+1$ indices such that*

$$d_{j_0} = \sum_{i=1}^{l} \alpha_i d_{j_i}$$

for some nonnegative numbers $\alpha_1, \dots, \alpha_l$. Then for vector representation b of an FDH the following condition holds:

$$b_{j_0} \leq \sum_{i=1}^{l} \alpha_i b_{j_i}$$

Proof. Consider supporting hyperplane $\{x | d_{j_0}^\top x = b_{j_0}\}$ of the FDH. From Proposition 1 we know that it intersects the FDH in some point x. Since x is in the FDH it must satisfy inequalities $d_i^\top x \leq b_i$ for any $i = 1, \dots, n$. We consider only inequalities for $d_{j_1}, \dots, d_{j_l}$. Since the values α_i are nonnegative we can multiply the inequalities by these numbers and they will still hold. Thus we get

$$\alpha_i d_{j_i}^\top x \leq \alpha_i b_{j_i} \text{ for } i = 1, \dots, l$$

By summing all these inequalities and moving x out of the sum we obtain

$$\sum_{i=1}^{l} (\alpha_i d_{j_i})^\top x = (\sum_{i=1}^{l} \alpha_i d_{j_i})^\top x \leq \sum_{i=1}^{l} \alpha_i b_{j_i}$$

However by the assumption of the lemma we can replace the sum in the front of x by d_{j_0}. Moreover x is on the supporting hyperplane, thus $d_{j_0}^\top x = b_{j_0}$ and hence

$$b_{j_0} = d_{j_0}^\top x = (\sum_{i=1}^{l} \alpha_i d_{j_i})^\top x \leq \sum_{i=1}^{l} \alpha_i b_{j_i}$$

□

Given matrix D as above we can consider every $(l+1)$-tuple of indices from the proposition. For each of them we get a linear inequality that has to be satisfied by any vector representation of an D-induced FDH.

Graphical presentation of the meaning of the proposition is in Figure 2.3. The left part shows redundant situation that is prevented by the proposition, while the right shows situation that actually may happen.

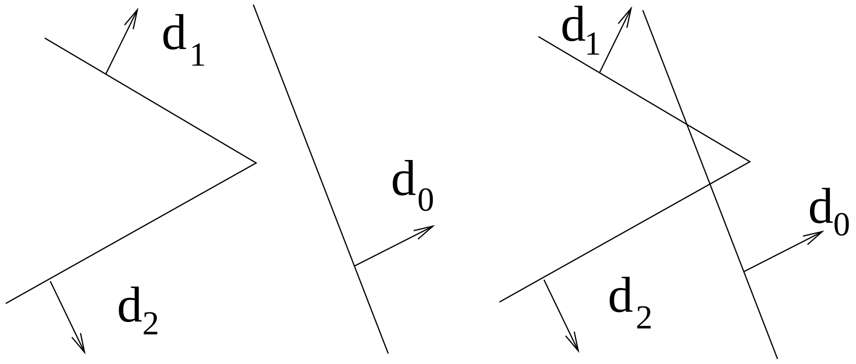

Figure 2.3: Redundant and non-redundant hyperplanes

The next observation could be the fact that independently on the choice of D we can define some meaningful operations on FDHs. These operations work directly on the vectors in $\mathbb{R}^n$ and have interesting geometrical interpretations. The first of these operations is *join*.

$$b^1 \vee b^2 = (max\{b_i^1, b_i^2\})$$

This operation is closely related to the union of FDHs.

Proposition 3. *The join $b^1 \vee b^2$ of FDHs b^1 and b^2 is an FDH of the union of their defining sets.*

Proof. Assume $b^1 = b_D(X)$, $b^2 = b_D(Y)$ and consider FDH $b = b_D(X \cup Y)$. We shall prove that $b_d = \max\{b_d^1, b_d^2\}$ for every $d \in D$. By the definition of FDH we get $b_d = \sup\{d^\top x | x \in X \cup Y\}$ and from properties of $\sup$ we obtain $b_d = \max\{\sup\{d^\top x|x \in X\}, \sup\{d^\top x|x \in Y\}\} = \max(b_d^1, b_d^2)$, i.e. the claim of the lemma. $\square$

There are two operations on vector representations that are of less practical importance, however they bring insight into the structure of the set of vector representations and thus are presented here. The first of them is scalar multiplication by nonnegative value, the second is vector addition. To show that they are well defined we have to analyze their behavior on defining sets of FDHs. For any $c \in \mathbb{R}$ and sets $X, Y \subseteq \mathbb{R}^k$ we define $cX = \{cx|x \in X\}$ and $X + Y = \{x + y|x \in X, y \in Y\}$.

Lemma. *For every nonempty and bounded set $X \subseteq \mathbb{R}^k$ and for every $c \geq 0$ it holds: $b_D(cX) = cb_D(X)$.*

Proof. From the definition we get that for any $d \in D$ it holds $b_d(cX) = \sup\{d^\top(cx)|x \in X\}$. Then clearly $b_d(cX) = c\sup\{d^\top x|x \in X\} = c\ b_d(X)$. This is the claim of the lemma. □

Lemma. *For every pair of nonempty and bounded sets X and Y it holds:* $b_D(X+Y) = b_D(X) + b_D(Y)$

Proof. For any $d \in D$ we get by definition: $b_d(X+Y) = \sup\{d^\top(x+y)|x \in X, y \in Y\}$. Since the choices of x and y are independent we immediately get $b_d(X+Y) = \sup\{d^\top x|x \in X\} + \sup\{d^\top y|y \in Y\} = b_d(X) + b_d(Y)$ and the lemma follows. □

The last two lemmas have two meanings. The first meaning is the fact that the set of vector representations of FDHs is closed under addition and multiplication by nonnegative constant. The second meaning is that these operations preserve minimality of FDHs for original sets transformed accordingly.

Using these properties we can also show that FDHs are closed under scaling and translation. The scaling comes immediately from the first lemma. For translation it suffices to realize that if we express it as a vector $t \in \mathbb{R}^k$, then we can create its FDH. Its vector representation is actually equal to Dt and by previous lemma we get that sum of Dt and vector representation of any FDH is also FDH. Moreover this sum represents translation of FDH along vector t.

From correctness of the last two operations we also get following information about structure of the set of vector representations:

Lemma. *Vector representations of FDHs form convex cone in* $\mathbb{R}^n$.

2.3 Transformation of FDHs

In Section 3.3 we shall see that to implement proximity computation using BV-trees we need to compute distances between pairs of possibly transformed bounding volumes. This presents no problem if the class of bounding volumes used in the hierarchy is closed under transformations applied to objects. For rigid motions such classes are for instance oriented bounding boxes or spheres. On the other hand FDHs are closed only under translation and scale (see Section 2.1). For the purposes of proximity computation we will analyze only rigid motions here. These transformations are the most important from the practical point of view as they do not deform the object.

Another important issue we will take into account is the fact that the BV-tree algorithms have to consider many objects undergoing the same transformation. It would clearly help to precompute as much information about the transformation as possible, since the time needed for this precomputation does not depend on the (usually large) number of objects in the hierarchy.

