An Efficient Collision Detection Algorithm for Polytopes in Virtual Environments

Chung Tat Leung, Kelvin
M. Phil THESIS

The University of Hong Kong
1996

Abstract of thesis entitled

'An Efficient Collision Detection Algorithm for Polytopes in Virtual Environments'

submitted by

Chung Tat Leung, Kelvin
for the degree of Master of Philosophy
at the University of Hong Kong
in September, 1996.

The problem of collision detection is fundamental to interactive applications such as computer animation and virtual environments. In these fields, prompt recognition of possible impacts and the closest pair of points between two polytopes in collision are important for computing real-time response. We present a simple exact collision detection algorithm for convex polytopes. The algorithm finds quickly a separating plane between two polytopes if they are non-colliding. The validity of this separating plane is verified in constant time instead of linear time as in previous methods [13]. Besides, our algorithm continues to find another separating plane if the current one is not a valid separating plane until collision is detected. Our algorithm guarantees to find a separating plane in a finite number of steps if there is no collision between the two polytopes. In the case of collision, the algorithm reports the closest pair of points between them since the contact points are useful for computing response. In the case of non-collision, the separating plane found for one time frame is cached as a witness for the next time frame. This use of time coherence further speeds up the algorithm in dynamic applications. Both temporal and geometric coherences are exploited to make this algorithm run in expected constant time empirically. In practice, our algorithm is significantly faster than existing methods.

An Efficient Collision Detection Algorithm for Polytopes in Virtual Environments

by

Chung Tat Leung, Kelvin

B.S. Computer Science (University of Hong Kong) 1994

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Master of Philosophy
in
Department of Computer Science
at the
University of Hong Kong

September 1996

Acknowledgments

The successful completion of this thesis is the result of the help, cooperation, love and support of many people. I would like to take this opportunity to express my deepest gratitude to all the people who have given invaluable advices and supports to my research.

First of all, I would like to thank my supervisor, Dr. Wenping Wang. He helped me with precious advices, comments and insightful discussions. Thanks to his guidance during my research and his patience to revise and review my writing many times. Without his patient and insightful guidance, continuous support, and his wide knowledge, this thesis would have been impossible. It is my greatest pleasure to be the research student of Dr. Wang.

Thanks go to Dr. C. L. Wang and Dr. G. Baciu for their agreeing to be the examiners and careful reading of this thesis.

Special thanks should go to my family, who are always supportive and caring. I would like to thank my mother. Her love, concern and encouragement give me inspiration, joy and courage to work hard and finish my study.

Lastly, I wish to express my gratitude to The Department of Computer Science, The University of Hong Kong and The Hong Kong and China Gas Co. Ltd in granting me the studentship, the research grant, and the scholarships for my financial support.

Declaration

I hereby declare that the dissertation, submitted in partial fulfillment of the requirements for the degree of Master of Philosophy and entitled "An Efficient Collision Detection Algorithm for Polytopes in Virtual Environments", represents my own work and has not been previously submitted to this or any other institution for any degree, diploma and other qualification.

Chung Tat Leung
September, 1996

To My Family

Contents

List of Figures

Chapter 1

Introduction

Collision detection problems and their variants are of vital importance in many fields, such as computer animation, physical simulation, computer simulated environments, solid modeling and robot motion planning, especially with the emergent fields of virtual reality [12, 17, 2, 9, 27, 28, 29, 50]. The problems concern the fact that two impenetrable objects cannot share a common region. In computer animation, objects simulated in the environments change motions according to the contact constraints and impact dynamics. It is critical to compute the response in time when objects collide. In physical simulation, complex interactions of hundreds of parts in the virtual prototyping system are simulated based on physics and geometry. It is important to locate the intersection points when parts collide in order to provide proper reaction. In robotics, the collision detection between robots and obstacles is important for robot motion planning and collision avoidance. In solid modeling, complex objects are usually formed from intersection of primitive objects such as in CSG. It is important to identify the intersection area for efficient modeling. In virtual reality, a physical environment is simulated such that humans can readily visualize, explore and interact with the virtual objects in the environment. The virtual world will seem more believable if objects can receive expectable natural behaviour presented as feedback from the objects in the virtual environment such as push, pull and grasp. The fundamental principle behind is that solid objects do not penetrate one another when colliding, instead they should react according to the law of physics.

In all these fields, prompt recognition of possible impacts is necessary for computing responses as an effect of collision. However, the virtual environment usually consists of hundreds or even thousands of simulated moving/static objects, with each object modeled by hundreds of patches. So collision detection is widely recognized to be one of the major bottlenecks towards real time virtual environment simulations [27]. We believe that successful derivation and implementation of fast and efficient collision detection algorithms will significantly enhance the real-time performance of the above applications.

Typically the above collision handling problem has two parts. The *detection algorithm* and the *response algorithm*. The detection algorithm identifies whether objects collide. The response algorithm determines appropriate action based on the contact points and impact dynamics when there is a collision. In this thesis, we will focus on the detection algorithm. For discussions of response algorithms, see [12, 45, 46].

In general we can distinguish between three types of collision detection (see [39]) : *static collision detection*, *pseudo-dynamic collision detection* and *dynamic collision detection*. In static collision detection, the moving object is checked for intersection with the environment at one particular position and orientation. In pseudo-dynamic collision detection, the moving object is checked for intersection with the environment at any of a set of discrete position/orientation pairs along the path corresponding to the object's motion. In dynamic collision detection, the volume swept out by the moving object is checked for intersection with the environment. The problem of static collision detection is studied extensively in computational geometry [40, 41, 42, 14, 43, 22]. There are a number of algorithms with good asymptotic bounds, however they are not practical when implemented in a realistic environment. Solving the pseudo-dynamic collision detection has an advantage that temporal coherence can be exploited to achieve a better expected running time as in [1, 12]. In this thesis, we focus on the use of temporal coherence to solve the problem of pseudo-dynamic collision detection. The problem of dynamic collision detection is addressed in [16, 44].

Objects in a simulated environment are usually modeled by some analytic parametric surfaces, such as NURBS surfaces. Because the solution to the problem of polygonal collision detection is extremely efficient and the objects must usually be represented in polygonal form before being displayed, a hybrid approach is employed as in [46]. In this approach, the boundary surface is first tessellated and the object is represented by boundary representations, then an intersection point of the polygonal surfaces is used as an initial point for numerical methods. For a discussion of intersection between parametric surfaces, refer to [34, 35, 37, 36].

The problem of collision detection is usually coupled with determining the minimum Euclidean distance or the closest pair of points/features between two objects [48, 49, 15, 47, 43], since the closest pair of points provides necessary information for the spatial position of the two objects in order to determine whether they collide or not. However, the problem of finding the closest pair of points between two objects is a harder problem than the collision detection problem itself in the sense that we can solve the collision detection problem without any knowledge about the closest pair of points between them. As a result, it is expected that more computation is needed if one wants to find the closest pair of points between the two objects. However, existing methods on collision detection attempt to find the closest pair of points to solve the above harder problem when their bounding boxes overlap.

In this thesis, we present a new and efficient approach to solving the collision detection problem without computing the closest pair of points between the two objects involved. Although the information about the closest pair of points is important in computing the response when objects collide, such information is usually not useful when there is no collision. Hence we propose in this thesis to calculate the closest pair of points only when collision occurs.

We propose an efficient algorithm, called the *separating vector algorithm*, to handle collision detection of polytopes. This algorithm is a major improvement on existing ones in terms of running time, implementation simplicity, and memory requirement. Our algorithm extends the idea in [13], which caches a separating plane H between two non-colliding polytopes after the closest pair of points $\mathbf{p}$ and $\mathbf{q}$ of polytopes P and Q are computed. This cached plane H is a plane passing through the point $(\mathbf{p}+\mathbf{q})/2$ with normal $\mathbf{q}-\mathbf{p}$. In the next time frame, the algorithm uses linear time to check if all vertices of P lie in one side of H and all vertices of Q lie in the opposite side of H. If the above condition is satisfied then P and Q do not collide and there is no need to find the closest pair of points. Otherwise the closest pair of points are computed and another plane H is found.

Our algorithm finds a separating plane with a different strategy that does not rely on computing the closest pair of points. Besides, our algorithm verifies the validity of the cached separating plane in expected constant time instead of linear time as in [13]. Moreover, instead of giving up and spending considerable amount of time on computing the closest pair of points, our algorithm will continue to find another separating plane by using a simple iterative method if the current one is not a valid separating plane. If there is no collision, our algorithm will find a proper separating plane quickly in a finite number of steps. Otherwise it will report collision after testing some simple conditions. In the case of collision, information from the preceding time frame is used to compute the closest pair of points.

Since the closest pair of points may be useful when there is a collision, we have improved and integrated Gilbert's algorithm [18] into our separating vector algorithm so that the closest pair of points can be found quickly. In the typical virtual reality environments, the running time of the improved algorithm is nearly independent of polytope's size when temporal coherence is exploited. Without making use of temporal coherence, we also improve the running time of Gilbert's algorithm by modifying the way supporting vertices are searched from time complexity $O(n)$ to $O(\log n)$, where n is the number of vertices. As a consequence, we greatly improved the running time of the Gilbert's algorithm for collision detection of two polytopes.

Our algorithm makes use of temporal coherence by caching a separating plane for successive time frames. A special property of our algorithm is that the closest pair of points/features is computed only when there is a collision, as opposed to [10] in which the closest pair of features between two polytopes are computed for every time frame. Besides, our algorithm considers vertices of polyhedra only, rather than all boundary features (vertices, edges, and faces) as in [10], so it is more efficient and simpler to implement. Temporal and geometric coherences are exploited to make the algorithm run in expected constant time empirically. We have shown that the performance of our algorithm meets requirements for real-time interactive collision detection of polytopes in virtual environments.

1.1 Previous Work

The problem of collision detection has been extensively studied in many fields, such as robotics and computational geometry. Most of the research makes use of axis-aligned bounding boxes or a hierarchy of them as the first step to quickly eliminate most non-interfering objects, assuming that no self-intersection occurs as in deformable objects. For n bounding boxes, the $O(n^2)$ pairwise checks have been reduced to $O(n \log n + e)$, where e is the number of intersections, using the octree [9] and sorting [30]. In the octree methods, the space is divided into 8 sub-spaces with each one recursively subdivided further. All the faces that are in one sub-space do not intersect with faces in the other 7 sub-spaces. Based on this observation, the collision of each face is searched only in the sub-trees where possibly intersecting faces exist. Recently, the bound $O(n+e)$ is achieved in [1] by projecting the endpoints of three-dimensional bounding boxes onto the x, y, z axes and sorting them at each time instant. Due to geometrical coherence, the expected sorting time is linear. Two bounding boxes overlap if and only if their intervals overlap in all of the three dimensions.

Spatial subdivision or spatial partitioning [3] is another method to facilitate the bounding box tests. The space is divided into cells of equal volume, and each object is assigned to one or more cells. Collision is checked between all object pairs belonging to a particular cell. Yet another method makes use a scheduling scheme such that objects that are close to each other are checked more frequently at each time instant [4, 5]. The idea is to calculate the shortest possible time before the next possible collision for every pair, assuming that velocity is bounded. Another approach using time bound is described in [16]. This is a progressive refinement approach which is based on two forms of approximate geometry, a sphere-tree and a four-dimensional structure called a space-time bound. The algorithm is interruptible and can trade quality for speed when necessary.

The next step for the collision detection algorithm is to locate possible intersection of faces between objects whose bounding boxes overlap. As each object can be made up of hundreds of triangular patches, efficient algorithms have to be utilized. In [7], faces of objects that intersect the overlapping region of bounding boxes are determined first using clipping. Then a face octree is built for the above faces to check for possible intersection. This method has the advantage that it can deal with concave and deformable objects. In [8], the rectangular box bounding an object is subdivided into voxels, ordered in a 3D array, in its modeling reference system. Each element of the array is a pointer to a list of facets that intersect the voxel. Intersection is done by transformation of the voxel from one reference frame to another.

In [11], a data structure, called a "BRep-Index", is used for quick spatial access of polyhedra in order to localize contact regions between two objects. In [18], an expected linear time algorithm which computes the minimum distance and a separating plane of two objects is proposed. In [13], separating planes for pairs of objects are found by the above expected linear time algorithm and cached [12] to yield a reply of non-collision most of the time using temporal coherence. However, it also takes linear time in the following time frame to test the validity of the cached separating plane. In [19], a sub-quadratic running time algorithm to detect collision between polytopes is proposed.

When the motion is restricted to be translational only, the best theoretical running time so far for detecting collision between two polytopes is $O(\log p * \log q)$, where p and q are the number of faces of two polytopes respectively, using the hierarchical representation of convex polyhedra [14]. The algorithm needs $O(p+q)$ preprocessing time to build the hierarchical structure. Then the closest pair of points is found incrementally starting from the lowest hierarchical level to the highest one.

In [31], back-face culling is used to remove roughly half of the faces of objects from being checked for interference. The basic idea is that for any two objects, the polygon faces of one object facing backwards with respect to the relative direction of motion cannot collide with the other object. In [20], the ideas of [1] and [10] are extended to deal with concave polytopes. In [32, 33] the idea of z-buffer visible surface algorithm is used to perform interference detection through rasterization. Other methods to detect collision usually decompose the object into a hierarchical structure, so that concave polyhedra can be handled. Those methods include octree [2], BSP tree [3], C-Tree [38], sphere trees [23], strip trees and boxtrees [25], R-trees [24] and OBB-Tree (Oriented Bounding Box Tree)[26]. Among them, OBB-Tree is more efficient than other hierarchical trees because of its tightest bounding box. In OBB-Tree, the rectangular bounding box is not axis-aligned. Instead it makes use of statistical techniques to analysis the distribution of vertices in space. In this way it yeilds a tight fitting oriented bounding box. A top-down approach is used to build the bounding boxes tree by recursively subdividing a group of polygons until all leaf nodes are indivisible. The overall time to build the tree is $O(n \log n)$. Besides, a Separating Axis Theorem is described in [26] that can determine if two OBBs overlap in less than 200 operations. The theorem states that if two OBBs are not in contact, then there exists a separating axis which is the cross product of two distinct vectors taken from the six box axes. Although OBB-Tree is already quite efficient for general polyhedral, for convex polyhedra geometric coherence can be exploited to devise faster algorithm without decomposing the polyhedra.

In the implemention of our collision detection library, we use the sweep and prune technique with axis-aligned bounding boxes instead of OBB. It is because our library is modified from I_COLLIDE library, an implementation of the closest features tracking algorithm in [1], which uses axis-aligned bounding boxes. Besides, the complexity of testing the intersection of n OBBs is $O(n \log n)$, while the sweep and prune technique for axis-aligned bounding boxes requires only expected $O(n)$

time.

The method in [10] maintains a closest pair of features (vertices, edges, or faces) for each pair of polytopes and calculates the Euclidean distance between them. The method is based on the fact that two features F_1 and F_2 are the closest pair of features if and only if F_1 lies in the Voronoi region of F_2 and F_2 lies in the Voronoi region of F_1. This method takes advantage of geometric coherence and runs in expected constant time if the polytopes do not move swiftly. Since the algorithm needs to compute and store the Voronoi region for each vertex, edge and face on the boundary, and to handle different cases when walking around on the boundary in order to find the closest pair of features, the implementation is complicated. Moreover, in some applications the pair of closest features are not of great interest to the program when polytopes do not collide. So sometimes it is not worth continuing to compute the closest pair of features once it is known that a separating plane exists between the two polytopes.

As for curved objects, a general algorithm is described in [34] for time dependent parametric surfaces. The algorithm uses a subdivision technique in the resulting space. A similar method using interval arithmetic and subdivision is presented in [35]. However, for commonly used spline patches, computing and representing the implicit representations are computationally expensive [36]. In [37] implicit functions are used to represent shapes and the property of the "inside-outside" functions is employed for collision detection. In [6] homogeneous bounding boxes are used as tools in intersection algorithms of rational Bézier curves and surfaces.

In summary, existing algorithms are not as efficient, simple, and practical as ours for collision detection of polytopes in virtual environments. Moreover, existing algorithms tend to solve the harder problem of finding the closest pair of points between two polytopes but not exactly the problem of determining if two polytopes collide.

1.2 Main Contributions

The main contributions of this thesis are :

1. A new simple and efficient algorithm, called *separating vector algorithm*, for collision detection of polytopes in virtual environments.
2. An improved Gilbert's algorithm to compute the closest pair of points more robustly and efficiently by exploiting geometric and temporal coherences.
3. A new $O(\log n)$ algorithm to find a supporting vertex of a polytope with $O(n)$ preprocessing where n is the total number of vertices in polytopes.
4. A technique that makes use of the hierarchical representation of polytopes to find a supporting vertex by local search.
5. An implementation of a collision detection library, called Q_COLLIDE (Quick collision detection library), based on our algorithm.
6. Experiments show that our collision detection library Q_COLLIDE is more efficient than the currently fastest collision detection library I_COLLIDE in all situations, especially when the rotational/translational velocity of polytopes in the environment is high or the complexity of polytopes is high. In particular, Q_COLLIDE is at least one order of magnitude faster then I_COLLIDE when the number of vertices of each polytope exceeds 200.

1.3 Overview of the Thesis

This thesis is organized in a way that each chapter is based on the foundation provided by the preceding chapters. We begin in Chapter 2 to describe basic concepts used in the development of the algorithms presented later. The computational geometry and modeling concepts we describe are convex polyhedra, hyperplanes, Minkowski sum and boundary representation.

Chapter 3 presents the main idea of our separating vector algorithm, which is a simple and practical algorithm for collision detection between two polytopes. In this chapter, we describe how the algorithm can be used to detect non-interference polytopes efficiently - by exploiting geometric and temporal coherences to find a separating plane between two polytopes in expected constant time. The algorithm is particularly suitable for dynamic collision detection. Experiments show that it can eliminate about 99% of non-interference polytopes with at most four iterations of the algorithm, nearly independent of polytopes' shape and complexity, before a usually more time-consuming exact collision detection algorithm must be invoked.

Chapter 4 extends the separating vector algorithm to detect collision between two polytopes by enforcing a termination condition so that the algorithm can detect if a collision occurs. Then we prove that the algorithm is guaranteed to find a separating plane if it exists. The later parts of this chapter deal with the complexity of the algorithm and its possible improvements.

Chapter 5 describes how to extend the separating vector algorithm so that the closest pair of points is also reported when there is a collision, which is useful in computing responses when collision. We first review the idea of Gilbert's algorithm and Johnson's algorithm [51]. Then we describe how to improve Gilbert's algorithm so that it is more robust and supporting vertices can be found in $O(\log n)$ time instead of $O(n)$ time. Afterwards, we describe how to use local search to further improve the complexity of Gilbert's algorithm when both geometric and temporal coherences are exploited. Finally, we describe how to integrate our separating vector algorithm with the improved Gilbert's algorithm seamlessly so that the whole algorithm can run in expected constant time empirically.

In Chapter 6, we will discuss the implementation details of our separating vector algorithm and the improved Gilbert's algorithm. We implemented our algorithm as a collision detection library, called Q_COLLIDE. Experiments have been carried out to verify the nice properties of our algorithm. To test the efficiency and effectiveness of our algorithm, we compare it with I_COLLIDE - an implementation of the closest feature tracking algorithm, which is the fastest collision detection library for polytopes so far [1]. Our results show that Q_COLLIDE is more efficient in all cases, especially when the rotational/translational velocity of polytopes in the environment is high or the complexity of polytopes is high.

The thesis is concluded in Chapter 7.

Chapter 2

Background

In this chapter, we will describe some basic concepts used in the development of our algorithms. We consider rigid polyhedral objects with bounding surfaces described by polygonal meshes. This representation is commonly used in rendering simulated objects in virtual environments. Some of the material presented here can be found in [36, 21].

2.1 Convex Polyhedra

An object is represented by a compact set $P_c \subset E^n$. In E^n a set P_c is convex if for any two points in P_c, A and B, the line segment $\overline{AB}$ is entirely contained in P_c. A point $\mathbf{p}$ is affinely dependent on a set of points $P = \{\mathbf{p_1}, \mathbf{p_2}, \ldots, \mathbf{p_r}\}$ in E^n if there exist real numbers $\lambda_1, \lambda_2, \ldots, \lambda_r$ such that $\mathbf{p} = \lambda_1\mathbf{p_1} + \lambda_2\mathbf{p_2} + \cdots + \lambda_r\mathbf{p_r}$ and $\lambda_1 + \lambda_2 + \cdots + \lambda_r = 1$. If, in addition, the constraint $\lambda_i \geq 0$ holds for $i = 1, 2, \ldots, r$, then $\mathbf{p}$ is convexly dependent on P, i.e. $\mathbf{p}$ is inside the convex hull of P.

