Efficient Collision Detection for Animation and Robotics

Ming C. Lin

Department of Electrical Engineering

and Computer Science

University of California, Berkeley

Berkeley, CA,

Efficient Collision Detection for Animation and Robotics

by

Ming Chieh Lin

B.S. (University of California at Berkeley) 1988
M.S. (University of California at Berkeley) 1991

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Engineering - Electrical Engineering
and Computer Sciences

in the

GRADUATE DIVISION

of the

UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor John F. Canny, Chair
Professor Ronald Fearing
Professor Andrew Packard

1993

Efficient Collision Detection for Animation and Robotics

by

Ming Chieh Lin

Abstract

Efficient Collision Detection for Animation and Robotics

by

Ming Chieh Lin

Doctor of Philosophy in Electrical Engineering
and Computer Science

University of California at Berkeley

Professor John F. Canny, Chair

We present efficient algorithms for collision detection and contact determination between geometric models, described by linear or curved boundaries, undergoing rigid motion. The heart of our collision detection algorithm is a simple and fast incremental method to compute the distance between two convex polyhedra. It utilizes convexity to establish some local applicability criteria for verifying the closest features. A preprocessing procedure is used to subdivide each feature's neighboring features to a constant size and thus guarantee expected constant running time for each test.

The expected constant time performance is an attribute from exploiting the geometric coherence and locality. Let n be the total number of features, the expected run time is between $O(\sqrt{n})$ and $O(n)$ depending on the shape, if no special initialization is done. This technique can be used for dynamic collision detection, planning in three-dimensional space, physical simulation, and other robotics problems.

The set of models we consider includes polyhedra and objects with surfaces described by rational spline patches or piecewise algebraic functions. We use the expected constant time distance computation algorithm for collision detection between convex polyhedral objects and extend it using a hierarchical representation to distance measurement between non-convex polytopes. Next, we use global algebraic methods for solving polynomial equations and the hierarchical description to devise efficient algorithms for arbitrary curved objects.

We also describe two different approaches to reduce the frequency of collision detection from $\binom{N}{2}$ pairwise comparisons in an environment with n moving objects. One of them is to use a priority queue sorted by a lower bound on time to collision; the other uses an overlap test on bounding boxes.

Finally, we present an opportunistic global path planner algorithm which uses the incremental distance computation algorithm to trace out a one-dimensional skeleton for the purpose of robot motion planning.

The performance of the distance computation and collision detection algorithms attests their promise for real-time dynamic simulations as well as applications in a computer generated virtual environment.

Approved: John F. Canny

Acknowledgements

The successful completion of this thesis is the result of the help, cooperation, faith and support of many people.

First of all, I would like to thank Professor John Canny for the insightful discussions we had, his guidance during my graduate studies at Berkeley, his patience and support through some of the worst times in my life. Some of the results in this thesis would not have been possible without his suggestions and feedbacks.

I am also grateful to all my committee members (Professor R. Fearing, A. Packard, and J. Malik), especailly Professor Ronald Fearing and Andrew Packard for carefully proofreading my thesis and providing constructive criticism.

I would like to extend my sincere appreciation to Professor Dinesh Manocha for his cheerful support and collaboration, and for sharing his invaluable experience in "job hunting". Parts of Chapter 4 and a section of Chapter 5 in this thesis are the result of our joint work.

Special thanks are due to Brian Mirtich for his help in re-implementing the distance algorithm (described in Chapter 3) in ANSI C, thorough testing, bug reporting, and his input to the robustness of the distance computation for convex polyhedra.

I wish to acknowledge Professor David Baraff at Carnegie Mellon University for the discussion we had on one-dimensional sweeping method. I would also like to thank Professor Raimond Seidel and Professor Herbert Edelsbrunner for comments on rectangle intersection and convex decomposition algorithms; and to Professor George Vanecek of Purdue University and Professor James Cremer for discussions on contact analysis and dynamics.

I also appreciate the chance to converse about our work through electronic mail correspondence, telephone conversation, and in-person interaction with Dr. David Stripe in Sandia National Lab, Richard Mastro and Karel Zikan in Boeing. These discussions helped me discover some of the possible research problems I need to address as well as future application areas for our collision detection algorithms.

I would also like to thank all my long time college pals: Yvonne and Robert Hou, Caroline and Gani Jusuf, Alfred Yeung, Leslie Field, Dev Chen and Gautam Doshi. Thank you all for the last six, seven years of friendship and support, especially when I was at Denver. Berkeley can never be the same wthout you!!!

And, I am not forgetting you all: Isabell Mazon, the “Canny Gang”, and all my (30+) officemates and labmates for all the intellectual conversations and casual chatting. Thanks for the #333 Cory Hall and Robotics Lab memories, as well as the many fun hours we shared together.

I also wish to express my gratitude to Dr. Colbert for her genuine care, 100% attentiveness, and buoyant spirit. Her vivacity was contagious. I could not have made it without her!

Last but not least, I would like to thank my family, who are always supportive, caring, and mainly responsible for my enormous amount of huge phone bills. I have gone through some traumatic experiences during my years at CAL, but they have been there to catch me when I fell, to stand by my side when I was down, and were *ALWAYS* there for me no matter what happened. I would like to acknowledge them for being my “moral backbone”, especially to Dad and Mom, who taught me to be strong in the face of all adversities.

Ming C. Lin

Contents

List of Figures

Chapter 1

Introduction

The problem of collision detection or contact determination between two or more objects is fundamental to computer animation, physical based modeling, molecular modeling, computer simulated environments (e.g. virtual environments) and robot motion planning as well [3, 7, 12, 20, 43, 45, 63, 69, 70, 72, 82, 85]. (Depending on the content of applications, it is also called with many different names, such as interference detection, clash detection, intersection tests, etc.) In robotics, an essential component of robot motion planning and collision avoidance is a geometric reasoning system which can detect potential contacts and determine the exact collision points between the robot manipulator and the obstacles in the workspace. Although it doesn't provide a complete solution to the path planning and obstacle avoidance problems, it often serves as a good indicator to steer the robot away from its surrounding obstacles before an actual collision occurs.

Similarly, in computer animation, interactions among objects are simulated by modeling the contact constraints and impact dynamics. Since prompt recognition of possible impacts is a key to successful response to collisions in a timely fashion, a simple yet efficient algorithm for collision detection is important for fast and realistic animation and simulation of moving objects. The interference detection problem has been considered one of the major bottlenecks in acheiving real-time dynamic simulation.

Collision detection is also an integral part of many new, exciting technological developments. Virtual prototyping systems create electronic representations of mechanical parts, tools, and machines, which need to be tested for interconnectivity, functionality, and reliability. The goal of these systems is to save processing and manufacturing costs by avoiding the actual physical manufacture of prototypes. This is similar to the goal of CAD tools for VLSI, except that virtual prototyping is more demanding. It requires a complete test environment containing hundreds of parts, whose complex interactions are based on physics and geometry. Collision detection is vital component of such environments.

Another area of rising interests is synthetic environment, commonly known as "virtual reality" (which is a comprehensive term promoting much hypes and hopes). In a synthetic environment, virtual objects (most of them stationary) are created and placed in the world. A human participant may wish to move the virtual objects or alter the scene. Such a simple action as touching and grasping involves geometric contacts. A collision detection algorithm must be implemented to achieve any degree of realism for such a basic motion. However, often there are hundreds, even thousands of objects in the virtual world, a naive algorithm would probably take hours just to check for possible interference whenever a human participant moves. This is not acceptable for an interactive virtual environment. Therefore, a simple yet efficient collision detection algorithm is almost indispensable to an interactive, realistic virtual environment.

The objective of collision detection is to automatically report a geometric contact when it is about to occur or has actually occurred. It is typically used in order to simulate the physics of moving objects, or to provide the geometric information which is needed in path planning for robots. The static collision detection problem is often studied and then later extended to a dynamic environment. However, the choice of step size can easily affect the outcome of the dynamic algorithms. If the position and orientation of the objects is known in advance, the collision detection can be solved as a function of time.

A related problem to collision detection is determining the minimum Euclidean distance between two objects. The Euclidean distance between two objects is a natural measurement of *proximity* for reasoning about their spatial relationship.

A dynamic solution to determining the minimum separation between two moving objects can be a useful tool in solving the interference detection problem, since the distance measurement provides all the necessary local geometric information and the solution to the proximity question between two objects. However, it is not necessary to determine the exact amount of separation or penetration between two objects to decide whether a collision has taken place or not. That is, determining the minimum separation or maximum penetration makes a much stronger statement than what is necessary to answer the collision detection problem. But, this additional knowledge can be extremely useful in computing the interaction forces and other penalty functions in motion planning.

Collision detection is usually coupled with an appropriate response to the collision. The collision response is generally application dependent and many algorithms have been proposed for different environments like motion control in animation by Moore and Wilhelm [63], physical simulations by Baraff, Hahn, Pentland and Williams [3, 43, 70] or molecular modeling by Turk [85]. Since simplicity and ease of implementation is considered as one of the important factors for any practical algorithm in the computer graphics community, most collision detection algorithms used for computer animation are rather simple but not necessary efficient. The simplest algorithms for collision detection are based upon using bounding volumes and spatial decomposition techniques. Typical examples of bounding volumes include cuboids, spheres, octrees etc., and they are chosen due to the simplicity of finding collisions between two such volumes. Once the two objects are in the vicinity of each other, spatial decomposition techniques based on subdivision are used to solve the interference problem. Recursive subdivision is robust but computationally expensive, and it often requires substantially more memory. Furthermore the convergence to the solution corresponding to the contact point is linear. Repeating these steps at each time instant makes the overall algorithm very slow. The run time impact of a subdivision based collision detection algorithm on the physical simulation has been highlighted by Hahn [43].

As interest in dynamic simulations has been rising in computer graphics and robotics, collision detection has also received a great deal of attention. The importance of collision detection extends to several areas like robot motion planning, dynamic simulation, virtual reality applications and it has been extensively studied in robotics, computational geometry, and computer graphics for more than a decade [3, 4, 11, 13, 17, 41, 43, 45, 51, 53, 63]. Yet, there is no practical, efficient algorithm available yet for general geometric models to perform collision detection in real time. Recently, Pentland has listed collision detection as one of the major bottlenecks towards real time virtual environment simulations [69].

In this thesis, we present an efficient algorithm for collision detection between objects with linear and curved boundaries, undergoing rigid motion. No assumption is made on the motion of the object to be expressed as a closed form function of time. At each instant, we only assume the knowledge of position and orientation with respect to a global origin. We first develop a fast, incremental distance computation algorithm which keeps track of a pair of closest features between two convex polyhedra. The expected running time of this algorithm is constant.

Next we will extend this incremental algorithm to non-convex objects by using sub-part hierarchy tree representation and to curved objects by combining local and global equation solving techniques. In addition, two methods are described to reduce the frequency of interference detections by (1) a priority queue sorted by lower bound on time to collisions (2) sweeping and sorting algorithms and a geometric data structure. Some examples for applications of these algorithms are also briefly mentioned. The performance of these algorithms shows great promise for real-time robotics simulation and computer animation.

1.1 Previous Work

Collision detection and the related problem of determining minimum distance has a long history. It has been considered in both static and dynamic (moving objects) versions in [3], [11], [13], [17], [26], [27], [39], [40], [41], [65].

One of the earlier survey on "clash detection" was presented by Cameron [12]. He mentioned three different approaches for dynamic collision detection. One of them is to perform static collision detection repetitively at each discrete time steps, but it is possible to miss a collision between time steps if the step size is too large. Yet, it would be a waste of computation effort if the step size is too small. Another method is to use a space-time approach: working directly in the four-dimensional sets which form the abstract modes of the motion shapes (swept volumes in $4D$). Not only is it difficult to visualize, but it is also a challenging task to model such sets, especially when the motion is complex. The last method is to using sweeping volume to represent moving objects over a period of time. This seems to be a rather intuitive approach, but rather restrictive. Unless the motion of the objects are already known in advance, it is impossible to sweep out the envelope of the moving objects and it suppresses the temporal information. If two objects are both moving, the intersection of two sweeping volume does not necessarily indicate an actual "clash" between two objects.

Local properties have been used in the earlier motion planning algorithms by Donald, Lozano-Pérez and Wesley [28, 53] when two features come into contact. In [53], Lozano-Pérez and Wesley characterized the collision free motion planning problem by using a point robot navigating in the configuration space by growing the stationary obstacles with the size of the robot. As long as the point robot does not enter a forbidden zone, a collision does not take place.

A fact that has often been overlooked is that collision *detection* for convex polyhedra can be done in linear time in the worst case by Sancheti and Keerthi [77]. The proof is by reduction to linear programming. If two point sets have disjoint convex hulls, then there is a plane which separates the two sets. Letting the four parameters of the plane equations be variables, add a linear inequality for each vertex of polyhedron A that specifies that the vertex is on one side of the plane, and an inequality for each vertex of polyhedron B that specifies that it is on the other side. Megiddo and Dyers work [30], [58], [59] showed that linear programming is solvable in linear time for any fixed number of variables. More recent work by Seidel [79] has shown that linear time linear programming algorithms are quite practical for a small number of variables. The algorithm of [79] has been implemented, and seems fast in practice.

Using linear-time preprocessing, Dobkin and Kirkpatrick were able to solve the collision detection problem as well as compute the separation between two convex polytopes in $O(log|A| \cdot log|B|)$ where A and B are polyhedra and $|\cdot|$ denotes the total number of faces [27]. This approach uses a hierarchical description of the convex objects and extension of their previous work [26]. This is one of the best known theoretical bounds.

The capability of determining possible contacts in dynamic domains is important for computer animation of moving objects. We would like an animator to perform impact determination by itself without high computational costs or much coding efforts. Some algorithms (such as Boyse's [11] and Canny's [17]) solve the problem in more generality than is necessary for computer animation; while others do not easily produce the exact collision points and contact normal direction for collision response [34]. In one of the earlier animation papers addressing the issue of collision detection, Moore and Wilhelms [63] mentioned the method based on the Cyrus-Beck clipping algorithm [74], which provides a simple, robust alternative but runs in $O(n^2m^2)$ time for m polyhedra and n vertices per polyhedron. The method works by checking whether a point lies inside a polygon or polyhedron by using a inner product calculation test. First, all vertices from polyhedron B are tested against polyhedron A, and then all vertices from A are tested for inclusion in B. This approach along with special case treatments is reasonably reliable. But, the computation runs in $O(n^2)$ time where n is the number of vertices per polyhedron.

Hahn [43] used a hierarchical method involving bounding boxes for intersection tests which run in $O(n^2)$ time for each pair of polyhedra where n is the number of vertices for each polyhedron. The algorithm sweeps out the volume of bounding boxes over a small time step to find the exact contact locations. In testing for interference, it takes every edge to check against each polygon and vice versa. Its performance is comparable to Cyrus-Beck clipping algorithm. Our algorithm is a simple and efficient method which runs in expected constant time for each pair of polyhedra, independent of the geometric complexity of each polyhedron. (It would only take $O(m^2)$ time for m polyhedra with any number of vertices per polyhedron.) This provides a significant gain in speed for computer animation, especially for polyhedra with a large number of vertices.

In applications involving dynamic simulations and physical motion, geometric coherence has been utilized to devise algorithms based on local features [3]. This has significantly improved the performance of collision detection algorithms in dynamic environments. Baraff uses cached edges and faces to find a separating plane between two convex polytopes [3]. However, Baraff's assumption to cache the last "witnesses" does not hold when relative displacement of objects between successive time steps are large and when closest features changes, it falls back on a global search; while our method works fast even when there are relatively large displacements of objects and changes in closest features.

As for curved objects, Herzen and etc. [45] have described a general algorithm based on time dependent parametric surfaces. It treats time as an extra dimension and also assumes bounds on derivatives. The algorithm uses subdivision technique in the resulting space and can therefore be slow. A similar method using interval arithmetic and subdivision has been presented for collision detection by Duff [29]. Duff has extended it to dynamic environments as well. However, for commonly used spline patches computing and representing the implicit representations is computationally expensive as stated by Hoffmann [46]. Both algorithms, [29, 46], expect a closed form expression of motion as a function of time. In [70], Pentland and Williams proposes using implicit functions to represent shape and the property of the "inside-outside" functions for collision detection. Besides its restriction to implicits only, this algorithm has a drawback in terms of robustness, as it uses point samples. A detailed explanation of these problems are described in [29]. Baraff has also presented an algorithm for finding closest points between two convex closed objects only [3].

In the related problem of computing the minimum separation between two objects, Gilbert and his collaborators computed the minimum distance between two convex objects with an expected linear time algorithm and used it for collision detection. Our work shares with [38], [39], and [41] the calculation and maintenance of closest points during incremental motion. But whereas [38], [39], and [41] require expected linear time to verify the closest points, we use the properties of convex sets to reduce this check to constant time.

Cameron and Culley further discussed the problem of interpenetration and provided the intersection measurement for the use in a penalty function for robot motion planning [13]. The classical non-linear programming approaches for this problem are presented in [1] and [9]. More recently, Sancheti and Keerthi [77] discussed the computation of proximity between two convex polytopes from a complexity viewpoint, in which the use of quadratic programming is proposed as an alternative to compute the separation and detection problem between two convex objects in $O(n)$ time in a fixed dimension d, where n is the number of vertices of each polytopes. In fact, these techniques are used by researchers Karel Zikan [91], Richard Mastro, etc. at the Boeing Virtual Reality Research Laboratory as a mean of computing the distance between two objects.

Meggido's result in [58] stated that we can solve the problem of minimizing a convex quadratic function, subject to linear constraints in $\mathbb{R}^3$ in linear time by transforming the quadratic programming using an appropriate affine transformation of $\mathbb{R}^3$ (found in constant time) to a linear programming problem. In [60], Megiddo and Tamir have further shown that a large class of separable convex quadratic transportation problems with a fixed number of sources and separable convex quadratic programming with nonnegativity constraints and a fixed number of linear equality constraints can be solved in linear time. Below, we will present a short description of how we can reduce the distance computation problem using quadratic programming to a linear programming problem and solve it in linear time.