Any position during rigid motion can be described by an unitary matrix and a translation vector. Thus we can assume that we are given two unitary matrices R, S of the order k and two vectors $r, s \in \mathbb{R}^k$. These matrices and vectors represent linear isomorphisms $f : x \mapsto Rx + r$ and $g : x \mapsto Sx + s$ on $\mathbb{R}^k$. These isomorphism and their inverses are clearly length and angle preserving. The input of the distance function are vector representations b^1, b^2 of FDHs undergoing transformations f and g respectively. Let $H_1 = \{x|Dx \leq b^1\}$ and $H_2 = \{x|Dx \leq b^2\}$ be corresponding intersections of half-spaces. The first thing we can use is removing one of the transformations. This is done by transforming one of FDHs into coordinate system of the other:

$$\delta(f(H_1), g(H_2)) = \delta(g^{-1}(f(H_1)), g^{-1}(g(H_2))) = \delta(g^{-1}(f(H_1)), H_2)$$

We can simplify the transformations and replace original matrices and vectors by single matrix $U = S^\top R$ and the translation vector $t = S^\top(r - s)$, getting new length and angle preserving isomorphism $f : x \mapsto Ux + t$. Hence it is sufficient to solve the problem for one transformed and one unchanged FDH.

There is a number of possible solutions of this problem. I mention just the most important of them. The solution that provides best approximation of bounded object is certainly recomputing the bounding volume from the original set transformed by f. Although this might seem as too expensive approach it has been successfully used in [He-96], though only in the root of the hierarchy. It is important to note that we need not consider whole input set when we are computing its FDH, it is sufficient to replace it by its convex hull and the algorithm presented there really uses boundary representation of the convex hull and a simple hill-climbing procedure to update its FDH after transformation. This approach is quiet efficient in dynamic environment if the transformations change only in small steps. In this case the number of operations required by the hill-climbing is proportional to $|D|$. The worst case complexity, however, depends on the number of vertices of the convex hull.

The second best solution with respect to the approximation tightness is computing distance of transformed FDHs. I will show how to simplify this process, however the final solution is based on quadratic programming methods and is quiet inefficient for our purposes. To compute the distance of FDHs we start by its definition:

$$\delta(f(H_1), H_2) \quad = \quad \inf\{\|x - y\| | x \in f(H_1), y \in H_2\}$$

If we substitute definitions of H_1, H_2 and f, we get (after simplification) the following result:

$$\begin{aligned}\delta(f(H_1), H_2) &= \inf\{\|Ux + t - y\| | Dx \leq b^1, Dy \leq b^2\} \\ &= \inf\{\|Ux - y'\| | Dx \leq b^1, D(t + y') \leq b^2\} \\ &= \inf\{\|Ux - y\| | Dx \leq b^1, Dy \leq b^2 - Dt\}\end{aligned}$$

And finally after we use the definition of distance we get:

$$\delta(f(H_1), H_2) = \sqrt{\inf\{\sum_{i=1}^{k}((Ux)_i - y_i)^2 | Dx \leq b^1, Dy \leq b^2 - Dt\}}$$

The expression under square root represents optimization problem with quadratic cost function and linear constraints, it thus can be solved by applying methods of quadratic programming.

The last but not least important approach presented here is replacing transformed FDH by its FDH. This method has been used in [He-96, Za-98], though the implementation is quiet different from what is presented here. The same method as presented in [Za-98] actually works only in 2D and for some very special cases also in higher dimensions, though the author does not explicitly mention that. The implementation in [He-96] is a general solution, however it requires memory in the order of magnitude larger than our approach. A special case of solution presented here can be found in our previous work [Ko-97].

As above assume that the transformation $f : x \mapsto Ux+t$ is defined by the unitary matrix U and the vector t. Moreover let b be the vector representation of FDH on the input. The goal is to find vector representation b' of FDH of the set $\{Ux + t | Dx \leq b\}$. By the definition we have to compute

$$b'_d = \sup\{d^\top(Ux + t) | Dx \leq b\}$$

for every $d \in D$. Since FDH is a compact subset of $\mathbb{R}^k$ we know that we can replace $\sup$ by $\max$. Simplifying the equation we get

$$b'_d = d^\top t + \max\{(d^\top U)x | Dx \leq b\}$$

The values $d^\top t$ as well as the vectors $d^\top U$ are fixed for given U and t and can be precomputed for the transformation and all vectors $d \in D$. We shall denote them t_d and U_d respectively. So far we have reduced the transformation of FDH to computing following numbers for every $d \in D$.

$$b'_d = t_d + \max\{U_d x | Dx \leq b\} \tag{2.2}$$

A reader acquainted with linear programming has probably noticed that this is a problem from this theory. Naturally general methods for solving these problems (for instance simplex algorithm) could be used to solve this problem. However in the case of FDHs there exists more efficient solution. Before presenting it I will devote couple of pages to solution of problems based on unchanged FDHs.

2.4 Collision of FDHs

Now we should examine how to compute distance of two FDHs and how to detect their intersection. The problem of collision detection of these objects has been addressed in [Za-98, He-96]. The approach used there is an extension of collision test for AABBs. Every box is defined as an intersection of three pairwise orthogonal slabs. To check whether two such objects intersect it is sufficient to test whether every pair of parallel slabs intersect. To apply this method on FDH we need to constrain the set of directions used in the definition. From now we will require that the set D is closed under additive inverse, i.e. whenever $d \in D$ then also $-d \in D$. This constraint is not too limiting and moreover all of the sets D used in this thesis satisfy this property.

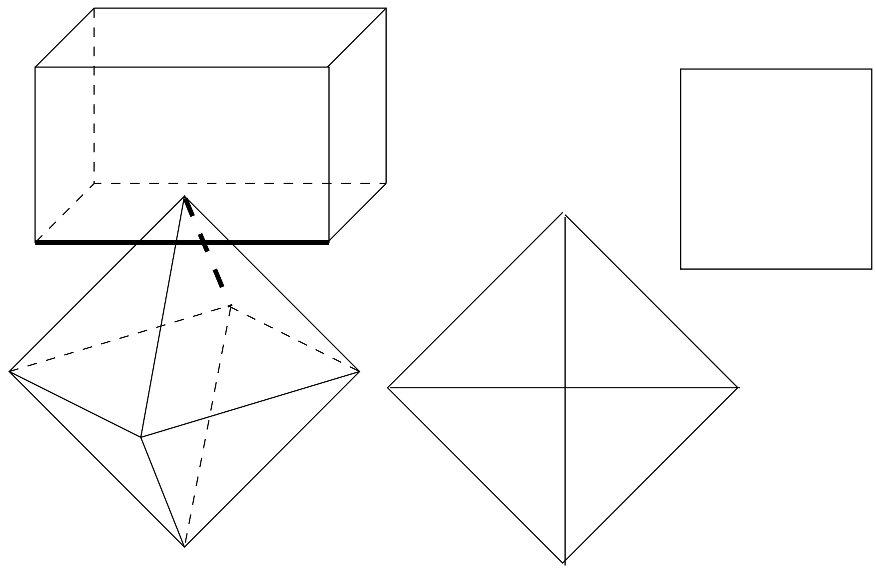

Figure 2.4: Not real collision

Although the approach based on intersections of slabs works well for AABBs, it cannot be extended to FDHs in general. In case of AABBs it holds that two boxes intersect if and only if every pair of parallel slabs intersects. This is not true in general, it may happen that every pair of parallel slabs defining two FDHs intersects while FDHs do not intersect.

An example of such situation is displayed in Figure 2.4. The left part shows instances of FDH_{14} that are in a position where the test reports collision while the objects actually do not intersect. For clarity the right part of the figure shows whole situation from side view.