We define the affine hull and convex hull of a set of points $P \subset E^n$ as :

$$\mathrm{aff}(P) = \sum_i^r \lambda_i \mathbf{p_i}, \quad \text{where} \quad \sum_i^r \lambda_i = 1,\ \mathbf{p_i} \in P,\ \lambda_i \in R \tag{2.1}$$

$$\mathrm{conv}(P) = \sum_i^r \lambda_i \mathbf{p_i}, \quad \text{where} \quad \sum_i^r \lambda_i = 1,\ \lambda_i \geq 0,\ \mathbf{p_i} \in P,\ \lambda_i \in R \tag{2.2}$$

A set of points is affinely independent if no point in the set is affinely dependent on the other members of the set. The convex hull of a set of finite points is called a convex polyhedron, or simply a *polytope*.

2.2 Hyperplanes

A hyperplane H is an affine subspace of E^n of dimension $n-1$. It is defined by $\{\mathbf{p} \in E^n \mid \mathbf{p} \cdot \mathbf{S} = \beta\}$ for some n-vector $\mathbf{S}$, where $\mathbf{x} \cdot \mathbf{y}$ is the inner product of vectors $\mathbf{x}$ and $\mathbf{y}$. The two half-closed spaces determined by a hyperplane H are defined as :

$$H^+ = \{\mathbf{p} \in E^n \mid \mathbf{p} \cdot \mathbf{S} \geq \beta\} \text{ and } H^- = \{\mathbf{p} \in E^n \mid \mathbf{p} \cdot \mathbf{S} \leq \beta\} \tag{2.3}$$

A hyperplane H is a supporting hyperplane of a convex set P if (i) $P \cap H \neq \emptyset$ and (ii) P is contained in one of the closed half-spaces determined by H.

The supporting function $H_P : R^n \mapsto R$ of P is defined by

$$H_P(\mathbf{S}) = \max\{\mathbf{p} \cdot \mathbf{S} \mid \mathbf{p} \in P\}, \quad \mathbf{S} \in R^n \tag{2.4}$$

Since there may be more than one supporting point, we define the contact function $C_P(\mathbf{S}) : E^n \mapsto P$ as one of the solution of the above equations :

$$C_P(\mathbf{S}) = \mathbf{p}' \text{ such that } \mathbf{p}' \in \{\mathbf{p} \mid \mathbf{p} \cdot \mathbf{S} = H_P(\mathbf{S}), \mathbf{p} \in P\} \tag{2.5}$$

Geometrically, $C_P(\mathbf{S})$ is one of the points farthest in P in the direction $\mathbf{S}$ (see Figure 2.1). We define it as the *supporting point* of P in the direction $\mathbf{S}$. Note

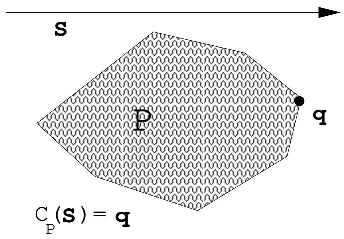

Figure 2.1: Supporting vertex of P in the direction $\mathbf{S}$.

that this point is not unique in general. It has the property that the hyperplane H passing through it with normal $\mathbf{S}$ is a supporting hyperplane of P.

If P is a finite set of points $\{\mathbf{p_1}, \mathbf{p_2}, \ldots, \mathbf{p_r}\}$ in E^n, then the support and contact functions for conv(P) are defined similarly :

$$H_{conv(P)}(\mathbf{S}) = \max\{\mathbf{p_i} \cdot \mathbf{S} \mid i = 1, \ldots, r\}, \quad \mathbf{S} \in E^n \tag{2.6}$$

$$C_{conv(P)}(\mathbf{S}) = \mathbf{p_j}, \quad \text{such that} \quad \mathbf{p_j} \in \{\mathbf{p_i} \mid \mathbf{p_i} \cdot \mathbf{S} = H_{conv(P)}(\mathbf{S}),\ 1 \leq i \leq r\} \tag{2.7}$$

We define $C_{conv(P)}(\mathbf{S})$ as a *supporting vertex* of P in the direction $\mathbf{S}$. Again $C_{conv(P)}(\mathbf{S})$ is not unique but it doesn't affect our algorithm to be developed later.

With the above definition in mind, we can now state the necessary and sufficient condition for there to be a collision between two polytopes P and Q. If there exists a hyperplane H such that P and Q belong to opposite half-spaces of H, i.e. $P \subset H^+$ and $Q \subset H^-$ or vice versa, then P and Q are said to be *separated*. If in addition P and Q do not both intersect H, then P and Q are properly separated. Otherwise P and Q may touch each other without overlapping. It is shown in [21] that if P and Q are convex sets, then the necessary and sufficient condition for the existence of a hyperplane separating P and Q properly is that they have no common points, i.e. the minimum distance between P and Q is positive. In other words, a hyperplane exists that properly separates P and Q is the necessary and sufficient condition that P and Q do not collide.

2.3 Minkowski Sum

The Minkowski sum Z of two polytopes Q and P is defined as

$$Z = Q + P = \{\mathbf{q} + \mathbf{p} \mid \mathbf{q} \in Q, \mathbf{p} \in P\} \tag{2.8}$$

It is well known that the Minkowski sum of two convex polytopes is also convex polytope. From the definition, if P has n vertices and Q has m vertices, then Z has at most $O(nm)$ vertices. An example of Minkowski sum is shown in Figure 2.2.

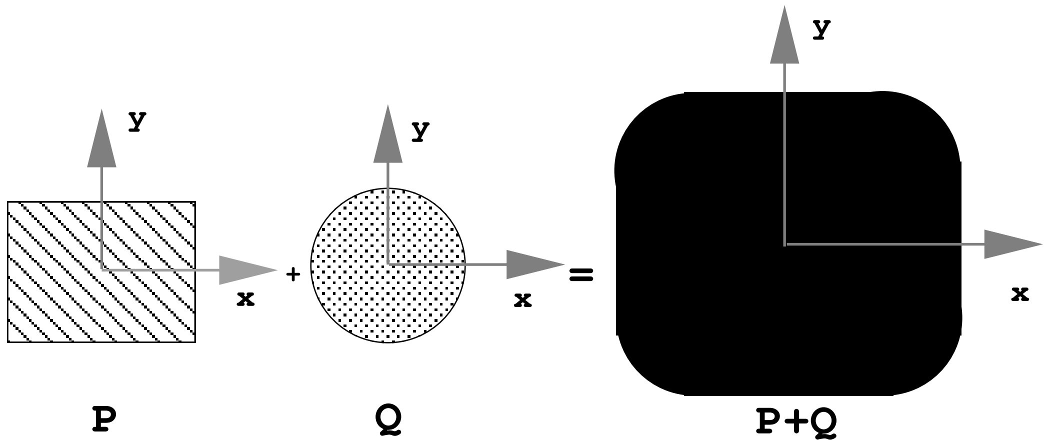

Figure 2.2: An example of Minkowski sum.

If we define $-P = \{-\mathbf{p} \mid \mathbf{p} \in P\}$, then the set $M = Q - P = \{\mathbf{q} - \mathbf{p} \mid \mathbf{q} \in Q, \mathbf{p} \in P\}$ is the Minkowski sum of Q and $-P$. The set M contains the origin if and only if P and Q collide, since

$$\begin{aligned} P \text{ and } Q \text{ collide} \quad &\Leftrightarrow \quad P \text{ and } Q \text{ have a common point} \\ &\Leftrightarrow \quad \mathbf{p} = \mathbf{q} \text{ for some } \mathbf{p} \in P \text{ and } \mathbf{q} \in Q \\ &\Leftrightarrow \quad \mathbf{q} - \mathbf{p} = \mathbf{0} \text{ for some } \mathbf{p} \in P \text{ and } \mathbf{q} \in Q \\ &\Leftrightarrow \quad \mathbf{0} \in M \end{aligned}$$

Hence the problem of collision detection between P and Q is reduced to the problem of determining whether the origin is contained in M. Besides, the problem of finding the closest pair of points between P and Q is reduced to the problem of finding the closest point of M to the origin. Later we will show how to compute the closest pair of points using this idea.

Although the Minkowski sum $M = Q - P$ needs $O(nm)$ time forming, the support and contact functions of M can be computed in $O(n + m)$ time [18], since

$$H_M(\mathbf{S}) = H_Q(\mathbf{S}) + H_P(-\mathbf{S}) \tag{2.9}$$

$$C_M(\mathbf{S}) = C_Q(\mathbf{S}) - C_P(-\mathbf{S}) \tag{2.10}$$

This reduces greatly the effort of computing the supporting and contact functions of the Minkowski sum of two sets. An example of the use of Minkowski sum to solve the collision detection problem can be found in [18].

2.4 Boundary Representation

A polyhedral object can be represented by boundary representation unambiguously by describing its surface and its topological orientation in such that we have the complete information about the interior and exterior of an object. This descriptions has two parts, a topological description of the connectivity and orientation of vertices, edges and faces, and a geometric description of coordinates needed to describe vertices, edges and faces. The topological description specifies the incidences and adjacencies of vertices, edges and faces. For example, in boundary representation a vertex structure has a field that specifies the edges incident to the vertices and the faces incident to the vertices, both in clockwise order when viewing from outside the solid.

In this thesis, polytopes are represented by boundary representation but the topological and geometric description of edges and faces are removed, as this information is not used in our algorithm. The removal of edges and faces topological and geometric information greatly reduces the coding and memory requirement of our algorithm. In our algorithm, the topological description that specifies a vertex v are the vertices adjacent to v.

2.5 Hierarchical Representation

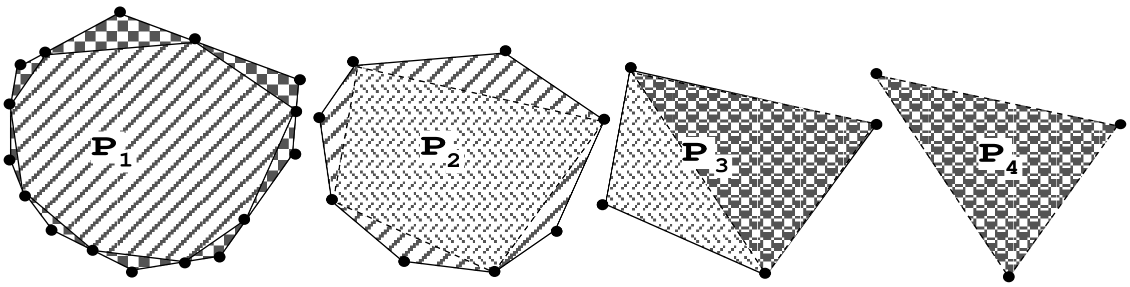

Figure 2.3: A hierarchical representation of a polygon.

A hierarchical representation of a polytope P [14] with vertex set $\mathbf{V}(P)$ is defined as a sequence of polytopes hier$(P) = \{P_1, \ldots, P_h\}$ such that

(i) $P_1 = P$ and P_h is a simplex;

(ii) $P_{i+1} \subset P_i$, for $1 \leq i < h$;

(iii) $\mathbf{V}(P_{i+1}) \subset \mathbf{V}(P_i)$, for $1 \leq i < h$; and

(iv) the vertices in $\mathbf{V}(P_i) - \mathbf{V}(P_{i+1})$, which is a set difference, form an independent set in P_i for $1 \leq i < h$.

Let h and $\sum_{i=1}^{h} |V(P_i)|$ be the height and size of hier(P) respectively ($|V(P)| =$ total number of vertices of P). It is proved in [14] that the height of the hierarchical representation, h, is $O(\log n)$, assuming the degrees of all vertices are bounded. Moreover, it takes $O(n)$ time and space to build a hierarchical representation of polytope.

This representation has been used to derive the $O((\log n)^2)$ algorithm in [43] for computing the closest pair of points between two polytopes (n is the total number of vertices). An example of hierarchical representation in 2D case is shown in Figure 2.3. Note that the hierarchical representation of P is not unique.

Chapter 3

Separating Vector Algorithm

In this chapter we present a simple and efficient algorithm to quickly eliminate non-interference polytopes in a virtual environment. This method is especially suitable for repetitive collision detection when the objects do not move swiftly so temporal coherence can be exploited in the algorithm. When objects move very fast, the algorithm will take slightly longer time. The basic idea is to detect collision between polytopes using the fact that two polytopes are separated if and only if there exists a plane such that they belong to the opposite half-spaces of this plane [21]. In each iteration, the algorithm finds a candidate plane and uses constant time to verify whether this plane is a separating plane. If it is a separating plane then the polytopes do not collide, and this plane is cached to be used as the initial plane in the search for a separating plane in the next time frame; otherwise the algorithm continues to search for a separating plane.

Our algorithm is efficient and simple to implement. We will give a thorough discussion on how the algorithm utilizes caching, preprocessing and local searching so to run in expected constant time empirically. Some properties of the separating vector algorithm will be proved at the end of this chapter. An important property is that the plane found by the separating vector algorithm in each iteration is guaranteed to be closer than the plane in the previous iteration to any fixed separating plane, if it exists.

In the next chapter we will extend this algorithm to an exact collision detection algorithm by adding a termination condition. In such a way, if the algorithm has determined that a separating plane cannot possibly exist, it reports collision.

3.1 Collision Detection in Large-Scale Virtual Environments

Our algorithm is an exact collision detection algorithm between convex polytopes. It can be used as a standalone algorithm for collision detection between polytopes in virtual environments. However, in a large scale virtual environment it is more efficient to combine our algorithm with a bounding box algorithm in order to reduce pairwise collision tests. With some preprocessing, our algorithm can integrate nicely with a bounding box algorithm to detect collision efficiently. We propose to handle collision detection of convex polytopes in a large-scale virtual environment in four steps :

1. Use spatial partitioning to eliminate non-interference objects that belong to different regions.
2. Within each region, use the sweep and prune technique [1] to quickly identify object pairs whose rectangular bounding boxes overlap.
3. Use the separating vector algorithm to detect collision between polytopes whose bounding boxes overlap.
4. Use the improved Gilbert's algorithm when collision occurs to compute the closest pair of points in the immediately preceding non-collision time frame.

Later, we will describe how to use the results of each step to initialize the computation of the next step so that collision detection can be done efficiently. The details of our separating vector algorithm are given below. The improved Gilbert's algorithm will be presented in Chapter 5.

3.2 The Algorithm

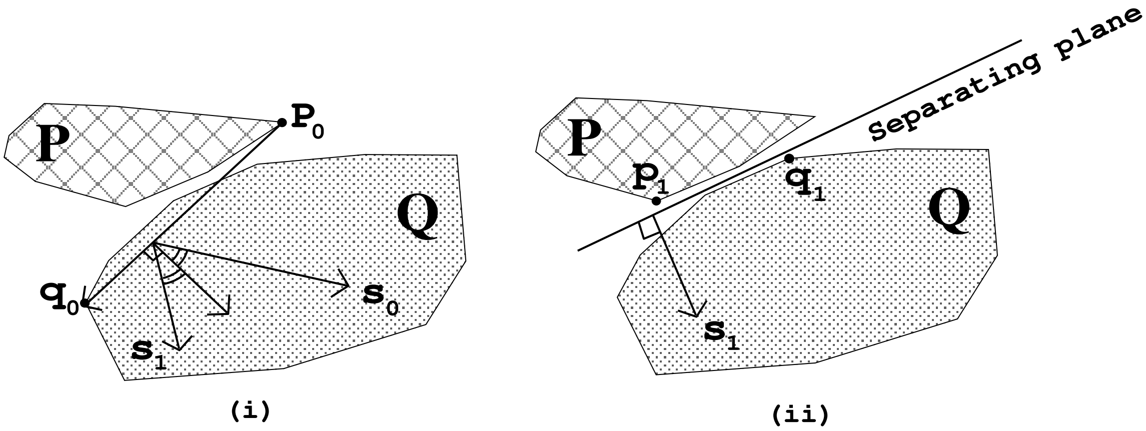

Figure 3.1: The idea of searching for a separating vector.

In this section we present the main idea of our algorithm. Recall that if $\mathbf{V}(P)$ denotes the set of vertices of polytope P, a supporting vertex of P in the direction $\mathbf{S}$ is given by $\mathbf{p} \in \mathbf{V}(P)$ where $\mathbf{S} \cdot \mathbf{p} = \max\{\mathbf{S} \cdot \mathbf{p}' \mid \mathbf{p}' \in \mathbf{V}(P)\}$. In fact, for a supporting vertex $\mathbf{p}$ of P, we have $\mathbf{S} \cdot \mathbf{p} = \max\{\mathbf{S} \cdot \mathbf{p}' \mid \mathbf{p}' \in P\}$ since $\mathbf{p}$ is a convex polyhedron.

Lemma 1: *For a vector* $\mathbf{S}$, *let* $\mathbf{p}$ *be a supporting vertex of polytope* P *in the direction* $\mathbf{S}$ *and* $\mathbf{q}$ *be a supporting vertex of polytope* Q *in the direction* $-\mathbf{S}$. *If* $\mathbf{S} \cdot (\mathbf{q} - \mathbf{p}) > 0$, *then* P *and* Q *do not intersect.*

Proof: Since $\mathbf{S} \cdot (\mathbf{q} - \mathbf{p}) > 0$, we have $\mathbf{S} \cdot \mathbf{q} > \mathbf{S} \cdot \mathbf{p}$, which implies $\mathbf{S} \cdot \mathbf{q} > \mathbf{S} \cdot (\mathbf{p} + \mathbf{q})/2 > \mathbf{S} \cdot \mathbf{p}$. By definition, $\mathbf{S} \cdot \mathbf{p} \geq \mathbf{S} \cdot \mathbf{p}'$ for any $\mathbf{p}' \in P$ and $\mathbf{S} \cdot \mathbf{q}' \geq \mathbf{S} \cdot \mathbf{q}$ for any $\mathbf{q}' \in Q$. So $\mathbf{S} \cdot \mathbf{q}' > \mathbf{S} \cdot (\mathbf{p} + \mathbf{q})/2 > \mathbf{S} \cdot \mathbf{p}'$. Hence the plane containing the point $(\mathbf{p} + \mathbf{q})/2$ with normal vector being $\mathbf{S}$ separates properly P and Q. □

To explain the idea of our algorithm, a 2D version is first presented. Figure 3.1 shows two non-overlapping convex polygons P and Q. Note that Lemma 1 also holds for the 2D case with the term "polytope" being replaced by "convex polygon".

Briefly, the algorithm works as follows. Given two convex polygons P and Q, initially a unit vector $\mathbf{S_0}$ is chosen and a supporting vertex $\mathbf{p_0}$ of P in the direction

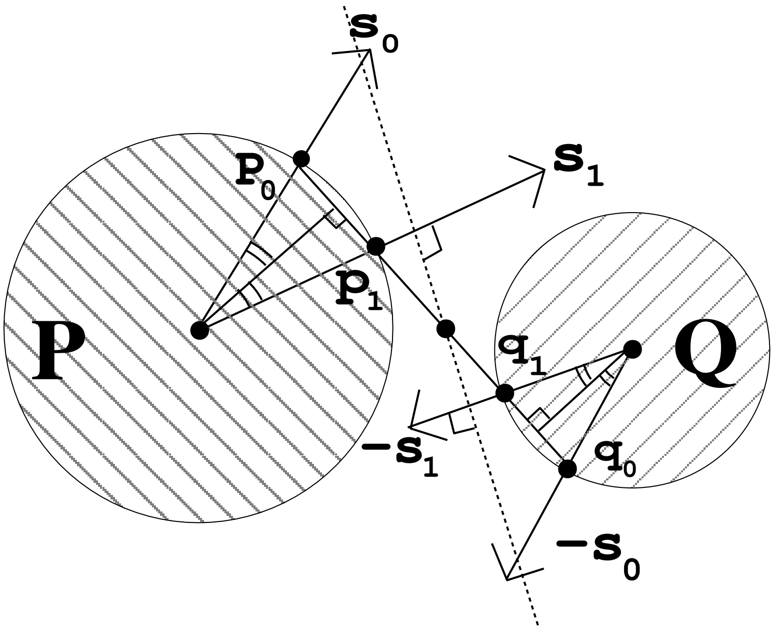

Figure 3.2: Searching for a separating vector in the case of circles.