The convex optimization problem of computing the distance between two convex polyhedra A and B by quadratic programming can be formulated as follows:

$$\begin{array}{llll}
\text{Minimize} & \|v\|^2 = \|q \Leftrightarrow p\|^2, & \text{s.t. } p \in A, & q \in B \\
\text{subject to} & \sum_{i=1}^{n_1} \lambda_i p_i = p, & \sum_{i=1}^{n_1} \lambda_i = 1, & \lambda_i \geq 0 \\
 & \sum_{j=1}^{n_2} \kappa_j q_j = q = p + v, & \sum_{j=1}^{n_2} \kappa_j = 1, & \kappa_j \geq 0
\end{array}$$

where p_i and q_j are vertices of A and B respectively. The variables are p, v, λ_i's and κ_j's. There are (n_1+n_2+6) constraints: n_1 and n_2 linear constraints from solving λ_i's and κ_j's and 3 linear constraints each from solving the $x, y, z\Leftrightarrow$coordinates of p and v respectively. Since we also have the nonnegativity constraints for p and v, we can displace both A and B to ensure the coordinates of $p \geq 0$ and to find the solutions of 8 systems of equations (in 8 octants) to verify that the constraints, the $x, y, z\Leftrightarrow$ coordinates of $v \geq 0$, are enforced as well. According to the result on separable quadratic programming in [60], this QP problem can be solved in $O(n_1 + n_2)$ time.

Overall, no good collision detection algorithms or distance computation methods are known for general geometric models. Moreover, most of the literature has focussed on collision detection and the separation problem between a pair of objects as compared to handling environments with multiple object models.

1.2 Overview of the Thesis

Chapter 3 through 5 of this thesis deal with algorithms for collision detection, while chapter 6 gives an application of the collision detection algorithms in robot motion planning. We begin in Chapter 2 by describing the basic knowledge necessary to follow the development of this thesis work. The core of the collision detection algorithms lies in Chapter 3, where the incremental distance computation algorithm is described. Since this method is especially tailored toward convex polyhedra, its extension toward non-convex polytopes and the objects with curved boundaries is described in Chapter 4. Chapter 5 gives a treatment on reducing the frequency of collision checks in a large environment where there may be thousands of objects present. Chapter 6 is more or less self contained, and describes an opportunistic global path planner which uses the techniques described in Chapter 3 to construct a one-dimensional skeleton for the purpose of robot motion planning.

Chapter 2 described some computational geometry and modeling concepts which leads to the development of the algorithms presented in this thesis, as well as the object modeling to the input of our algorithms described in this thesis.

Chapter 3 contains the main result of thesis, which is a simple and fast algorithm to compute the distance between two polyhedra by finding the closest features between two convex polyhedra. It utilizes the geometry of polyhedra to establish some local applicability criteria for verifying the closest features, with a preprocessing procedure to reduce each feature's neighbors to a constant size, and thus guarantees expected constant running time for each test. Data from numerous experiments tested on a broad set of convex polyhedra in $\mathbb{R}^3$ show that the expected running time is *constant* for finding closest features when the closest features from the previous time step are known. It is linear in total number of features if no special initialization is done. This technique can be used for dynamic collision detection, physical simulation, planning in three-dimensional space, and other robotics problems. It forms the heart of the motion planning algorithm described in Chapter 6.

In Chapter 4, we will discuss how we can use the incremental distance computation algorithm in Chapter 3 for dynamic collision detection between non-convex polytopes and objects with curved boundary. Since the incremental distance computation algorithm is designed based upon the properties of convex sets, extension to non-convex polytopes using a sub-part hierarchical tree representation will be described in detail here. The later part of this chapter deals with contact determination between geometric models described by curved boundaries and undergoing rigid motion. The set of models include surfaces described by rational spline patches or piecewise algebraic functions. In contrast to previous approaches, we utilize the expected constant time algorithm for tracking the closest features between convex polytopes described in Chapter 3 and local numerical methods to extend the incremental nature to convex curved objects. This approach preserves the coherence between successive motions and exploits the locality in updating their possible contact status. We use local and global algebraic methods for solving polynomial equations and the geometric formulation of the problem to devise efficient algorithms for non-convex curved objects as well, and to determine the exact contact points when collisions occur. Parts of this chapter represent joint work with Dinesh Manocha of the University of North Carolina at Chapel Hill.

Chapter 5 complements the previous chapters by describing two methods which further reduce the frequency of collision checks in an environment with multiple objects moving around. One assumes the knowledge of maximum acceleration and velocity to establish a priority queue sorted by the lower bound on time to collision. The other purely exploits the spatial arrangement without any other information to reduce the number of pairwise interference tests. The rational behind this work comes from the fact that though each pairwise interference test only takes expected constant time, to check for all possible contacts among n objects at all time can be quite time consuming, especially if n is in the range of hundreds or thousands. This n^2 factor in the collision detection computation will dominate the run time, once n increases. Therefore, it is essential to come up with either heuristic approaches or good theoretical algorithms to reduce the n^2 pairwise comparisons.

Chapter 6 is independent of the other chapters of the thesis, and presents a new robot path planning algorithm that constructs a global skeleton of free-space by the incremental local method described in Chapter 3. The curves of the skeleton are the loci of maxima of an artificial potential field that is directly proportional to distance of the robot from obstacles. This method has the advantage of fast convergence of local methods in uncluttered environments, but it also has a deterministic and efficient method of escaping local extremal points of the potential function. We first describe a general roadmap algorithm, for configuration spaces of any dimension, and then describe specific applications of the algorithm for robots with two and three degrees of freedom.

Chapter 2

Background

In this chapter, we will describe some modeling and computational geometry concepts which leads to the development of the algorithms presented later in this thesis, as well as the object modeling for the input to our collision detection algorithms. Some of the materials presented in this chapter can be found in the books by Hoffmann, Preparata and Shamos [46, 71].

The set of models we used include rigid polyhedra and objects with surfaces described by rational spline or piecewise algebraic functions. No deformation of the objects is assumed under motion or external forces. (This may seem a restrictive constraint. However, in general this assumption yields very satisfactory results, unless the object nature is flexible and deformable.)

2.1 Basic Concenpts

We will first review some of the basic concepts and terminologies which are essential to the later development of this thesis work.

2.1.1 Model Representations

In solid and geometric modeling, there are two major representations schemata: B-rep (boundary representation) and CSG (constructive solid geometry). Each has its own advantages and inherent problems.

• **Boundary Representation:** to represent a solid object by describing its surface, such that we have the complete information about the interior and exterior of an object. This representation gives us two types of description: (a) geometric – the parameters needed to describe a face, an edge, and a vertex; (b) topological – the adjacencies and incidences of vertices, edges, and faces. This representation evolves from the description for polyhedra.

• **Constructive Solid Geometry:** to represent a solid object by a set-theoretic Boolean expression of *primitive* objects. The CSG operations include rotation, translation, *regularized union*, *regularized intersection* and *regularized difference*. The CSG standard primitives are the sphere, the cylinder, the cone, the parallelepiped, the triangular prism, and the torus. To create a primitive part, the user needs to specify the dimensions (such as the height, width, and length of a block) of these primitives. Each object has a *local coordinate frame* associated with it. The configuration of each object is expressed by the basic CSG operations (i.e. rotation and translation) to place each object with respect to a global *world coordinate frame*.

Due to the nature of the distance computation algorithm for convex polytopes (presented in Chapter 3), which utilizes the adjacencies and incidences of features as well as the geometric information to describe the geometric embedding relationship, we have chosen the (modified) boundary representation to describe each polyhedron (described in the next section). However, the basic concept of CSG representation is used in constructing the subpart hierarchical tree to describe the nonconvex polyhedral objects, since each convex piece encloses a volume (can be thought of as an object primitive).

2.1.2 Data Structures and Basic Terminology

Given the two major different representation schema, next we will describe the modified *boundary* representation, which we use to represent convex polytope in our algorithm, as well as some basic terminologies describing the geometric relationship between geometric entities.

Let A be a polytope. A is partitioned into features $f_1, \ldots, f_n$ where n is the total number of features, i.e. n = f + e + v where f, e, v stands for the total number of faces, edges, vertices respectively. Each feature (except vertex) is an open subset of an affine plane and does not contain its boundary. This implies the following relationships:

$$\cup f_i = A, i = 1, \ldots, n$$
$$f_i \cap f_j = \emptyset, i \neq j$$

Definition: B is in the *boundary* of F and F is in *coboundary* of B, if and only if B is in the closure of F, i.e. $B \subseteq \overline{F}$ and B has one fewer dimension than F does.

For example, the coboundary of a vertex is the set of edges touching it and the coboundary of an edge is the two faces adjacent to it. The boundary of a face is the set of edges in the closure of the face.

We also adapt the winged edge representation. For those readers who are not familiar with this terminology, please refer to [46]. Here we give a brief description of the winged edge representation:

The edge is oriented by giving two incident vertices (the head and tail). The edge points from tail to head. It has two adjacent faces cobounding it as well. Looking from the the tail end toward the head, the adjacent face lying to the right hand side is what we called the "right face" and similarly for the "left face" (please see Fig.2.1).

Each polyhedron data structure has a field for its features (faces, edges, vertices) and Voronoi cells described below in Sec 2.1.4. Each feature (a vertex, an edge, or a face) is described by its geometric parameters. Its data structure also includes a list of its boundary, coboundary, and *Voronoi regions* (defined later in Sec 2.1.4).

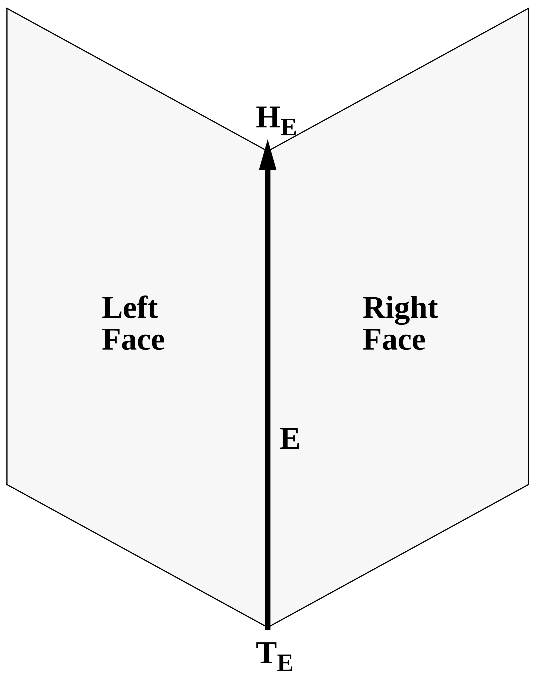


Figure 2.1: A winged edge representation

In addition, we will use the word "above" and "beneath" to describe the relationship between a point and a face. In the homogeneous representation where a point P is presented as a vector $(P_x, P_y, P_z, 1)$ and F's normalized unit outward normal N_F is presented as vector $(a, b, c, \Leftrightarrow d)$, where $\Leftrightarrow d$ is the directional distance from the origin and the vector $n = (a, b, c)$ has the magnitude of 1. The plane which F lies on is described by the plane equation $ax + by + cz + d = 0$.

A point P is *above* $F \Leftrightarrow P \cdot N_F > 0$
A point P is *on* $F \Leftrightarrow P \cdot N_F = 0$
A point P is *beneath* $F \Leftrightarrow P \cdot N_F < 0$

So, if $\Leftrightarrow d > 0$, then the origin lies $\Leftrightarrow d$ units beneath the face F and vice versa. Similarly, let $\vec{e}$ be the vector representing an edge E. Let $H_E = (H_x, H_y, H_z)$ and $T_E = (T_x, T_y, T_z)$. $\vec{e} = H_E \Leftrightarrow T_E = (H_x \Leftrightarrow T_x, H_y \Leftrightarrow T_y, H_z \Leftrightarrow T_Z, 0)$ where H_E and T_E are the head and tail of the edge E respectively.

An edge E points *into* a face $F \Leftrightarrow \vec{e} \cdot N_F < 0$
An edge E is *parallel* to a face $F \Leftrightarrow \vec{e} \cdot N_F = 0$
An edge E points *out* of a face $F \Leftrightarrow \vec{e} \cdot N_F > 0$

2.1.3 Voronoi Diagram

The proximity problem, i.e. "given a set S of N points in $\mathbb{R}^2$, for each point $p_i \in S$ what is the set of points (x, y) in the plane that are closer to p_i than to any other point in S ?", is often answered by the retraction approach in computational geometry. This approach is commonly known as constructing the *Voronoi diagram* of the point set S. This is an important concept which we will revisit in Chapter 3. Here we will give a brief definition of Voronoi diagram.

The intuition to solve the proximity problem in $\mathbb{R}^2$ is to partition the plane into regions, each of these is the set of the points which are closer to a point $p_i \in S$ than any other. If we know this partitioning, then we can solve the problem of proximity directly by a simple query. The partition is based on the set of closest points, e.g. bisectors which have 2 or 3 closest points.

Given two points p_i and p_j, the set of points closer to p_i than to p_j is just the half-plane $H_i(p_i, p_j)$ containing p_i. $H_i(p_i, p_j)$ is defined by the perpendicular bisector to $\overline{p_i p_j}$. The set of points closer to p_i than to any other point is formed by the intersection of at most $N \Leftrightarrow 1$ half-planes, where N is the number of points in the set S. This set of points, V_i, is called the *Voronoi polygon associated with* p_i.

The collection of N Voronoi polygons given the N points in the set S is the *Voronoi diagram*, $Vor(S)$, of the point set S. Every point (x, y) in the plane lies within a Voronoi polygon. If a point $(x, y) \in V(i)$, then p_i is a *nearest point* to the point (x, y). Therefore, the Voronoi diagram contains all the information necessary to solve the proximity problems given a set of points in $\mathbb{R}^2$. A similar idea applies to the same problem in three-dimensional or higher dimensional space [15, 32].

The extension of the Voronoi diagram to higher dimensional *features* (instead of points) is called the generalized Voronoi diagram, i.e. the set of points closest to a *feature*, e.g. that of a polyhedron. In general, the generalized Voronoi diagram has quadratic surface boundaries in it. However, if the objects are convex, then the generalized Voronoi diagram has planar boundaries. This leads to the definition of *Voronoi regions* which will be described next in Sec 2.1.4.

2.1.4 Voronoi Region

A *Voronoi region* associated with a *feature* is a set of points exterior to the polyhedron which are closer to that feature than any other. The Voronoi regions form a partition of space outside the polyhedron according to the closest feature. The collection of Voronoi regions of each polyhedron is the Voronoi diagram of the polyhedron. Note that the Voronoi diagram of a convex polyhedron has linear size and consists of polyhedral regions. A *cell* is the data structure for a Voronoi region. It has a set of constraint planes which bound the Voronoi region with pointers to the neighboring cells (which share a constraint plane with it) in its data structure. If a point lies on a constraint plane, then it is equi-distant from the two features which share this constraint plane in their Voronoi regions.

Using the geometric properties of convex sets, applicability criteria (explained in Sec.3.2) are established based upon the Voronoi regions. If a point P on object A lies inside the Voronoi region of f_B on object B, then f_B is a closest feature to the point P. (More details will be presented in Chapter 3.)

In Chapter 3, we will describe our incremental distance computation algorithm which utilizes the concept of Voronoi regions and the properties of convex polyhedra to perform collision detection in expected constant time. After giving the details of polyhedral model representations, in the next section we will describe more general object modeling to include the class of non-polyhedral objects as well, with emphasis on curved surfaces.

2.2 Object Modeling

The set of objects we consider, besides convex polytopes, includes non-convex objects (like a torus) as well as two dimensional manifolds described using polynomials. The class of parametric and implicit surfaces described in terms of piecewise polynomial equations is currently considered the state of the art for modeling applications [35, 46]. These include free-form surfaces described using spline patches, primitive objects like polyhedra, quadrics surfaces (like cones, ellipsoids), torus and their combinations obtained using CSG operations. For arbitrary curved objects it is possible to obtain reasonably good approximations using B-splines

Most of the earlier animation and simulation systems are restricted to polyhedral models. However, modeling with surfaces bounded by linear boundaries poses a serious restriction in these systems. Our contact determination algorithms for curved objects are applicable on objects described using spline representations (Bézier and B-spline patches) and algebraic surfaces. These representations can be used as primitives for CSG operations as well.

Typically spline patches are described geometrically by their control points, knot vectors and order continuity [35]. The control points have the property that the entire patch lies in the convex hull of these points. The spline patches are represented as piecewise Bézier patches. Although these models are described geometrically using control polytopes, we assume that the Bézier surface has an algebraic formulation in homogeneous coordinates as:

$$\mathbf{F}(s,t) = (X(s,t), Y(s,t), Z(s,t), W(s,t)). \tag{2.1}$$

We also allow the objects to be described implicitly as algebraic surfaces. For examples, the quadric surfaces like spheres, cylinders can be simply described as a degree two algebraic surface. The algebraic surfaces are represented as $f(x,y,z) = 0$.

2.2.1 Motion Description

All objects are defined with respect to a global coordinate system, the *world coordinate frame*. The initial configuration is specified in terms of the origin of the system. As the objects undergo rigid motion, we update their positions using a 4×4 matrix, M, used to represent the rotation as well as translational components of the motion (with respect to the origin). The collision detection algorithm is based only on local features of the polyhedra (or control polytope of spline patches) and does not require the position of the other features to be updated for the purpose of collision detection at every instant.

2.2.2 System of Algebraic Equations

Our algorithm for collision detection for algebraic surface formulates the problem of finding closest points between object models and contact determination in terms of solving a system of algebraic equations. For most instances, we obtain a zero dimensional algebraic system consisting of n equations in n unknowns. However at times, we may have an overconstrained system, where the number of equations is more than the number of unknowns or an underconstrained system, which has infinite number of solutions. We will be using algorithms for solving zero dimensional systems and address how they can be modified to other cases. In particular, we are given a system of n algebraic equations in n unknowns:

$$\begin{aligned} F_1(x_1, x_2, \ldots, x_n) &= 0 \\ &\vdots \\ F_n(x_1, x_2, \ldots, x_n) &= 0 \end{aligned} \tag{2.2}$$

Let their degrees be d_1, d_2, ..., d_n, respectively. We are interested in computing all the solutions in some domain (like all the real solutions to the given system).

Current algorithms for solving polynomial equations can be classified into local and global methods. Local methods like Newton's method or optimization routines need good initial guesses to each solution in the given domain. Their performance is a function of the initial guesses. If we are interested in computing all the real solutions of a system of polynomials, solving equations by local methods requires that we know the number of real solutions to the system of equations and good guesses to these solutions.