Nonetheless the converse is true, i.e. whenever two FDHs intersect then every pair of parallel slabs intersects. I use this approach in this work too, especially because exact collision detection of FDHs can be relatively expensive and the gain obtained by the exact test is not significant. An exact method, however, could be based on the following lemma.

Lemma. *Two convex polyhedra do not intersect if and only if there exists separating hyperplane that is parallel to two edges of polyhedra.*

This lemma can of course resolve also the problem in Figure 2.4. Bold lines represent edges that can be considered to find a separating plane of the FDHs

The algorithm can check for each pair of edges if there is a separating hyperplane parallel to these edges. Since the number of such pairs can be quadratic in $|D|$, this could clearly be costly in the case of large sets D. However, it is quiet efficient when the set D is very small. It has been successfully used in [Go-96] for collision test of oriented bounding boxes. Only nine separating planes have to be considered in this case. The lemma can be also used to prove correctness of collision test for AABBs.

2.5 Distance Bounds

Because we want to cope not only with the problem of collision detection, but also with the problem of proximity computation, we must be able to compute the distance of FDHs or its lower bound. Our method is based on special norm in space that bounds euclidean norm from below and can be efficiently approximated for polyhedral sets.

Before explaining new concepts I will briefly mention methods that can be used to compute real distance. Since FDH is an intersection of limited number of halfspaces, we know that it forms a polyhedron with limited number of faces. Thus methods of computational geometry that work for general convex polyhedra can be used even for FDHs. Their performance, however, is questionable since the number of faces, edges and vertices of FDH is usually too small to show the advantage of asymptotically fast methods.

Another solution of distance computation can be found in the theory of mathematical programming. The problem can be transformed to the problem of maximization quadratic function with $2 * k$ variables bounded by $2 * |D|$ linear inequalities, the theory of quadratic programming then gives us an algorithm to find the solution. However also these methods work especially well only for large problems. This is just an instance of the method from Section 2.3 for the case $f = \mathrm{id}$.

Let D be a set of directions as above, define *D-induced norm* of $x \in \mathbb{R}^k$ by

$$\|x\|_D = \max_{d \in D} d^\top x$$

Of course we have to make sure that this functional is really a norm in $\mathbb{R}^k$, actually there are Ds for which it is not a norm. For instance whenever $|D| < k$ there exists nonzero vector $x \in \mathbb{R}^k$ such that $d^\top x = 0$ for every $d \in D$, thus $\|x\|_D = 0$, this is not possible for any norm. Before asserting condition on D we will explore the relation between D-induced norm and euclidean norm.

Lemma. *If $\|.\|_D$ is a norm then there is $c > 0$ such that for any $x \in \mathbb{R}^k$ the inequalities $\|x\| \geq \|x\|_D \geq c\|x\|$ hold.*

Proof. For the first inequality it is sufficient to realize that for $D = \{x \in \mathbb{R}^k | \|x\| = 1\}$ the D-induced norm and euclidean norm coincide. The claim then comes from the monotonicity of $\max$.

For the second inequality consider set $M = \{\|x\|_D | \|x\| = 1\}$. M is an image of a compact set in $\mathbb{R}^k$ under continuous mapping, thus M is compact and $\inf M = \min M$. By assumption of the lemma we have $m > 0$ for any $m \in M$. From this we get that $c = \inf M \geq 0$, moreover $c = \min M$ and thus $c > 0$. Now let $x \in \mathbb{R}^k$ be an arbitrary vector, then $x = \|x\|e$ for some unit length vector e. Hence $\|x\|_D = \|x\|\|e\|_D \geq \|x\|c$. □

The previous lemma can be used to measure largest relative error of approximation of euclidean norm by D-induced norm for any chosen D. Moreover in its proof we have constructed a c from the claim of the lemma. Applying this choice on D_8 from the beginning of the chapter we get that $c = \cos \pi/8 \doteq 0.9238$. Having this lemma we can assert a condition that D has to satisfy to induce a norm in $\mathbb{R}^k$.

Lemma. $\|.\|_D$ *is a norm in* $\mathbb{R}^k$ *if and only if the set* $M = \{x \in \mathbb{R}^k | \forall d \in D : d^\top x \leq 1\}$ *is bounded in* $\mathbb{R}^k$.

Proof. Assume that $\|.\|_D$ is a norm and for the sake of contradiction assume that M is unbounded. Then for any $a \in \mathbb{R}^k$ there is $x \in M$ satisfying $\|x\| \geq a$. Let us choose $a = 2/c$ (c defined in the previous proof) and fix one such x. Then $\|x\| \geq 2/c$, however $x \in M$ and thus $d^\top x \leq 1$ for any $d \in D$. Hence $\|x\|_D \leq 1$. By the previous lemma we get $2 \leq c\|x\| \leq \|x\|_D \leq 1$ and we have a contradiction. Therefore M is bounded.

On the contrary suppose that M is bounded. Let $x, y \in \mathbb{R}^k$ be vectors and $c > 0$ a real number. We have to prove several conditions to make sure that $\|.\|_D$ is a norm.

1. $\|x\|_D \geq 0$: Assume that $\|x\|_D \leq 0$, then also $\|cx\|_D \leq 0$. Hence $cx \in M$ and therefore M is unbounded iff $x \neq 0$, a contradiction. When $x = 0$ we have immediately $\|x\|_D = 0$ and the claim is trivial.

2. $\|x\|_D = 0$ iff $x = 0$: From the previous part.

3. $\|cx\|_D = c\|x\|_D$: Since $d^\top(cx) = c(d^\top x)$ for any $d \in \mathbb{R}^k$ we have $\|cx\|_D = \max_{d\in D} d^\top(cx) = c \max_{d\in D} d^\top x = c\|x\|_D$.

4. $\|x+y\|_D \leq \|x\|_D + \|y\|_D$: Almost trivial $\|x+y\|_D = \max_{d\in D} d^\top(x + y) \leq \max_{d\in D} d^\top x + \max_{d\in D} d^\top y = \|x\|_D + \|y\|_D$.

□

The set M defined in the previous lemma is an intersection of finite number of closed halfspaces, thus it is bounded iff it forms a polyhedron.

The concept of D-induced norm can be naturally extended to sets in $\mathbb{R}^k$. As with euclidean distance we can define the distance of two sets as the distance of their nearest points, i.e. given the set of directions D and two nonempty sets $X, Y \subseteq \mathbb{R}^k$ we define D-induced distance of X and Y as

$$\delta_D(X, Y) = \inf_{x\in X, y\in Y} \|x - y\|_D$$

We have seen that D-induced distance is bounded from above and below by euclidean distance multiplied by factors depending solely on D, thus this functional is bounded in the same manner. This means that we can use D-induced distance to approximate euclidean distance. To compute D-induced distance of the sets X and Y we have to solve the following optimization problem

$$\min\{\max_{d\in D} d^\top(x - y) | x \in X, y \in Y\}$$

However this problem is still too complicated. We can change the order of $\min$ and $\max$ and bound the solution from below by

$$\max_{d\in D}\min\{d^\top(x-y)|x\in X, y\in Y\}$$

Since the set D is relatively small we can perform minimization for each d. Since the choice of x and y in the minimization are independent we can split it into two subproblems.

$$\min\{d^\top(x-y)|x\in X, y\in Y\} = -\max\{-d^\top x|x\in X\} - \max\{d^\top y|y\in Y\}$$

The title of this section suggests that we need to solve this problem when X and Y are D-induced FDHs. By the assumption of the previous section we get that for any ordering of vectors in D and each index i there is i' such that $d_i = -d_{i'}$. From the definition of FDH and the assumption of previous section we get that vectors d and $-d$ from maximization problems above are normals of some bounding halfspaces of X and Y. Maximal values then can be taken directly from vector representations of FDHs, i.e. when $d = d_i$ for some i then $\max\{d^\top x|x\in X\} = b_i$ for b being vector representation of FDH X. The whole optimization process can then be implemented by following algorithm with the input consisting of two FDHs b^1 and b^2:

Algorithm 1 Lower bound of distance of FDHs

```
m := −∞
for i = 1 to |D| do
   m := max(m, −(b¹_i + b²_i'))
end for
return m
```

We described this algorithm also in [Ko-97], though in little bit different way.