$\mathbf{S_0}$ is found. Similarly, a supporting vertex $\mathbf{q_0}$ of Q in the direction $-\mathbf{S_0}$ is found. Then the following condition is tested. By Lemma 1, with $i = 0$, P and Q do not collide if

$$\mathbf{S_i} \cdot (\mathbf{q_i} - \mathbf{p_i}) \geq 0. \tag{3.1}$$

Any vector $\mathbf{S_i}$ satisfying the above condition is called a *separating vector* of P and Q, or just a separating vector since P and Q are often clear from the context. We also call the vertex $\mathbf{p_i}$ or $\mathbf{q_i}$ satisfying the above condition a *separating vertex* of P or Q respectively. Note that we consider the case where $\mathbf{S_0} \cdot (\mathbf{q_0} - \mathbf{p_0}) = 0$ to be non-collision, although in this case P and Q may touch each other. A separating vector $\mathbf{w}$ of P and Q has the property that $\mathbf{w} \cdot (\mathbf{q}' - \mathbf{p}') \geq 0$ for any $\mathbf{p}' \in P$ and any $\mathbf{q}' \in Q$.

In general, if the above test fails for $\mathbf{S_i}$, P and Q may still not collide. In this case we find a new direction $\mathbf{S_{i+1}}$ from $\mathbf{S_i}$, $\mathbf{p_i}$ and $\mathbf{q_i}$ by

$$\mathbf{S_{i+1}} = \mathbf{S_i} - 2(\mathbf{r_i} \cdot \mathbf{S_i})\mathbf{r_i}, \quad i = 0, 1, \ldots, \tag{3.2}$$

where $\mathbf{r_i} = (\mathbf{q_i} - \mathbf{p_i})/\|(\mathbf{q_i} - \mathbf{p_i})\|$ (see Figure 3.3). Note that $\mathbf{S_{i+1}}, \mathbf{S_i}$, and $\mathbf{r_i}$ lie

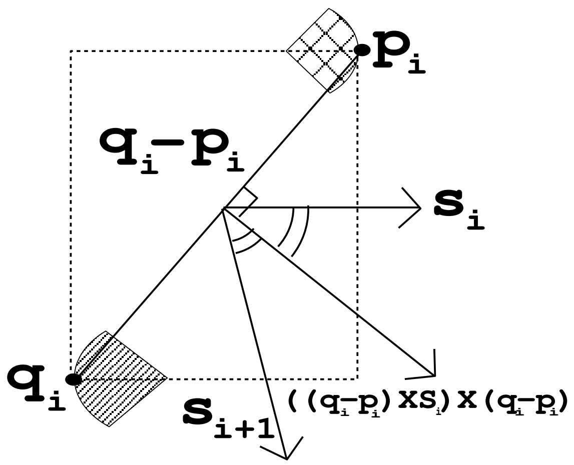

Figure 3.3: Choosing the next searching direction $\mathbf{S_{i+1}}$.

on the same plane, and the angle between $\mathbf{S_{i+1}}$ and $\mathbf{S_i}$ is bisected by the vector $((\mathbf{q_i} - \mathbf{p_i}) \times \mathbf{S_i}) \times (\mathbf{q_i} - \mathbf{p_i})$ perpendicular to $\mathbf{r_i}$.

This choice of $\mathbf{S_{i+1}}$ from $\mathbf{S_i}$ is based on the following observation. Consider two non-intersecting circular disks P and Q in the plane (see Figure 3.2). If $\mathbf{S_0}$ is not a separating vector, then the vector $\mathbf{S_1}$ computed by formula (3.2) is always a separating vector of the two circles. This is because the line segment $\mathbf{p_0 q_0}$ intersects P at $\mathbf{p_1}$, which must be the supporting vertex of P in the direction $\mathbf{S_1}$ due to the special geometric property of circles. Similarly the line segment $\mathbf{p_0 q_0}$ intersects Q at $\mathbf{q_1}$ which is also the supporting vertex of Q in the direction $-\mathbf{S_1}$. Now the line segment $\mathbf{p_1 q_1}$ does not intersect any other points of P and Q. Hence a line (or a plane in 3D case) passing through $(\mathbf{p_1} + \mathbf{q_1})/2$ with normal $\mathbf{S_1}$ separates properly P and Q. This argument is also true of two non-intersecting spheres, by considering the cross-sections of two spheres with the plane determined by $\mathbf{S_0}$ and the centers of the spheres. So in the general case of polytopes we choose $\mathbf{S_{i+1}}$ by formula (3.2) in the hope that $\mathbf{S_{i+1}}$ thus chosen converges quickly to some separating vector, provided that P and Q do not collide.

If condition (3.1) does not hold and collision conditions (to be given later) are not satisfied, the above procedure is repeated. The first $\mathbf{S_k}$ that satisfies condition (3.1) is a separating vector of P and Q, and k is the number of iterations performed by the algorithm.

This algorithm works exactly the same way in 3D case. It is proved in the next section that if the two polytopes do not collide and the condition (3.1) does not hold, $\mathbf{S_i}$ will get closer and closer to any fixed separating vector by each iteration. Because of convexity, local search (to be explained in next section) is sufficient to locate the supporting vertices. In the following sections, we will explain how local searching, caching, preprocessing can help the separating vector searching step achieve expected constant running time empirically.

In practice, this algorithm, without using exact collision testing, is useful to eliminate non-interference polytopes. If after a prescribed number of iterations the algorithm cannot find a separating plane, then an exact collision detection algorithm can be used. Experiment shows that about 99% of non-interference polytopes are identified as such within 4 iterations. Conditions for reporting collision when the two polytopes collide will be discussed in the next chapter.

3.3 Searching for Supporting Vertices

We use the searching algorithm outlined in [1] to find supporting vertices $\mathbf{p_i}$ on P and $\mathbf{q_i}$ on Q, with respect to $\mathbf{S_i}$ and $-\mathbf{S_i}$, respectively. In the search the current vertex $\mathbf{p}$ on P is compared to its neighboring vertices to see if $\mathbf{S_i} \cdot \mathbf{p}$ is the largest. If this is the case, the current vertex is a supporting vertex; otherwise this vertex is replaced by a neighboring vertex $\mathbf{p}'$ with the largest $\mathbf{S_i} \cdot \mathbf{p}'$. This process is repeated until a supporting vertex is found. Notice that the supporting vertex may not be unique but this does not affect our algorithm. Moreover, the search is performed locally on the surface of the polytopes. Because of convexity, this search can always find a supporting vertex in a finite number of steps. If we assume that polytopes do not move very fast between successive time frames (which is usually the case in a virtual environment), then the separating vertex found for the preceding time frame is usually close to a separating vertex in the current time frame. So if we use the separating vertex of the preceding time frame as an initial point of search, then the required supporting vertices can be found quickly. Strictly speaking, the time complexity to locating a supporting vertex using the above strategy depends on how much the polytopes rotate/translate between successively time frame. In practice, we observe that the running time is nearly constant as polytopes usually do not move swiftly in a virtual environment, which has been verified by experiments. A supporting vertex $\mathbf{q}$ on Q can be found similarly.

To speed up the searching process, we can use a timestamp associated with each vertex to remember which vertices have already been visited. There is also a global counter which increments by one every time the local search is performed. When searching for the new supporting vertices, a vertex is ignored if its timestamp matches the current global counter. Otherwise the dot product is evaluated and its timestamp is set equal to the current global counter. On average, about one third of computation in the search for supporting vertices is saved by using this approach.

Another advantage of using local search is that, in implementation, there is no need to transform each vertex of polytope P or Q from its defining coordinate system to the world coordinate system and then take the dot product with $\mathbf{S_i}$ in order to find a supporting vertex. Instead, a more efficient way is to transform vector $\mathbf{S_i}$ to the defining coordinate system of the polytope by the inverse of the rotation matrix of the polytope, and the search is performed in the defining coordinate system. After a supporting vertex is found, it is transformed to the world coordinate system. Thus only two coordinate transformations are required in our method for locating each supporting vertex.

3.4 Choosing the Initial Searching Direction

When the bounding boxes of two polytopes overlap for the first time, there are three cases in general (see Figure 3.4). In each of these cases, there are many ways to choose the initial searching direction $\mathbf{S_0}$ such as the normal vector of the plane formed by some contact points between the two bounding boxes as shown in Figure 3.4.

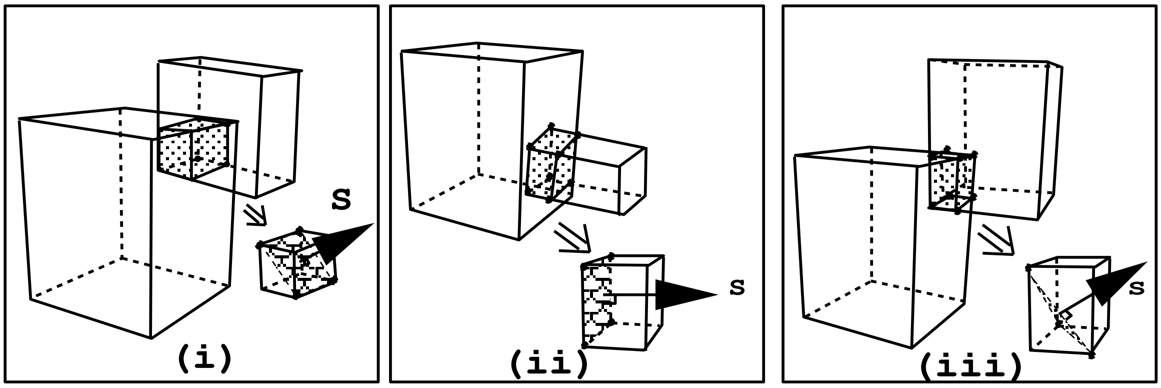

Figure 3.4: Three cases to find the initial separating vector.

For simplicity and efficiency, the line segment connecting the two polytopes' centers $\mathbf{S_0} = (\mathbf{q_c} - \mathbf{p_c})/||\mathbf{q_c} - \mathbf{p_c}||$ is chosen in our implementation, where $\mathbf{p_c}$ and $\mathbf{q_c}$ are the centroids of P and Q respectively. A centroid can be approximated by the average of all vertices of the polytope or by $(\sum_i A_i * C_i)/(\sum_i A_i)$ where C_i and A_i are the centroids and areas of face i respectively. This initial $\mathbf{S_0}$ is chosen because the separating vector is likely to be close to this direction.

3.5 Preprocessing

When bounding boxes of polytopes overlap for the first time, an arbitrary vertex can be used as an initial vertex for searching for supporting vertices of P and Q. For better efficiency, we pre-compute supporting vertices in a number of pre-defined directions and store them in a 2D table. In Figure 3.5, the center point in each region is used to compute an approximate supporting vertex in that region. Then, for any given direction $\mathbf{S_0}$, a supporting vertex in the table with the direction close to $\mathbf{S_0}$ is retrieved in constant time. This supporting vertex is used as the initial vertex to search for a supporting vertex with respect to $\mathbf{S_0}$. The larger is the size of this 2D table, the better approximation of this initial point, and the more quickly does this searching algorithm locate a supporting vertex. In our implementation, a table of size 8×16 is used.

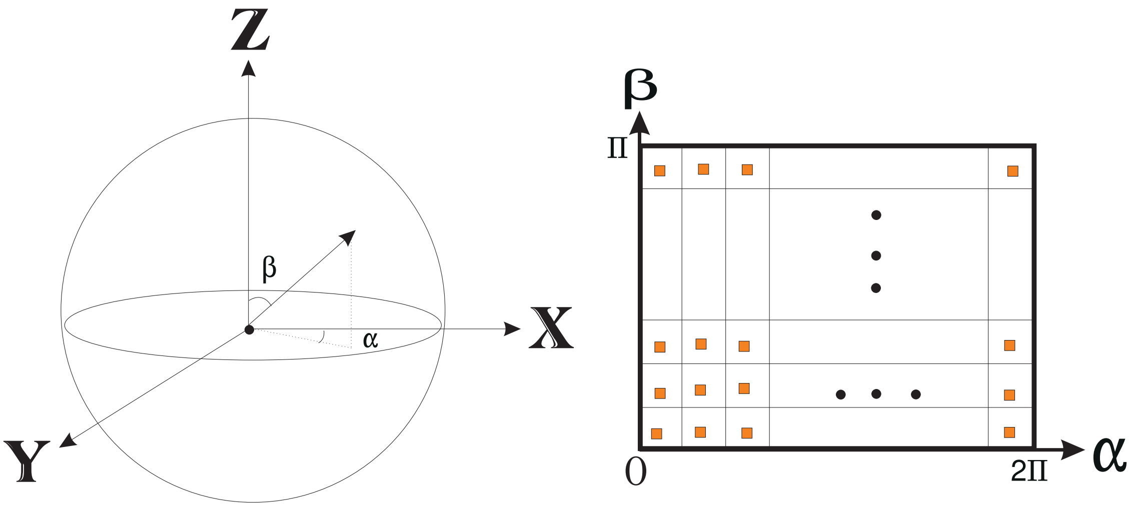

Figure 3.5: Precomputing a supporting vertex in various directions.

3.6 Caching

In each time frame, if the separating vector algorithm detects that two polytopes do not collide, the two supporting vertices and the separating vector found are cached. This separating vector is used as the initial vector $\mathbf{S_0}$ in the next time frame. If objects in the virtual environment do not move swiftly, this vector is likely to be the separating vector in the next time frame or as an initial vector it can help get a report on collision more quickly (collision report will be discussed in next Chapter). When objects do move swiftly, the algorithms still works though a bit more slowly. Similarly, the separating vertices found in the preceding time frame are used as initial points to search for the new separating vertices in the next time frame. In the subsequent iterations, the supporting vertex found in the last iteration is used as an initial vertex to search for a new supporting vertex. Thus, even though the previous iteration of separating vector searching algorithm may not find a separating vertex, the effort is not wasted since the supporting vertex found in the last iteration can help locate a separating vertex more quickly in subsequent iterations. To summarize, this caching mechanism reduces the search computations of the separating vector due to temporal and spatial coherences.

3.7 Properties of the Separating Vector Algorithm

In this section, we derive some important properties of the separating vector algorithm. We start with the convergence property which states that the vector $\mathbf{S}$ we compute using formula (3.2) is closer and closer to the separating vector in each searching step if one exists. Then two additional properties of this algorithm are derived, followed by a discussion on other possible ways to choose the next searching direction $\mathbf{S}$. In the next section, further extensions of the way to choose $\mathbf{S}$ are discussed.

Lemma 2: *If polytopes P and Q do not collide and $\mathbf{S_i} \cdot \mathbf{r_i} < 0$ in the i-th searching step, then for any separating vector $\mathbf{w}$ of P and Q,*

$$\mathbf{S_{i+1}} \cdot \mathbf{w} > \mathbf{S_i} \cdot \mathbf{w}, \quad i = 0, 1, \ldots. \tag{3.3}$$

Proof: By formula (3.2),

$$\mathbf{S_{i+1}} \cdot \mathbf{w} = \mathbf{S_i} \cdot \mathbf{w} - 2(\mathbf{r_i} \cdot \mathbf{S_i})(\mathbf{r_i} \cdot \mathbf{w}).$$

Since $\mathbf{r_i} \cdot \mathbf{w} > 0$ as $\mathbf{w}$ is a separating vector,

$$\mathbf{S_{i+1}} \cdot \mathbf{w} - \mathbf{S_i} \cdot \mathbf{w} = -2(\mathbf{r_i} \cdot \mathbf{S_i})(\mathbf{r_i} \cdot \mathbf{w}) > 0. \quad \square$$

Hence if the two polytopes do not collide and $\mathbf{S_i}$ is not a separating vector, then $\mathbf{S_{i+1}}$ given by formula (3.2) is closer to any separating vector $\mathbf{w}$ than $\mathbf{S_i}$ is. It is because by Lemma 2 the angle between $\mathbf{S_{i+1}}$ and $\mathbf{w}$ is smaller than the angle between $\mathbf{S_i}$ and $\mathbf{w}$.

Another property of the algorithm is that if the pair $\mathbf{p_i}$ and $\mathbf{q_i}$ appear in two consecutive steps, i.e. $\mathbf{r_{i+1}} = \mathbf{r_i}$, then P and Q do not collide, as indicated by the following lemma.

Lemma 3: *If $\mathbf{S_i} \cdot (\mathbf{q_i} - \mathbf{p_i}) < 0$ and $\mathbf{p_{i+1}} = \mathbf{p_i}$, $\mathbf{q_{i+1}} = \mathbf{q_i}$, then $\mathbf{S_{i+1}} \cdot \mathbf{r_{i+1}} > 0$, i.e. P and Q do not collide.*

Proof : Since

$$\begin{aligned}\mathbf{S_{i+1}} \cdot \mathbf{r_{i+1}} &= \mathbf{S_i} \cdot \mathbf{r_{i+1}} - 2(\mathbf{r_i} \cdot \mathbf{S_i})(\mathbf{r_i} \cdot \mathbf{r_{i+1}}) \\ &= \mathbf{S_i} \cdot \mathbf{r_i} - 2(\mathbf{r_i} \cdot \mathbf{S_i})(\mathbf{r_i} \cdot \mathbf{r_i}) = -\mathbf{S_i} \cdot \mathbf{r_i} > 0\end{aligned}$$

by Lemma 1, P and Q do not collide. $\Box$.

Besides, if we consider the Minkowski sum $Q-P$, the distance from the origin to the supporting vertex in each step is less than the distance between supporting vertices found in successive steps as shown in the following property.

Lemma 4: *Let* $\mathbf{m_i} = \mathbf{q_i} - \mathbf{p_i}, i = 0, 1, \ldots$ *If* $\mathbf{S_i} \cdot \mathbf{m_i} < 0$ *and* $\mathbf{S_{i+1}} \cdot \mathbf{m_{i+1}} < 0$*, then*

$$\|\mathbf{m_{i+1}} - \mathbf{m_i}\| > \|\mathbf{m_{i+1}}\| \tag{3.4}$$

Proof : Since $\mathbf{p_i}$ is the supporting vertex in the direction $\mathbf{S_i}$, $\mathbf{S_i} \cdot \mathbf{p_i} > \mathbf{S_i} \cdot \mathbf{p_{i+1}}$. Similarly $\mathbf{S_i} \cdot \mathbf{q_i} < \mathbf{S_i} \cdot \mathbf{q_{i+1}}$. Hence $\mathbf{S_i} \cdot \mathbf{m_i} < \mathbf{S_i} \cdot \mathbf{m_{i+1}}$.

$$\begin{aligned}\mathbf{S_{i+1}} \cdot \mathbf{m_{i+1}} < 0 &\Rightarrow (\mathbf{S_i} - \frac{2(\mathbf{S_i} \cdot \mathbf{m_i})}{\|\mathbf{m_i}\|^2}\mathbf{m_i}) \cdot \mathbf{m_{i+1}} < 0 \\ &\Rightarrow \mathbf{S_i} \cdot \mathbf{m_{i+1}} < 2\frac{(\mathbf{S_i} \cdot \mathbf{m_i})(\mathbf{m_{i+1}} \cdot \mathbf{m_i})}{\|\mathbf{m_i}\|^2} \\ &\Rightarrow \mathbf{S_i} \cdot \mathbf{m_i} < 2\frac{(\mathbf{S_i} \cdot \mathbf{m_i})(\mathbf{m_{i+1}} \cdot \mathbf{m_i})}{\|\mathbf{m_i}\|^2} \\ &\Rightarrow \|\mathbf{m_i}\|^2 > 2(\mathbf{m_{i+1}} \cdot \mathbf{m_i}) \\ &\Rightarrow \|\mathbf{m_{i+1}} - \mathbf{m_i}\|^2 > \|\mathbf{m_{i+1}}\|^2 \\ &\Rightarrow \|\mathbf{m_{i+1}} - \mathbf{m_i}\| > \|\mathbf{m_{i+1}}\| \quad \Box\end{aligned}$$

Besides choosing $\mathbf{S_{i+1}}$ as in formula (3.2), another alternative searching direction is

$$\mathbf{S_{i+1}} = \langle((\mathbf{q_i} - \mathbf{p_i}) \times \mathbf{S_i}) \times (\mathbf{q_i} - \mathbf{p_i})\rangle \tag{3.5}$$

where $\langle x \rangle$ denotes $x/\|x\|$.

This searching direction is perpendicular to $\mathbf{S_i}$ and $\mathbf{q_i} - \mathbf{p_i}$. It is another good choice for searching direction as indicated by the following Lemma.