The global approaches do not need any initial guesses. They are based on algebraic approaches like resultants, Grőbner bases or purely numerical techniques like the homotopy methods and interval arithmetic. Purely symbolic methods based on resultants and Grőbner bases are rather slow in practice and require multiprecision arithmetic for accurate computations. In the context of finite precision arithmetic, the main approaches are based on resultant and matrix computations [55], continuation methods [64] and interval arithmetic [29, 81]. The recently developed algorithm based on resultants and matrix computations has been shown to be very fast and accurate on many geometric problems and is reasonably simple to implement using linear algebra routines by Manocha [55]. In particular, given a polynomial system the algorithm in [55] reduces the problem of root finding to an eigenvalue problem. Based on the eigenvalue and eigenvectors, all the solutions of the original system are computed. For large systems the matrix tends to be sparse. The order of the matrix, say N, is a function of the algebraic complexity of the system. This is bounded by the Bezout bound of the given system corresponding to the product of the degrees of the equations. In most applications, the equations are sparse and therefore, the order of the resulting matrix is much lower than the Bezout bound. Good algorithms are available for computing the eigenvalues and eigenvectors of a matrix. Their running time can be as high as $O(N^3)$. However, in our applications we are only interested in a few solutions to the given system of equations in a corresponding domain, i.e. real eigenvalues. This corresponds to finding selected eigenvalues of the matrix corresponding to the domain. Algorithms combining this with the sparsity of the matrix are presented in [56] and they work very well in practice.

The global root finders are used in the preprocessing stage. As the objects undergo motion, the problem of collision detection and contact determination is posed in terms of a new algebraic system. However, the new algebraic system is obtained by a slight change in the coefficients of the previous system of equations. The change in coefficients is a function of the motion between successive instances and this is typically small due to temporal and spatial coherence. Since the roots of an algebraic system are a continuous function of the coefficients, the roots change slightly as well. As a result, the new set of roots can be computed using local methods only. We can either use Newton's method to compute the roots of the new set of algebraic equations or inverse power iterations [42] to compute the eigenvalues of the new matrix obtained using resultant formulation.

Chapter 3

An Incremental Distance Computation Algorithm

In this chapter we present a simple and efficient method to compute the distance between two convex polyhedra by finding and tracking the closest points. The method is generally applicable, but is especially well suited to repetitive distance calculation as the objects move in a sequence of small, discrete steps due to its incremental nature. The method works by finding and maintaining a pair of closest features (vertex, edge, or face) on the two polyhedra as the they move. We take advantage of the fact that the closest features change only infrequently as the objects move along finely discretized paths. By preprocessing the polyhedra, we can verify that the closest features have not changed or performed an update to a neighboring feature in expected constant time. Our experiments show that, once initialized, the expected running time of our incremental algorithm is *constant* independent of the complexity of the polyhedra, provided the motion is not abruptly large.

Our method is very straightforward in its conception. We start with a candidate pair of features, one from each polyhedron, and check whether the closest points lie on these features. Since the objects are convex, this is a local test involving only the neighboring features (boundary and coboundary as defined in Sec. 2.1.2) of the candidate features. If the features fail the test, we step to a neighboring feature of one or both candidates, and try again. With some simple preprocessing, we can guarantee that every feature has a constant number of neighboring features. This is how we can verify a closest feature pair in expected constant time.

When a pair of features fails the test, the new pair we choose is guaranteed to be closer than the old one. Usually when the objects move and one of the closest features changes, we can find it after a single iteration. Even if the closest features are changing rapidly, say once per step along the path, our algorithm will take only slightly longer. It is also clear that in any situation the algorithm must terminate in a number of steps at most equal to the number of feature pairs.

This algorithm is a key part of our general planning algorithm, described in Chap.6 That algorithm creates a one-dimensional roadmap of the free space of a robot by tracing out curves of maximal clearance from obstacles. We use the algorithm in this chapter to compute distances and closest points. From there we can easily compute gradients of the distance function in configuration space, and thereby find the direction of the maximal clearance curves.

In addition, this technique is well adapted for dynamic collision detection. This follows naturally from the fact that two objects collide if and only if the distance between them is less than or equal to zero (plus some user defined tolerance). In fact, our approach provide more geometric information than what is necessary, i.e. the distance information and the closest feature pair may be used to compute inter-object forces.

3.1 Closest Feature Pair

Each object is represented as a convex polyhedron, or a union of convex polyhedra. Many real-world objects that have curved surfaces are represented by polyhedral approximations. The accuracy of the approximations can be improved by increasing the resolution or the number of vertices. With our method, there is little or no degradation in performance when the resolution is increased in the convex case. For nonconvex objects, we rely on subdivision into convex pieces, which unfortunately, may take $O((n+r^2)logr)$ time to partition a nondegenerate simple polytope of genus 0, where n is the number of vertices and r is the number of *reflex edges* of the original nonconvex object [23, 2]. In general, a polytope of n vertices can always be partitioned into $O(n^2)$ convex pieces [22].

Given the object representation and data structure for convex polyhedra described in Chapter 2, here we will define the term *closest feature pair* which we quote frequently in our description of the distance computation algorithm for convex polyhedra.

The closest pair of features between two general convex polyhedra is defined as the pair of features which contain the closest points. Let polytopes A and B denote the convex sets in $\mathbb{R}^3$. Assume A and B are closed and bounded, therefore, compact. The distance between objects A and B is the shortest Euclidean distance d_{AB}:

$$d_{AB} = \inf_{p \in A, q \in B} \mid p \Leftrightarrow q \mid$$

and let $P_A \in A$, $P_B \in B$ be such that

$$d_{AB} = \mid P_A \Leftrightarrow P_B \mid$$

then P_A and P_B are a pair of closest points between objects A and B.

For each pair of features (f_A and f_B) from objects A and B, first we find the pair of nearest points (say P_A and P_B) between these *two features*. Then, we check whether these points are a pair of closest points *between A and B*. That is, we need to verify that f_B is truly a closest feature on B to P_A and f_A is truly a closest feature on A to P_B This is verification of whether P_A lies inside the Voronoi region of f_B and whether P_B lies inside the Voronoi region of f_A (please see Fig. 3.1). If either check fails, a new (closer) feature is substituted, and the new pair is checked again. Eventually, we must terminate with a closest pair, since the distance between each candidate feature pair decreases, as the algorithm steps through them.

The test of whether one point lies inside of a Voronoi region of a feature is what we call an "applicability test". In the next section three intuitive geometric applicability tests, which are the essential components of our algorithm, will be described. The overall description of our approach and the completeness proof will be presented in more detail in the following sections.

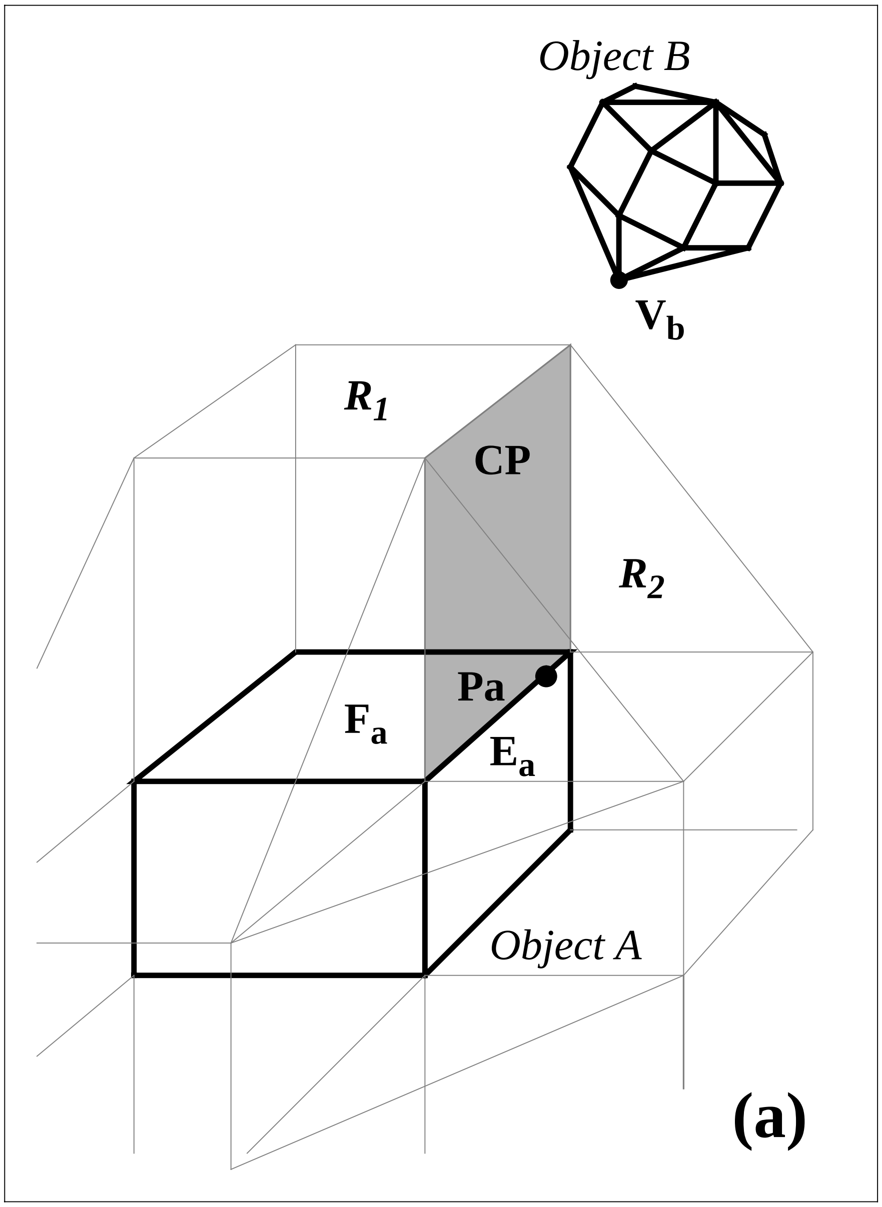


Figure 3.1: Applicability Test: $(F_a, V_b) \to (E_a, V_b)$ since V_b fails the applicability test imposed by the constraint plane CP. R_1 and R_2 are the Voronoi regions of F_a and E_a respectively.

3.2 Applicability Criteria

There are three basic applicability criteria which we use throughout our distance computation algorithm. These are (i) point-vertex, (ii) point-edge, and (iii) point-face applicability conditions. Each of these applicability criteria is equivalent to a membership test which verifies whether *the point* lies in the Voronoi region of the *feature*. If the *nearest points* on two features both lie inside the Voronoi region of the other feature, then the two features are a closest feature pair and the points are closest points between the polyhedra.

3.2.1 Point-Vertex Applicability Criterion

If a vertex V of polyhedron B is truly a closest feature to point P on polyhedron A, then P must lie within the Voronoi region bounded by the constraint planes which are perpendicular to the coboundary of V (the edges touching V). This can be seen from Fig.3.2. If P lies outside the region defined by the constraint planes and hence on the other side of one of the constraint planes, then this implies that there is at least one edge of V's coboundary closer to P than V. This edge is normal to the violated constraint. Therefore, the procedure will walk to the corresponding edge and iteratively call the closest feature test to verify whether the *feature containing* P and the *new edge* are the closest features on the two polyhedra.

3.2.2 Point-Edge Applicability Criterion

As for the point-edge case, if edge E of polyhedron B is really a closest feature to the point P of polyhedron A, then P must lie within the Voronoi region bounded by the four constraint planes of E as shown in Fig.3.3. Let the head and tail of E be H_E and T_E respectively. Two of these planes are perpendicular to E passing through the head H_E and the tail T_E of E. The other two planes contain E and one of the normals to the coboundaries of E (i.e. the right and the left faces of E). If P lies inside this wedge-shaped Voronoi region, the applicability test succeeds. If P fails the test imposed by H_E (or T_E), then the procedure will walk to H_E (or

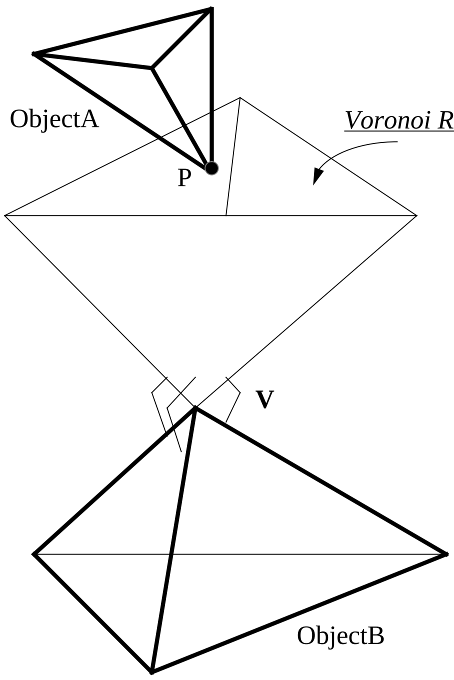


Figure 3.2: Point-Vertex Applicability Criterion

T_E) which must be closer to P and recursively call the general algorithm to verify whether the *new vertex* and the *feature containing* P are the two closest features on two polyhedra respectively. If P fails the applicability test imposed by the right (or left) face, then the procedure will walk to the right (or left) face in the coboundary of E and call the general algorithm recursively to verify whether the *new face* (the right or left face of E) and the *feature containing* P are a pair of closest features.

3.2.3 Point-Face Applicability Criterion

Similarly, if a face F of polyhedron B is actually a closest feature to P on polyhedron A, then P must lie within F's Voronoi region defined by F's prism and above the plane containing F, as shown in Fig.3.4. F's **prism** is the region bounded by the constraint planes which are perpendicular to F and contain the edges in the boundary of F.

First of all, the algorithm checks if P passes the applicability constraints imposed by F's edges in its coboundary. If so, the feature containing P and F

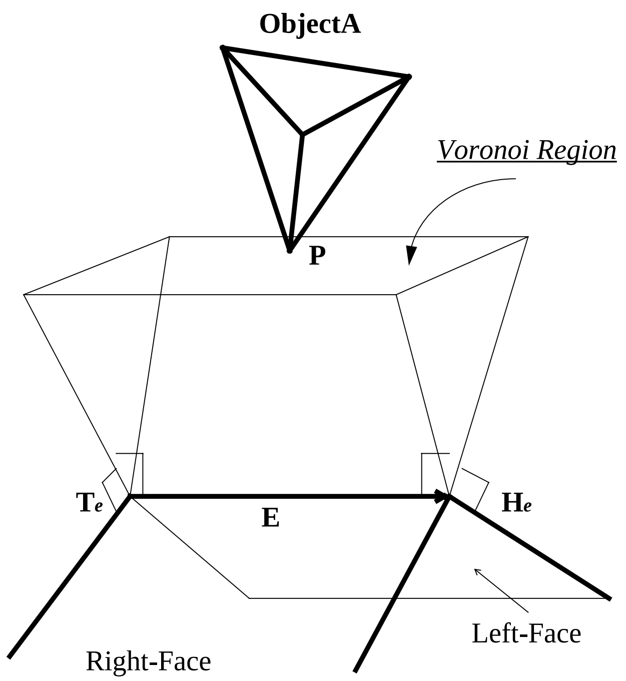


Figure 3.3: Point-Edge Applicability Criterion

are a pair of closest features; else, the procedure will once again walk to the edge corresponding to a failed constraint check and call the general algorithm to check whether the *new edge* in F's boundary (i.e. E_F) and the *feature containing* P are a pair of the closest features.

Next, we need to check whether P lies above F to guarantee that P is not inside the polyhedron B. If P lies beneath F, it implies that there is at least one feature on polyhedron B closer to the feature containing P than F or that collision is possible. In such a case, the nearest points on F and the feature containing P may define a local minimum of distance, and stepping to a neighboring feature of F may increase the distance. Therefore, the procedure will call upon a $O(n)$ routine (where n is the number of features of B) to search for the closest feature on the polyhedron B to the feature containing P and proceed with the general algorithm.

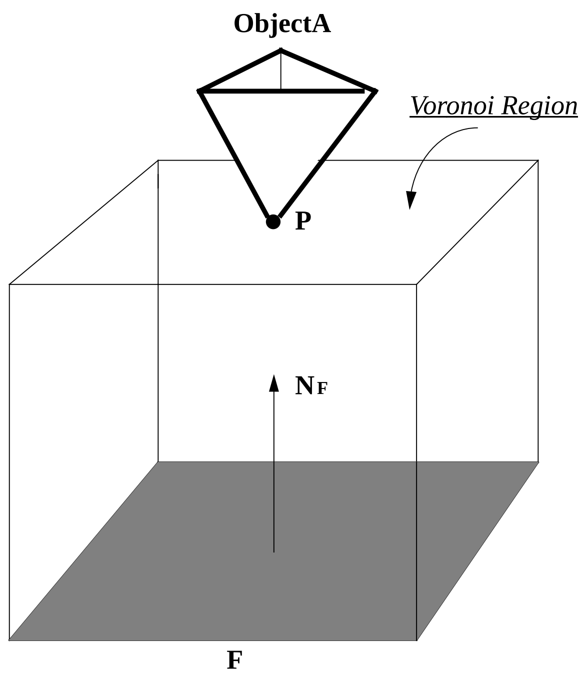


Figure 3.4: Vertex-Face Applicability Criterion

3.2.4 Subdivision Procedure

For vertices of typical convex polyhedra, there are usually three to six edges in the coboundary. Faces of typical polyhedra also have three to six edges in the boundaries. Therefore, frequently the applicability criteria require only a few quick tests for each round. When a face has more than five edges in its boundary or when a vertex has more than five edges in its coboundary, the Voronoi region associated with each feature is preprocessed by subdividing the whole region into smaller cells. That is, we subdivide the prismatic Voronoi region of a face (with more than five edges) into quadrilateral cells and divide the Voronoi region of a vertex (with more than five coboundaries) into pyrmidal cells. After subdivision, each Voronoi cell is defined by 4 or 5 constraint planes in its boundary or coboundary. Fig.3.5 shows how this can be done on a vertex's Voronoi region with 8 constraint planes and a face's Voronoi region with 8 constraint planes. This subdivision procedure is a simple algorithm which can be done in linear time as part of preprocessing, and it guarantees that when the algorithm starts, each Voronoi region has a constant number of constraint planes. Consequently, the three applicability tests described above run in *constant time*.