2.6 Linear Optimization over FDH

In this section we will examine the problem that resulted from analysis of the rigid transformations of FDHs in Section 2.3. Our goal is to develop efficient preprocessing and query algorithms to solve the optimization problem:

$$\max\{c^\top x|x\in\mathbb{R}^k, Dx\leq b\} \tag{2.3}$$

for fixed matrix D and vector c, where b is vector representation of D-induced FDH. I remind that D is matrix with n rows and k

columns, $c \in \mathbb{R}^k$ and $b \in \mathbb{R}^n$. The first important step in the process is using *duality* in linear programming (LP). The problem (2.3) is equivalent to the minimization problem:

$$\min\{b^\top y | y \in \mathbb{R}^n, y \geq 0, D^\top y = c\} \tag{2.4}$$

By the definition of FDH we know that the set $P = \{x | Dx \leq b\}$ of primal feasible solutions is non-empty and bounded. Thus the set $Q = \{y | y \in \mathbb{R}^n, y \geq 0, D^\top y = c\}$ of dual feasible solutions is also non-empty and bounded. Moreover both these sets are expressed as intersection of hyperplanes and halfspaces in corresponding dimensions and hence form convex polyhedra. From the theory of LP we get that in such case the optimum is achieved in a vertex of the polyhedron. Each vertex of a polyhedron in $\mathbb{R}^l$ can be given by intersection of l hyperplanes that contain hyperfaces of the polyhedron.

2.6.1 Simple Solutions

This leads to brute-force method for computing optimum. We consider every k-tuple of supporting hyperplanes of FDH, check if they intersect in a point and compute the value in this point. Since there are only finitely many vertices this will certainly end in an optimal vertex. However the number of vertices of FDH can be $\Omega(\binom{n}{k})$ in the worst case and the algorithm would perform really badly if it would use this method. One possibility is to precompute vertices of FDH and store them together with (or instead of) the vector representation. Then we can search only these points for maximum. This approach was used in [He-96].

To speed up this solution we can create boundary representation of the convex polyhedron and use hill-climbing algorithm to find the optimum. This roughly corresponds to simplex method for solving LP problems. To solve the problem it is sufficient to store only vertices and edges of polyhedron to use this method. Thus this speed up can dramatically increase memory requirements.

The major problem of last two methods is the amount of memory they consume. As stated above the number of vertices can be very large, and the number of edges can be even larger.

While the previous approach uses the set of primal feasible solutions, it is possible to use the dual vertices. The advantage is apparent, since the set Q does not depend on particular FDH we can precompute its structure for given D and c and then given vector representation b of FDH we can compute minimum using this structure. Though this solution can be easily implemented there are some observations that have to be made before an attempt to do so.

First of all we should analyze the structure of dual vertices. At this point we have to remind some notions from LP. The *base solution* of LP is a solution whose non-zero coordinates correspond to linearly independent columns of the matrix D. The *base* is a set of indices that have non-zero value in a base solution. The vertices of Q are exactly the base solutions of (2.4). As such they have at most k non-zero coordinates. To compute the value of the cost function $b^\top y$ in the vertex y it obviously suffices to use only the positive coordinates[3] of y. Since n is always larger than k it is useful to store only the values of positive components of vertices and their indices.

2.6.2 Redundancy

So far we did not employ the fact that the vector representations of FDHs form a proper subset of $\mathbb{R}^n$. By using this fact we can remove significant number of vertices from consideration. To do it we start by noting that every vertex of P can be given by intersection of k supporting hyperplanes of FDH with linearly independent normals.

Let x be an optimal vertex of the problem (2.3), let $I = \{j_1, \dots, j_k\}$ be the set of indices that identify supporting hyperplanes defining x. Thus the following equation holds for any $i = 1, \dots, k$:

$$d_{j_i}^\top x = b_{j_i} \tag{2.5}$$

Assume that there is an index j_0 and $\alpha_1, \dots, \alpha_k \geq 0$ such that $d_{j_0} = \sum_{i=1}^k \alpha_i d_{j_i}$. Let us examine the value of $d_{j_0}^\top x$. Trivially $d_{j_0}^\top x = \sum_{i=1}^k \alpha_i d_{j_i}^\top x$. However from equations above we get $d_{j_0}^\top x = \sum_{i=1}^k \alpha_i b_{j_i}$ and by Proposition 2 we get that $d_{j_0}^\top x \geq b_{j_0}$. Moreover $Dx \leq b$, hence also $d_{j_0}^\top x \leq b_{j_0}$. Thus $d_{j_0}^\top x = b_{j_0}$. If we replace any of equations in (2.5) by $d_{j_0}^\top x = b_{j_0}$ the vertex will not change. Hence we need not consider the solution corresponding to the base $\{j_1, \dots, j_k\}$, because if it is optimal we will get it from some other base. We will call such base $\{j_1, \dots, j_k\}$ *redundant base*. Moreover the primal bases correspond to the vertices of the dual problem, thus the vertices corresponding to redundant bases are not needed to solve the problem, we will call them *redundant vertices*.

It remains to show how to compute the vertices of the dual problem (2.4). Since we noted that every vertex has at most k nonzero values it is clearly sufficient to select those $n-k$ components that will be zero and try to complete the remaining coordinates. After selecting $n-k$ zero components we can extract square submatrix B of $D^\top$ that corresponds to k remaining elements. The equality $By' = c$

[3] Remember that $y \geq 0$.

must hold for some $y' \in \mathbb{R}^k, y' \geq 0$. The following algorithm finds all vertices of the dual problem for given D and c and rejects those that are redundant.

Algorithm 2 Dual vertices computation

for each $I = \{j_1, \ldots, j_k\} \subseteq \{1, \ldots, n\}, |I| = k$ **do**
  $B := (d_{j_1}, \ldots, d_{j_k})$
  if B is invertible **then**
    $M := B^{-1}$
    $r := 0$
    for $i \in \{1, \ldots, n\} \setminus I$ **do**
      if $Md_i \geq 0$ **then**
        $r := 1$
        break
      end if
    end for
    if $r \neq 1$ **then**
      $y := Mc$
      if $y \geq 0$ **then**
        store vertex $j_1, \ldots, j_k; y$
      end if
    end if
  end if
end for

Thus to solve (2.3) for a large number of different vectors b we precompute non-redundant vertices of the dual problem and then use brute-force search for minimum. Experiments show that the number of such vertices grows slowly for properly chosen sets D. The following table shows how this works for some choices of FDHs found in literature (see [He-96]). In the table FDH_6 corresponds to

	FDH_6	FDH_8	FDH_{12}	FDH_{14}	FDH_{18}	FDH_{26}
total	8	24	120	48	96	336
bases	8	24	80	48	80	192
min	1	2	4	2	2	3
max	1	2	4	2	4	5

Table 2.1: Number of vertices for selected FDHs

axis-aligned bounding box, FDH_8 and FDH_{12} use normals of faces of regular octahedron and dodecahedron respectively, FDH_{14} was defined above, FDH_{18} corresponds to cube with edges cut off, and finally FDH_{26} is intersection of FDH_8 and FDH_{18}. The row labeled *total* contains total number of k-tuples considered, *bases* is number of those that are invertible, *min* and *max* are minimum and maximum number of vertices corresponding to any c.