Lemma 5 *If P and Q do not collide and $\mathbf{S_i} \cdot (\mathbf{q_i} - \mathbf{p_i}) < 0$ in the i-th searching step, then for any separating vector $\mathbf{w}$ of P and Q with $\mathbf{S_0} \cdot \mathbf{w} > 0$,*

$$\mathbf{S_{i+1}} \cdot \mathbf{w} > \mathbf{S_i} \cdot \mathbf{w} > 0, \quad i = 1, 2, \ldots \tag{3.6}$$

where $\mathbf{S_{i+1}}$ is given by formula (3.5).

Proof: Let $\mathbf{r_i} = \langle \mathbf{q_i} - \mathbf{p_i} \rangle$. Then we have $\mathbf{r_i} \cdot \mathbf{S_i} < 0$.

$$\begin{aligned}
\mathbf{S_{i+1}} \cdot \mathbf{w} &= \langle (\mathbf{r_i} \times \mathbf{S_i}) \times \mathbf{r_i} \rangle \cdot \mathbf{w} \\
&= \langle \mathbf{S_i} - (\mathbf{r_i} \cdot \mathbf{S_i}) \mathbf{r_i} \rangle \cdot \mathbf{w} \\
&= ((\mathbf{S_i} \cdot \mathbf{w}) - (\mathbf{r_i} \cdot \mathbf{S_i})(\mathbf{r_i} \cdot \mathbf{w}))/L_i
\end{aligned}$$

where $L_i^2 = ((\mathbf{r_i} \times \mathbf{S_i}) \times \mathbf{r_i}) \cdot ((\mathbf{r_i} \times \mathbf{S_i}) \times \mathbf{r_i}) = 1 - (\mathbf{S_i} \cdot \mathbf{r_i})^2$.

Because for any separating vector $\mathbf{w}$, $(\mathbf{q'} - \mathbf{p'}) \cdot \mathbf{w} > 0$ for any point $\mathbf{p'} \in P, \mathbf{q'} \in Q$, $\mathbf{r_i} \cdot \mathbf{w} > 0$. Since $\mathbf{r_i} \cdot \mathbf{S_i} < 0$,

$$L_i(\mathbf{S_{i+1}} \cdot \mathbf{w}) - \mathbf{S_i} \cdot \mathbf{w} = -(\mathbf{r_i} \cdot \mathbf{S_i})(\mathbf{r_i} \cdot \mathbf{w}) > 0$$

So

$$L_i(\mathbf{S_{i+1}} \cdot \mathbf{w}) > \mathbf{S_i} \cdot \mathbf{w}. \tag{3.7}$$

Since $0 \le L_i \le 1$ and $\mathbf{S_0} \cdot \mathbf{w} > 0$, we have, by induction, $\mathbf{S_{i+1}} \cdot \mathbf{w} > \mathbf{S_i} \cdot \mathbf{w} > 0$. $\square$

Note that by choosing $\mathbf{S_0} = \mathbf{q_c} - \mathbf{p_c}$, where $\mathbf{p_c}$ and $\mathbf{q_c}$ are centers of P and Q respectively, the condition $\mathbf{S_0} \cdot \mathbf{w} > 0$ holds. Hence this searching direction also guarantees that $\mathbf{S_{i+1}}$ is closer to any separating vector $\mathbf{w}$ than $\mathbf{S_i}$ is. Besides, if we consider the Minkowski sum $Q - P$ and let $\mathbf{m_i} = \mathbf{q_i} - \mathbf{p_i}$, we have that the length of projection from $\mathbf{m_{i+1}}$ to $\mathbf{m_i}$ is less than the length of $\mathbf{m_i}$ as shown below :

Lemma 6: *If $\mathbf{S_i} \cdot \mathbf{m_i} < 0$ and $\mathbf{S_{i+1}} \cdot \mathbf{m_{i+1}} < 0$, then*

$$\|\mathbf{m_i}\|^2 \ge \mathbf{m_i} \cdot \mathbf{m_{i+1}}\ , \ i = 1, 2, \ldots \tag{3.8}$$

Proof : Since $\mathbf{p_i}$ is the supporting vertex in the direction $\mathbf{S_i}$, $\mathbf{p_i} \cdot \mathbf{S_i} > \mathbf{p_{i+1}} \cdot \mathbf{S_i}$. Similarly $\mathbf{q_i} \cdot \mathbf{S_i} < \mathbf{q_{i+1}} \cdot \mathbf{S_i}$. Hence $(\mathbf{q_i} - \mathbf{p_i}) \cdot \mathbf{S_i} < (\mathbf{q_{i+1}} - \mathbf{p_{i+1}}) \cdot \mathbf{S_i}$. Using this inequality, we have

$$
\begin{aligned}
&\mathbf{S_{i+1}} \cdot \mathbf{m_{i+1}} \leq 0 \\
&\Rightarrow (\|\mathbf{m_i}\|^2 \mathbf{S_i} - (\mathbf{m_i} \cdot \mathbf{S_i})\mathbf{m_i}) \cdot \mathbf{m_{i+1}} \leq 0 \\
&\Rightarrow \|\mathbf{m_i}\|^2 (\mathbf{S_i} \cdot \mathbf{m_{i+1}}) \leq (\mathbf{m_i} \cdot \mathbf{m_{i+1}})(\mathbf{S_i} \cdot \mathbf{m_i}) \\
&\Rightarrow \|\mathbf{m_i}\|^2 (\mathbf{S_i} \cdot \mathbf{m_i}) \leq (\mathbf{m_i} \cdot \mathbf{m_{i+1}})(\mathbf{S_i} \cdot \mathbf{m_i}) \\
&\Rightarrow \|\mathbf{m_i}\|^2 \geq \mathbf{m_i} \cdot \mathbf{m_{i+1}} \quad \square
\end{aligned}
$$

This choice of $\mathbf{S_{i+1}}$ by formula (3.5) is more conservative than the vector $\mathbf{S_{i+1}}$ given by formula (3.2) since the latter is more greedy in finding a separating vector. It is greedy in the sense that the deviation of $\mathbf{S_{i+1}}$ from $\mathbf{S_i}$ is bigger using formula (3.2). The following example shows how the two strategies differ when P is a long thin strip and Q is a single point in 2D space.

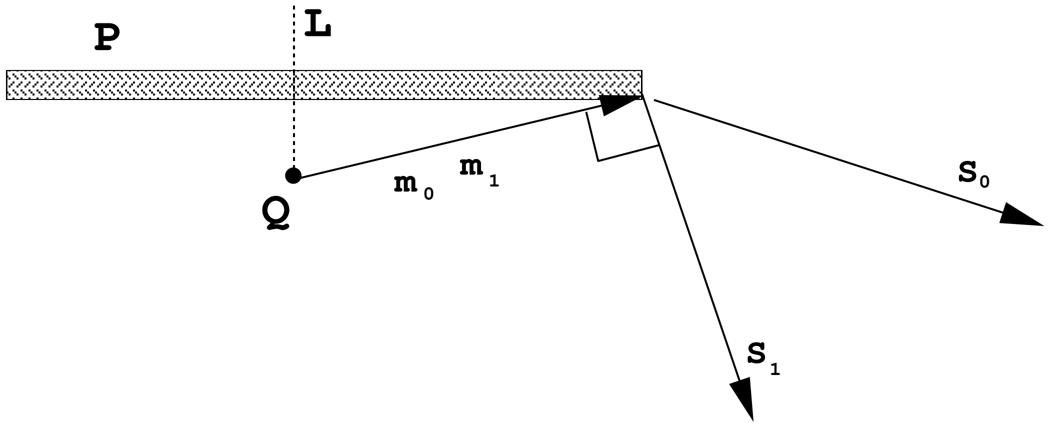

Figure 3.6: The conservative searching directions by formula (3.5).

Using formula (3.5) the separating vector can be found in two steps but using formula (3.2) more than two steps are needed. (In contrast, more steps are needed using formula (3.5) than using formula (3.2) when P and Q are non-overlapping circles.) Besides, the sequence of supporting vertices found using formula (3.2) will

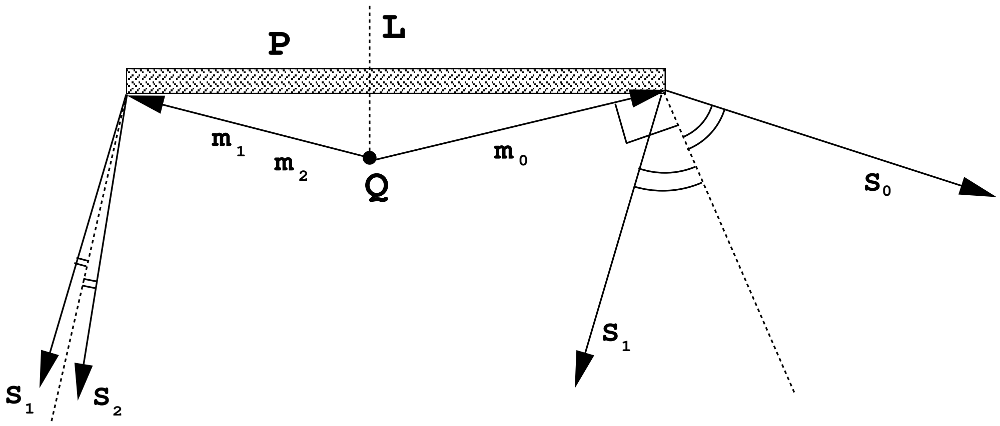

Figure 3.7: The greedy searching directions by formula (3.2).

pass across the line L, which connects the closest pair of points between P and Q. In general, the sequence of supporting vertices found by formula (3.5) will converge to the closest point monotonously along one side but the sequence of supporting vertices found by formula (3.2) may jump around the closest point.

In practice, as most objects in virtual environments are of ellipsoid shape, we find that using formula (3.2) is a better choice on average. Besides, formula (3.2) is more efficient to compute. Hence it is used in our implementation to compute $\mathbf{S_{i+1}}$.

3.8 Possible Extensions

Let $\mathbf{S_i^G}$ be the searching direction chosen using the greedy approach as in formula (3.2) and $\mathbf{S_i^C}$ be the searching direction chosen using the conservative approach as in formula (3.5) in the ith step. Then any searching direction lying between $\mathbf{S_i^G}$ and $\mathbf{S_i^C}$, denoted by $\mathbf{S_i^T}$, satisfies the convergence property given by Equation (3.3). The vector $\mathbf{S_i^T}$ is defined by

$$\begin{cases} \mathbf{S_0^T} \cdot \mathbf{w} > 0 \ , \quad \text{for any separating vector } \mathbf{w} \\ \mathbf{S_{i+1}^T} = \langle \alpha_{i+1}\mathbf{S_{i+1}^G} + (1-\alpha_{i+1})\mathbf{S_{i+1}^C} \rangle, \ \ 0 \le \alpha_{i+1} \le 1 \ , \ i = 0, 1, \ldots \quad , \ \textit{where} \\ \mathbf{S_{i+1}^G} = \mathbf{S_i^T} - 2(\mathbf{r_i} \cdot \mathbf{S_i^T})\mathbf{r_i}, \quad i = 0, 1, \ldots \\ \mathbf{S_{i+1}^C} = \langle (\mathbf{r_i} \times \mathbf{S_i^T}) \times \mathbf{r_i} \rangle \ , \quad i = 0, 1, \ldots \end{cases} \tag{3.9}$$

Note that the $\alpha_\mathbf{i}$ are arbitrary numbers in $[0,1]$.

Lemma 7: *If polytopes P and Q do not collide and $\mathbf{S_i^T} \cdot \mathbf{r_i} < 0$, then for any separating vector $\mathbf{w}$ with $\mathbf{S_0^T} \cdot \mathbf{w} > 0$,*

$$\mathbf{S_{i+1}^T} \cdot \mathbf{w} > \mathbf{S_i^T} \cdot \mathbf{w} \ , \ i = 0, 1, \ldots$$

Proof : For $i = 1, 2, \ldots,$

$$\|\alpha_i \mathbf{S_i^G} + (1-\alpha_i)\mathbf{S_i^C}\|^2 = \alpha_i^2 + (1-\alpha_i)^2 + 2\alpha_i(1-\alpha_i)\mathbf{S_i^G} \cdot \mathbf{S_i^C}$$

Now, as $\mathbf{r_{i-1}} \cdot \mathbf{S_{i-1}^T} < 0$

$$\begin{aligned}
\mathbf{S_i^G} \cdot \mathbf{S_i^C} &= (\mathbf{S_{i-1}^T} - 2(\mathbf{r_{i-1}} \cdot \mathbf{S_{i-1}^T})\mathbf{r_{i-1}}) \cdot \langle(\mathbf{r_{i-1}} \times \mathbf{S_{i-1}^T}) \times \mathbf{r_{i-1}}\rangle \\
&= (\mathbf{S_{i-1}^T} - 2(\mathbf{r_{i-1}} \cdot \mathbf{S_{i-1}^T})\mathbf{r_{i-1}}) \cdot \frac{\mathbf{S_{i-1}^T} - (\mathbf{r_{i-1}} \cdot \mathbf{S_{i-1}^T})\mathbf{r_{i-1}}}{\sqrt{1 - (\mathbf{r_{i-1}} \cdot \mathbf{S_{i-1}^T})^2}} \\
&= \frac{1 - (\mathbf{r_{i-1}} \cdot \mathbf{S_{i-1}^T})^2 - 2(\mathbf{r_{i-1}} \cdot \mathbf{S_{i-1}^T})^2 + 2(\mathbf{r_{i-1}} \cdot \mathbf{S_{i-1}^T})^2}{\sqrt{1 - (\mathbf{r_{i-1}} \cdot \mathbf{S_{i-1}^T})^2}} \\
&= \sqrt{1 - (\mathbf{r_{i-1}} \cdot \mathbf{S_{i-1}^T})^2} \\
&< 1
\end{aligned}$$

Hence

$$\begin{aligned}
\|\alpha_i \mathbf{S_i^G} + (1-\alpha_i)\mathbf{S_i^C}\|^2 &= \alpha_i^2 + (1-\alpha_i)^2 + 2\alpha_i(1-\alpha_i)\mathbf{S_i^G} \cdot \mathbf{S_i^C} \\
&< \alpha_i^2 + (1-\alpha_i)^2 + 2\alpha_i(1-\alpha_i) \\
&< 1
\end{aligned}$$

So $\|\alpha_i \mathbf{S_i^G} + (1-\alpha_i)\mathbf{S_i^C})\| < 1 \ \ , \quad i = 1, 2, \ldots$

When $i = 0$, since by Lemma 2, $\mathbf{S_1^G} \cdot \mathbf{w} > \mathbf{S_0^T} \cdot \mathbf{w} > 0$ and by Lemma 5, $\mathbf{S_1^C} \cdot \mathbf{w} > \mathbf{S_0^T} \cdot \mathbf{w} > 0$.

$$\begin{aligned}
\mathbf{S_1^T} \cdot \mathbf{w} &= \frac{\alpha_1(\mathbf{S_1^G} \cdot \mathbf{w}) + (1-\alpha_1)(\mathbf{S_1^C} \cdot \mathbf{w})}{\|\alpha_1 \mathbf{S_1^G} + (1-\alpha_1)\mathbf{S_1^C}\|} \\
&> \frac{\alpha_1(\mathbf{S_0^T} \cdot \mathbf{w}) + (1-\alpha_1)(\mathbf{S_0^T} \cdot \mathbf{w})}{\|\alpha_1 \mathbf{S_1^G} + (1-\alpha_1)\mathbf{S_1^C}\|} \\
&= \frac{\mathbf{S_0^T} \cdot \mathbf{w}}{\|\alpha_1 \mathbf{S_1^G} + (1-\alpha_1)\mathbf{S_1^C}\|} \\
&> \mathbf{S_0^T} \cdot \mathbf{w} \\
&> 0
\end{aligned}$$

Assume $\mathbf{S_i^T} \cdot \mathbf{w} > \mathbf{S_{i-1}^T} \cdot \mathbf{w} > 0$ is true for all $i \leq k$.
When $i = k+1$, since by Lemma 2, $\mathbf{S_{k+1}^G} \cdot \mathbf{w} > \mathbf{S_k^T} \cdot \mathbf{w} > 0$ and by Lemma 5, $\mathbf{S_{k+1}^C} \cdot \mathbf{w} > \mathbf{S_k^T} \cdot \mathbf{w} > 0$.

$$\begin{aligned}
\mathbf{S_{k+1}^T} \cdot \mathbf{w} &= \frac{\alpha_{k+1}(\mathbf{S_{k+1}^G} \cdot \mathbf{w}) + (1-\alpha_{k+1})(\mathbf{S_{k+1}^C} \cdot \mathbf{w})}{\|\alpha_{k+1}\mathbf{S_{k+1}^G} + (1-\alpha_{k+1})\mathbf{S_{k+1}^C}\|} \\
&> \frac{\alpha_{k+1}(\mathbf{S_k^T} \cdot \mathbf{w}) + (1-\alpha_{k+1})(\mathbf{S_k^T} \cdot \mathbf{w})}{\|\alpha_{k+1}\mathbf{S_{k+1}^G} + (1-\alpha_{k+1})\mathbf{S_{k+1}^C}\|} \\
&= \frac{\mathbf{S_k^T} \cdot \mathbf{w}}{\|\alpha_{k+1}\mathbf{S_{k+1}^G} + (1-\alpha_{k+1})\mathbf{S_{k+1}^C}\|} \\
&> \mathbf{S_k^T} \cdot \mathbf{w} \\
&> 0
\end{aligned}$$

By induction, $\mathbf{S_{i+1}^T} \cdot \mathbf{w} > \mathbf{S_i^T} \cdot \mathbf{w}$ is true for all $i \geq 0$. $\square$

Hence we have a family of searching directions $\mathbf{S_i^T}$ to choose from when α_i varies. It is still not known how to determine α_i adaptively so that the number of searching step is minimized. Our experiments show that in most cases the vector given by formula (3.2) yields slightly faster running time than the vector given by formula (3.5) does. For simplicity we always choose $\alpha_i = 1$ for all i.

Chapter 4

Collision Detection

Obviously, a termination condition is needed in the above separating vector searching step, for otherwise the search might run without stopping when there is a collision. This chapter extends the separating vector algorithm so that it not only eliminates non-interference polytopes but performs exact collision detection between polytopes. With the addition of a termination condition, our separating vector algorithm guarantees to find a separating vector in a finite number of steps if one exists, otherwise the algorithm will report collision.

4.1 The Collision Condition

It is shown in Section 2.3 that two polytopes P and Q collide if and only if the origin $O \in M$ where $M = Q - P$. So, as M is convex, if the origin is outside M there exists $\mathbf{w}$ such that

$$\mathbf{m} \cdot \mathbf{w} \geq 0 \quad \text{for all} \quad \mathbf{m} \in M \tag{4.1}$$

and this $\mathbf{w}$ is a separating vector. Conversely, $\mathbf{w}$ does not exist if there is a collision. The existence of $\mathbf{w}$ is related to the problem of determining whether there is a $\mathbf{w}$ such that $\mathbf{w} \cdot \mathbf{r_i} \geq 0$ for all i where $\mathbf{r_i} = \mathbf{m_i}/||\mathbf{m_i}|| = (\mathbf{q_i} - \mathbf{p_i})/||\mathbf{q_i} - \mathbf{p_i}||$. Instead of solving this problem, we simplify it by exploring the geometrical property behind the problem.

Geometrically, Equation (4.1) implies that there exists a separating plane H passing through the origin such that all the points $\mathbf{r_0}, \ldots, \mathbf{r_k}$ on the unit sphere lie on one side of the plane H (see Figure 4.1(i)). Therefore, if it is not possible to find such a plane H, then the two polytopes collide.

4.2 Existence of a Separating Plane

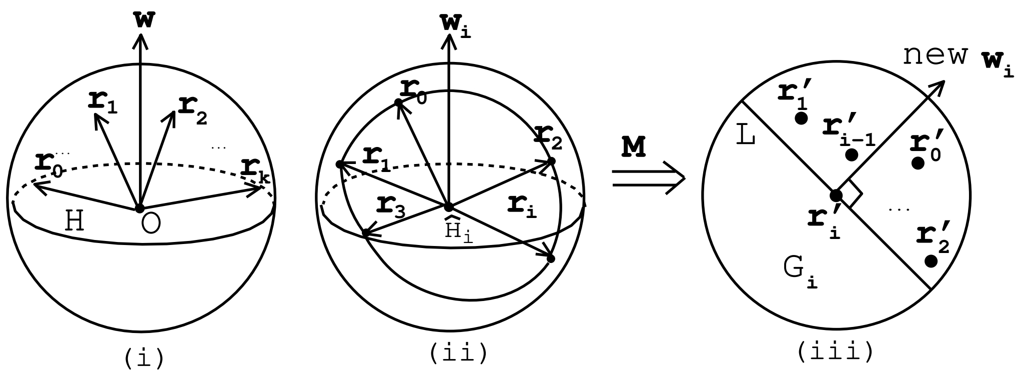

Figure 4.1: Determining the existence of a separating vector.