3.2.5 Implementation Issues

In order to minimize online computation time, we do all the subdivision procedures and compute all the Voronoi regions first in a one-time computational effort. Therefore, as the algorithm steps through each iteration, all the applicability tests would look uniformly alike — all as "point-cell applicability test", with a slight difference to the point-face applicability criterion. There will no longer be any distinction among the three basic applicability criteria.

Each feature is an open subset of an affine plane. Therefore, end points are not consider part of an edge. This is done to simplify the proof of convergence. Though we use the winged edge representation for our implementation, all the edges are not deliberately oriented as in the winged edge representation.

We also use hysteresis to avoid cycling in degenerate cases when there is more than one feature pair separated at approximately the same distance. We achieve this by making the Voronoi regions overlap each other with small and "flared" displacements. To illustrate the "flared Voronoi cells", we demonstrate them by an example in Fig. 3.6. Note that all the constraint planes are tilted outward by a small angular displacement as well as infinitesimal translational displacement. The angular displacement is used to ensure the overlap of adjacent regions. So, when a point falls on the borderline case, it will not flip back and forth between the two adjacent Voronoi regions. The small translational displacement in addition to angular displacement is needed for numerical problem arising from point coordinate transformations. For an example, if we transform a point p by a homogeneous matrix T and transform it back by T^{-1}, we will not get the same point p but p' with small differences in each of its coordinate, i.e. $T^{-1}(Tp)/neqp$.

All faces are convex, *but* because of subdivision they can be coplanar. Where the faces are coplanar, the Voronoi regions of the edges between them is null and face constraints point directly to the next Voronoi region associated with their neighboring

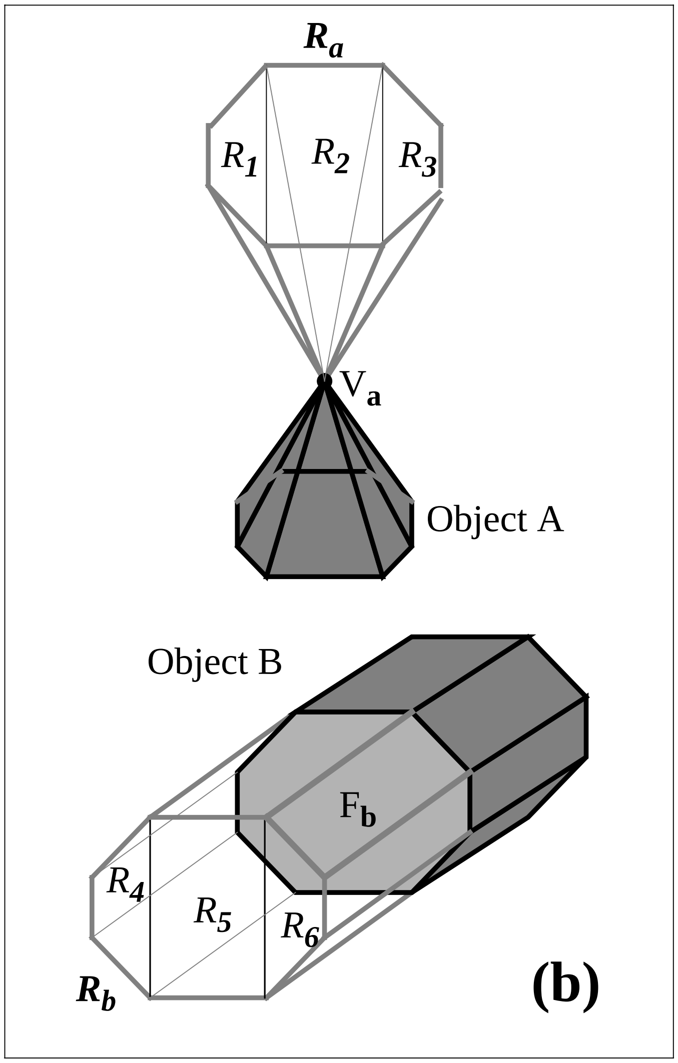


Figure 3.5: Preprocessing of a vertex's conical applicability region and a face's cylindrical applicability region

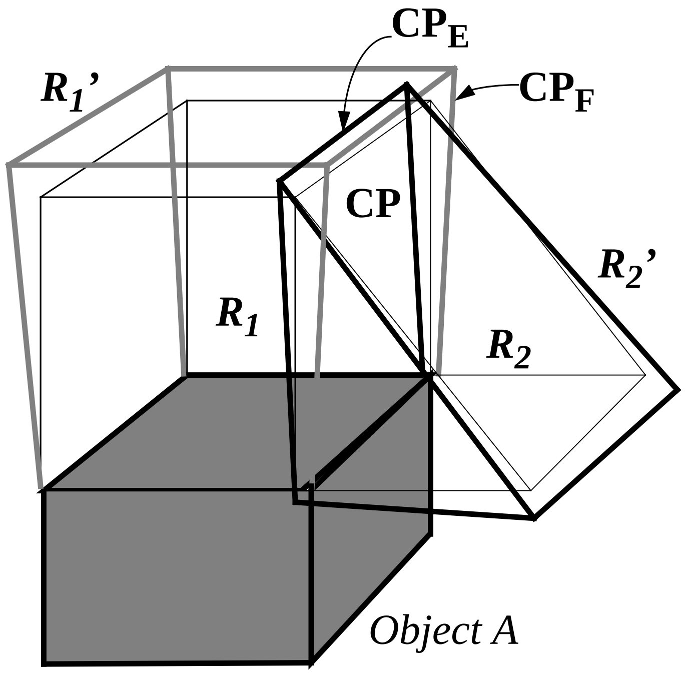


Figure 3.6: An example of Flared Voronoi Cells: CP_F corresponds to the flared constraint place CP of a face and CP_E corresponds to the flared constraint plane CP of an edge. R_1' and R_2' are the *flared* Voronoi regions of unperturbed R_1 and R_2. Note that they overlap each other.

coplanar faces.

In addition, the order of the applicability test in each case does not affect the final result of the algorithm. However, to achieve faster convergence, we look for the constraint plane which is violated the most and its associated neighboring feature to continue the verification process in our implementation.

We can also calculate how frequently we need to check for collision by taking into consideration the maximum magnitude of acceleration and velocity and initial separation. We use a priority queue based on the lower bound on time to collision to detect possible intersection at a timely fashion. Here the step size is adaptively set according to the lower bound on time to collision we calculate. For more information on this approach, please see Chapter 5 of this thesis.

In the next section, we will show how these applicability conditions are used to *update* a pair of closest features between two convex polyhedra in expected *constant time*.

3.3 The Algorithm

Given a pair of features, there are altogether 6 possible cases that we need to consider: (1) a pair of vertices, (2) a vertex and an edge, (3) a vertex and a face, (4) a pair of edges, (5) an edge and a face, and (6) two faces. In general, the case of two faces or edge-face rarely happens. However, in applications such as path planning we may end up moving along maximal clearance paths which keep two faces parallel, or an edge parallel to a face. It is important to be able to detect when we have such a degenerate case.

3.3.1 Description of the Overall Approach

Given a pair of features f_A and f_B each from convex polyhedra A and B respectively, except for cases (1), (5) and (6), we begin by computing the nearest points between f_A and f_B. The details for computing these nearest points between f_A and f_B are rather trivial, thus omitted here (please refer to Appendix A if necessary).

However, we would like to reinforce one point. We do not assume infinitely long edges when we compute these nearest points. The nearest points computed given two features are the actual points on the features, not some virtual points on infinitely extended edges, i.e. nearest points between edges may be their endpoints.

(1) If the features are a pair of vertices, V_A and V_B, then both V_A and V_B have to satisfy the point-vertex applicability conditions imposed by each other, in order for them to be the closest features. If V_A fails the point-vertex applicability test imposed by V_B or vice versa, then the algorithm will return a new pair of features, say V_A and E_B (the edge whose corresponding constraint was violated in the point-vertex applicability test), then continue verifying the new feature pair until it finds a closest pair.

(2) Given a vertex V_A and an edge E_B from A and B respectively, the algorithm will check whether the vertex V_A satisfies the point-edge applicability conditions imposed by the edge E_B *and* whether the nearest point P_E on the edge E_B to V_A satisfies the point-vertex applicability conditions imposed by the vertex V_A. If both checks return "true", then V_A and E_B are the closest features. Otherwise, a corresponding new pair of features (depending on which test failed) will be returned and the algorithm will continue to walk closer and closer toward a pair of closest features.

(3) For the case of a vertex V_A and a face F_B, both of the point-face applicability tests imposed by the face F_B to the vertex V_A *and* the point-vertex applicability criterion by V_A to the nearest point P_F on the face must be satisfied for this pair of features to qualify as a closest-feature pair. Otherwise, a new pair of features will be returned and the algorithm will be called again until the closest-feature pair is found.

(4) Similarly, given a pair of edges E_A and E_B as inputs, if their nearest points P_A and P_B satisfy the point-edge applicability criterion imposed by E_B and E_A, then they are a pair of closest features between two polyhedra. If not, one of the edges will be changed to a neighboring vertex or a face and the verification process will be done again on the new pair of features.

(5) When a given pair of features is an edge E_A and a face F_B, we first need to decide

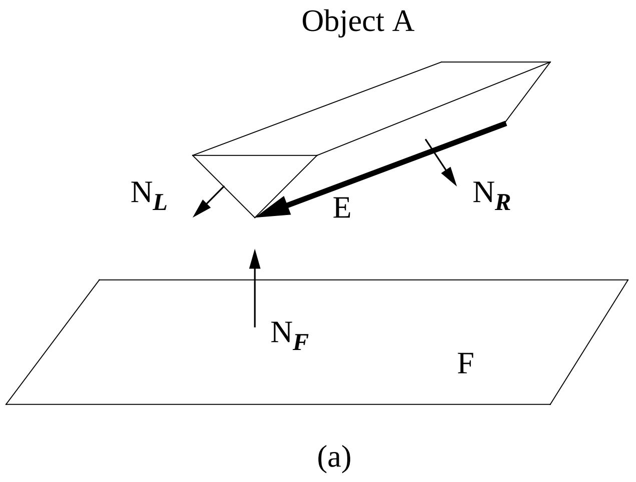
whether the edge is parallel to the face. If it is not, then either one of two edge's endpoints and the face, *or* the edge and some other edge bounding the face will be the next candidate pair. The former case occurs when the head or tail of the edge satisfies the point-face applicability condition imposed by the face, and when one endpoint of E is closer to the plane containing F than the other endpoint of E. Otherwise, the edge E and the closest edge E_F on the face's boundary to E will be returned as the next candidate features.

If the edge and the face are parallel, then they are the closest features provided three conditions are met. (i) The edge must cut the "applicability prism" $Prism_F$ which is the region bounded by the constraint planes perpendicular to F_B and passing through F_B's boundary (or the region swept out by F_B along its normal direction), that is $E \cap Prism_F \neq \emptyset$ (ii) the face normal must lie between the face normals of the faces bounding the edge (see Fig. 3.7, Sec. 3.4, and the pseudo code in Appendix B), (iii) the edge must lie above the face. There are several other situations which may occur, please see Appendix B or the proof of completeness in Sec. 3.4 for a detailed treatment of edge-face case.

(6) In the rare occasion when two faces F_A and F_B are given as inputs, the algorithm first has to decide if they are parallel. If they are, it will invoke an overlap-checking subroutine which runs in linear time in the total number of edges of F_A and F_B. (Note: it actually runs in constant time after we preprocess the polyhedra, since now all the faces have constant size of boundaries.) If they are both parallel, F_A's projection down onto the face F_B overlaps F_B and each face is above the other face relative to their outward normals, then they are in fact the closest features.

But, if they are parallel yet not overlapping, then we use a linear time procedure [71] to find the closest pair of edges, say E_A and E_B between F_A and F_B, then invoke the algorithm again with this new pair of candidates (E_A, E_B).

If F_A and F_B are not parallel, then first we find the closest feature (either a vertex or an edge) f_A of F_A to the plane containing F_B and vice versa. Then, we check if this closest feature f_A satisfy the applicability constraint imposed by F_B. If so, the new candidate pair will be this closest feature f_A and F_B; otherwise, we



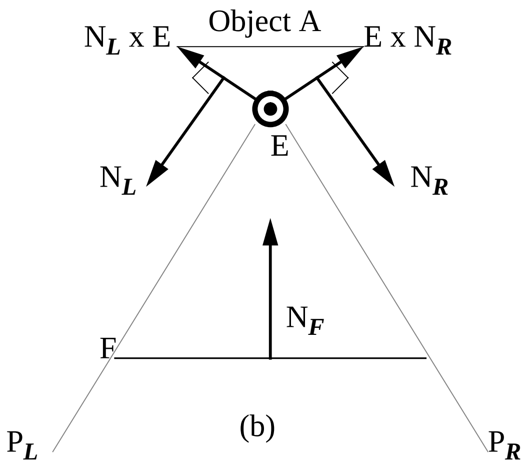


Figure 3.7: (a) An overhead view of an edge lying above a face (b) A side view of the face outward normal bounded by the two outward normals of the edge's left and right faces.

enumerate over all edge-pairs between two faces to find a closest pair of edges and proceed from there.

If one face lies on the "far side" of the other (where the face is beneath the other face relative to their face outward normals) or vice versa, then the algorithm invokes a linear-time routine to step away from this local minimum of distance, and provides a new pair of features much closer than the previous pair. Please refer to Appendix B for a complete description of face-face case.

3.3.2 Geometric Subroutines

In this section, we will present some algorithms for geometric operations on polytopes. They are used in our implementation of the algorithms described in this thesis.

• **Edge & Face's Prism Intersection:**

To test for the intersection of a *parallel* edge E_A to F_B within a prism $Prism_F$ swept out by a face F_B along its face normal direction N_F, we check whether portion of this edge lies within the interior region of $Prism_F$. This region can be visualized as the space bounded by n planes which are perpendicular to the face F_B and passing through the n edges in its boundary.

Suppose F_i is a face of the prismatic region $Prism_F$ and perpendicular to the face F_B and passing through the edge E_i bounding F_B. To test for E_A's intersection within the prism $Prism_F$, we need to check whether E_A intersects some face F_i of R_F.

We can perform this test with easy and efficient implementation[1] of this intersection test. We first parameterize the edge E_A between 0 and l where l is the length of E_A. The idea is to intersect the intervals of the edge E_A contained in the *exterior*[2] of each half-space with each constraint plane in the prism of a face F_B.

[1]suggested by Brian Mirtich

[2]We define the exterior half-space of each constraint plane to be the half-space where all the points satisfy the applicability constraints imposed by F_B's boundary. The "point-cell-checkp" routine in the pseudo code can clarify all the ambiguity. Please refer to Appendix B if necessary.

First, we decide whether the edge E_A points inward or outward of each hyperplane F_i of the region by taking the inner product of E_A's unit directional vector with F_i's outward normal N_{F_i}.

Then, we can decide which portion of E_A intersect the *exterior* half-space of F_i by taking another inner product of the head of E_A and F_i's outward normal normalized by the inner product of E_A and N_{F_i} to calculate the relative "inclusion" factor of E_A inside of the prism region. The relative (beginning and ending) inclusion factors should be in the range of 0 and l for an indication of an edge-face's prism intersection.

Pseudo code for this approach just described is listed in Appendix B. This geometric test is used in our implementation of distance computation algorithm as a subroutine to check whether a given edge intersects the Voronoi prism of a given face to test their eligibilities be a pair of closest features.

• **Face & Face's Prisim Intersection:**

To check if two convex polygons A and B are overlapping, one of the basic approaches is to verify if there is some vertex of A lying on the "inside" of *all* edges in the boundary of B or vice versa. The "inside" refers to the *interior half-plane* of the edge, not necessary the interior of the polygon. A n-sided convex polygon is formed by the intersection of interior half-planes of n edges. We can use the fact that two polygons do no overlap if there exists a separating line passing through one edge in the boundary of either face. The polygons overlap, if and only if *none* of the edges in the boundary of both polygons forms a separating line.

For our own application, we are interested to verify whether a face A's projection down to the plane containing the face B overlaps with the face B, where A and B are *parallel* and in $\mathbb{R}^3$. To find a separating line between two such polygons in $\mathbb{R}^3$, we check if there exists a supporting plane passing through an edge in the boundary of one polygon such that all vertices of the other polygon lies in the "exterior half-space" of the supporting plane. The supporting plane is perpendicular the polygon and containing an edge in the polygon's boundary. This approach runs in $O(N^2)$ time where N is the number of edges of each polygon.

An elegant approach which runs in $O(M + N)$ can be found in [66, 71], where the two polygons A and B each has M, N vertices respectively. This approach not only determines whether two convex polygon intersect or not, it also finds the intersections between them. The basic approach is to "advance" along the edges of A and B to find the "internal chain" of vertices which form the intersection polygon. Please refer to [66, 71] for all the details.

3.3.3 Analysis of the Algorithm

A careful study of all of the above checks shows that they all take time in proportion to the size of the boundary and coboundary of each feature. Therefore, after subdivision preprocessing, all local tests run in constant time since each feature now has constant number of neighboring features. The only exception to this is when f_A is a face, and f_B lies under the plane containing f_A (or vise versa). In this case, we can not use a local feature change, because this may lead to the procedure getting stuck in a loop. Geometrically, we are moving around the local minimum of the distance function, in which we may become trapped. When this situation occurs, we search among all the features of object A to find a closest feature to the f_B[3]. This is not a constant time step, but note that it is impossible for the algorithm to move to such an opposing face once it is initialized and given a reasonable step size (not more than a few degrees of rotation). So this situation can only occur when the algorithm is first called on an arbitrary pair of features.

The algorithm can take any arbitrary pair of features of two polyhedra and find a true pair of closest features by iteratively checking and changing features. In this case, the running time is proportional to the number of feature pairs traversed in this process. It is not more than the product of the numbers of features of the two polyhedra, because the distance between feature pairs must always decrease when a switch is made, which makes cycling impossible. Empirically, it seems to be not worse than linear when started from an arbitrary pair of features. However, once

[3]We can also use a very expensive *preprocessing* step to find a feature on the opposite side of the polyhedron of the current face. But, this involves searching for all facets whose the outward normals pointing in the opposite direction, i.e. the dot product of the two face outward normals is negative.

it finds the closest pair of features *or* a pair in their vicinity, it only takes expected constant time to update a closest feature pair as the two objects translate and rotate in three-space. The overall computational time is shorter in comparison with other algorithms available at the present time.