The numbers in the column of FDH_{14} agree with our previous result in [Ko-97] where only this special case was analyzed.

A major improvement of this method would be an algorithm to find non-redundant bases in some efficient manner. The brute force search we use seems to be too slow. The above article shows that this is possible for FDH_{14}. It seems that some general concepts can be extracted from it and applied also on other cases. If we consider only single query our algorithm is in the order of magnitude better than the approach used in [He-96] both in speed and memory consumption. However brute-force used in the preprocessing time makes it useful only for large number of FDHs.

2.7 Other Operations

There are also other important geometric operations that can be used in applications. For raytracing such operation is intersection of FDH and line. Our approach is essentially the same as the one used in [Ka-86] for same purposes. We use Cyrus-Back algorithm to find the intersection of line and FDH.

For completeness we provide it here – Algorithm 3. The input of the algorithm is a vector representation b of FDH and a line given by point p and vector v. The result is the interval $[f, l]$ of values t for which $p + t * v \in$ FDH. The algorithm is based on the fact that FDH is an intersection of halfspaces. It constructs intersection of the line with each of these halfspaces and computes intersection of resulting line segments. Should the algorithm return values $f > l$ it means that corresponding intersection is empty.

Graphical presentation of 2D version of the algorithm is shown in the Figure 2.5. Bold line is actual intersection, small dots represent boundaries of line segments considered by the algorithm.

Algorithm 3 Intersection of line and FDH - LineFDH

$f := -\infty$
$l := \infty$
for $i := 1$ to $|D|$ **do**
  $s := d_i^\top v$
  if $s = 0$ **then**
    if $d_i^\top p > b_i$ **then**
      $f := \infty$
      $l := -\infty$
      return
    end if
  else
    $t := (b_i - d_i^\top p)/s$
    if $s > 0$ **then**
      $l := \min(l, t)$
    else
      $f := \max(f, t)$
    end if
  end if
end for

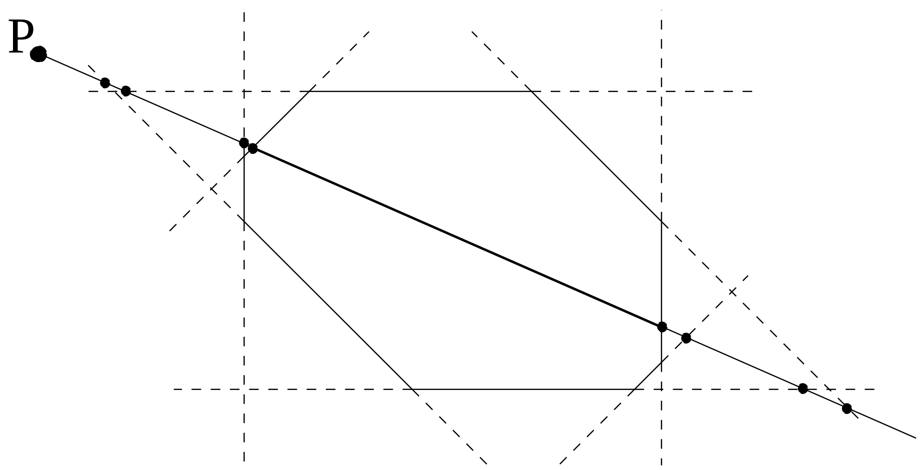

Figure 2.5: Intersection of line and FDH

Chapter 3

Bounding Volume Hierarchies

In this chapter I shall develop the notion of hierarchical approximations of objects consisting of a large number of primitives. I choose to do it without choice of particular bounding volume, I just assume that we are given a class of bounding volumes and that for any number of primitive objects we are able to find their common bounding volume in this class. Later we shall see some other properties of bounding volumes that can help us to speed up algorithms.

Using bounding volume hierarchies has two aspects. The first of them are algorithms to build them and the second are methods that use hierarchies to solve various problems. The aim is to build a hierarchy that approximates object and keep it in persistent storage together with original object for future use. Clearly the efficiency of creating the data structure is of less importance than the cost of one particular solution. However for practical use we have to keep the time of construction and the size of resulting auxiliary data under some reasonable bound depending on the size and the complexity of original objects.

Majority of approaches to hierarchical object approximation uses trees of bounding volumes. One suitable definition of this concept can be found in [He-96]. Assume we are given set S of geometric primitive objects. A *BV-tree* of set S is a tree, BVT(S). Each node ν of BVT(S) corresponds to a set $S_\nu \subseteq S$. The root of the tree is associated to whole set S. Each non-leaf node has two or more children; as usual the maximum number of children for every node is called *degree* of BVT(S) and is denoted by δ. The subsets of S that correspond to the children of node ν form a partition of the set S_ν of objects associated with ν. We say that BVT(S) is *complete* iff the set assigned to each leaf is a singleton subset of S. From the definition we get that BVT(S) contains at most $2|S|-1$ nodes and if it is complete it contains at least $|S|$ nodes. The height of a complete tree is thus at least $\lceil \log_\delta |S| \rceil$. If this depth is achieved the tree is called *balanced*. Also associated with each node ν of BVT(S) is a *bounding volume*, $b(S_\nu)$, that is an (outer) approximation to the set S_ν. An example of complete balanced binary BV-tree with AABBs chosen as bounding volumes is depicted in Figure 3.1. Primitive objects are triangles and bold lines represent links in the tree. It is important to realize that primitive objects are actually stored only in leafs. The sets assigned to internal nodes are defined only to make the explanation easier.

Contrary to the original definition we do not require the bounding volume of a set to be minimal. In practice, however, this will be common case. At this place I would like to note that the requirement on the sets assigned to children of the node can be relaxed. For correctness of algorithms it is sufficient that union of the sets associated with children of node ν is a subset of the set assigned to node ν. According to this definition, though, it would not be possible to limit the size of the hierarchy, since any primitive object could be assigned to multiple nodes. This kind of definition was used for instance in [Za-95].

3.1 Building BV-trees

Most methods for building BV-trees use either bottom-up or top-down approach to construct them. Bottom-up approach starts with the input primitives as the leafs of the tree and tries to group some of them together to form a node on level higher in the hierarchy, until it reaches single node – the root of the tree. Efficient algorithm based on this approach has to use local information about the objects. An example of this method can be found in [Ba-96].

If we use FDHs we can avoid recomputation of the bounding volumes over increasing sets of primitive objects. After the algorithm decides to collect some of the objects into one node of the tree it need not recompute bounding volume of the node from the scratch. By Proposition 3 it is sufficient to apply join operation on the FDHs of collected nodes. This method can be used even if the chosen BVs were not FDHs. For instance we can easily compute bounding sphere of two given spheres. However in most cases the result would not be the smallest bounding sphere of the union of the original sets. We would miss this minimality even in the case of oriented bounding boxes. However from Proposition 3 we know that in the case of FDHs the result is minimal bounding volume of primitives.