When a point $\mathbf{r_i}$ is added, an incremental algorithm is used to find a plane H_i with normal vector $\mathbf{w_i}$ such that $\mathbf{r_0}, \ldots, \mathbf{r_i}$ all lie on the positive half space of H_i.

We propose two methods to determine if the plane H_i exists. The first one is called the *hemisphere method* which reduces the problem into determining if all points are lie on a hemisphere. The second one is called the *half-plane intersection method* which projects the hemisphere associated with the normal $\mathbf{r_i}$ into a halfplane and computes the intersection of all such halfplanes.

4.2.1 Hemisphere Method

Initially, $\mathbf{w_1}$ is chosen to be the bisector of $\mathbf{r_0}$ and $\mathbf{r_1}$. At the i-th iteration, if $\mathbf{r_i} \cdot \mathbf{w_{i-1}} > 0$, then $\mathbf{r_i}$ also lies on the positive half space of H_{i-1}, so we set $\mathbf{w_i} = \mathbf{w_{i-1}}$;

otherwise, $\mathbf{r_i}$ must be one of the boundary points on the convex hull (a spherical polygon) formed by $\mathbf{r_0}$, ..., $\mathbf{r_i}$ on the surface of the sphere (see Figure 4.1(ii)). If there exists a plane $\hat{H}_i$ passing through the origin such that all the above points lie on one side of $\hat{H}_i$, then we can always rotate $\hat{H}_i$ into a plane $\tilde{H}_i$ such that $\tilde{H}_i$ touches $\mathbf{r_i}$ and all the points $\mathbf{r_j}$, $j = 0, 1, \ldots, i$, are on one side of $\tilde{H}_i$.

Let G_i be the plane passing through the origin with normal vector $\mathbf{r_i}$. Then project $\mathbf{r_0}, \ldots, \mathbf{r_{i-1}}$ along the vector $\mathbf{r_i}$ into points $\mathbf{r'_0}, \ldots, \mathbf{r'_{i-1}}$ on the plane G_i where vectors parallel to $\mathbf{r_i}$ are ignored ($\mathbf{r_i}$ will be projected to the center). If there exists a line L on the plane G_i that passes through the origin such that all the points $\mathbf{r'_0}, \ldots, \mathbf{r'_{i-1}}$ on G_i lie on one side of the line L, then $\mathbf{w_i}$, which is the normal vector of $\tilde{H}_i$, is taken to be the vector on the plane G_i perpendicular to L (see Figure 4.1(iii)). Conversely, if such a line L does not exist, then there does not exist a vector $\mathbf{w_i}$ such that Equation (4.1) is satisfied; that is, the two polytopes collide.

The existence of the line L can be determined in $O(i)$ time as follows. Let M be the rotation matrix that transforms $\mathbf{r_i}$ to the z axis (note that M is not unique). To be specific, denoting $\mathbf{r_i} = (x, y, z)$, one may choose

$$M = \begin{pmatrix} xz/d & yz/d & -(x^2+y^2)/d \\ -y/d & x/d & 0 \\ x & y & z \end{pmatrix}, \text{ where } d = \sqrt{x^2+y^2} \tag{4.2}$$

Then the projection of $\mathbf{r_j}$, $j = 0, \ldots, i-1$, along $\mathbf{r_i}$ are determined by dropping the z value of point $M * \mathbf{r_j}$. The algorithm is as follows (see Figure 4.1(iv)):

If $\mathbf{r_i} \cdot \mathbf{w} > 0$, return non-collision.
Let $\mathbf{r_a} = \mathbf{r'_0}$ and $\mathbf{r_b} = \mathbf{r'_1}$ such that $\mathbf{r_b}$ is in clockwise direction with respect to $\mathbf{r_a}$.
For each $\mathbf{r'_j}$, $j = 2, \ldots, i-1$
  Substitute $\mathbf{r'_j}$ in the equations of line $O\mathbf{r_a}$. Let the sign of the result be $Sign_a$.
  Substitute $\mathbf{r'_j}$ in the equations of line $O\mathbf{r_b}$. Let the sign of the result be $Sign_b$.
  (i) If $Sign_a \geq 0$ and $Sign_b \leq 0$, continue.
  (ii) If $Sign_a \geq 0$ and $Sign_b > 0$, $\mathbf{r_b} = \mathbf{r'_j}$.
  (ii) If $Sign_a < 0$ and $Sign_b \leq 0$, $\mathbf{r_a} = \mathbf{r'_j}$.

(ii) else return collision.

Update $\mathbf{w} = M^{-1}(< \mathbf{r_a} + \mathbf{r_b} >)$

Return non-collision

4.2.2 Half-Plane Intersection Method

The existence of a separating vector can also be determined in another way. Without loss of generality assume $\mathbf{r_0} = (0, 0, -1)$. Let G denote the plane $z = -1$.

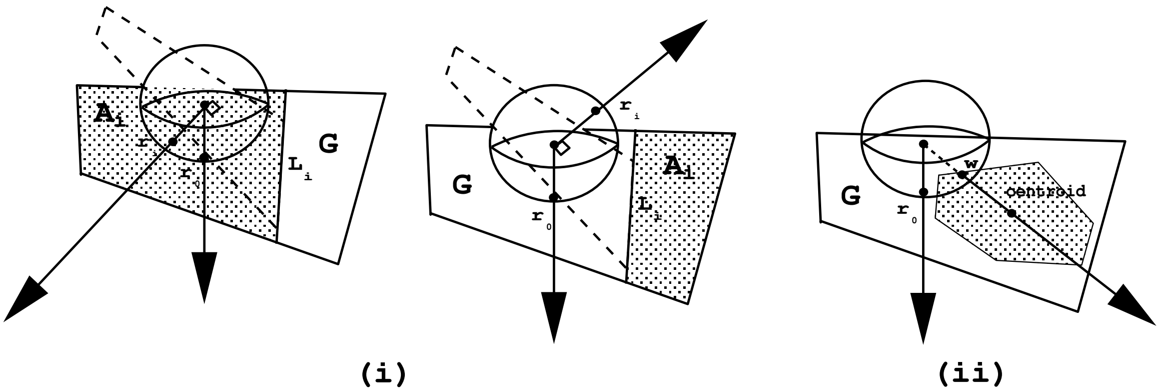

Figure 4.2: (i) Two cases of the region A_i on G (ii) Choosing the new $\mathbf{w}$.

A plane through the origin with normal $\mathbf{r_i}$ intersects the plane G in a line (see Figure 4.2(i)), which cuts the plane G into two half planes. Let A_i denote the half plane satisfying $x_iX + y_iY > -z_i$ where $\mathbf{r_i} = (x_i, y_i, z_i)$. Then the existence of a plane H through the origin such that all the $\mathbf{r_i}$ lie on one side of H is equivalent to that $\bigcap_{j=0}^{k} A_j \neq \emptyset$. Note that $\bigcap_{j=0}^{i} A_j$, $i = 0, 1 \ldots, k$, are convex polygons. Although the intersection of a half plane and a convex polygon with i vertices can be found in $O(\log i)$ time [19], the reconstruction of the polygon $\bigcap_{j=0}^{i} A_j$ may take $O(i)$ time in each step where $i \leq k$. Afterwards, we can set $\mathbf{w_i}$ as a vector connecting the origin to any point inside $\bigcap_{j=0}^{i} A_j$, provided that it is non-empty.

This method has the advantage that it reduces the problem from 3D space to 2D plane directly. Besides, this method can give a better approximation of $\mathbf{w_i}$ in each step if it is chosen as in Figure 4.2(ii). This $\mathbf{w_i}$ can help the separating vector algorithm terminate faster as discussed in the next section. However, it is more complicated to implement and the overhead is larger. Since in a virtual environment, the separating vector algorithm terminates in less than four steps in most cases, our implementation uses the first method which has a smaller overhead.

4.3 Termination

In the above searching process, by Lemma 3, if $\mathbf{m_i} = \mathbf{q_i} - \mathbf{p_i} \in Q - P$ repeats itself in two consecutive steps, P and Q do not collide. However, if $\mathbf{m_i}$ reoccurs after more than one step before Equation (3.1) is satisfied, we cannot conclude that P and Q do not collide. In this case, in order to prevent the algorithm from running without stop, we set $\mathbf{S_{i+1}} = \mathbf{w_i}$ which is found in the above section. Then the vector $\mathbf{m_{i+1}} = \mathbf{q_{i+1}} - \mathbf{p_{i+1}}$ thus found with $\mathbf{S_{i+1}} = \mathbf{w_i}$ has the following property.

Lemma 8: *If* $\mathbf{m_{i+1}} = \mathbf{m_j}$ *for some* j, $0 \leq j \leq i$, *then* $\mathbf{S_{i+1}} = \mathbf{w_i}$ *is a separating vector of* P *and* Q, *that is,* P *and* Q *do not collide.*

Proof : Since

$$\mathbf{S_{i+1}} \cdot \mathbf{m_{i+1}} = \mathbf{w_i} \cdot \mathbf{m_j} \geq 0,$$

by Lemma 1, P and Q do not collide. □

Lemma 8 implies that either the algorithm stops with $\mathbf{S_{i+1}}$ being a separating vector or $\mathbf{m_{i+1}}$ is a new vertex of M that has not been visited before. This guarantees that the total number of vertex pairs repeated during the search is at most the number of vertices of M. So the algorithm will terminate in a finite number of steps.

To summarize, the vector $\mathbf{S_{i+1}}$ is either generated from $\mathbf{S_i}$ by formula (3.2) or set to be $\mathbf{w_i}$ when there is a reoccurrence of $\mathbf{m_i} = \mathbf{q_i} - \mathbf{p_i}$. For a sequence of vectors $\{\mathbf{S_i}\}$ thus defined, when $\mathbf{S_i} \cdot \mathbf{r_i} \geq 0$ for some i for the first time, we can conclude that $\mathbf{S_i}$ is a separating vector, and the polytopes P and Q do not collide. The polytopes P and Q collide if there does not exist $\mathbf{w_i}$ such that $\mathbf{w_i} \cdot \mathbf{r_j} \geq 0$, $j = 0, 1, \ldots, i$ for some i. Note that when $\mathbf{m_i}$ reoccurs in two consecutive steps, P and Q do not collide by Lemma 3. Moreover Lemma 3 and Lemma 8 are necessary because of numerical errors in implementation. For simplicity, they do not appear in the pseudo code of the separating vector algorithm in the appendix of this thesis.

4.4 Complexity

As pointed out earlier, the dynamic version of our algorithm makes use of time coherence between successive frames so to run in expected constant time empirically. Let the number of vertices in P and Q be n and m respectively. Without making use of coherence, our implementation uses $O(n+m)$ time to search for new supporting vertices in P and Q, and $O(i)$ time to detect collision at the i-th iteration. So the worst time complexity is $O((n+m+k)*k)$, where k is the total number of iterations performed.

The time complexity for searching for a new supporting vertex in P can be reduced to $O(\log n)$ using the hierarchical representation of polytopes which will be discussed in Section 5.3. Besides, it takes constant time to check whether a pair of supporting vertices has been visited previously in the algorithm. The method is to keep track of a 2D array with each entry being a timestamp for a pair of vertices. Initially all the entries are reset to zero. There is also a global variable called *counter*, which is incremented every time the collision detection algorithm is called. During the search for a separating vector, if the timestamp for a pair of supporting vertices is not equal to the counter, that timestamp is set to the counter; if it is equal to the counter, the pair has been visited before. When the maximum limit for the counter is reached, which is the maximum long integer of the language used, all the timestamps are reset to zero.

Hence the worst case running time of the separating vector algorithm can be reduced to $O((\log n + \log m + k) * k)$. So far the only upper-bound known to us for k is $O(mn)$. However, with temporal coherence being exploited in a virtual environment, it is found empirically that k is very small even for very large n. For a pair of ellipsoid-shaped polytopes, the maximum value of k is less than 25 when $n < 1000$. In practice, it is observed that the empirical running time of this algorithm in a dynamic environment is almost constant.

Chapter 5

Finding the Closest Pair of Points

When there is a collision, the information of the closest pair of points is useful in computing the response. In this chapter, we will further extend the separating vector algorithm so that it can compute the closest pair of points when polytopes collide. We will first review the idea of Gilbert's algorithm, which computes the closest pair of points between two polytopes in expected $O(n)$ time. Then we will review Johnson's algorithm which is the underlying layer of Gilbert's algorithm to compute the closest pair of points between two simplices. Afterwards, we will describe how to improve the theoretical running time of Gilbert's algorithm from expected $O(n)$ time to expected $O(\log n)$ time using the hierarchical representation of polytopes. Besides, we will describe how this complexity can be further improved when temporal coherence is exploited and how to modify the termination condition of Gilbert's algorithm so that it is more robust. Finally, we will integrate the improved Gilbert's algorithm into our separating vector algorithm so that the separating vector algorithm is capable of reporting contact points.

5.1 Review of Gilbert's Algorithm

The idea of Gilbert's algorithm is as follows. Initially a set S with ≤ 4 points is chosen arbitrarily from the vertex set of M, where $M = Q - P$. Then the closest

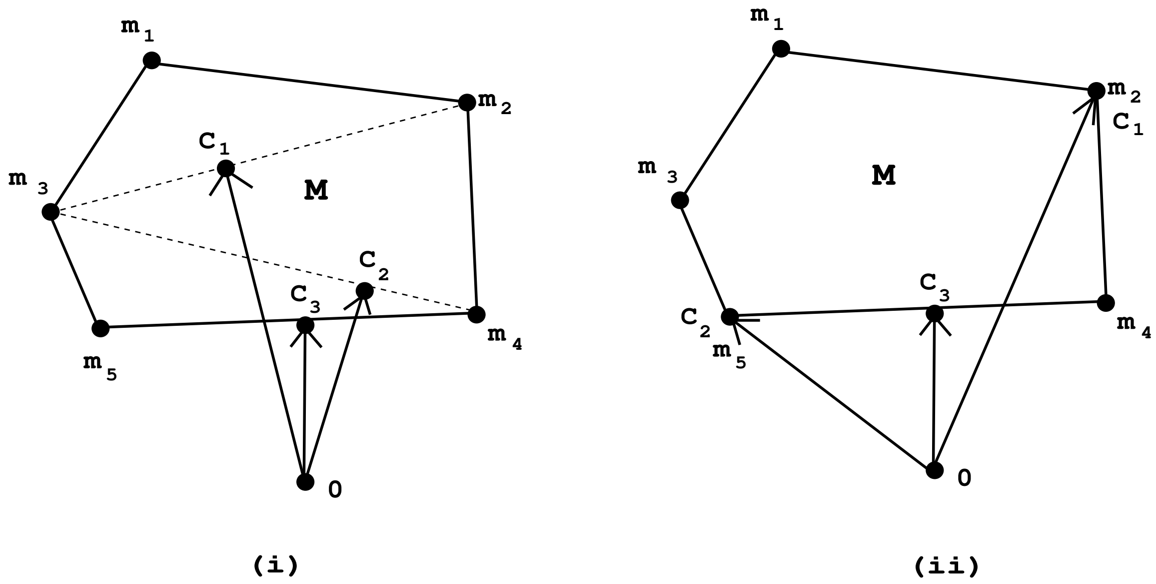

Figure 5.1: A 2D example of how Gilbert's algorithm works.

point $\mathbf{C_m}$ from the convex hull of S (which is a simplex) to the origin is computed using constant time [18]. This algorithm is described in the next section. The closest point $\mathbf{C_m}$ computed can be expressed as :

$$\mathbf{C_m} = \sum_{i=1}^{r} \lambda_i \mathbf{m_i} \quad \text{where} \ \sum_{i=1}^{r} \lambda_i = 1, \quad \lambda_i > 0, \quad \mathbf{m_i} \in S \tag{5.1}$$

Let V be the convex hull of vertices $\mathbf{m_i}$ in the above equation. Then $V \subset S$. When there is no collision, the number of vertices in V must be ≤ 3.

After $\mathbf{C_m}$ is computed, a supporting vertex $\mathbf{m}$ of M in the direction $-\mathbf{C_m}$ is found. This is done by finding the supporting vertices $\mathbf{p}$ and $\mathbf{q}$ of P and Q in the direction $\mathbf{C_m}$ and $-\mathbf{C_m}$, respectively, and setting $\mathbf{m} = \mathbf{q} - \mathbf{p}$. Note that this step takes linear time in Gilbert's algorithm because it simply evaluates the inner product between all the vertices and the supporting direction, then chooses the maximum one. If the condition $\mathbf{m} \cdot \mathbf{m} = \mathbf{m} \cdot \mathbf{C_m}$ is satisfied, the closest point is given by Equation (5.1) and the algorithm stops. The closest pair of points are given by

$$\begin{aligned} \mathbf{C_p} &= \sum_{i=1}^{r} \lambda_i \mathbf{p_i} \ , \ \mathbf{p_i} \in P \\ \mathbf{C_q} &= \sum_{i=1}^{r} \lambda_i \mathbf{q_i} \ , \ \mathbf{q_i} \in Q \text{ , where } \mathbf{m_i} = \mathbf{q_i} - \mathbf{p_i} \end{aligned}$$

If $\mathbf{m} \cdot \mathbf{m} = \mathbf{m} \cdot \mathbf{C_m}$ does not hold, the algorithm sets $S = V \cup \{\mathbf{m}\}$ and repeats. From the above equation, we see that not only we can find the closest pair of points but also the closest pair of features between them. For instance, if there are two distinct $\mathbf{p_i}$ and three distinct $\mathbf{q_i}$ in V, then the closest pair of features is an edge in P and a face in Q. Gilbert observed empirically that the algorithm will terminate within a constant number of iterations [18]. In other words, the algorithm runs in expected linear time.

The complete algorithm is as follows:

1. Set $S = \{\mathbf{m_1}, \mathbf{m_2}, \ldots, \mathbf{m_r}\}$, $r \leq 4$.
2. Compute the closest point $\mathbf{C_m}$ and the convex hull, V, of $\mathbf{C_m}$.
3. Compute the contact function $\mathbf{m} = C_M(-\mathbf{C_m}) = C_Q(-\mathbf{C_m}) - C_P(\mathbf{C_m})$.
4. If $\mathbf{m} \cdot \mathbf{m} = \mathbf{m} \cdot \mathbf{C_m}$, return $\mathbf{C_m}$.
5. Otherwise, set $S = V \cup \mathbf{m}$ and goto step 2.

As an example of how the algorithm runs, see Figure 5.1(i). If we set $S = \{\mathbf{m_1}, \mathbf{m_2}, \mathbf{m_3}\}$ initially, then after one iteration we have $V = \{\mathbf{m_2}, \mathbf{m_3}\}$, $\mathbf{m} = \mathbf{m_4}$ and $S = \{\mathbf{m_2}, \mathbf{m_3}, \mathbf{m_4}\}$. After two iterations we have $V = \{\mathbf{m_3}, \mathbf{m_4}\}$, $\mathbf{m} = \mathbf{m_5}$ and $S = \{\mathbf{m_3}, \mathbf{m_4}, \mathbf{m_5}\}$. Finally, we have $V = \{\mathbf{m_4}, \mathbf{m_5}\}$ and the closest pair of points are given by Equation (5.1).

If we set $S = \{\mathbf{m_2}\}$ initially (see Figure 5.1(ii)), then we have $V = \{\mathbf{m_2}\}$, $\mathbf{m} = \mathbf{m_5}$ and $S = \{\mathbf{m_2}, \mathbf{m_5}\}$ after one iteration. After two iterations we have $V = \{\mathbf{m_5}\}$, $\mathbf{m} = \mathbf{m_4}$ and $S = \{\mathbf{m_5}, \mathbf{m_4}\}$. Finally we have $V = \{\mathbf{m_4}, \mathbf{m_5}\}$ and the algorithm stops.