If the two objects are just touching or intersecting, it gives an error message to indicate collision and terminates the procedure with the contacting-feature pair as returned values. A pseudo code is given in Appendix B. The completeness proof of this algorithm is presented in Sec. 3.4 and it should give readers a better insight to the algorithm.

3.3.4 Expected Running Time

Our approach works well by taking advantage of "continuity of motion" or geometric coherence between discrete time steps. When the motion is abruptly large, the performance doesn't achieve the potential of this algorithm though it still typically finds a pair of closest features in sub-linear time. One of the frequently asked questions is how to choose step size so that we can preserve the geometric coherence to exploit the locality of closest feature pairs in small discrete steps. This question can be addressed indirectly by the following analysis:

For relatively spherical objects, the expected running time t is proportional to $\sqrt{n}$. For cylindrical objects, the expected running time t is proportional to n, where n is the number of faces (or vertices) per polyhedron.

This is derived from the following reasoning: Given a tessellated sphere which has N faces around its equator, when the objects are rotated X degrees, it takes roughly $\frac{k*X}{360/N} = K*N$ steps to update the closest feature pair, where k and K are the adjustment constants. And the total number of features (vertices) for the sphere is $O(N^2)$. Therefore, the number of steps (or accumulated running time) is proportional to $\sqrt{n}$, where $n = O(N^2)$ is the total number of features (vertices or faces). A similar argument holds for the cylindrical objects.

With this observation, we can set the step size to preserve the coherence at each step. However, it is not necessary since the overall runtime is governed by the number of intermediate feature pairs traversed in the process and the algorithm can always find the pair of closest features, given any step size.

3.4 Proof of Completeness

The algorithm takes any pair of features on two polyhedra and returns a pair of closest features by iteratively checking and changing features. Through each applicability test, the algorithm steps closer and closer to a pair of closest features. That is, after each applicability check, the distance between the updated feature-pair is smaller than that of the previous feature-pair if a switch of feature-pairs is made. To verify if a given pair of features is the closest pair, each step takes running time proportional to the number of neighboring features (boundaries and/or coboundaries of the features). Since each feature has a constant number of neighboring features, each verification step takes only constant time in all cases except when a feature lies beneath a face and inside the face's Voronoi region (this corresponds to a local minimum of distance function). Thus we call this pair of features as an "exceptional" feature pair. This situation only occurs after a large motion in a single step when the actual closest feature of A is on the other (or opposite) side of the last closest feature on A.

Let n be the number of features from each object. If the algorithm never encounters a local minimum, the algorithm will return a pair of closest feature in $O(n^2)$ time since there are at most n^2 feature-pairs to verify using constant-time applicability criteria and other subroutines. If the algorithm runs into an exceptional pair of features, then it is necessary to invoke a linear-time routine to continue the verification process. We will show later that the algorithm calls the linear-time routine at most $2n$ times. Hence, the algorithm converges in $O(n^2)$ time.

Lemma 1 Each applicability test and closest feature-pair verifying subroutine will necessarily return a new pair of features closer in distance than the previous pair when a switch of feature-pairs is made.

Proof:

The algorithm works by finding a pair of closest features. There are two categories: (1) non-degenerate cases – vertex-vertex, vertex-edge, and vertex-face (2) degenerate cases – edge-edge, edge-face, and face-face.

For non-degenerate and edge-edge cases, the algorithm first takes any pair of features f_A and f_B, then find the nearest points P_A and P_B between them. Next, it checks each nearest point against the applicability constraints of the other feature. The distance between two features is the distance between the nearest points of two features, which is the same as the distance between one nearest point to the other feature. Therefore, if we can find another feature f'_A closer to P_B or another feature f'_B closer to P_A, we have shown the distance between each candidate feature pair is less after the switch.

Please note that edge-edge is a special degenerate case, since it can possibly have either one pair or many pairs of nearest points. When a pair of edges has one pair of nearest points, it is considered as all other non-degenerate cases. When both edges are parallel, they can possibly have infinitely many pairs of nearest points. In such a case, we still treat it as the other non-degenerate cases by choosing the nearest points to be the midpoints of the line segments which contains the entire sets of nearest points between two edges. In this manner, the edge-edge degenerate case can be easily treated as in the other non-degenerate cases and the proof of completeness applies here as well.

The other two degenerate cases must be treated differently since they may contain infinitely many closest point pairs. We would like to recall that edges and faces are treated as open subsets of lines and planes respectively. That is, the edge is considered as the set of points between two endpoints of the edge, excluding the head and the tail of the edge; similarly, a face is the set of points interior to its boundary (excluding the edges and vertices in its boundary). This guarantees that when a switch of features is made, the new features are *strictly* closer.

We will first show that each applicability test returns a pair of candidate features closer in distance than the previous pair when a switch of feature pairs is made. Then, we show the closest feature verifying subroutines for both edge-face and face-face cases necessarily return a closer pair of features when a switch of feature pairs is made.

(I) Point-Vertex Applicability Criterion

If vertex V is truly the closest feature to point P, then P must lie within the region bounded by the planes which are perpendicular to the coboundaries of V (the edges touching V). If so, the shortest distance between P and the other object is clearly the distance between P and V (by the definition of Voronoi region) . When P lies outside one of the plane boundaries, say C_{E_1}, the constraint plane of an edge E_1 touching V, then there is at least one point on E_1 closer to P than V itself, i.e. the distance between the edge E_1 (whose constraint is violated) and P is shorter than the distance between V and P. This can be seen from Fig.3.8. When a point P lies directly on the bisector C_{E_1} between V and E_1, then P is equi-distant from both features; else, P is closer to V if P is inside of V's Voronoi region, and vice versa. Therefore, each Point-Vertex Applicability test is guaranteed to generate a pair of features that is closer in distance than the previous pair (which fails the Point-Vertex Applicability test), when a switch of feature pairs occurs.

(II) Point-Edge Applicability Criterion

If edge E is really the closest feature to point P, then P must lie within the region bounded by the four constraint planes generated by the coboundary (the left and right faces) and boundary (the two end points) of E. This is the Voronoi region of E. If this is the case, the shortest distance between P and the other object is the distance between P and E (see Fig.3.3 and Sec. 3.2.2 for the construction of point-edge constraint planes.) When P fails an applicability constraint, say C_F, imposed by the left (or right) face of the edge E, there is at least one point on the corresponding face closer to P than E, i.e. the distance between P and the corresponding face is shorter than the distance between P and E (as in Fig.3.9). If P fails the applicability criterion C_V imposed by the head (or tail) of E, then P is closer to the corresponding endpoint than to E itself (as in Fig.3.10). Therefore, the distance between the new pair of features is guaranteed to be shorter than that of the previous pair which fails

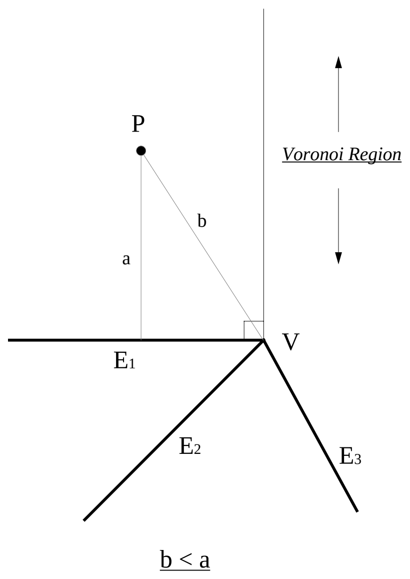


Figure 3.8: A Side View for Point-Vertex Applicability Criterion Proof

the Point-Edge Applicability Criterion, when a switch of feature pairs is made.

(III) Point-Face Applicability Criterion

If the face F is actually the closest feature to a point P, then P must lie within the prism bounded by the constraint planes which are orthogonal to F and containing the edges in F's boundary and above F (by the definition of Voronoi region and point-face applicability criterion in Sec. 3.2.3). If P fails one applicability constraint, say C_{E_1} imposed by one of F's edges, say E_1, then E_1 is closer to P than F itself.

When a point lies beneath a face F and within the prism bounded by other constraint planes, it is possible that P (and of course, the object A which contains P) lies inside of the object B containing F. But, it is also possible that P lies out side of B and on the other side of B from F. This corresponds to a local minimum of distance function. It requires a linear-time routine to step to the next feature-pair. The linear-time routine enumerates all features on the object B and searches for the closest one to the feature f_A containing P. This necessarily decreases the distance

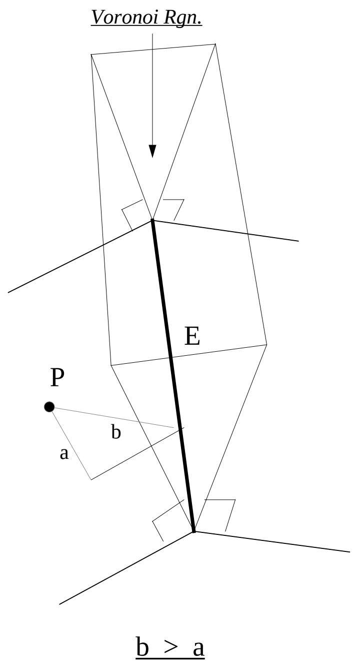


Figure 3.9: An Overhead View for Point-Edge Applicability Criterion Proof

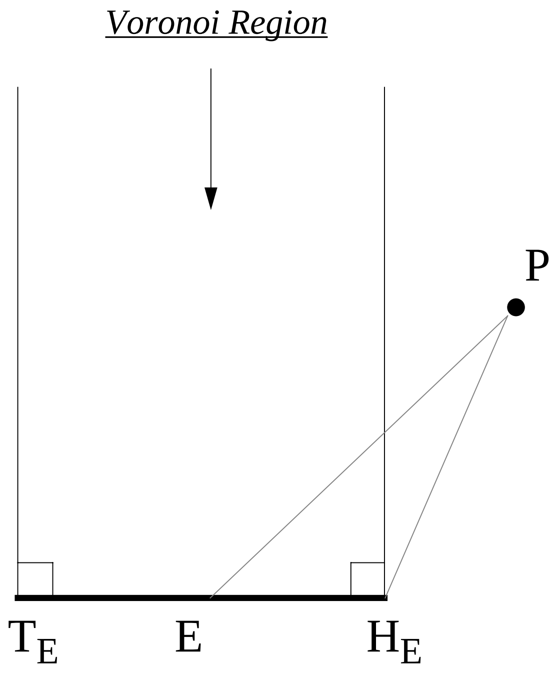


Figure 3.10: A Side View for Point-Edge Applicability Criterion Proof

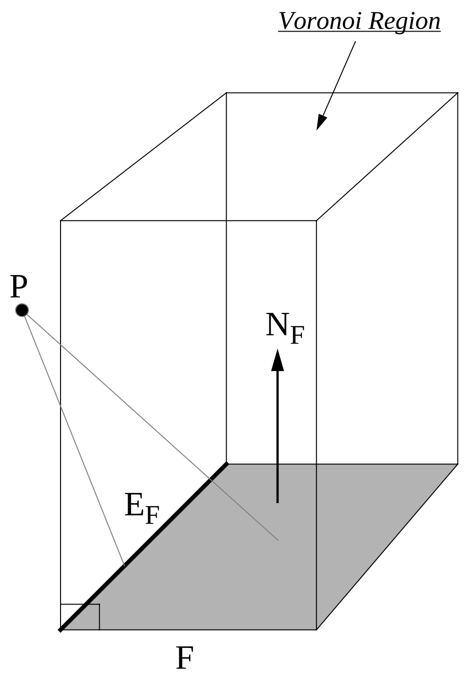


Figure 3.11: A Side View for Point-Face Applicability Criterion Proof

between the closest feature candidates base upon the minimum distance information during the search.

Therefore, we can conclude that the distance decreases when a switch of feature pairs is made as the result of Point-Face Applicability Test.

(IV) Verifying Subroutines in Edge-Face Case

With the exception of edge-edge (as mentioned earlier in the beginning of the proof), only edge-face and face-face may have multiple pairs of closest points and require a more complex treatments. Whereas, the closest feature verification for the other non-degenerate cases and edge-edge solely depends on point-vertex, point-edge, and point-face applicability tests which we have already studied. Now we will examine each of these two cases closely.

• *Edge-Face* (E, F)

The indentation corresponds to the level of nested loops in the pseudo code (Appendix B). Please cross-reference if necessary.

Given an edge E and a face F, the algorithm first checks **IF E AND F ARE PARALLEL**, i.e. the two endpoints H_E and T_E are equi-distant from the face F (i.e. their signed distance is the same). The *signed* distance between H_E and F can be computed easily by $H_E \cdot N_F$ where N_F is F's unit outward normal and similarly for the signed distance between F and H_T.

THEN, when the head H_E and tail T_E of the edge E are equi-distant from the face F, the subroutine checks **IF E INTERSECTS F'S PRISM REGION**, (i.e. the region bounded by F's constraint planes constructed from the edges in F's boundary).

> **THEN**, if E intersects the prism region of F, the algorithm checks **IF THE HEAD H_E (OR TAIL T_E) OF E LIES ABOVE THE FACE F.**
>
> > **THEN**, it will check **IF THE FACE OUTWARD NORMAL N_F IS "BOUNDED" BY THE NORMALS OF THE EDGE'S LEFT AND RIGHT FACES**, say N_L and N_R respectively, to verify that the face F lies inside the Voronoi region of the edge E, not in the Voronoi region of E's left or right face (F_L or F_R). This is done by checking if $(N_L \times \vec{E}) \cdot N_F > 0$ and $(\vec{E} \times N_R) \cdot N_F > 0$, where N_F is the outward normal of the face F, $\vec{E}$ is the edge vector (as described in Sec. 2.1.2), and N_R, N_L denote the outward normal of the edge's right and left face respectively.
> >
> > > **THEN**, the edge E and the face F will be returned as the closest feature pair, since E and F lie within the Voronoi region of each other.
> > >
> > > **ELSE**, it returns the right or left face (F_R or F_L) of E, which ever is closer to F.
> >
> > **ELSE**, when the head H_E (implies E also) lies beneath F, then it will invoke a subroutine *e-find-min* which finds the closest feature on the polyhedron B containing F to E by enumerating all the features on B. Then the algorithm will return E and this new feature f_B as the next candidates for the closest feature pair verification.
>
> **ELSE**, E does not intersect the prism defined by F's boundary edges, then there is at least one edge of F's boundary which is closer to E

than the interior of F is to E (by the definition of Voronoi region). So, when the test is failed, it calls a constant time routine (which takes time proportional to the constant number of face's boundary) and returns the edge E and the closest edge or vertex, say f_F, on the F's boundary to E.

ELSE, the algorithm checks **IF ONE END POINT LIES ABOVE THE FACE** F **AND THE OTHER LIES BENEATH** F.

> **THEN**, if so there is one edge on F's boundary which is closer to E than the interior of F. So, the algorithm will find the closest edge or vertex f_B, which has the minimum distance to E among all edges and vertices in F's boundary, and return (E, f_B) as the next candidate pair.
>
> **ELSE**, one endpoint H_E (or T_E) of E is closer (the magnitude $\mid V_E \cdot N_F \mid$ instead of the signed distance is used for comparison here) to F than the other endpoint, the algorithm will check **IF THE CLOSER ENDPOINT** V_E **LIES INSIDE OF** F**'s PRISM REGION** (bounded by constraint planes generated by edges in F's boundary), $Cell_F$, as in the vertex-face case.
>
> > **THEN**, if closer endpoint V_E lies inside of F's prism region, then the algorithm will next verify **IF THIS ENDPOINT LIES ABOVE THE FACE** F (to detect a local minimum of distance function).
> >
> > > **THEN**, if V_E (the closer endpoint to F) is above F the algorithm next verifies **IF THE CLOSEST POINT** P_F **ON** F **TO** V_E **SATISFIES** V_E**'S APPLICABILITY CONSTRAINTS**.
> > >
> > > > **THEN**, if so the algorithm returns V_E and F as the closest feature pair.
> > > >
> > > > **ELSE** some constraint plane in V_E's Voronoi cell $Cell_{V_E}$ is violated. This constraint plane is generated by an edge, say E', in V_E's coboundary. This edge E' is closer to F, so the algorithm will return (E', F) as the next candidate pair.
> > >
> > > **ELSE**, if the closer endpoint V_E of E lies beneath F, then the algorithm finds the closest feature f_B on the polyhedron B to V_E The algorithm returns (V_E, f_B) as the next candidate feature pair.

ELSE, if the closer endpoint V_E of E to F lies outside of F's prism region, then there is one edge on F's boundary which is closer to E than the interior of F. So, the algorithm will find the closest edge or vertex f_B, which has the minimum distance to E among all edges and vertices in F's boundary, and return $(E,\, f_B)$ as the next candidate pair.

(V) Verifying Subroutines in Face-Face Case

• *Face-Face* $(F_A,\, F_B)$

The numbering system (n) in this proof is annotated for easy reference within the proof. The indentation corresponds to the level of nested loops in the pseudo code (Appendix B).

(0) Given two faces, F_A and F_B, the algorithm first checks **IF THEY ARE PARALLEL**.

(1) **THEN**, if they are parallel, it invokes a subroutines which runs in constant time (proportional to the product of numbers of boundaries between two faces, which is a constant after subdivision) to check **IF THE ORTHOGONAL PROJECTION OF F_A ON TO THE PLANE OF F_B OVERLAPS** F_B (i.e. F_A intersects the prism region of F_B).

(2) **THEN**, if they overlapped, the algorithm next checks **IF** P_A (implies F_A as well, since two faces are parallel) **LIES ABOVE** F_B, where P_A is one of the points on F_A and its projection down to the F_B is in the overlap region.

(3) **THEN**, if P_A (thus F_A) lies above F_B, the algorithm next checks **IF** P_B **(THUS** F_B**) ALSO LIES ABOVE** F_A, where P_B is one of the points and its projection down to the F_B is in the overlap region.