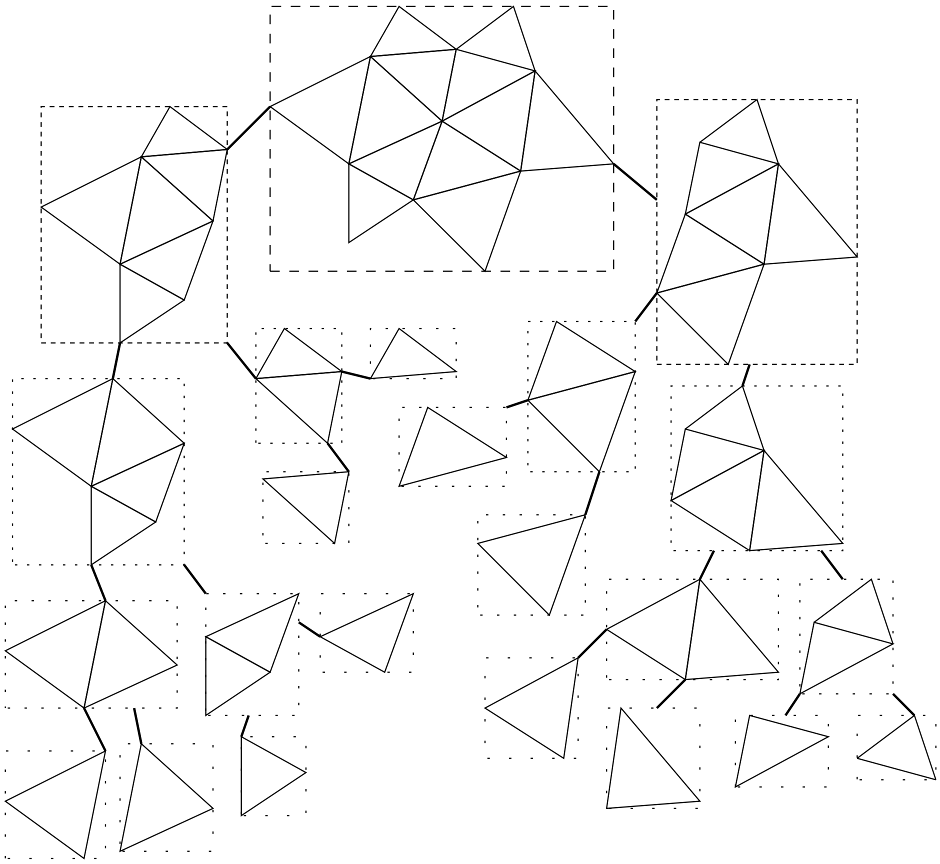

Figure 3.1: Complete balanced binary BV-tree

Top-down methods start with whole set S. This set is partitioned into disjoint subsets. For each of these subsets the bounding volume is created and the algorithm proceeds recursively towards leafs. These methods are far more popular in the literature then their bottom-up counterparts (see [Go-96, He-96, Za-98, Za-95]). This can be explained by the simplicity of implementation of these algorithms. The relation between these two approaches is not very well explored and it is not clear which of these methods provide better approximations. The aim of this work is not to resolve this problem and I resort to methods with easier implementation. One advantage of this attitude is robustness of resulting algorithms as they are easier to analyze.

The core of a top-down algorithm is procedure to partition input set into multiple subsets. The first decision we have to make is choosing degree of the tree. Here we can use analysis from [He-96].

The authors argue that the best choice is binary tree. The first reason they provide is that the cost of the search in δ-ary balanced tree along a single path from root to leaf is proportional to $f(\delta) = \delta \log_\delta n$ and that this function is minimized over integers for $\delta = 2$. This analysis is actually slightly incorrect as the minimum is achieved for $\delta = 3$, although the difference is only marginal (5%). Later we shall see that for our implementation the cost of search along one root-to-leaf path is actually proportional to $f(\delta) = \delta^2 \log_\delta n$ which is really minimized in $\delta = 2$. This argument is of course only of heuristic nature since the search would usually need more than single path to leaf. Another, more legitimate reason is that binary trees are easier to build as there are fewer options to split set into parts. Also some preliminary experiments show that this choice is the best for our purposes.

Having assigned bounding volume to each node of BV-tree we created problem unknown in the case of normal trees. There the balancedness is the ultimate property to measure quality of the tree. Although we have defined this property also for BV-trees we have to reconsider its meaning in the new context. The issue is that balanced tree may not be optimal for our purposes. We have to take into account the geometric properties of underlying sets. The worst case would happen if bounding volumes of all children of every node were the same. Then the search algorithm would need to consider every node in the tree, because the collision tests of bounding volumes would not reject any unpromising candidates.

The ideal solution would be a balanced tree such that intersection of bounding volumes of children of every node is empty. This goal, however, usually cannot be achieved (see Figure 3.1).

3.2 Splitting Rules

To achieve the best solution we can choose some measure of quality of the partitioning and try to optimize it. For our purposes it will be sufficient to use a function of number of primitives in each part and bounding volumes of these parts. There are number of options that were explored in the literature:

- minimize sum/maximum of volumes
- minimize sum/maximum of surfaces
- minimize volume/surface of intersection
- minimize volume/surface of union

- maximize separation
- minimize penetration
- combinations of previous methods

In the case of binary trees there are $2^{|S_\nu|-1} - 1$ ways to partition the input set into parts, thus we clearly can consider only very small portion of them. This problem is usually resolved by choosing *splitting hyperplane* and using geometric position of primitives with respect to the hyperplane to split the input set [He-96, Za-95, Go-96]. Choice of this method is justified by the fact that bounding volumes are usually convex, the aim is to find a hyperplane that would separate bounding volumes of parts after splitting.

One step of this approach after making a choice of the hyperplane is apparent. Objects that are completely “on the left” are put into one part and objects that are completely “on the right” are put into the other. The methods differ in handling objects that intersect the hyperplane. Here the optimization criteria described above are often employed [Za-95].

The hyperplane is usually chosen in two steps. In the first step the algorithm chooses axis orthogonal to the hyperplane. In the second step the hyperplane is positioned according to some optimization criteria.

3.2.1 Choice of Axis

The choice of axis often depends on particular bounding volume. The most simple approach is choosing one of coordinate axes as in [He-96]. Authors of [Go-96] use longest side of oriented bounding box enclosing the set of primitives, while in [Za-98] a line passing through two most distant points is employed. The choice of the splitting axis in our implementation is closely related to the class of bounding volumes we use. The algorithm in our implementation uses combination of statistical method similar to the one in [Go-96] and fixed number of axes used in [He-96]. High level description of our method is the Algorithm 4. It computes variance of projections of centroids of primitive objects on the axes given by all directions in D. Then the one that has maximal variance is chosen.

3.2.2 Positioning the Hyperplane

After picking split axis the algorithm has to find the exact position of the plane used for splitting. We have avoided explicit computation of such position by directly partitioning the set of triangles into

Algorithm 4 Choice of split axis

```
for i := 1 to |D| do
  s[i] := 0
  s_2[i] := 0
end for
for p ∈ primitives do
  for i := 1 to |D| do
    v := d_i^⊤ * centroid(p)
    s[i] := s[i] + v
    s_2[i] := s_2[i] + v^2
  end for
end for
m := −1
for i := 1 to |D| do
  v := s_2[i] − (s[i])^2
  if v > m then
    m := v
    axis := d_i
  end if
end for
```

two parts of the same size. This approach was used also in [Za-95]. The result will be balanced tree and in the case that input primitives are approximately of the same size, it usually has also good geometric properties. Rather than using deterministic algorithm for finding median we decided to use well known randomized algorithm, because it has better average time complexity and quite small probability of being too slow.