5.2 Review of Johnson's Algorithm

For completeness, we describe the algorithm that computes the closest point from the origin to an elementary polytope P where $P = \{\mathbf{p_1}, \mathbf{p_2}, \ldots, \mathbf{p_r}\}$, $\mathbf{p_i} \in E^n$, $1 \leq r \leq n + 1$. This algorithm is the underlying layer that Gilbert's algorithm relies on. The complete algorithm can be found in [51]. The algorithm is specially designed to be efficient when the number of points in P is small and when the set P is affinely independent. When the algorithm is terminated, the solution is expressed as

$$\mathbf{p}_{closest} = \sum_{i \in I_s} \lambda_i \mathbf{p_i} \tag{5.2}$$

where

$$\lambda_i > 0, \quad \sum_{i \in I_s} \lambda_i = 1, \quad i \in I_s \subset \{1, \ldots, r\}$$

and the set $P_s = \{\mathbf{p_i} : i \in I_s\} \subset P$ is affinely independent.

Since r is small, it is effective to take a combinatoric approach where all subsets of P are successively tested until a representation of the above form is found. Define real numbers $\Delta_i(P_s), i \in I_s$ and $\Delta(P_s)$ by

$$\Delta_i(\{\mathbf{p_i}\}) = 1, \ i \in \{1, \ldots, r\} \tag{5.3}$$

$$\Delta_j(P_s \cup \{\mathbf{p_j}\}) = \sum_{i \in I_s} \Delta_i(P_s)\mathbf{p_i} \cdot (\mathbf{p_k} - \mathbf{p_j}), \ j \in I_s^C, \ k = \min\{i | i \in I_s\}. \tag{5.4}$$

$$\Delta(P_s) = \sum_{i \in I_s} \Delta_i(P_s) \tag{5.5}$$

where I_s^C is the complement of I_s.

The above equation can be viewed as a recursion which determines $\Delta_i(P_s)$ in order of increasing cardinality of P_s. The conditions that $\lambda_i = \Delta_i(P_s)/\Delta(P_s)$ is the solution of Equation (5.2) are

$$\begin{cases} \Delta(P_s) > 0; \\ \Delta_i(P_s) > 0 \quad \text{for each } i \in I_s; \\ \Delta_j(P_s \cup \{\mathbf{p_j}\}) \leq 0 \quad \text{for each } j \in I_s^c. \end{cases} \tag{5.6}$$

The algorithm to compute the closest point is therefore to evaluate all subsets of P until all the above conditions hold. On rare occasions the algorithm will not find a P_s that satisfies the above three conditions. This is due to numerical roundoff error and the case that P is affinely dependent or nearly so. A back-up method is used as describe in [18] when this situations occurs by choosing the numerically best P_s. This is the one with the minimum value from the origin to aff(P). The formula is given by

$$\sqrt{\Delta(P_s)^{-1}\sum_{i\in I_s}\Delta_i(P_s)\mathbf{p_i}\cdot\mathbf{p_k}}\ ,\ k=\min i\in I_s \tag{5.7}$$

subject to $\Delta(P_s) > 0$ and $\Delta_j(P_s) > 0$ for all $j \in I_s$.

Since P is a simplex with $n \leq 3$, the algorithm finds the closest point from the origin to P in constant time.

5.3 Using Hierarchical Representation to Search for a Supporting Vertex

The time complexity for searching a new supporting vertex in a polytope P can be reduced to $O(\log n)$ with $O(n)$ preprocessing time where n is the number of vertices in P, using the hierarchical representation of polytopes as described in Section 2.5. Let hier$(P) = \{P_1, \ldots, P_h\}$ be the hierarchical representation of polytope P where h, the height of the hierarchical representation, is $O(\log n)$.

Using this representation, we can find a supporting vertex of polytope P in direction $\mathbf{S}$ by first finding the supporting vertex $\mathbf{v_h}$ of polytope P_h, which takes constant time since P_h is a simplex. Then local search is used to find the supporting vertex $\mathbf{v_i}$ of P_i in direction $\mathbf{S}$ starting from $\mathbf{v_{i+1}}$, for $i = h-1, h-2, \ldots, 1$.

Lemma 9: *For* $i = 1, \ldots, h-1$, *let* $\mathbf{v_{i+1}}$ *be a supporting vertex of* P_{i+1} *with respect to direction* $\mathbf{S}$. *Then either* $\mathbf{v_{i+1}}$ *is also a supporting vertex* P_i *with respect to direction* $\mathbf{S}$ *or a supporting vertex* $\mathbf{v_i}$ *of* P_i *with respect to* $\mathbf{S}$ *is a neighbor of* $\mathbf{v_{i+1}}$ *in* P_i.

Proof : Let $\mathbf{v_i}$ be a supporting vertex of P_i with respect to $\mathbf{S}$. Since $P_{i+1} \subset P_i$, $\mathbf{S}\cdot\mathbf{v_i} \geq \mathbf{S}\cdot\mathbf{v_{i+1}}$. When $\mathbf{S}\cdot\mathbf{v_i} = \mathbf{S}\cdot\mathbf{v_{i+1}}$, clearly, $\mathbf{v_{i+1}}$ is also a supporting vertex of P_i with respect to $\mathbf{S}$.

When $\mathbf{S}\cdot\mathbf{v_i} > \mathbf{S}\cdot\mathbf{v_{i+1}}$, we must have $\mathbf{v_i} \notin P_{i+1}$; for otherwise $\mathbf{v_i}$ would be a supporting vertex in P_{i+1}, instead of $\mathbf{v_{i+1}}$. Suppose $\mathbf{v_i}$ is not a neighbor of $\mathbf{v_{i+1}}$ in

P_i. By property (iv) of the hierarchical representation, all the neighbors of $\mathbf{v_i}$ are vertices of P_{i+1}, and these neighbors do not include $\mathbf{v_{i+1}}$. Therefore the open line segment $\overline{\mathbf{v_i v_{i+1}}}$ has nonempty intersection with the polytope P_{i+1}. Let a point in this nonempty intersection $\overline{\mathbf{v_i v_{i+1}}} \bigcap P_{i+1}$ be $\mathbf{v}_\lambda = (1-\lambda)\mathbf{v_i} + \lambda \mathbf{v_{i+1}}$ for some λ with $0 < \lambda < 1$. Then

$$\mathbf{S} \cdot \mathbf{v}_\lambda = (1-\lambda)\mathbf{S} \cdot \mathbf{v_i} + \lambda \mathbf{S} \cdot \mathbf{v_{i+1}} > \mathbf{S} \cdot \mathbf{v_{i+1}}.$$

This contradicts that $\mathbf{v_{i+1}}$ is a supporting vertex of P_{i+1} with respect to $\mathbf{S}$. Hence $\mathbf{v_i}$ is a neighbor of $\mathbf{v_{i+1}}$ in P_i. $\square$

Since $h = O(\log n)$, a supporting vertex of $P = P_1$ can be found in $O(\log n)$ time, assuming that the degrees of vertices in P_i are bounded by a constant.

In a virtual environment we can make use of temporal coherence to further improve the running time of finding a supporting vertex. The idea is to use a good estimation of the supporting vertex as an initial vertex, then employ local search to find a supporting vertex. This vertex chosen is described in Section 3.5 and Section 3.6, basically this is the cached supporting vertex in the preceding time frame. Using local search, each supporting vertex searching step takes $O(d)$ time where d is the number of vertices visited when walking from the initial vertex to a supporting vertex.

We are currently investigating whether local search can be combined with the hierarchical representation of polytopes to achieve a better running time of $O(\log d)$. So far we have proved that it is true for 2D case. The question of whether it also holds for 3D case is subjected to further research.

5.4 The Improved Gilbert's Algorithm

We have proved in the above section that using the hierarchical representation of polytopes, the time complexity for searching a new supporting vertex can be reduced from $O(n)$ to $O(\log n)$ with $O(n)$ preprocessing time and $O(n)$ space. Gilbert claims that the expected number of iterations of his algorithm is constant. If this is true, the running time of our improved Gilbert's algorithm is expected $O(\log n)$, which is the first known expected $O(\log n)$ algorithm for finding the closest pair of points between two polytopes. Besides, it is more practical than the current best theoretical time of $O((\log n)^2)$ in [43] for finding the closest pair of points.

In a virtual environment, we can make use of temporal coherence to further increase the efficiency of searching a supporting vertex. The idea is to give a good estimate of the closest points as an initial set S and use local search to find a supporting vertex. This initial set S chosen is described in the next section. As a result, each supporting vertex searching step takes $O(d)$ time where d is the number of vertices visited when walking from the initial vertex to the supporting vertex. Using local search, the time complexity of the improved Gilbert's algorithm is expected $O(d)$ time which is nearly independent of polytope's complexity.

To make the algorithm robust, we modified the termination condition $\mathbf{m} \cdot \mathbf{m} = \mathbf{m} \cdot \mathbf{C_m}$ to avoid numerical imprecision of the floating point comparison encountered. To solve this problem, the supporting vertex $\mathbf{m}$ found in each step of the algorithm is recorded. If there is a reoccurrence of supporting vertices then the algorithm terminates. Since the expected number of iterations performed in the algorithm is constant, checking the reoccurrences of supporting vertices does not change the complexity of the algorithm. This new termination condition holds because if the closest point is found in the current step, the supporting vertex found in the next iteration must be in V. Eventually, there is a reoccurrence of supporting vertices. Besides, there is no reoccurrence of supporting vertices in the original Gilbert's algorithm. Therefore, this modified termination condition is simpler and more robust.

The improved Gilbert's algorithm is as follows:

1. Set $S = \{\mathbf{m_1}, \ldots, \mathbf{m_r}\},\ r \leq 4$.

2. Compute the closest point $\mathbf{C_m}$ and the convex hull, V, of $\mathbf{C_m}$.

3. Compute the contact function $\mathbf{m} = C_M(-\mathbf{C_m}) = C_Q(-\mathbf{C_m}) - C_P(\mathbf{C_m})$ using local search.

4. If $\mathbf{m}$ appears before, return $\mathbf{C_m}$.

5. Otherwise, save $\mathbf{m}$, set $S = V \cup \mathbf{m}$ and goto step 2.

Experiments have shown that if the initial point is chosen to be the closest point found in the preceding time frame, the performance of this improved Gilbert's algorithm alone (i.e. without using the separating vector algorithm) outperforms the closest feature algorithm [1] especially when the complexity of polytopes is high (see Section 6.2).

5.5 The Combined Algorithm

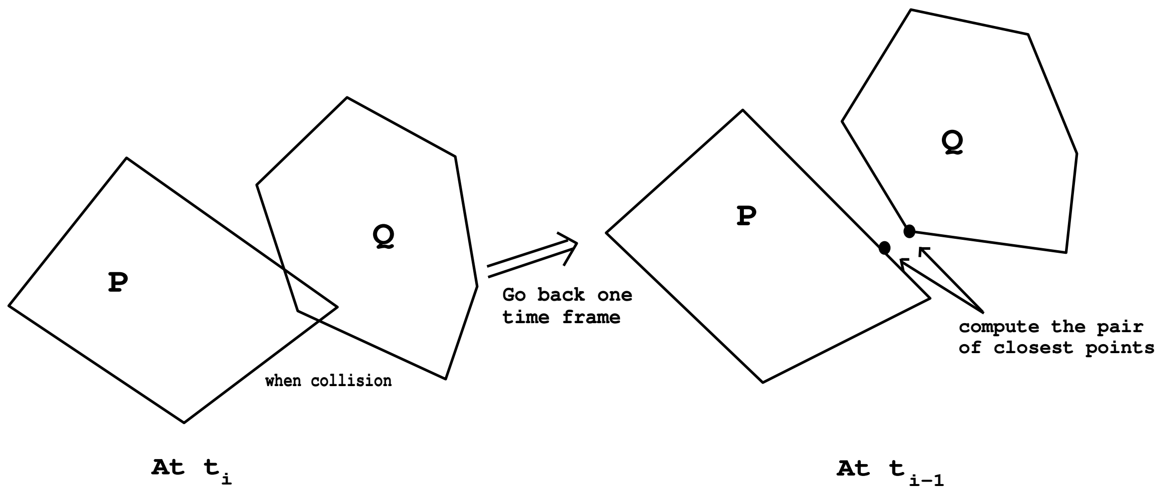

Figure 5.2: Computing the closest pair of points in the case of collision.

In animation and robotics, the closest pair of points is important for computing real-time responses when there is a collision. But it is usually the case that the closest pair of points is of no use when there is no collision, e.g. in motion path planning [52] and virtual reality fly-throughs [39]. The choice of not computing the closest pair of points in every time frame is also adopted in the method of separating plane [12] and various bounding boxes algorithm.

On the other hand, it is often too late to compute the closest pair of points once collision has been detected. This contradictory requirements suggest us to adopt a new strategy that computes the closest pair of points only at the latest non-collision time frame when collision is detected (see Figure 5.2). This is achieved by saving the transformation matrix of the preceding time frame in the algorithm. If the separating vector algorithm detects that there is collision in the current time frame and the closest pair of points is useful to the program, then the closest pair of points is computed by the improved Gilbert's algorithm using the transformation matrix of the immediately preceding time frame.

To reduce the time needed to save the transformation matrix at each time frame, we can just swap the two pointers which point to the memory occupied by the previous and current transformation matrices. This achieves the same effect as saving the old transformation matrices but with much less costs.

Since the separating vector algorithm does not find the closest pair of points at every time frame, we cannot use the closest pair of points in the preceding time frame to initialize the improved Gilbert's algorithm in the current time frame. Nevertheless, the separating vector algorithm usually finds some reoccurrence vertices when there is a collision, so these vertices are used to initialize the improved Gilbert's algorithm. In rare cases if there are no reoccurrences vertices and there is a collision, the last two supporting vertices encountered are used. When the improved Gilbert's algorithm returns, a vertex on the resultant convex hull (select any one) and the line connecting the closest pair of points are cached. They are used as initial vertices and initial searching direction for the separating vector algorithm in the next time frame. Thus the results in the separating vector algorithm are useful to initialize the improved Gilbert's algorithm and the results in the improved Gilbert's algorithm are useful to initialize the separating vector algorithm in the next time frame.

The reasons why we combine the separating vector algorithm and the improved Gilbert's algorithm for collision detection and finding the closest pair of points are

1. Both algorithms use local search to locate vertices for collision detection. The local search exploits both temporal and geometric coherences in a virtual environment. Therefore the time complexity of both algorithms are nearly inde-

pendent of polytopes' complexity.

2. Both algorithms consider only vertices of polytopes and the neighboring relationship between them. Therefore the data structure of polytopes consists of merely vertices and a list of neighboring vertices for each vertex, without other information such as edges, faces and the relationship between them. This greatly reduces memory usage and simplifies the coding of the algorithm.

3. Both algorithms compute good initial points for each other when there is a collision. Therefore, they are highly dependent on each other and the effort spent in one algorithm is useful to the other.

5.6 Dealing with Concave Polyhedra

In real world, there are many non-convex polyhedral objects. The problem of collision detection between them cannot therefore be tackled directly with our algorithm. However, it is observed that many concave polyhedral objects can be represented by a union of convex polyhedra or composed of several non-convex subparts, where each non-convex subpart may be further represented as a union of convex polyhedra or non-convex objects. Therefore we suggest using the hierarchical tree representation of objects as proposed in [15]. Each node of the tree represents non-convex object that can be decomposed further, while each leaf of the tree represent a convex object. We propose to extend the tree so that each leaf of the tree can also be a non-convex object that cannot be decomposed further (e.g. a torus).

The convex hull is computed for each object in the node of the tree. The separating vector algorithm is used to check the convex hulls of two objects. If there is a collision, their children will be expanded and all children of one parent node are checked against all children of the other parent node. This expansion will be done recursively if there is also a collision between the children. For a non-convex leaf object (which cannot be further decomposed), we propose to find its convex hull first. If there is a collision between the convex hulls of two objects, then the method using OBB-Tree [26] is used. Thus for non-convex objects, we propose to preprocess it with hierarchical tree representation and OBB-Tree representation first, then apply our separating vector algorithm or the OBB-Tree algorithm. We believe that using this approach collision detection of concave objects can be done efficiently and concave objects are decomposed (in OBB-Tree algorithm) only when necessary.

Chapter 6

Implementation

In this chapter we will discuss the implementation details of our algorithms. Besides, we will carry out various experiments to verify the properties of our algorithm. To test how efficient our algorithm is, we will give detailed comparisons between the two collision detection libraries : Q_COLLIDE - the implementation of our separating vector algorithm, and I_COLLIDE - the fastest collision detection library for polytopes so far.

6.1 Quick Collision Detection Library

We have implemented the separating vector algorithm and the improved Gilbert's algorithm in a collision detection library package called Q_COLLIDE - Quick Collision Detection Library, in order to test the efficiency of our algorithm and verify some of its properties. This library is a modification of another collision detection library package called I_COLLIDE - Interactive Collision Detection Library [1]. I_COLLIDE is a fast collision detection library for polytopes written by M. C. Lin. Besides, the source code is publicly available [1]. I_COLLIDE uses the sweep and prune technique to determine whether two bounding boxes overlap. Temporal coherence is exploited so that the algorithm runs in expected $O(n + e)$ time for n bounding boxes and e

[1]http://www.cs.unc.edu/~manocha/I_COLLIDE.html

intersections of bounding boxes. If bounding boxes of two polytopes overlap, the closest features tracking algorithm, which runs in expected constant time, is used to keep track of the closest features between them. The method works by finding and maintaining a closest pair of features on the two polytopes as they move. If the algorithm fails to maintain the closest features as in the case of collision, another linear programming algorithm is invoked in I_COLLIDE to tell whether the two polytopes actually collide or not.

We replace the underlying layer of I_COLLIDE which tracks the closest pair of features and the linear programming algorithm in I_COLLIDE by our separating vector algorithm and the improved Gilbert's algorithm. When the sweep and prune technique determines that bounding boxes overlap, our separating vector algorithm is used. If the separating vector algorithm detects that there is a collision, then the improved Gilbert's algorithm is used to compute the closest pair of points between the two colliding polytopes using the previous transformation matrices of polytopes. For simplicity, we do not use the hierarchical representation of polytopes. Besides, the line connecting the two polytopes' centers is used as the initial searching direction when bounding boxes overlap for the first time. A precomputed 8×16 table for the supporting vertices is used as explained in Section 3.5. We have made the source code of Q_COLLIDE, and testing data publicly available for interested researchers [2].

The data structure of polytopes used in Q_COLLIDE is simple. It holds a list of vertices, the coordinates of the centroid, the transformation matrix and the axis-aligned bounding box. The data structure of each vertex contains geometric information about vertices in the local coordinate system of the polytope and a list of its neighboring vertices referenced by pointers. The list of neighboring vertices can be arranged in any order. There is also a data structure for every pair of polytopes with overlapping bounding boxes. This structure contains information of a cached separating vector, a cached supporting vertex for each polytope and the closest pair of points.

[2] http://www.cs.hku.hk/~tlchung/collision_library.html

The differences between Q_COLLIDE and I_COLLIDE are :

1. Q_COLLIDE uses the separating vector algorithm whereas I_COLLIDE uses the closest features tracking algorithm as the main collision detection algorithm when bounding boxes overlap.

2. I_COLLIDE computes the closest pair of features at every time frame whereas Q_COLLIDE computes the closest features/points pair only at the time frame immediately before a collision occurs.

3. Q_COLLIDE computes only the closest pair of features whereas I_COLLIDE computes both the closest pair of points and the closest pair of features.

4. Q_COLLIDE uses less memory and is more efficient then I_COLLIDE.

For flexibility, a procedure is available in Q_COLLIDE as an option to find the closest pair of points at any time using the improved Gilbert's algorithm. Besides there is an option available in Q_COLLIDE which uses the improved Gilbert's algorithm directly (i.e. without using the separating vector algorithm) so that it can compute the closest pair of points between two polytopes in every time frame. In the next section, we will show that the performance of this improved Gilbert's algorithm alone is already better than I_COLLIDE. In the case that the closest pair of points is not useful even when there is a collision, another option is available in Q_COLLIDE which can turn off the improved Gilbert's algorithm altogether and only the separating vector algorithm is used.

6.2 Experiments

Experiments have been carried out to investigate some properties of our algorithm. Besides, we will compare the performance of our algorithms with the closest feature tracking algorithms. The simulation uses polytopes of the same number of vertices moving in a closed environment. The simulation is done on SGI/Indy machine (R4600). To find the real difference in performance between the two libraries, only the simulation time for collision detection is measured. This is done by not rendering the scene with the graphics pipeline.

Polytopes of three different shapes are used: an ellipsoid, a thin rod, and a flat plate, obtained by randomly sampling points on the surface of an ellipsoid, a thin rod, and a flat plate, respectively. They provide a variety of different shapes for testing. Unless otherwise specified, each object has its translational velocity equal to 5% of its radius and rotational velocity 10 degrees per time frames. When there is a collision between two polytopes, their rotational and translational velocities are reversed. Two snapshots of the simulated environment are shown in Figure 6.1 and Figure 6.2.