(4) **THEN**, if P_B (thus F_B) is above F_A and vice versa, we have a pair of closest points $(P_A,\, P_B)$ satisfy the respective applicability constraints of F_B and F_A. So, P_A and P_B are the closest points on A and B; and the

features containing them, F_A and F_B, will be returned as the closest features.

(5) **ELSE**, if P_B (thus F_B) lies beneath F_A, then the algorithm finds the closest feature f_A on polyhedron A containing F_A to F_B by enumerating all features of A to search for the feature which has minimum distance to F_B. So, the new pair of features (f_A, F_B) is guaranteed to be closer than (F_A, F_B).

(6) **ELSE**, if P_A (thus F_A) lies beneath F_B, the algorithm finds the closest feature f_B on polyhedron B containing F_B to F_A and returns (F_A, f_B) as the next candidate feature pair. This pair of new features has to be closer than F_A to F_B since f_B has the shortest distance among all features on B to F_A.

(7) **ELSE**, if F_A and F_B are parallel but the projection of F_A does not overlap F_B, a subroutine enumerates all possible combination of edge-pairs (one from the boundary of each face respectively) and computes the distance between each edge-pair to find a pair of closest edges which has the minimum distance among all. This new pair of edges (E_A, E_B) is guaranteed to be closer than the two faces, since two faces F_A and F_B do not contain their boundaries.

(8) **ELSE**, if F_A and F_B are not parallel, this implies there is at least one vertex or edge on F_A closer to F_B than F_A to F_B *or* vice versa. So, the algorithm first finds the closest vertex or edge on F_A to the plane containing F_B, Φ_B.

(9) **VERTEX:** if there is only a single closest vertex V_A to the plane containing F_B, V_A is closer to F_B than F_A is to F_B since two faces are not parallel and their boundaries must be closer to each other than their interior. Next, the algorithm checks if V_A lies within the Voronoi region of F_B.

(10) **THEN**, if so V_A and F_B will be returned as the next candidate pair.

(11) **ELSE**, the algorithm finds the closest vertex (or edge) on F_B to Φ_A, and proceeds as before.

(12) **VERTEX:** if there is only a single vertex V_B closest to the plane Φ_A containing F_A, the algorithm checks if V_B lies inside of F_A's Voronoi region bounded by the constraint planes generated by the edges in F_A's boundary and above F_A.

(13) **THEN**, if V_B lies inside of F_A's Voronoi region, then F_A and V_B will naturally be the next candidate pair. This new pair of features is closer than the previous pair as already analyzed in (10).

(14) **ELSE**, the algorithm performs a search of all edge pairs from the boundary of F_A and F_B and returns the closest pair (E_A, E_B) as the next candidate feature pair.

(15) **EDGE:**, similarly if the closest feature on F_B to Φ_A is an edge E_B, the algorithm once again checks if E_B cuts F_A's prism to determine whether F_A and E_B are the next candidate features.

(16) **THEN**, the algorithm checks **IF THE END POINT OF E_B LIES ABOVE F_A**.

(17) **THEN**, the new pair of candidate features is F_A and E_B since E_B lies within F_A's Voronoi region.

(18) **ELSE**, the algorithm searches for the closest feature f_A which has the minimum distance among all other features on A to E_B and returns (f_A, E_B) as the next pair of candidate features.

(14) **ELSE**, if not, then this implies that neither face contains the closest points since neither F_A nor F_B's Voronoi regions contain a closest point from the other's boundary. So, the algorithm will enumerate all the edge-pairs in the boundaries of F_A and F_B to find the pair of edges which is closest in distance. This new pair of edges (E_A, E_B) is guaranteed to be closer than the two faces.

(19) **EDGE:** if there are two closest vertices on F_A to Φ_B, then the edge E_A containing the two vertices is closer to Φ_B than F_A is. Next, the algorithm checks **IF E_A INTERSECTS THE PRISM REGION OF F_B**, as in the edge-face case.

(20) **THEN**, the algorithm checks **IF THE END POINT OF E_A LIES ABOVE F_B**.

(21) **THEN**, if so the edge E_A and F_B will be returned as the next candidate pair.

(22) **ELSE**, the algorithm searches for the closest feature f_B which has the minimum distance among all other features on B to E_A and returns (E_A, f_B) as the next pair of candidate features.

(23) **ELSE**, if this is not the case, then the algorithm finds the closest vertex (or edge) on F_B to the plane containing F_A, say Φ_A and proceeds as before, from block (11) to block (18).

In either degenerate case (edge-face or face-face), a pair of features closer in distance is always provided when a switch of feature pairs is made. All the verifying subroutines run in *constant* time (proportional to the number of boundary or coboundary of features *or* the product of numbers of boundaries between two features), except where an edge lies beneath a face or when a face lies beneath a face. In such local cases, a linear-time routine is invoked.

To sum up, all the applicability tests and closest feature-pair verifying subroutines generate a pair of new features closer in distance than the previous pair when a switch of feature-pairs is made. □

Corollary The algorithm takes $O(n^3)$ time to find the closest feature pair, since each pair of features is visited at most once, and each step takes at most linear time.

$O(n^3)$ is a rather loose upper bound. We can find a tighter bound on the running time if we study the local minimum case in greater details. As a result of a new analysis, we claim in Lemma 2 that the algorithm should converge in $O(n^2)$ time. The proof is stated below.

Lemma 2 The algorithm will converge in $O(n^2)$ time.

Proof:

Except for the case of a local minimum which corresponds to the situation where one feature lies under the plane containing the other face, all the applicability tests and subroutines run in constant time. Thus the overall running time is $O(n^2)$ in this case, since the algorithm can only exhaustively visit $O(n^2)$ pairs of features and no looping can possibly exist if it doesn't encounter local minima (by Lemma 1). As mentioned earlier, when a local minimum is encountered, a linear-time routine is invoked. But it is not necessary to call this linear-time routine more than $2n$ times to find the closest feature-pair. The reasons are the following:

We claim that we will not revisit the same feature twice in a linear time check. We will prove by contradiction. Assume that the linear time routine is called to find a closest feature in B to f_A. The routine will never again be called with (f_A, B) as argument. Suppose it was, i.e. suppose that f_B was the result of the call to the linear time routine with input (f_A, B) and after several iterations from (f_A, f_B) $\rightarrow \ldots \rightarrow$ (f'_A, f'_B) we come to a feature pair (f_A, f'_B) and that the linear routine is invoked to find the closest feature on B to f_A. Once again, f_B must be returned as the closest feature to f_A. But, this violates Lemma 1 since the distance must strictly decrease when changes of features are made, which is not true of the regular sequence $(f_A, f_B) \rightarrow \ldots \rightarrow (f_A, f'_B) \rightarrow \ldots \rightarrow (f_A, f_B)$.

Similarly, the routine will not be called more than once with (f_B, A) as arguments. Thus, the linear-time routine can be called at most $2n$ times, where n is the total number of features of object B (or A).

Therefore, the closest-feature algorithm will converge in $O(n^2)$ time, even in the worst case where local minima are encountered. $\square$

3.5 Numerical Experiments

The algorithm for convex polyhedral objects described in this chapter has been implemented in both Sun Common Lisp and ANSI C for general convex objects. (The extension of this algorithm to non-convex objects is presented in the next chapter.) The input data are arbitrary pairs of features from two given polyhedral objects in three dimensional space. The routine outputs a pair of the closest features (and a pair of nearest points) for these two objects and the Euclidean distance between them.

Numerous examples in three dimensional space have been studied and tested. Our implementation in ANSI C gives us an expected constant time performance of

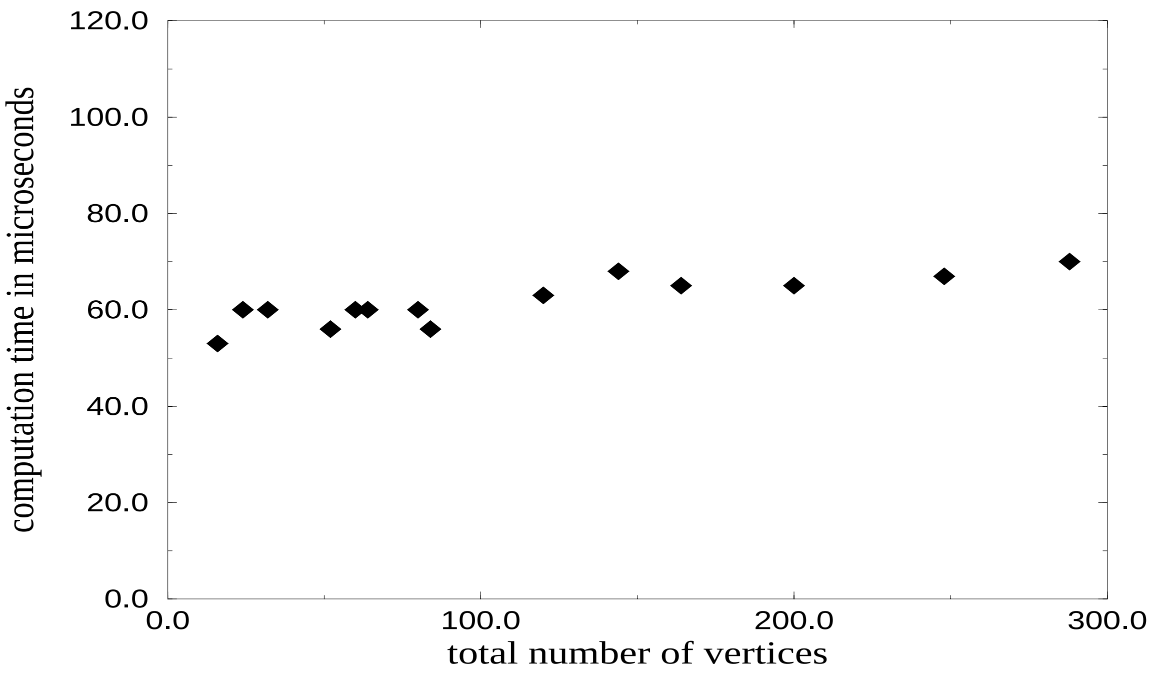


Figure 3.12: Computation time vs. total no. of vertices

50-70 usec per object pair of arbitrary complexity on SGI Indigo2 XZ (200-300 usec per object pair on a 12.5 Mips 1.4 Mega flops Sun4 Sparc Station) for relative small rotational displacement. Fig.3.12 shows the expected constant time performance for polygonized spherical and cylindrical objects of various resolutions. Each data point is taken as the average value of 1000 trials with the relative rotation of 1 degree per step. (The closest features rarely change if two objects are translated along the line connecting the closest feature pair.)

We would like to mention that the accumulated runtime for 1000 trials shows slightly increasing trend for total vertices over 1000, this is due to the fact that the number of feature pair changes for the same amount of rotation increases if the resolution of objects increases. For spherical and most objects, the runtime is roughly $O(\sqrt{n})$ where n is number of vertices for each object with a extremely small constant. For the cylindrical objects where the total number of facets to approximate the quadratic surface is linear with the number of vertices, the total accumulated runtime is expectedly sub-linear with the total number of vertices.

For two convex hulls of teapots (each has 1047 vertices, 3135 edges, and 2090 faces), the algorithm takes roughly 50, 61, 100, 150 usec for 1, 5, 10, 20 degrees of relative rotation and displacements. Note that when the relative movement between 2 objects becomes large, the runtime seems to increase as well, this is due to the fact that the geometric coherence doesn't hold as well in the abrupt motions. Nevertheless, the algorithm continues to perform well compared to other algorithms.

The examples we used in our simulation include a wide variety of polytopes: cubes, rectangular boxes, cylinders, cones, frustrums, and a Puma link of different sizes as shown in Fig.3.13. In particular, the number of facets or the resolution for cylinders, cones and frustrums have been varied from 12, 20, 24, 48, up to 120 in order to generate a richer set of polytopes for testing purpose. Due to the programming language's nature, Lisp code usually gives a longer run time (roughly 10 times slower than the implementation written in ANSI C).

To give readers a feeling of speed difference the expected run time obtained from our Lisp code is 3.5 msec which is roughly 10 times slower than our C implementation (from the same Sun4 SPARC station described above), independent of object complexity. The experiment results obtained from our Lisp code are briefly summarized in Table 1. M1 and M2 stand for the number of vertices for each object respectively; and N stands for the total number of vertices between the two objects. The average CPU time is taken over 50 trials of experimental runs. (Each trial picks all 18 arbitrary permuted combination of feature pairs to initialize the computation and calculates the average run time based upon these 18 runs.) With initialization to the previous closest feature, the routine can almost always keep track of the closest features of two given polytopes in expected constant time (about 3 to 4 msec).

Without initialization (i.e. no previous closest feature pair is given), the algorithm runs in average time not worse than linear in the total number of vertices. This is what we would expect, since it seems unlikely that the algorithm would need to visit a given feature more than once. In practice, we believe our algorithm compares very favorably with other algorithms designed for distance computations or collision detection. Since the nature of problem is not the same, we cannot provide exact comparisons to some of the previous algorithms. But, we can briefly comment on the

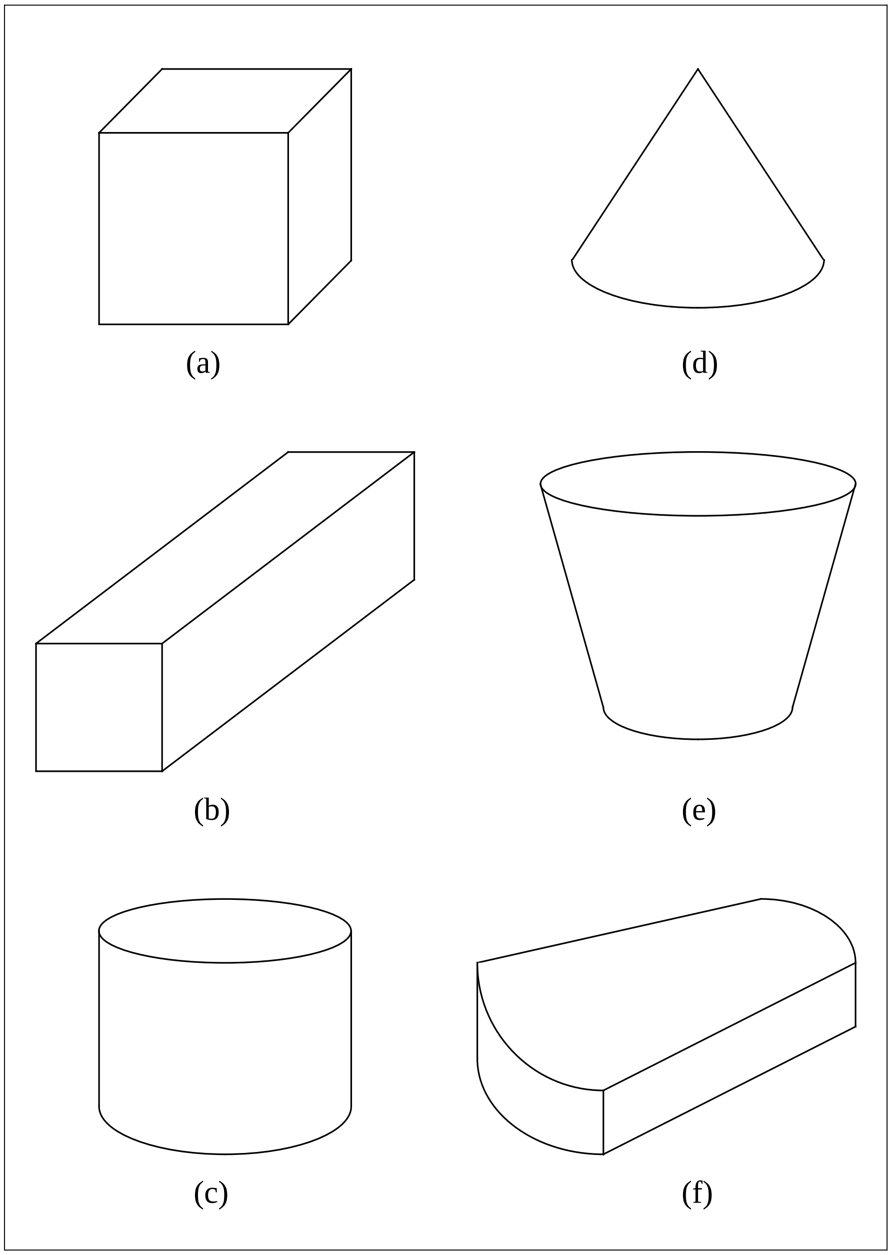


Figure 3.13: Polytopes Used in Example Computations

Objects	M1 + M2 = N	w/o Init	w/ Init
(a),(b)	8 + 8 = 16	21	3.2
(a),(f)	8 + 16 = 24	23	3.3
(a),(c)	8 + 24 = 32	26	3.5
(d),(e)	48 + 21 = 69	41	3.5
(c),(c)	96 + 48 = 144	67	3.0

Table 3.1: Average CPU Time in Milliseconds

number of operations involved in a typical calculation. For an average computation to track or update a pair of closest features, each run takes roughly 50-70 usec on a SGI Indigo2 XZ or 92-131 arithmetic operations (roughly 12-51 operations to find the nearest points, 2 x 16 operations on 2 point transformations, 2 x 4 x 6 operations for two applicability tests and each involves 4 inner product calculations) independent of the machines. By comparing the number of arithmetic operations in the previous implemented algorithms, we believe that our implementation gives better performance and probably is the fastest implemented collision detection algorithm. (Please see [65], [13], [17], [40], [39], [38], [72], and [90].)

This is especially true in a dynamic environment where the trajectories are not known (nor are they in closed form even between impacts, but are given by elliptic integrals). The *expected constant time* performance comes from the fact that *each update* of the closest feature pair involves only the constant number of neighboring features (after the initialization and preprocessing procedures). The speed of algorithm is an attribute from the *incremental* nature of our approach and the geometric coherence between successive, discrete movements.

3.6 Dynamic Collision Detection for Convex Polyhedra

The distance computation algorithm can be easily used for detecting collision while the objects are moving. This is done by iteratively checking the distance between any pair of moving objects. If the distance is less than or equal to zero (plus ϵ - a small safety margin defined by the user), then a collision is declared. We can set the step size of the algorithm once we obtain the initial position, distance, velocity and acceleration among all object pairs. We also use a simple queuing scheme to reduce the frequency of collision checks by relying on the fact that only the object pairs which have a small (Chapter 5).