As mentioned above it is arguable whether the split at median is the best choice. The experiments in [He-96] show that split at mean is usually between 1% to 35% faster than split at median. This can be, however, caused by other factors than the choice of split plane. Especially the split axis was chosen by different method then our. The investigation of relations of these factors is not the aim of this work and is left to be accomplished by other experimenters. Of course, some other methods have been analyzed for positioning the hyperplane, for instance incremental algorithm was proposed in [Za-98].

3.3 Proximity Computation Using BV-trees

The reason for choosing BV-tree as a main data structure was to efficiently reject pairs of objects that cannot collide. In this chapter we shall analyze some algorithms that search hierarchies for colliding nodes. The optimization techniques used here often assume that FDHs are the class of bounding volumes used in the hierarchy and many of optimizations have no sense when applied to other objects.

Algorithms for collision detection using these hierarchies and their variants have been published in many papers [He-96, Za-95, Za-98, Za-94]. For future reference and analysis I provide the description of the one from [He-96] (see Algorithm 5). As is common with algorithms based on tree structures it is presented as a recursive procedure. The input to this function are two BV-trees and the output consists of calls to a user specified function for every pair of primitive objects that collide. In the case that only binary output is needed it should return from recursion in some language specific manner, e.g. by raising exception.

Algorithm 5 TraverseTree(ν_A,ν_B)

```
if b(ν_A) ∩ b(ν_B) ≠ ∅ then
  if ν_A is leaf then
    if ν_B is leaf then
      for p in primitives in ν_A do
        for q in primitives in ν_A do
          if p collides with q then
            report collision
          end if
        end for
      end for
    else
      for each ν_b child of ν_B do
        TraverseTree(ν_A,ν_b)
      end for
    end if
  else
    for each ν_a child of ν_A do
      TraverseTree(ν_a,ν_B)
    end for
  end if
end if
```

Though this algorithm is easy to implement and efficient it can be improved by a number of heuristics. The first observation we can make is that after we get to a leaf in one hierarchy we remain there in the corresponding branch of recursion. This property may not help in the case of static environments, however, we can speed up the algorithm if we use it in dynamic situation.

The improvement is hidden in the collision checks of the roots of input hierarchies. Since they are transformed together with objects we may precompute the transformed leaf and use only precomputed values for further tests. Because the algorithm traverses both trees sequentially this will usually improve performance especially in the case of collision. Moreover in Section 2.3 we have seen that with respect to our problems we can replace rigid motion of two objects by moving only one of them. Thus after transforming the leaf of the hierarchy we do not need to transform the other tree at all. The clean implementation of the idea is splitting the procedure into two parts. One part should handle the general case of inner nodes and the other should be used for a leaf of one hierarchy as one argument and whole hierarchy as the other.

3.3.1 Leaf versus Hierarchy

Our aim was to design not only the methods for collision detection but also algorithms for proximity computation. In the applications it is often desirable to specify a minimal separation of objects and test if it is satisfied for given objects. This opens new possibilities for optimization. For instance if the required result is only distance or binary (i.e. collision/no collision) information we can increase the speed of algorithm by changing the order in which we check the children for intersection. A simple heuristic is to check them in the order of increasing distance.

The next algorithm is an implementation of proximity computation between single object (i.e. leaf of the hierarchy) and an object described by BV-tree. Input to this algorithm is a leaf ν of one hierarchy altered by appropriate transformation, node ν_A of the other hierarchy and a number m that represents minimal separation we want to assure.

The algorithm uses auxiliary procedure MinIndex to extract the index of minimal value from an array. Since we choose to use binary trees the array has actually only two elements and the linear search is efficient enough.

This procedure can be used for any choice of bounding volumes. It uses only two "distance" functions δ and δ_l. The former is a function that we want to minimize over pairs of primitive objects, while the latter is a "distance" of bounding volumes. I used quotations

Algorithm 6 Leaf to node proximity computation - LeafProx

> **if** ν_A is leaf **then**
> $p := \min \delta(p, q)$ for all primitives p and q in ν and ν_A
> $p := \min(p, m)$
> **else**
>   **for** $i := 1$ to $\nu_A.NumberOfChildren$ **do**
> $d[i] = \delta_l(b(\nu), b(\nu_A.Child[i]))$
>   **end for**
> $p := m$
>   **loop**
> $j := \mathrm{MinIndex}(d)$
>   **if** $d[j] \geq p$ **then**
>   **return** p
>   **end if**
> $d[j] := \infty$
> $p := \mathrm{LeafProx}(\nu, \nu_A.Child[j], p)$
>   **end loop**
> **end if**
> **return** p

since these functions actually need not be distance measures in the strict sense. The correctness of the algorithm is based only on the condition:

$$\delta_l(\mathrm{BV}(A), \mathrm{BV}(B)) \leq \min_{a \in A, b \in B} \delta(a, b)$$

for any sets A and B of primitive objects. That is δ_l must be a lower bound for δ. Clearly the better this bound is the fewer nodes are visited during the search. In the case of FDHs, the distance defined in Chapter 2 obviously satisfies this condition, hence the algorithm correct.

Careful reader has probably noticed that the function does not compute distance of bounding volumes of input nodes. This seems suboptimal until we put the algorithm into proper context. Since it will never be called for such nodes, the test would be worthless.

The heuristic for children ordering is implemented by finding minimal value in an array of distances of bounding volumes and replacing it by ∞ after the search. The use of the value m is a noteworthy point, it is assigned to p as its initial value. This is justified by the fact that for the purpose of the search cases where the distance is greater or equal than m are considered to be equivalent, i.e. not colliding. An important point is that actual parameters m of the procedure may decrease during the search, this happens in the case that a pair of primitives with the distance smaller than original m is found. After this moment it is always equal to the distance of nearest primitives found so far. The value returned by the recursive call is also used to speed up the traversal, the procedure immediately exits if it discovers that the distance found so far cannot be decreased in the particular branch of the tree.

3.3.2 General Case

Having improved the simple case of one leaf and a node we can move to the more general case of two hierarchies. Although LeafProx can be used in Algorithm 5 there are still some improvements that can be made. First of all the order of recursive calls should be determined. For this purpose we use the same heuristic as in the special case.

Another questionable point of the original algorithm is the fact that before traversing the second tree it explores leafs of the first. Our choice is to traverse both trees simultaneously with the intention that it will proceed directly to the nearest leafs of hierarchies. To implement this approach efficiently we need to store transformed FDHs temporarily to prevent unnecessary recomputations.

An algorithm that uses simultaneous traversal of BV-trees similar to our algorithm can be found in [Za-94]. However it is presented without any optimization mentioned here and its description is not general with respect to the degree of BV-tree. We call our method *double recursion* as it goes one level down in both hierarchies at the same time. It is again described by recursive procedure, the input consists of: two nodes of BV-trees ν_A and ν_B, the transformation f that should be applied to the object A, and the number m with the same meaning as above.

The similar notes as for Algorithm 6 apply also to this algorithm. Namely it does not detect collision of bounding volumes associated with input nodes, since it is useful to do so only for roots of BV-trees. The function δ_l is again lower bound of the function δ that is to be minimized, the behavior of actual parameters m is also the same.