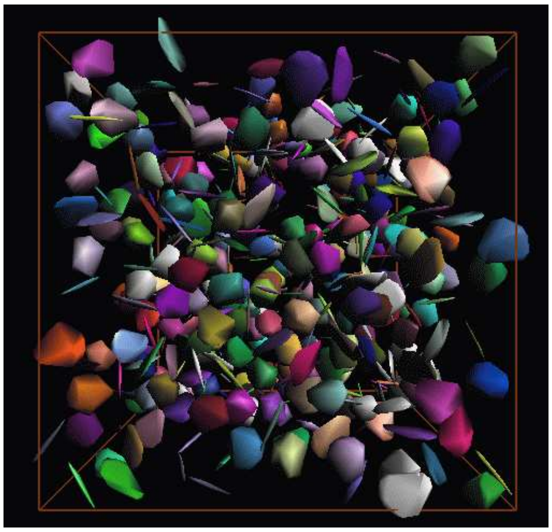

Figure 6.1: The experiment with 500 polytopes in the environment and $n = 20$.

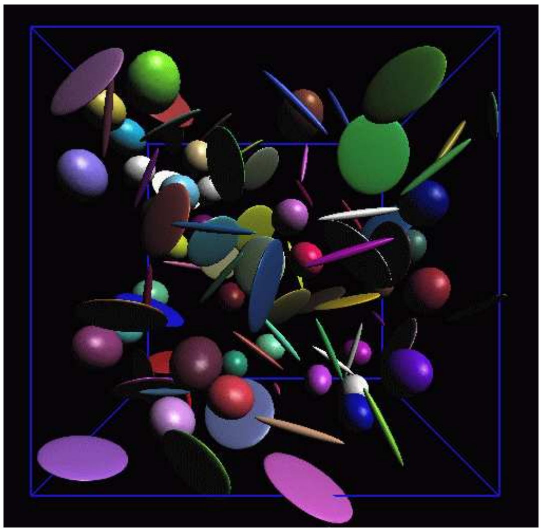

Figure 6.2: The experiment with 100 polytopes in the environment and $n = 1000$.

6.2.1 Properties of Separating Vector Algorithm

Experiments have been carried out to investigate the number of the searching steps k for polytopes with different numbers of vertices n. The simulation uses 500 polytopes of the same shape in the environment. In Figure 6.3, the value of k is measured when collision test is called between non-colliding objects of 500 vertices. This collision test is called only when the tightest rectangular bounding boxes (as found in [1]) of two polytopes overlap. The results show that more than 95% of non-colliding objects are identified in the first three steps for all three shapes. Moreover, in the case of ellipsoids more than 99% of non-interference objects are identified as non-colliding in four steps.

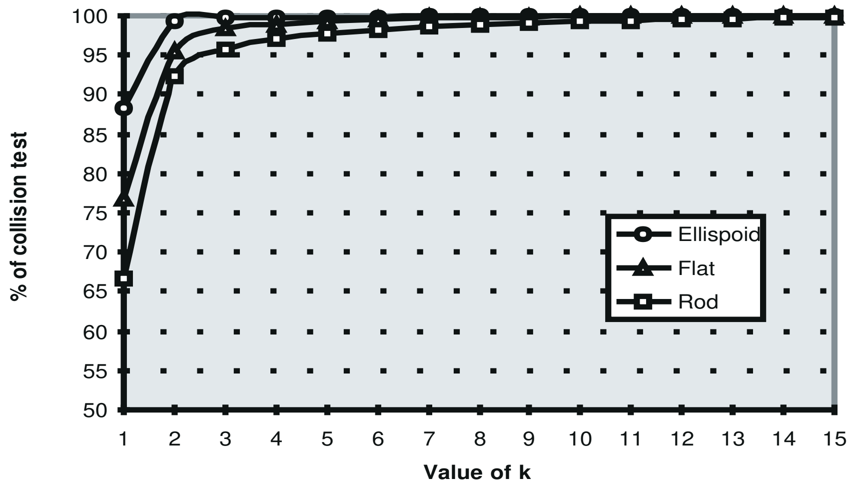

Figure 6.3: Number of searching steps when there is no collision.

Figure 6.4 measures the value of k when collision test is called for both colliding and non-colliding objects of 500 vertices. The results show that on average more than 80% of collision tests can be completed within three searching steps. Moreover, for polytopes of different numbers of vertices, a similar curve to that in Figure 6.3 and Figure 6.4 is obtained. This indicates that the algorithm runs in almost expected constant time. We also noticed that there are reoccurrences of supporting vertices during the search for separating plane in less than 0.1% of collision tests for the polytopes which do not collide.

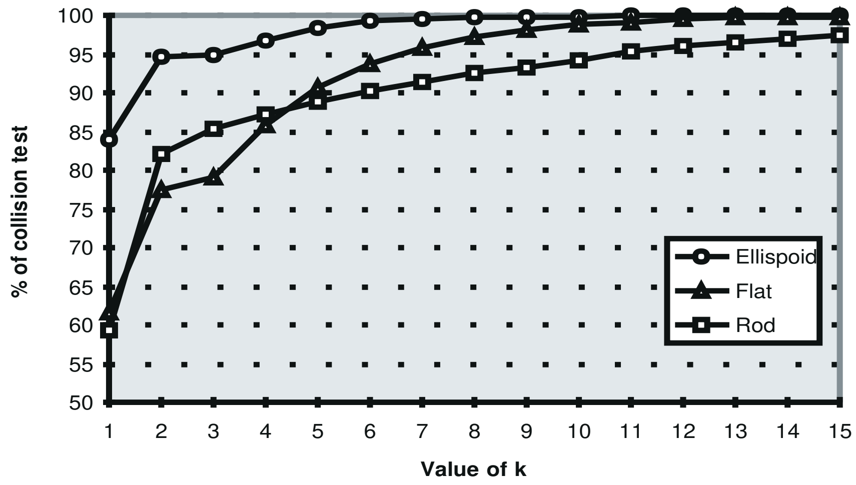

Figure 6.4: Overall number of searching steps used.

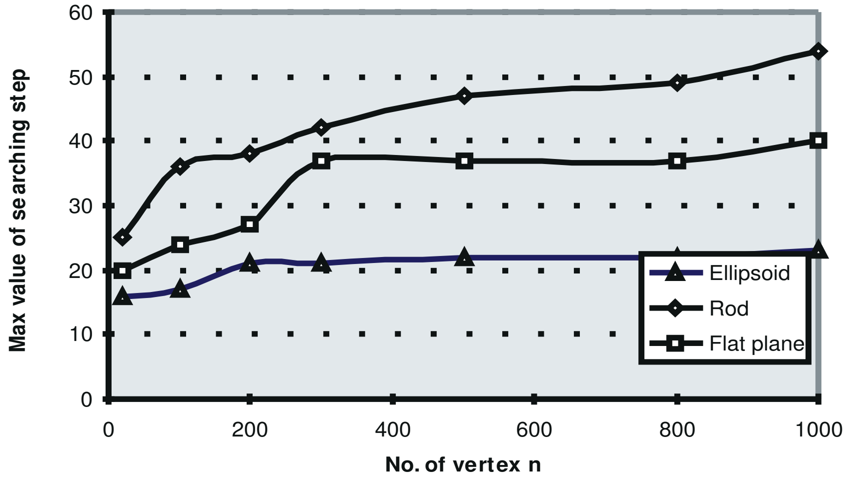

Figure 6.5: Maximum value of k.

In Figure 6.5, the maximum value of k for each case is recorded. From the figure, the maximum value of k is around 22 for ellipsoids, increases slightly from 20 to 40 for flat plates and increases from 25 to 55 for rods when n increases from 10 to 1000. It is noticed that the algorithm performs best for ellipsoid-shaped objects, and becomes less efficient for objects with plate-shape or rod-shape. The results also indicate that even in the worst example we have constructed involving thin rods and flat plates, the maximum value of k is small as compared to n. Besides, it is noted that this worst-case value of k happens very rarely (typically $\leq 0.01\%$ of collision test) in the experiments. This explains why our algorithm runs significantly faster than others on average as shown in the next experiment, especially in virtual environments where we can make use of temporal coherence.

6.2.2 Comparison with I_COLLIDE

We have compared our collision detection library Q_COLLIDE with I_COLLIDE, the fastest collision detection library so far. The closest features tracking algorithm (CF) which finds the closest pair of points in every time frame is compared against

1. The separating vector algorithm (SV), which does not find the closest pair of points at all.
2. The improved Gilbert's algorithm (Gilbert), which finds the closest pair of points in every time frames.
3. The combined algorithm (SV + Gilbert), which finds the closest pair of points only in the preceding non-collision time frame immediately before a collision occurs.

A total of 100 polytopes, with a mixture of the above three shapes and the same number of vertices, are used in the environment. Figure 6.6 and Figure 6.7 show the cases when $n = 20$ and $n = 500$ respectively, and the translational velocity is gradually increased from 2% to 20% of object radius per time frame. From the figures, the simulation time of SV algorithm increases slightly; however, the simulation time of CF algorithm increases substantially. Besides, the performance of the improved Gilbert's algorithm alone (without using separating vector algorithm) is better than CF algorithm when the translational velocity or the complexity of polytopes is high.

Figure 6.8 and Figure 6.9 show that, when the rotational velocity is increased from 5 degrees to 40 degrees per time frame, SV algorithm takes only a little longer time, while the CF algorithm takes substantially longer time. Here the set up is the same as above. In all cases, simulation time for the combined algorithm only needs a little more time than the SV algorithm. This indicates that the improved Gilbert algorithm is efficient. This fact also conforms with that in a virtual environment the probability of collision is relatively small compared with non-collision cases.

There are four reasons why our separating vector algorithm is significantly faster, and even the improved Gilbert's algorithm alone is faster, than the closest features tracking algorithm if translational/rotational velocity increases or complexity of polytopes is high.

1. When the translational/rotational velocity increases or the complexity increases, the CF algorithm needs more time to walk around on the boundary of polytopes in order to find the closest features, since each step may walk from vertex to edge, edge to face, face to vertex etc. However, in the SV algorithm and the improved Gilbert's algorithm a walk is always from vertex to vertex so they proceed "faster".

2. The condition for a walk to take place in the SV algorithm or the improved Gilbert's algorithm is simply a comparison between dot product of vectors. But in CF it requires an involved checking about whether one feature lies inside the Voronoi regions of other features.

3. CF transforms every feature along the walking path to the world coordinate system for comparison. However, in SV and the improved Gilbert's algorithm, the comparison is done in the local coordinate system of polytopes only. Only two coordinate transformation are required for a searching for a supporting vertex and there is no need to transform every vertex along the walking path to the world coordinate system during the search.

4. The I_COLLIDE library needs to call another linear programming algorithm for exact collision detection every time when there is a recycling of features (e.g. a collision occurs). Thus it runs in $O(n)$ time in the case of collision and cases where the closest features algorithm cannot resolve conflicts. However, Q_COLLIDE uses a nearly constant time algorithm even when there is a collision.

As a result, for velocity that is 20% of object radius per time frame and the number of vertices of each polytope is 500, we observe a nearly 28 times speedup by Q_COLLIDE over I_COLLIDE.

Lastly, Figure 6.10 and Figure 6.11 show the case when the density of the environment changes. Again, the combined algorithm is faster and more efficient than the CF algorithm in all cases. This indicates that our algorithm is more efficient in a highly dense virtual environment in which there are many moving objects.

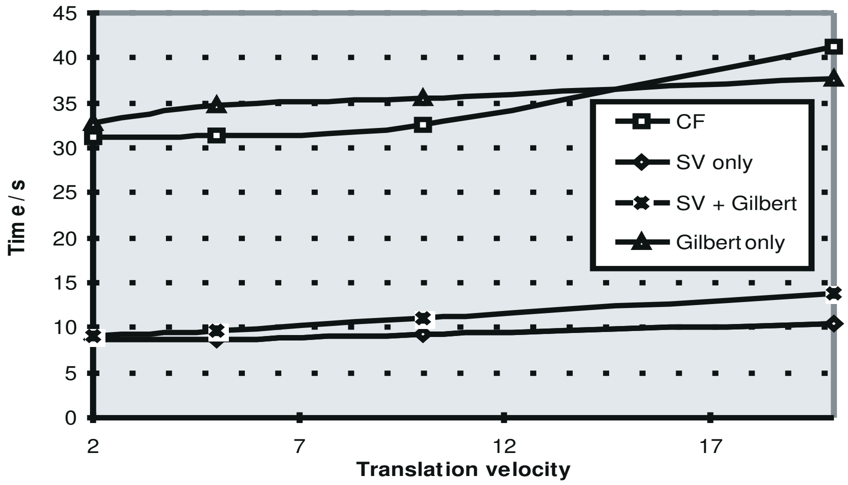

Figure 6.6: Simulation time when translation velocity increases for $n = 20$.

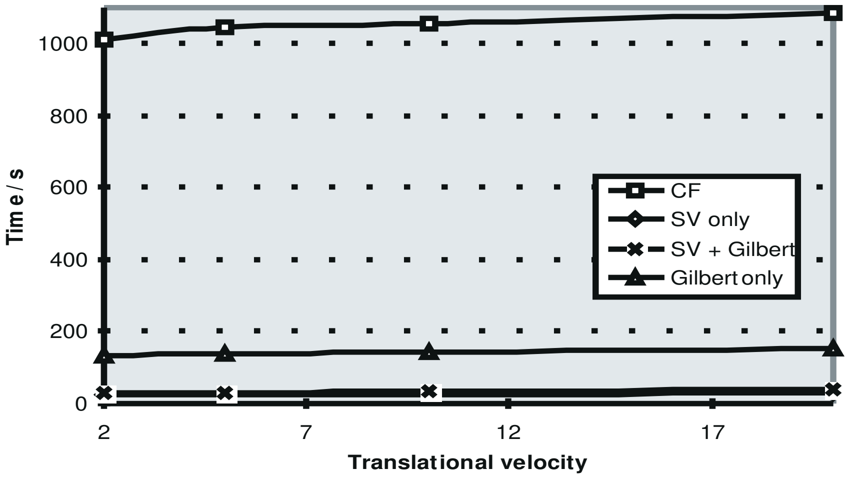

Figure 6.7: Simulation time when translation velocity increases for $n = 500$.

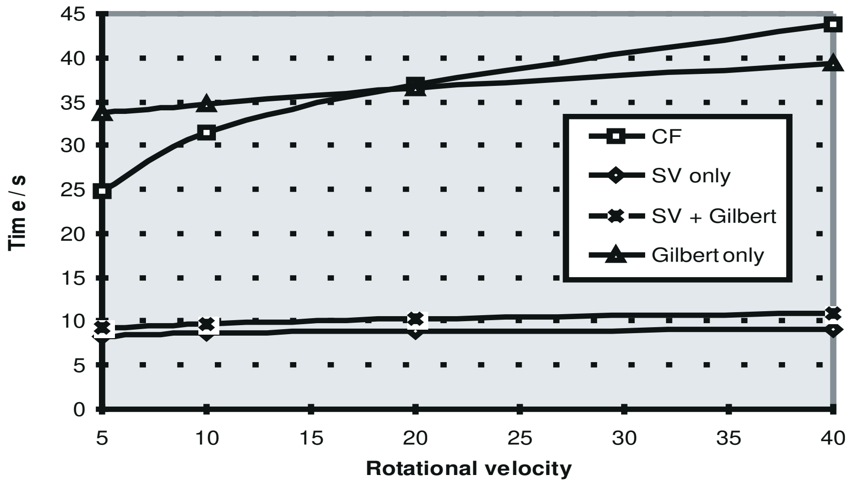

Figure 6.8: Simulation time when rotational velocity increases for $n = 20$.

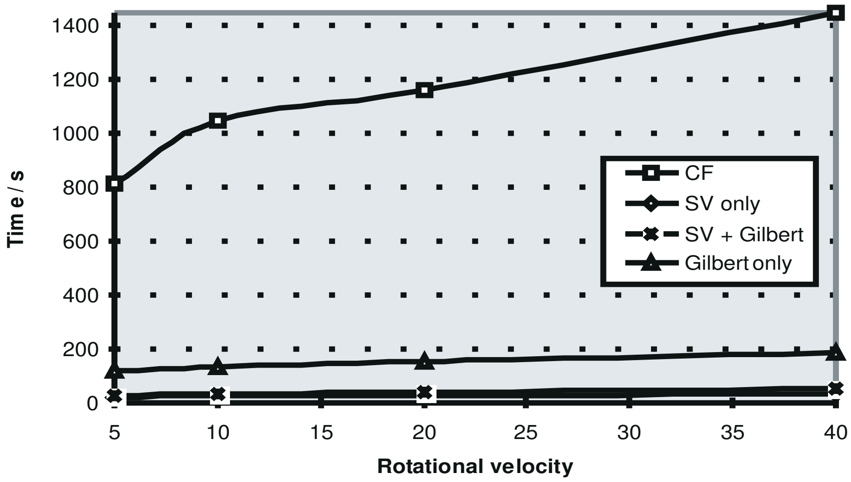

Figure 6.9: Simulation time when rotational velocity increases for $n = 500$.

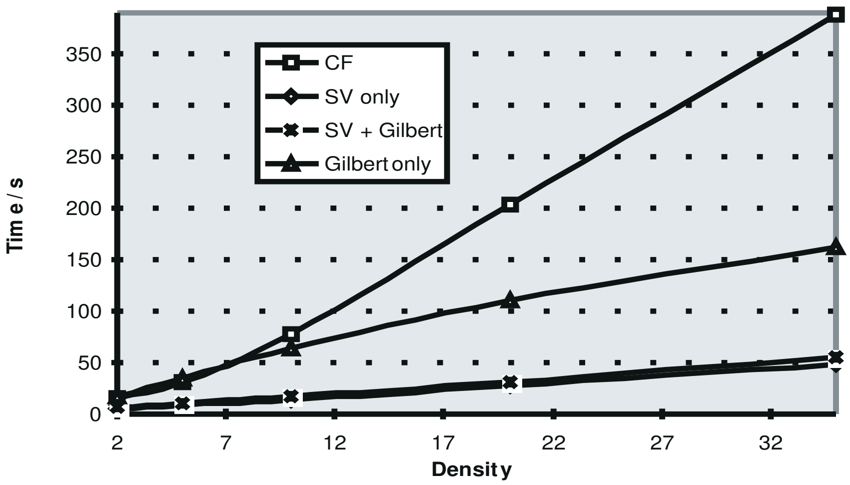

Figure 6.10: Simulation time when density increases for $n = 20$.

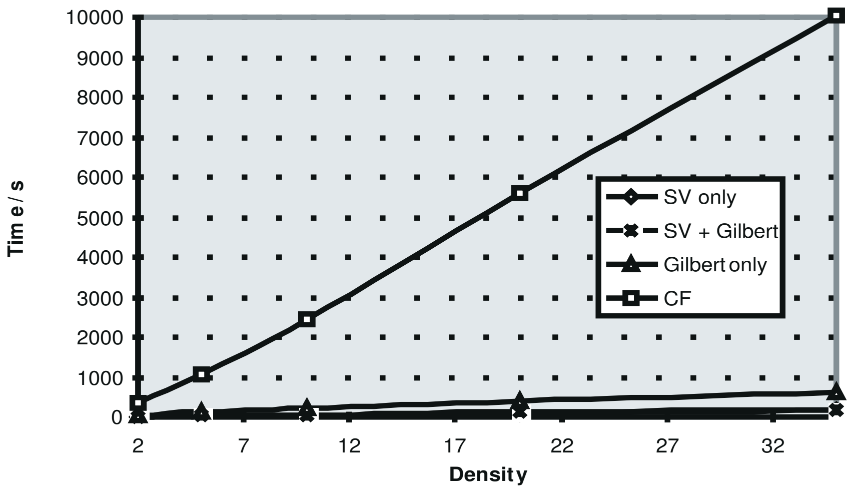

Figure 6.11: Simulation time when density increases for $n = 500$.

Chapter 7

Conclusion

We have proposed an efficient exact collision detection algorithm for polytopes in virtual environments. The algorithm is based on a simple technique to quickly locate a separating plane between two polytopes if they do not collide, or otherwise test some simple conditions to report collision. As the contact points between two objects when they collide provide useful information for impulse computation, we improved the Gilbert's algorithm and integrated it with our separating vector algorithm to compute the closest pair of points when there is a collision. Our algorithm is fast and simple to implement. Taking advantage of geometric and temporal coherences in a dynamic environment, our algorithm uses caching, preprocessing, and local search to run in expected constant time empirically. A collision detection library, called Q_COLLIDE, based on our algorithm is publicly available [1].