Furthermore, we will not need to transform all the Voronoi regions and all features of polyhedra but only the closest points and the new candidate features (if necessary), since local applicability constraints are all we need for tracking a closest pair of closest features. The transformation is done by taking the relative transformation between two objects. For example, given two objects moving in space, their motions with respect to the origin of the world frame can be characterized by the transformation matrices T_A and T_B respectively. Then, their relative motion can be represented by the homogeneous relative transformation $T_{AB} = T_B^{-1} T_A$.

Chapter 4

Extension to Non-Convex Objects and Curved Objects

Most of objects in the real world are not simple, convex polyhedral objects. Curved corners and composite objects are what we see mostly in the man-made environment. However, we can use the union of convex polyhedra to model a non-convex object and reasonably refined polyhedral approximation for curved boundaries as well. In this chapter, we will discuss how we extend the collision detection algorithm (described in Chapter 3) from convex objects to non-convex objects. Then, the extension to the curved objects will be described later in the chapter as well.

4.1 Collision Detection for Non-convex Objects

4.1.1 Sub-Part Hierarchical Tree Representation

Nonconvex objects can be either composite solid objects or they can be articulated bodies. We assume that each nonconvex object is given as a union of convex polyhedra *or* is composed of several nonconvex subparts, each of these can be further represented as a union of convex polyhedra or a union of non-convex objects. We use a sub-part hierarchy tree to represent each nonconvex object. At each node of the tree, we store either a convex sub-part (at a leave) or the union of several convex

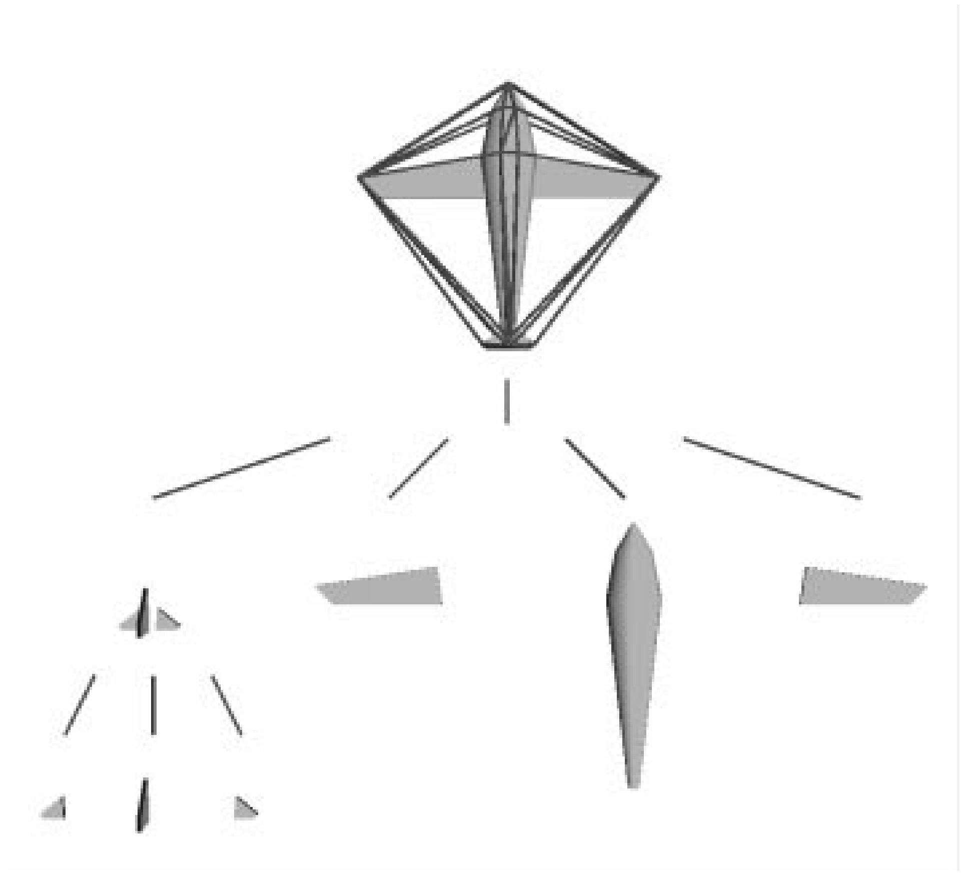


Figure 4.1: An example of sub-part hierarchy tree

subparts.

Depending on the applications, we first construct the convex hull of a rigid non-convex object or dynamic bounding box for articulated bodies at each node and work up the tree as part of preprocessing computation. We also include the convex hull or dynamic bounding box of the union of sub-parts in the data structure. The convex hull (or the bounding box) of each node is the convex hull (or bounding box) of the union of its children. For instance, the root of the sub-part hierarchy tree for a composite solid non-convex object is the nonconvex object with its convex hull in its data structure; each convex sub-part is stored at a leave. The root of the sub-part hierarchical tree for a Puma arm is the bounding box of the entire arm which consists of 6 links; and each leave stores a convex link.

For example, in Fig. 4.1, the root of this tree is the aircraft and the convex hull of the aircraft (outlined by the wire frames). The nodes on the first level of the tree store the subparts and the union of the subparts of the aircraft, i.e. the left and right wings (both convex), the convex body of airplane, and the union of the tail pieces. Please note that at the node where the union of the tail pieces are stored, a convex hull of union of these tail pieces is also stored there as well. This node, which stores the union of all tail pieces, further branches out to 3 leaves, each of which stores a convex tail piece.

For a robot manipulator like a Puma arm, each link can be treated as a convex subpart in a leave node. The root of this tree can be a simple bounding box of the entire Puma arm whose configuration may be changing throughout the duration of its task. (Therefore, the bounding box may be changing as well. Of course, the efficiency of our distance computation algorithm in Chapter 3 is lost when applied to a changing volume, since the Voronoi regions will be changing as the volume of the object changes and the preprocessing computation is not utilized to its maximum extend. However, in Chapter 5 we will present another simple yet efficient approach to deal with such a situation.)

In addition, not only can we apply the hierarchical subpart tree representation to polyhedral models, we can also use it for non-convex *curved* objects as well. Depending on the representation of the curved objects (parametric, algebraic, Bézier, or B-spline patches), each node of the tree will store a portion (such as a patch) of the curved surfaces and the root of the tree is the entire curved object and the convex hull of its polyhedral approximation. For more treatments on the non-convex curved objects, please refer to Sec. 4.3.3.

4.1.2 Detection for Non-Convex Polyhedra

Given the subpart tree representation, we examine the possible interference by a recursive algorithm at each time step. The algorithm will first check for collision between two parent convex hulls. Of course, if there is no interference between two parents, there is no collision and the algorithm updates the closest-feature pair and the distance between them. If there is a collision, then it will expand their children. All children of one parent node are checked against all children of the other parent node. If there is also a collision between the children, then the algorithm will recursively call upon itself to check for possible impacts among the children's children, and so on. In this recursive manner, the algorithm will only signal a collision if there is actually an impact between the sub-parts of two objects; otherwise, there is no collision between the two objects.

For instance, assume for simplicity that one nonconvex object is composed of m convex polyhedra and the other is composed of n convex polyhedra, the algorithm will first check for collision between the two convex hulls of these two objects. If there is a collision, mn convex polyhedra-pairs will be tested for possible contacts. If there is a collision between any one of these mn pairs of convex polyhedra, then the two objects interpenetrate; otherwise, the two objects have not yet intersected. Fig. 4.2 shows how this algorithm works on two simplified aircraft models.

In Fig. 4.2, we first apply the algorithm described in Chapter 3 to the convex hulls of the two aircrafts. If they do not intersect, then the algorithm terminates and reports no collision. In the next step, the algorithm continues to track the motion of the two planes and finds their convex hulls overlap. Then, the algorithm goes down the tree to the next level of tree nodes and check whether there is a collision among the subparts (the wings, the bodies, and the union of subparts, i.e. the union of tail pieces). If not, the algorithm will not report a collision. Otherwise, when there is an actual penetration of subparts (such as two wings interpenetrating each other), then a collision is reported.

We would like to clarify one fact that we never need to construct the Voronoi regions of the non-convex objects, since it is absolutely unnecessary given the subpart hierarchical tree. Instead, at the preprocessing step we only construct the Voronoi regions for each convex piece at each node of the subpart hierarchical tree. When we perform the collision checks, we only keep track of the closest feature pairs for the convex hulls of parent nodes, unless a collision has taken place between the convex hulls of two non-convex objects or the bounding boxes of articulated bodies. Under such a situation, we have to track the closest pair of features for not only the convex hulls or bounding boxes at the parent nodes, but also among all possible subpart pairs at the children nodes as well. This can lead to $O(n^2)$ of comparisons, if each object is consisted of n subparts and the trees only have two levels of leave nodes.

However, we can achieve a better performance if we have a good hierarchy of the model. For complex objects, using a deep hierarchical tree with lower branching factor will keep down the number of nodes which need to be expanded. This approach guarantees that we find the earliest collision between two non-convex objects while reducing computation costs. For example, given $2n$ subparts, we can further group them into two subassemblies and each has n subparts. Each subassembly is further divided into several smaller groups. This process generates a deep hierarchical tree with lower branching factor, thus reduces the number of expansions needed whenever the convex hulls (or bounding boxes) at the top-level parent nodes collide.

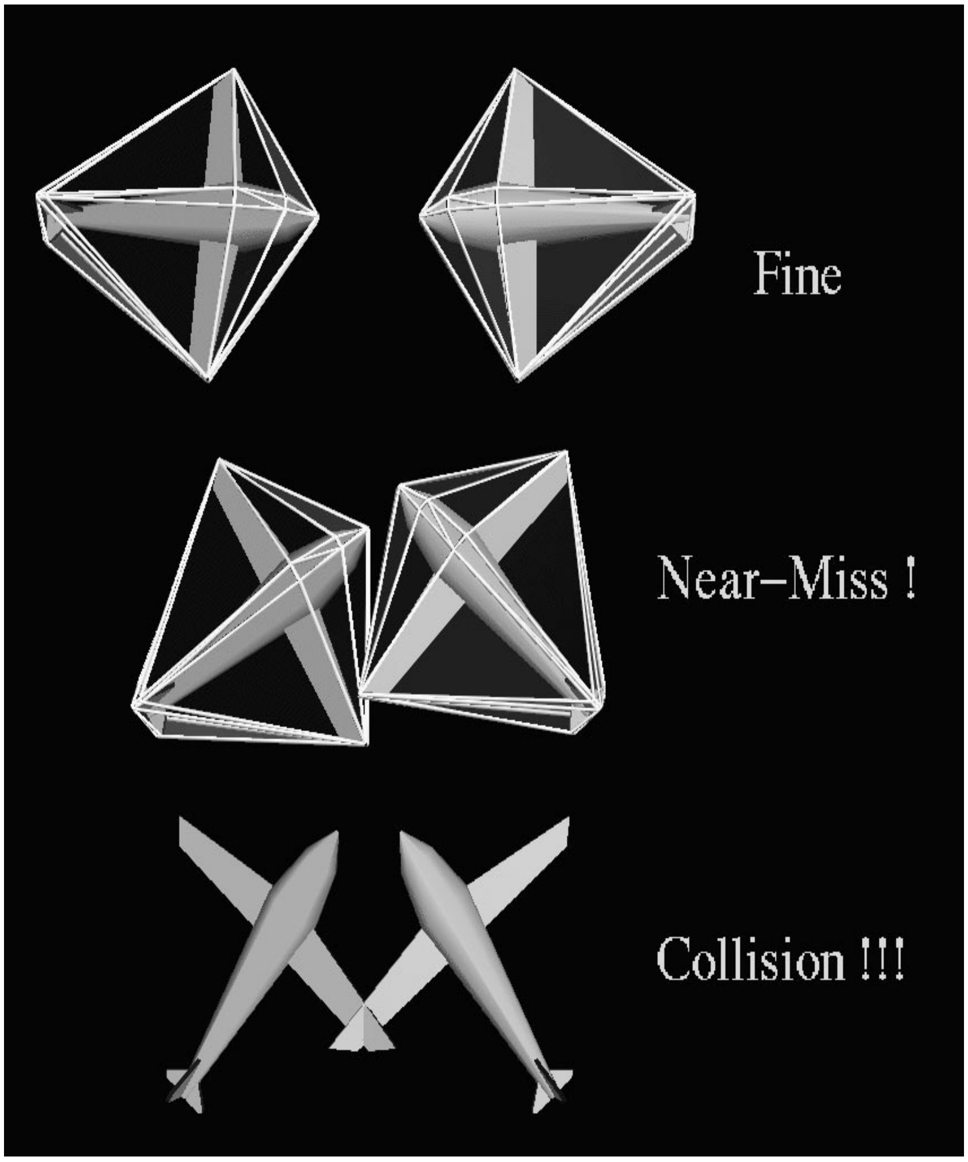


Figure 4.2: An example for non-convex objects

4.2 Collision Detection for Curved Objects

In this section[1], we analyze the problem of collision detection between curved objects represented as spline models or piecewise algebraic surfaces. Different types of surface representations have been desribed in Chapter 2, please refer to it if necessary. We show that these problems reduce to finding solutions of a system of algebraic equations. In particular, we present algebraic formulations corresponding to closest points determination and geometric contacts.

4.2.1 Collision Detection and Surface Intersection

In geometric and solid modeling, the problem of computing the intersection of surfaces represented as spline surfaces or algebraic surfaces has received a great deal of attention [46]. Given two surfaces, the problem corresponds to computing all components of the intersection curve robustly and accurately. However, for collision detection we are actually dealing with a restricted version of this problem. That is, given two surfaces we want to know whether they intersect.

In general, given two spline surfaces, there is *no* good and quick solution to the problem of whether they intersect or have a common geometric contact. The simplest solution is based on checking the control polytopes (a convex bounding box) for collision and using subdivision. However, we have highlighted the problems with this approach (in Chapter 1) in terms of performance when the two objects are close to touching. Our approach is based on formulating the problem in terms of solving systems of algebraic equations and using global methods for solving these equations, and local methods to update the solutions.

4.2.2 Closest Features

Given the homogeneous representation of two parametric surfaces, $\mathbf{F}(s,t) = (X(s,t),\ Y(s,t),\ Z(s,t),\ W(s,t))$ and $\mathbf{G}(u,v) = (\overline{X}(u,v),\ \overline{Y}(u,v), \overline{Z}(u,v),\ \overline{W}(u,v))$,

[1]This is the result of joint work with Prof. Dinesh Manocha at the University of North Carolina, Chapel Hill

the closest *features* between two polyhedra correspond to the closest point sets on the surface. The closest points are characterized by the property that the corresponding surface normals are collinear. This can be expressed in terms of the following variables. Let

$$\begin{aligned}
\mathbf{F_{11}}(s,t,u,v,\alpha_1) &= (\mathbf{F}(s,t) \Leftrightarrow \mathbf{G}(u,v)) \\
\mathbf{F_{12}}(s,t,u,v,\alpha_1) &= \alpha_1(\mathbf{G}_u(u,v) \times \mathbf{G}_v(u,v)) \\
\mathbf{F_{21}}(s,t,u,v,\alpha_2) &= (\mathbf{F}_s(s,t) \times \mathbf{F}_t(s,t)) \\
\mathbf{F_{22}}(s,t,u,v,\alpha_2) &= \alpha_2(\mathbf{G}_u(u,v) \times \mathbf{G}_v(u,v)),
\end{aligned}$$

where $\mathbf{F}_s, \mathbf{F}_t, \mathbf{G}_u, \mathbf{G}_v$ correspond to the partial derivatives. The closest points between the two surfaces satisfy the following equations:

$$\mathbf{F_1}(s,t,u,v,\alpha_1) = \mathbf{F_{11}}(s,t,u,v,\alpha_1) \Leftrightarrow \mathbf{F_{12}}(s,t,u,v,\alpha_1) = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}. \qquad (4.1)$$

$$\mathbf{F_2}(s,t,u,v,\alpha_2) = \mathbf{F_{21}}(s,t,u,v,\alpha_2) \Leftrightarrow \mathbf{F_{22}}(s,t,u,v,\alpha_2) = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}.$$

This results in 6 equations in 6 unknowns. These constraints between the closest features can also be expressed as:

$$\begin{aligned}
\mathbf{H}_1(s,t,u,v) &= (\mathbf{F}(s,t) \Leftrightarrow \mathbf{G}(u,v)) \bullet \mathbf{G}_u(u,v) = 0 \\
\mathbf{H}_2(s,t,u,v) &= (\mathbf{F}(s,t) \Leftrightarrow \mathbf{G}(u,v)) \bullet \mathbf{G}_v(u,v) = 0 \qquad (4.2) \\
\mathbf{H}_3(s,t,u,v) &= (\mathbf{F}(s,t) \Leftrightarrow \mathbf{G}(u,v)) \bullet \mathbf{F}_s(s,t) = 0 \\
\mathbf{H}_4(s,t,u,v) &= (\mathbf{F}(s,t) \Leftrightarrow \mathbf{G}(u,v)) \bullet \mathbf{F}_t(s,t) = 0,
\end{aligned}$$

where $\bullet$ corresponds to the dot product. This results in four equations in four unknowns.