The notable difference is precomputing values of transformed nodes during the search done by subroutine Transform. If applied to a node of the BV-tree it creates a new copy of the node by transforming the BV accordingly and recording reference to primitive objects of original node and the transformation used. The transformation is not applied to primitive objects until we need it for primitive object checks. Then the primitives are transformed and stored into the copy of node, thus they need not be recomputed again should the

Algorithm 7 Double Recursion - NodeProx

```
if ν_A is leaf then
  ν := Transform(f, ν_A)
  return LeafProx(ν, ν_B, m)
else if ν_B is leaf then
  ν := Transform(f^{-1}, ν_B)
  return LeafProx(ν, ν_A, m)
else
  for i = 1 to ν_A.NumberOfChildren do
    ν[i] := Transform(f, ν_A.Child[i])
    for j = 1 to ν_B.NumberOfChildren do
      d[i][j] := δ_l(b(ν[i]), b(ν_B.Child[j]))
    end for
  end for
  p := m
  loop
    (i, j) := MinIndex(d)
    if d[i][j] ≥ p then
      return p
    end if
    d[i][j] := ∞
    p := NodeProx(ν[i], ν_B.Child[j], f, p)
  end loop
end if
return p
```

leaf collide with another leaf of the second tree.

There are also other optimizations that we can use. The most remarkable one is employing spatial coherency during motion of the objects. This assumes that objects are transformed in small steps. Then the pair of closest primitives may often remain the same. If we remember the pair across steps of the motion we can use it as a *hot start* in the next step.

The necessary modification of the algorithm involves the point where we compute the real distance of primitives in the procedure LeafProx and the way we handle the initial value of m before we enter the procedure NodeProx. The former change consists of adding a statement to store the pair of leafs that are closest during the search. The latter change can be done externally. We compute exact distance of the leafs remembered in the previous step and we send minimum of this distance and user specified value m as the argument m of procedure NodeProx.

3.4 Raytracing and BV-trees

Having simple test for intersection of FDH and line from Section 2.7 we can construct algorithm for collision detection of ray and BV-tree. This can be considered to be special case of LeafProx computation from the Section 3.3. The ray will be considered as a leaf of hierarchy and the functions δ and δ_l will be defined in order to achieve correct order of traversal of the tree.

The first issue we have to take into account is the fact that we have not defined the notion of bounding volume for rays. However since we have an algorithm to compute intersection of ray and FDH we may consider the ray to be its own bounding volume.

As in Section 2.7 we will use the ray in the form $P+t*v$, however now $t \geq 0$. The functions δ and δ_l can be defined in a similar manner. Let r denote the ray and $X \subset \mathbb{R}^k$ be an obstacle. In the case that the ray intersects the set X we define:

$$\delta_l(r, X) = \delta(r, X) = \inf\{t \geq 0 | P + t * v \in X\}$$

otherwise we put $\delta_l(r, X) = \delta(r, X) = +\infty$. Of course we will use the function δ_l only for FDHs, while δ has to be implemented in the application dependent manner. We can of course create the function δ_l by using the Algorithm 3. The input to the algorithm will be the ray r and the vector representation b of FDH. The result is a number between 0 and $+\infty$ inclusive.

Algorithm 8 Function δ_l for ray and FDH

$[f, l] := \mathrm{LineFDH}(r, b)$
$f := \max(0, f)$
if $f > l$ **then**
  $f := +\infty$
end if
return f

The function first gets the intersection of the line that is containing the ray and FDH. Then it removes half of the line that is beyond the point p. Finally it checks whether the remaining part is empty. After implementing this function we can use it together with δ in procedure LeafProx to find intersection of the ray and the object.

Chapter 4

Conclusion

In this thesis I tried to analyze various aspects of using bounding volume hierarchies to solve proximity computation and collision detection problems. Small part is devoted to raytracing and its relation to these problems.

Chapter 2 brings insight into the structure and properties of the class of bounding volumes called *fixed direction hulls* (FDH) or discrete orientation polytopes (DOPs). We study algebraic properties of their vector representation and geometric meaning of these properties. The most important result of the analysis is the algorithm for linear optimization with FDH as a set of feasible solutions. This is algorithm is surprisingly efficient for solving large number of problems with the same cost function. Another issue discussed in the chapter are methods needed for the use of FDHs in bounding volume hierarchies. Most of these problems are transformed to linear optimization problem mentioned above.

In Chapter 3 we examine general trees of bounding volumes with special attention devoted to issues that arise from using FDHs. Classical algorithm for collision detection using BV-trees is studied and is extended to support also proximity computation. The resulting algorithm algorithm can be used also in areas not related to the computer graphics. It allows to minimize any function over pairs of objects from two given sets, provided that we have procedure to bound it from below.

A method to exploit coherency properties of dynamic problems is suggested as a modification of this algorithm. According to our experience it often dramatically decreases its running time.

Possible extensions of this work include experiments to verify that theoretical running times of algorithms can be achieved. Also the problem of finding dual vertices from Section 2.6.2 still remains to be solved.

Bibliography

[Ba-96] G. Barequet, B. Chazelle, L.J. Guibas, J.S.B. Mitchell, A. Tal. BOXTREE: A Hierarchical Representation for Surfaces in 3D. EuroGraphics'96, J. Rossignac and F. Sillion, eds., Blackwell Publishers, Eurographics Association, Volume 15, 1996, Number 3, pages C-387-C484.

[Be-90] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In Proc. ACM SIGMOD International Conference on Management of Data, pages 322-331, 1990

[Go-96] S. Gottschalk, M. Lin and D. Manocha. OBB-Tree: A Hierarchical Structure for Rapid Interference Detection. Appeared in Proc. of ACM Siggraph'96.

[He-95] M. Held, J. T. Klosowski, J.S.B. Mitchell. Evaluation of collision detection methods for virtual reality fly-throughs. In Proc. 7th Canad. Conf. Comput. Geom, pages 205-210, 1995.

[He-96] M. Held, J. T. Klosowski, J.S.B. Mitchell, H. Sowizral, K. Zikan. Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs. ACM SIGGRAPH'96 Visual Proceedings, New Orleans, LA, August 1996.

[Ka-86] T.L. Kay and J.T. Kaiya. Raytracing complex scenes. SIGGRAPH'86 Proceedings, pp. 296–278, 1986.

[Ko-97] P. Konečný, K. Zikan. Lower Bound of Distance in 3D. Proc. of WSCG'97, Pilsen, Czech Republic, February 1997, Volume III, pages 640-649, ISBN 80-7082-306-2. Also a technical report of FI MU, FIMU-RS-97-01, 1997 (http://www.fi.muni.cz/informatics/reports/, April 1998)

[Pl-90] Lineárne programovanie. Ján Plesník, Jitka Dupačová, Milan Vlach - 1. vyd. - Bratislava, Alfa 1990

[Za-94] G. Zachmann. Exact and Fast Collision Detection. Diploma Thesis, Fraunhofer Institute for Computer Graphics, Technische Hochschule Darmstadt, Fachbereich Informatik, Germany, 1994

[Za-95] G. Zachmann. The BoxTree: Exact and Fast Collision Detection of Arbitrary Polyhedra SIVE'95, University of Iowa, July 1995

[Za-98] G. Zachmann. Rapid Collision Detection by Dynamically aligned DOP-Trees. Proc. of IEEE, VRAIS'98, Atlanta, Georgia, March 1998.