[1] http://www.cs.hku.hk/~tlchung/collision_library.html

Bibliography

[1] D. J. Cohen, M.C. Lin, D. Manocha, and M. Ponamgi, I-Collide: An interactive and exact collision detection system for large-scale environments, *Proceeding of Symposium of Interactive 3D Graphics*, pp. 189-196, 1995.

[2] M. Moore and J. Wilhelms, Collision detection and response for computer animation, *ACM Computer Graphics*, Vol. 22, No. 4, pp. 289-298, 1988.

[3] W. Thibault and B. Naylor, Set operations on polyhedra using binary space partitioning trees, *ACM Computer Graphics*, 4, pp. 153-162, 1987.

[4] A. Foisy, V. Hayward, and S. Aubry, The use of awareness in collision prediction, *International Conference on Robotics and Automation*, pp. 338-343. IEEE, 1990.

[5] Ming C. Lin, Dinesh Manocha, Fast interference detection between geometric models, *The Visual Computer*, 11, pp. 542-561, 1995.

[6] Atsushi Yamada, Fujio Yamaguchi, Homogeneous bounding boxes as tools for intersection algorithms of rational Bézier curves and surfaces, *The Visual Computer*, 12, pp. 202-214, 1996.

[7] A. Smith, Yoshifumi Kitamu, Haruo Takemura, and Fumio Kishino, A simple and efficient method for accurate collision detection among deformable polyhedral objects in arbitrary motion, *Virtual Reality Annual International Symposium*, pp. 136-145, IEEE, 1995.

[8] A. Garcia-Alonso, N. Serrano and J. Flaquer, Solving the collision detection problem, *IEEE Computer Graphics and Applications*, 13(3), pp. 36-43, 1994.

[9] Y. Yang and N. Thalmann, An improved algorithm for collision detection in cloth animation with human body, *First Pacific Conference on Computer Graphics and Application*, pp. 237-251, 1993.

[10] M. Lin and J. Canny, Efficient collision detection for animation, *Proceedings of the Third Eurographics Workshop on Animation and Simulation*, Cambridge,

1991.

[11] W. Bouma and G. Vanecek, Collision detection and analysis in a physical based simulation, *Proceedings of Eurographics Workshop on Animation and Simulation*, pp. 191-203, September, 1991.

[12] D. Baraff, Curved surfaces and coherence for non-penetrating rigid body simulation, *ACM Computer Graphics*, Vol. 24, No. 4, pp. 19-28, 1990.

[13] Rich Rabbitz, Fast collision detection of moving convex polyhedra, *Graphics Gem IV*, AP Professional, pp. 83-109, 1994.

[14] D.P. Dobkin and D.G. Kirkpatrick, A linear algorithm for determining the separation of convex polyhedra, *Journal of Algorithms*, Vol. 6, pp. 381-392, 1985.

[15] M. C. Lin, *Efficient Collision Detection for Animation and Robotics*, PhD thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley, December 1993.

[16] Philip M. Hubbard, Collision Detection for Interactive Graphics Applications, *IEEE Transactions on Visualization and Computer Graphics*, Vol. 1, No. 3, pp. 218-228, 1995.

[17] Hahn JK, Realistic animation of rigid bodies, *Computer Graphics*, 22(4), pp. 299-308, 1988.

[18] E. G. Gilbert, D. W. Johnson, and S. S. Keerthi, A fast procedure for computing the distance between complex objects in three-dimensional space, *IEEE Journal of Robotics and Automation*, 4(2), pp. 193-203, 1988.

[19] Elmar Schomer and Christian Thiel, Efficient collision detection for moving polyhedra, *Proceedings of the Eleventh Annual Symposium on Computational Geometry*, pp. 51-60, 1995.

[20] M. Ponamgi, D. Manocha, and M. Lin, Incremental algorithms for collision detection between solid models, *Proceedings of ACM/Siggraph Symposium on Solid Modeling*, pp. 293-304, 1995.

[21] R. T. Rockafellar, *Convex Analysis*, Princeton University Press, 1970.

[22] B. Chazelle and D. Dobkin, Detection is easier than computation, *ACM Symposium on Theory of Comput.*, 12 , pp. 146-153, 1980.

[23] S. Quinlam. Efficient distance computation between non-convex objects. *Proceedings of International Conference on Robotics and Automations*, pp. 3324-3329, 1994.

[24] N. Beckmann, H. Kriegel, R. Schncider, and B. Seeger, The R*-tree: An efficient and robust access method for points and rectangles, *Proceedings of SIGMOD Conference on Management of Data*, pp. 322-331, 1990.

[25] G. Zachman and W. Felger, The boxtree: Enabling real-time and exact collision detection of arbitrary polyhedra, *Proceedings of SIVE'95*, 1995.

[26] S. Gottschalk, M. Lin and D. Manocha, OBB-Tree: A hierarchical structure for rapid interference detection, to appear in *Proceedings of SIGGRAPH'96*.

[27] A. Pentland, Computational Complexity versus simulated environment, *ACM Computer Graphics*, 22(2), pp. 185-192, 1990.

[28] A. Pentland, Williams J, Good vibrations: modal dynamics for graphics and animation, *Computer Graphics*, 23(3), pp. 185-192, 1990.

[29] Turk G, Interactive collision detection for molecular graphics, *Master's thesis*, Computer Science Department, University of North Carolina, Chapel Hill, 1989.

[30] W. Len and M. Fusco, Fast n-dimension extent overlap testing, *Graphics Gems III*, AP Professional, pp. 240-243, 1992.

[31] G. Vanecek, Back-face culling applied to collision detection of polyhedra, *The Journal of Visualization and Computer Animation*, vol. 1 , pp. 55-63, 1994.

[32] M. Shinya, and M. Forgue, Interference detection through rasterization, *The Journal of Visualization and Computer Animation*, Vol. 2, pp. 132-134, 1991.

[33] Karol Myszkowski, Oleg G. Okunev and Tosiyasu L. Kunii, Fast collision detection between complex solids using rasterizing graphics hardware, *The Visual Computer*, Vol, 11, pp. 497-511, 1995.

[34] B. V. Herzen., A. H. Barr, and H. R. Zatz, Geometric Collision for time-dependent parametric surfaces, *ACM Computer Graphics*, 24(4), August 1990.

[35] Tom Duff, Interval arithmetic and recursive subdivision for implicit functions and constructive solid geometry, *ACM Computer Graphics*, 26(2), pp.131-139, 1992.

[36] C. Hoffmann, *Geometric and Solid Modeling*, Morgan Kaufmann Publishers Inc., San Mateo, CA, 1989.

[37] A. Pentland and J. William, Good vibrations: Modal dynamics for graphics and animation, *ACM Computer Graphics* 23(3), pp. 185-192, 1990.

[38] Ji-Hoon Youn and K. Wohn, Realtime collision detection for virtual reality applications, *IEEE First Annual Virtual Reality Symposium*, pp. 415-421, 1993.

[39] Martin Held, James T. Klosowski and Joseph S.B. Mitchell, Evaluation of collision detection methods for virtual reality fly-throughs, *Proceedings of Seventh Canadian Conference on Computational Geometry*, 1995.

[40] P. K. Agarwal and M. Sharir, Red-blue intersection detection algorithms, with applications to motion planning and collision detection, *SIAM Journal of Computing*, vol. 19, pp. 297-321, 1990.

[41] P. K. Agarwal, M. van Kreveld, and M. Overmars, Intersection queries for curved objects, *Proc. 7th Annual ACM Symposium on Computational Geometry*, pp. 41-50, 1991.

[42] B. Chazelle, An optimal algorithm for intersecting three-dimensional convex polyhedra, *Proc. 30th Annual IEEE Symposium on Foundation Computer Science*, pp. 586-591, 1989.

[43] D. P. Dobkin and D. G. Kirkpatrick, Determining the separation of preprocessed polyhedra - a unified approach, *Proc. 17th International Colloq. Automata Lang. Program*, vol. 443 of Lecture Notes in Computer Science, pp. 400-413. Springer-Verlag, 1990.

[44] Stephen Cameron, Collision detection by four-dimensional intersection testing, *IEEE Transactions on Robotics and Automation*, vol. 6, No. 3, June 1990.

[45] V. V. Kamat, A survey of techniques for simulation of dynamic collision detection and response, *Computer & Graphics*, vol. 17, No. 4, pp. 379-385, 1993.

[46] John M. Snyder, An interactive tool for placing curved surfaces without interpenetration, *ACM Computer Graphics*, pp. 209-218, 1995.

[47] Gregory J. Hamlin, Robert B. Kelley and Josep Tornero, Efficient distance calculation using the spherically-extended polytope (S-tope) model, *IEEE International Conference on Robotics and Automation*, pp. 2502-2507, 1992.

[48] N. K. Sancheti and S. S. Keerthi, Computation of certain measures of proximity between convex polytopes: A complexity viewpoint, *IEEE International Conference on Robotics and Automation*, pp. 2508-2513, 1992.

[49] H. Edelsbrunner, Computing the extreme distances between two convex polygons, *Journal of Algorithms* 6, 213-224, 1985.

[50] S. Zeghloul, P. Rambeaud and J. P. Lallemand, A fast distance calculation between convex objects by optimization approach, *IEEE International Conference on Robotics and Automation*, pp. 2520-2525, 1992.

[51] Daniel Johnson, The optimization of robot motion in the presence of obstacles, University of Michigan, *PhD. Dissertation*, 1987.

[52] M. Sharir, Efficient algorithms for planning purely translational collision-free motion in two and three dimensions. In *Proceeding of IEEE International Conference in Robotic and Automation*, pp. 1326-1331, 1987.

Appendix

I. Pseudo Code of the Separating Vector Algorithm

```
/*
  Input  : Polytopes P and Q, with origin at the center in the local coordinate system.
           The rotation and inverse rotation matrices of P and Q: Rp, Rq, invRp, invRq.
           The translation vector of P and Q: Tp, Tq.
  Return : TRUE if P and Q collide, otherwise return FALSE.
*/
Boolean SeparatingVector(P, Q, Rp, Rq, invRp, invRq, Tp, Tq)
{
  If (bounding boxes overlap for the first time) {
     S = <Tq - Tp>;    /* where <x> is the normalized vector of x, section 3.4 */
     use S to get vertices p and q from the precomputed table;   /* section 3.5 */
  }
  Else {
     retrieve S, p, q from cache;                    /* section 3.6 */
  }
  k = 0;
  do {
     k++;
     p = SearchSupportVertex(P, p, invRp*S);     /* section 3.3 */
     q = SearchSupportVertex(Q, q, invRq*(-S));
     r_k = <(q*Rq + Tq) - (p*Rp + Tp)>;
     dp = DotProduct(S, r_k);   /* Equation (3.1) */
     If ( dp >= 0) {                             /* Lemma 1 */
        save S, p, q;                            /* cache the values */
        return FALSE;                            /* no collision */
     }
     If (the pair of vertices (p, q) has appeared before) {
      S = w;                                    /* Lemma 8 */
     }
     Else
     {
        If (k = 2)
           w = <r_1 + r_2>
        If (FindHalfPlane(SetR, k, w, r_k) = FALSE)
            return TRUE;                  /* collision, see section 4.2.1 */
        save the pair of vertices (p, q);
        S = S - 2*dp*r_k;                        /* Equation (3.2) */
     }
  } While (TRUE)
}
```

Procedure to search a supporting vertex using local search

```
/*
  Input : The data structure of polytope : P
          An initial vertex the above polytope : p
          A vector in the local coordinate system of the above polytope : S
 Output : None
 Return : A supporting vertex of P in the direction S
 Remark : currenttime is a global variable which is incremented by one
          every time this procedure is executed.
*/
Vertex SearchSupportVertex(P, p, S)
{
 newp = p;
 max = DotProduct(p, S);
 currenttime++;
 If (currenttime reach its maximum value)
 {/* This case happens only when the max value of integer is attained */
   currenttime = 0;
   reset all timestamp of vertices to zero;
 }
 timestamp(p) = currenttime;
 Do {
    p = newp;
    For (each neighborhood v vertex of p)
    {
      If (timestamp(v) <> currenttime)
      {                              /* v has not been visited before */
         If (DotProduct(v, S) > max)
         {
             max = DotProduct(v, S);
             newp = p;
         }
         timestamp(v) = currenttime;
      }
    }
 } While(p != newp)
 return p;
}
```

Procedure to determine if halfplane exists

```
/*
  Input : Array to hold vectors : SetR
          The size of the above array, must be > 2 : k
          A unit vector which is the normal of halfplane for SetR
          i.e. It satisfies DotProduct(w, r) >= 0 for all r in SetR : w
          The new vector to be added in SetR : newR
 Output : The new array with newR add to SetR : SetR
          The new size of the above array : k
          The new unit vector found which is the normal of halfplane for the
          new array SetR : w
 Return : TRUE if halfplane can be found after newR insert, FALSE otherwise.
 Remarks: Define A X B = x1*y2 - y1*x2, where A=(x1, y1) and B=(x2, y2).
*/
Boolean FindHalfPlane(SetR, k, w, newR)
{
  SetR[k] = newR;
  k = k + 1;
  If (DotProduct(w, newR) >= 0)
    return TRUE;
  Compute the matrix M by Equation (4.2) where newR=(x, y, z);
  ra = drop the z-value of M*SetR[0];   /* Now ra, rb is a 2D vector */
  rb = drop the z-value of M*SetR[1];
  If (ra X rb < 0)
    swap ra, rb;    /* Make sure rb is in clockwise direction w.r.t. ra */
  For (i = 2 to k - 1)
  {
    T =  M*SetR[i]
    If ( ra X T > 0)
    {
       If ( rb X T > 0)
         rb = T
    }
    Else
    {
       If (rb X T < 0)
         ra = T
       Else
         return FALSE;   /* Cannot find a half plane, there is a collision */
    }
  }
  w = Transpose(M)*<ra + rb>; /* where <x> = x /|x| */
  return TRUE;
}
```

II. Pseudo Code of the Improved Gilbert's Algorithm

```
/*
  Input : The initial vertex set of P and Q : VertexSetp[4], VertexSetq[4]
          The size of the above initial vertex set, must be <= 4 : n
          The transformation matrices of P and Q   : Mp, Mq
          The inverse rotation matrices of P and Q : invRp, invRq
          Empty array of size 4 : lambda[4]
 Output : The affine independent vertices set for the pair of closest points
                              : VertexSetp[4], vertexSetq[4]
          The size of the above affine independent vertices set : n
          The solution of lambda for Equation (5.1) : lambda[4]
 Return : none
 Remarks: The pair of closest points is given by
          closest P = VertexSetp[0]*lambda[0] + .. + VertexSetp[n-1]*lambda[n-1]
          closest Q = VertexSetq[0]*lambda[0] + .. + VertexSetq[n-1]*lambda[n-1]
*/
ImprovedGilbert(VertexSetp, VertexSetq, n, lambda, Mp, Mq, invRp, invRq);
{
   For ( i = 0 to n-1 )
       V[i] = VertexSetq[i]*Mq - VertexSetp[i]*Mp
   p = VertexSetp[0];
   q = VertexSetq[0];
  Do {
/* Find the closest point Cp for the simplex V using Johnson's Algorithm. */
/* The solution of the affine independent set is saved in V. */
/* The size of affine independent set is saved in n which must be <= 3 */
/* The solution of lambda and closest point is saved in lambda[] and Cp. */
/* Variable Is is the index of the resulting affine independent set V. */
        JohnsonAlgorithm( V, n, lambda, Cp, Is);
        Choose the set VertexSetp and VertexSetq base on Is ;
    /* Find a supporting vertex of P in the direction Cp */
         p = SearchSupportVertex(P, p, invRp*Cp);
    /* Find a supporting vertex of Q in the direction -Cp */
         q = SearchSupportVertex(Q, q, -invRq*Cp);
        If (the pair of vertices (p, q) has appear before)
           return;
        save the pair of vertices (p, q);
        Vertexsetp[n] = p;
        VertexSetq[n] = q;
        n = n + 1;
  } While (TRUE)
}
```

Procedure to find the pair of closest points

```
/*
  Input : A polytope P with vertex set given by V : V
          The size of the above polytope : n
          An empty array of size 4 to store the solution : lambda
          An empty position to store coordinate : Cp
          An empty integer array of size 4 to store index : Is
 Output : The convex hull of the set of solution in Equation (5.1): V
          The size of the above set : n
          The solution of lambda in Equation (5.1) : lambda
          The closest point from origin to P : Cp
          The index of the solution set V : Is
 Return : None
*/
JohnsonAlgorithm( V, n, lambda, Cp, Is)
{
  Choose an ordering Ps, s = 1, 2, ..., t of all subsets of P;
  For ( s = 1 to t )
  {
     Compute V, lambda, Is by Equation (5.3)-(5.5);
     If (all conditions in (5.6) are met)
     {
        Compute Cp by Equation (5.2);
        return;
     }
  }

  Use backup procedure to select one of s with minimum value
  given by Equation (5.7);
}
```

III. Pseudo Code of the Combined Algorithm

```
/*
  Input  : Polytopes P and Q, with origin at the center in the local coordinate system .
           The rotation and inverse rotation matrices of P and Q: Rp, Rq, invRp, invRq.
           The translation vector of P and Q, Tp, Tq.
  Output : The closest pair of points of P and Q if collide
  Return : TRUE if P and Q collide, otherwise return FALSE.
*/
Boolean Combined_Collision(P, Q, Rp, Rq, invRp, invRq, Tp, Tq)
{
  If (bounding boxes overlap for the first time) {
     S = <Tq - Tp>;    /* where <x> is the normalized vector of x, section 3.4 */
     use S to get vertices p and q from the precomputed table;   /* Section 3.5 */
  }
  Else {
     retrieve S, p, q from cache;                    /* section 3.6 */
  }
  save current transformation matrices Rp, Rq, Tp, Tq;
  k = n = 0;
  prevp = prevq = NULL;
  Do {
     k++;
     prevp = p;
     prevq = q;
     p = SearchSupportVertex(P, p, invRp*S);     /* Section 3.3 */
     q = SearchSupportVertex(Q, q, invRq*(-S));
     r_k = <(q*Rq + Tq) - (p*Rp + Tp)>;
     dp = DotProduct(S, r_k);                    /* Equation (3.1) */
     If ( dp >= 0) {                             /* Lemma 1 */
        save S, p, q in cache;                   /* cache the values */
        return non-collision;
     }
     If ( (p, q) = (prevp, prevq) )              /* Lemma 3 */
     {
        save S, p, q in cache;                   /* cache the values */
        return non-collision;
     }
     If (the pair of vertices (p, q) has appeared before)
     {
        If (RepeatFlag is TRUE)                  /* Lemma 8 */
        {
            save S, p, q in cache;
            return non-collision;
        }
```

```
        S = w;                                    /* Lemma 8 */
        RepeatFlag = TRUE;
        If ((n < 4) and (p, q) not in (VertexSetp, VertexSetq))
        {
             VertexSetp[n] = p ;                  /* Section 5.5 */
             VertexSetq[n] = q ;
             n = n + 1;
         }
     }
     Else
     {
        RepeatFlag = FALSE;
        If (k = 2)
           w = <r_1 + r_2> ;
        If (FindHalfPlane(SetR, k, w, r_k) = FALSE) /* Section 4.2.1 */
        {
            If (non-collision in the previous time frame)
            {
   /* Find the closest points, the result is expressed as in Equation (5.1) */
                    If (n = 0)
                    {                                 /* Section 5.5 */
                        VertexSetp[0] = p;
                        VertexSetq[0] = q;
                        VertexSetp[1] = prevp;
                        VertexSetq[1] = prevq;
                        n = 2;
                    }
                    Retrieve transformation matrices from previous time frame;
 /* Find the pair of closest points in previous time frame */
                    ImprovedGilbert(VertexSetp, VertexSetq, n, lambda,
                                    prev_Mp, prev_Mq, prev_invRp, prev_invRq);
  /* Initial values for Separating Vector Algorithm in next time frame */
            S = <closest_q - closest_p> ;   /* Section 5.5 */
            p = VertexSetp[0];
            q = VertexSetq[0];
            save S, p, q in cache;
            }
            return collision;
        }
        save the pair of vertices (p, q);
        S = S - 2*dp*r_k;                             /* Equation (3.2) */
     }
  } While (TRUE)
}
```