Let us analyze the algebraic complexity of these two system of equations corresponding to closest features. Lets consider the first system corresponding to (4.1). In particular, given 2 rational parametric surfaces $\mathbf{F}(s,t)$ and $\mathbf{G}(u,v)$, both of their numerators and denominators are polynomials of degree n, the degrees of numerators and denominators of the partials are $2n \Leftrightarrow 1$ and $2n$ respectively in the given variables (due to quotient rule). The numerator and denominator of $\mathbf{F_{11}}(s,t,u,v,\alpha_1)$ have degree $2n$ and $2n$ due to subtraction of two rational polynomials. As for $\mathbf{F_{12}}(s,t,u,v,\alpha_1)$, taking the cross product doubles the degrees for both the numerator and denominator; therefore, the degrees for the numerator and denominator of $\mathbf{F_{12}}(s,t,u,v,\alpha_1)$ are $4n \Leftrightarrow 2$ and $4n$ respectively. To eliminate α_1 from $\mathbf{F}_1(s,t,u,v,\alpha_1)$, we get

$$\frac{\mathbf{F}_{11}(X(s,t,u,v,\alpha_1))}{\mathbf{F}_{12}(X(s,t,u,v,\alpha_1))} = \frac{\mathbf{F}_{11}(Y(s,t,u,v,\alpha_1))}{\mathbf{F}_{12}(Y(s,t,u,v,\alpha_1))} = \frac{\mathbf{F}_{11}(Z(s,t,u,v,\alpha_1))}{\mathbf{F}_{12}(Z(s,t,u,v,\alpha_1))} \tag{4.3}$$

After cross multiplication to clear out the denominators we get two *polynomails* of degree $12n \Leftrightarrow 2$ each. Once again, by the same reasoning as stated above, both the numerators and denominators of $\mathbf{F_{21}}(s,t,u,v,\alpha_2)$ and $\mathbf{F_{22}}(s,t,u,v,\alpha_2)$ have degrees of $4n \Leftrightarrow 2$ and $4n$. By similar method mentioned above, we can eliminate α_2 from $\mathbf{F}_2(s,t,u,v,\alpha_2)$. We get two polynomial equations of degree $16n \Leftrightarrow 4$ each after cross multiplication. As a result the system has a Bezout bound of $(12n \Leftrightarrow 2)^2(16n \Leftrightarrow 4)^2$.

Each equation in the second system of equations has degree $4n \Leftrightarrow 1$ (obtained after computing the partials and addition of two rational polynomials) and therefore the overall algebraic complexity corresponding to the Bezout bound is $(4n \Leftrightarrow 1)^4$. Since the later system results in a lower degree bound, in the rest of the analysis we will use this system. However, we are only interested in the solutions in the domain of interest (since each surface is defined on a subset of the real plane).

Baraff has used a formulation similar to (4.1)to keep track of closest points between closed convex surfaces [3] based on local optimization routines. The main problem is finding a solution to these equations for the initial configuration. In general, these equations can have more than one solution in the associated domain, i.e. the real plane, (even though there is only one closest point pair) and the optimization routine may not converge to the right solution. A simple example is the formulation for the problem of collision detection between two spheres. There is only one pair of closest points, however equations (4.1) or (4.2) have four pairs of real solutions.

Given two algebraic surfaces, $f(x,y,z) = 0$ and $g(x,y,z) = 0$, the problem

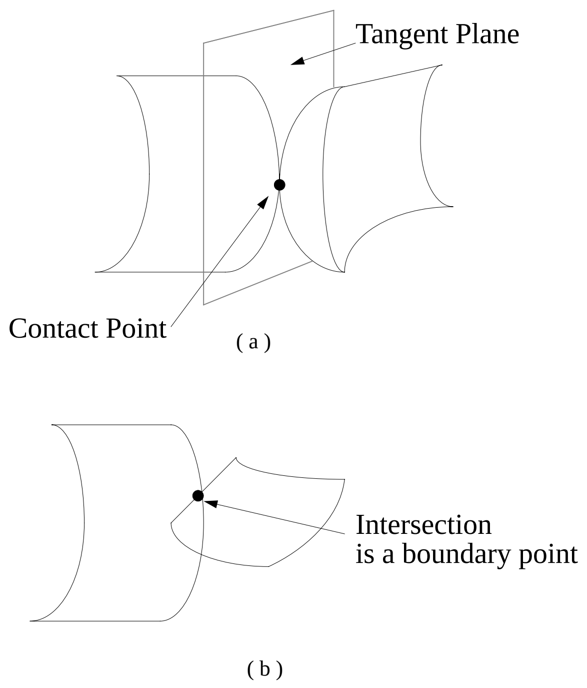


Figure 4.3: Tangential intersection and boundary intersection between two Bézier surfaces

of closest point determination can be reduced to finding roots of the following system of 8 algebraic equations:

$$
\begin{array}{rcl}
f(x_1, y_1, z_1) & = & 0 \\
g(x_2, y_2, z_2) & = & 0 \\
\begin{pmatrix} f_x(x_1, y_1, z_1) \\ f_y(x_1, y_1, z_1) \\ f_z(x_1, y_1, z_1) \end{pmatrix} & = & \alpha_1 \begin{pmatrix} g_x(x_2, y_2, z_2) \\ g_y(x_2, y_2, z_2) \\ g_z(x_2, y_2, z_2) \end{pmatrix} \\
\begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} \Leftrightarrow \begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} & = & \alpha_2 \begin{pmatrix} g_x(x_2, y_2, z_2) \\ g_y(x_2, y_2, z_2) \\ g_z(x_2, y_2, z_2) \end{pmatrix}
\end{array}
\tag{4.4}
$$

Given two algebraic surfaces of degree n, we can eliminate α_1 by setting $\frac{f_x(x1,y1,z1)}{g_x(x_2,y_2,z_2)} = \frac{f_y(x1,y1,z1)}{g_y(x2,y2,z2)}$. After cross multiplication, we have a polynomial equation of 2n-2, since each partial has degree of $n \Leftrightarrow 1$ and the multiplication results in the degree sum of $2n \Leftrightarrow 2$. To eliminate α_2, we set $\frac{x_1 - x_2}{g_x(x_2,y_2,z_2)} = \frac{y_1 - y_2}{g_y(x_2,y_2,z_2)}$ and the degree of the resulting polynomial equation is n. We have six quations after eliminating α_1 and α_2: two of degrees $(2n \Leftrightarrow 2)$ and four of degress n respectively (2 from eliminating α_2 and 2 from $f(x_1, y_1, z_1)$ and $g(x_2, y_2, z_2)$). Therefore, the Bezout bound of the resulting system can be as high as $N = (2n \Leftrightarrow 2)^2 n^4$. In general, if the system of equations is sparse, we can get a tight bound with Bernstein bound [8]. The Bernstein bound for Eqn. 4.4 is $n^2(n^2 + 3)(n \Leftrightarrow 1)^2$. Canny and Emiris calculate the Bernstein bounds efficiently by using sparse mixed resultant formulation [16]. For example, the Bernstein bounds[2] for the case of $n = 2, 3, 4, 5, 6, 7, 8, 9$ are $28, 432, 2736, 11200, 35100, 91728, 210112, 435456$, while the Bezout bounds are $64, 1296, 9216, 40000, 129600, 345744, \cdots$ respectively. Even for small values of n, N can be large and therefore, the algebraic complexity of computing the closest points can be fairly high. In our applications we are only interested in the real solutions to these equations in the corresponding domain of interest. The actual number of real solutions may change as the two objects undergo motion and some configurations can result in infinite solutions (e.g. when a closest pair corresponds to a curve on each surface, as shown for two cylinders in Fig. 4.4.). As a result, it is fairly non-trivial to keep track of all the closest points between objects and updating them as the objects undergo motion.

4.2.3 Contact Formulation

The problem of collision detection corresponds to determining whether there is any contact between the two objects. In particular, it is assumed that in the beginning the objects are not overlapping. As they undergo motion, we are interested in knowing whether there is any contact between the objects. There are two types of contacts. They are tangential contact and boundary contact. In this section, we formulate both of these problems in terms of a system of algebraic equations. In the next section, we describe how the algorithm tests for these conditions as the object

[2]These figures are calculated by John Canny and Ioannis Emiris using their code based on the sparse mixed resultant formulation.

undergoes rigid motion.

- *Tangential Intersection :* This corresponds to a tangential intersection between the two surfaces at a geometric contact point, as in Fig.4.3(a). The contact point lies in the interior of each surface (as opposed to being on the boundary curve) and the normal vectors at that point are collinear. These constraints can be formulated as:

$$\begin{aligned} \mathbf{F}(s,t) &= \mathbf{G}(u,v) \\ (\mathbf{F}_s(s,t) \times \mathbf{F}_t(s,t)) \bullet \mathbf{G}_u(u,v) &= 0 \\ (\mathbf{F}_s(s,t) \times \mathbf{F}_t(s,t)) \bullet \mathbf{G}_v(u,v) &= 0 \end{aligned} \tag{4.5}$$

  The first vector equation corresponds to a contact between the two surfaces and the last two equations represent the fact that their normals are collinear. They are expressed as scalar triple product of the vector The last vector equation represented in terms of cross product corresponds to three scalar equations. We obtain 5 equations in 4 unknowns. This is an overconstrained system and has a solution only when the two surfaces are touching each other tangentially. However, we solve the problem by computing all the solutions to the first four equations using global methods and substitute them into the fifth equation. If the given equations have a common solution, than one of the solution of the first four equation will satisfy the fifth equation as well. For the first three equations, after cross multiplication we get 3 polynomial equations of degree $2n$ each. The dot product results in the addition of degrees of the numerator polynomials. Therefore, we get a polynomial equation of degree $6n \Leftrightarrow 3$ from the fourth equation. Therefore, the Bezout bound of the system corresponding to the first four equations is bounded by $N = (2n)^3(6n \Leftrightarrow 3)$, where n is the parametric degree of each surface. Similarly for two algebraic surfaces, the problem of tangential intersection can be formulated as:

$$\begin{aligned} f(x,y,z) &= 0 \\ g(x,y,z) &= 0 \end{aligned} \tag{4.6}$$

$$\begin{pmatrix} f_x(x,y,z) \\ f_y(x,y,z) \\ f_z(x,y,z) \end{pmatrix} = \alpha \begin{pmatrix} g_x(x,y,z) \\ g_y(x,y,z) \\ g_z(x,y,z) \end{pmatrix}$$

In this case, we obtain 4 equations in 3 unknowns (after eliminating α) and these equations correspond to an overconstrained system as well. These overconstrained system is solved in a manner similar to that of parametric surfaces. The first two equations are of degrees n. To eliminate α from the third equation, we get a polynomial equation of degree 2(n-1) due to cross multiplication and taking partial with respect to x, y, z. The Bezout bound for the first three equations is $N = n^2(2n \Leftrightarrow 2)$.

- *Boundary Intersection :* Such intersections lie on the boundary curve of one of the two surfaces. Say we are given a Bézier surface, defined over the domain, $(s,t) \in [0,1] \times [0,1]$, we obtain the boundary curves by substituting s or t to be 0 or 1. The resulting problem reduces to solving the equations:

$$\mathbf{F}(s,1) = \mathbf{G}(u,v) \tag{4.7}$$

Other possible boundary intersections can be computed in a similar manner. The intersection points can be easily computed using global methods. An example has been shown in Figure 4.3(b)

Two objects collide if one of these sets of equations, (4.5) or (4.7) for parametric surfaces and (4.6) for algebraic surfaces, have a common solution in their domain.

In a few degenerate cases, it is possible that the system of equations (4.5) and (4.7) have an infinite number of solutions. One such example is shown for two cylinders in Fig.4.4. In this case the geometric contact corresponds to a curve on each surface, as opposed to a point. These cases can be detected using resultant methods as well [55].

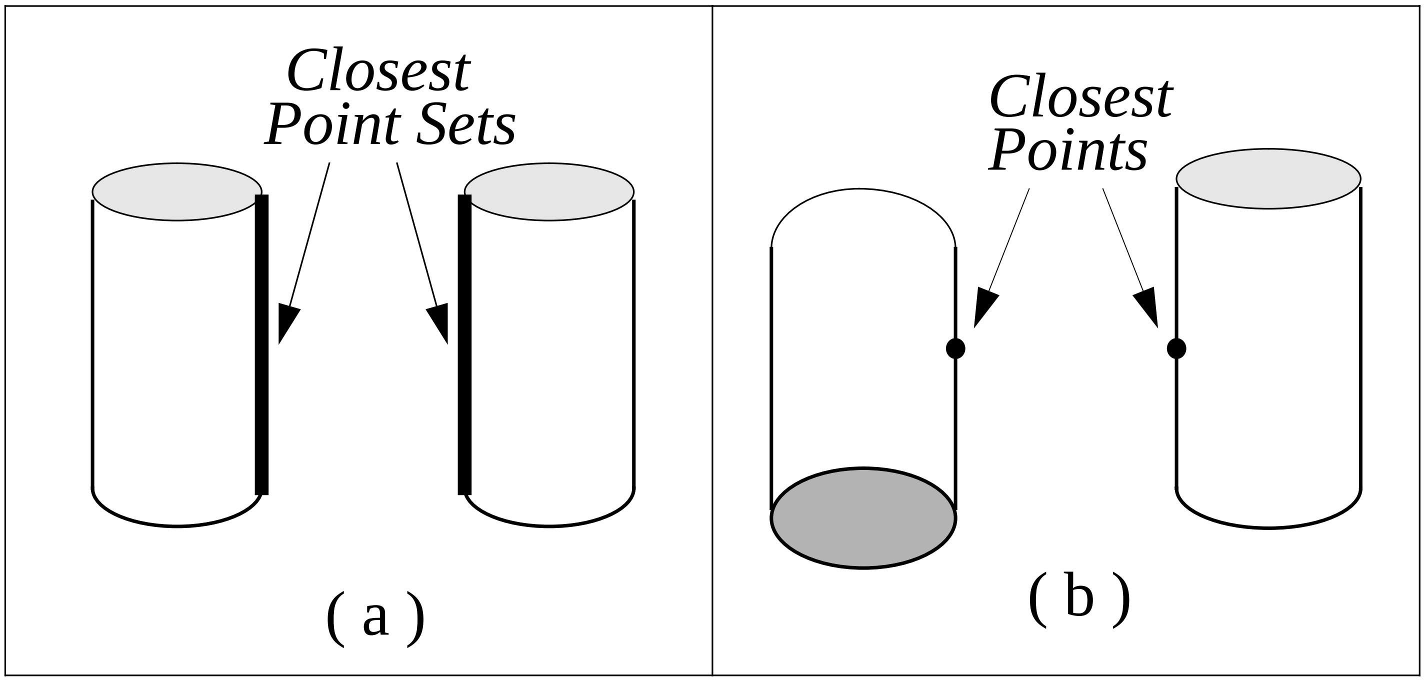


Figure 4.4: Closest features between two different orientations of a cylinder

4.3 Coherence for Collision Detection between Curved Objects

In most dynamic environments, the closest features or points between two moving objects change infrequently between two time frames. We have used this coherence property in designing the expected constant time algorithm for collision detection among convex polytopes and applied to non-convex polytopes as well. In this section we utilize this coherence property along with the algebraic formulations presented in the previous section for curved models.

4.3.1 Approximating Curved Objects by Polyhedral Models

We approximate each physical object with curved boundary by a polyhedra (or polygonal mesh). Such approximations are used by rendering algorithms utilizing polygon rendering capabilities available on current hardware. These polyhedral models are used for collision detection, based on the almost constant time algorithm utilizing local features. Eventually a geometric contact is determined by solving the equations highlighted in the previous section. We use an *ϵ-polytope* approximation for a curved surface. It is defined as:

Definition: Let S be a surface and P is ϵ-polytope approximation, if for all points, $\mathbf{p}$ on the boundary of polytope P, there is a point $\mathbf{s}$ on S such that $\| \mathbf{s} \Leftrightarrow \mathbf{p} \| \leq \epsilon$. Similarly for each point $\mathbf{s}$ on S, there is a point $\mathbf{p}$ on the boundary of P such that $\| \mathbf{p} \Leftrightarrow \mathbf{s} \| \leq \epsilon$.

An ϵ-polytope approximation is obtained using either a simple mesh generation algorithm or an adaptive subdivision of the surfaces. Given a user defined ϵ, algorithms for generating such meshes are highlighted for parametric B-spline surfaces in [36] and for algebraic surfaces in [44]. In our implementation we used an inscribed polygonal approximation to the surface boundary.

We use the ϵ-polytope approximations for convex surfaces only. In such cases the resulting polytope is convex as well. Often a curved model can be represented as a union of convex objects. Such models are represented hierarchically, as described in Section 4.1.1. Each polytope at a leaf node corresponds to an ϵ-polytope approximation.

4.3.2 Convex Curved Surfaces

In this section we present an algorithm for two convex spline surfaces. The input to the algorithm are two convex B-spline surfaces (say S_A and S_B) and their associated control polytopes. It is assumed that the surfaces have at least first order continuity. We compute an ϵ-polytope approximation for each spline surface. Let P_A and P_b be ϵ_A-polytope and ϵ_B-polytope approximations of S_A and S_B, respectively.

If the objects do not collide, there may be no need to find the exact closest points between them but a rough estimate of location to preserve the property of geometric coherence. In this case, we keep track of the closest points between P_A and P_B. Based on those closest points, we have a good bound on the actual distance between the two surfaces. At any instance let d_p be the minimum distance between P_A and P_B (computed using the polyhedral collision detection algorithm). Let d_s be the minimum distance between the two surfaces. It follows from the ϵ-approximation:

$$d_p \Leftrightarrow \epsilon_A \Leftrightarrow \epsilon_B \le d_s \le d_p. \tag{4.8}$$

The algorithm proceeds by keeping track of the closest points between P_A and P_B and updating the bounds on d_s based on d_p. Whenever $d_p \le \epsilon_A + \epsilon_B$, we use Gauss Newton routines to find the closest points between the surfaces S_A and S_B. In particular, we formulate the problem: For Gauss Newton routines, we want to minimize the function

$$\mathbf{H}(s,t,u,v) = \sum_{i=1}^{4} (H_i(s,t,u,v))^2,$$

where $\mathbf{H}_i$ are defined in (4.2). We use *Gauss-Newton* algorithm to minimize $\mathbf{H}$. The initial guess to the variables is computed in the following manner.

We use the line, say $L_{A,B}$, joining the closest points of P_A and P_B as an initial guess to the line joining the closest points of S_A and S_B (in terms of direction). The initial estimate to the variables in the equations in (4.2) is obtained by finding the intersection of the line $L_{A,B}$ with the surfaces, $\mathbf{F}(s,t)$ and $\mathbf{G}(u,v)$. This corresponds to a line-surface intersection problem and can be solved using subdivision or algebraic methods [35, 55]. As the surfaces move along, the coefficients of the equations in (4.2) are updated according to the rigid motion. The closest points between the resulting surfaces are updated using Gauss Newton routines. Finally, when these closest points coincide, there is a collision.

In practice, the convergence of the Gauss Newton routines to the closest points of S_A and S_B is a function of ϵ_A and ϵ_B. In fact, the choice of ϵ in the ϵ-polytope approximation is important to the overall performance of the algorithm. Ideally, as $\epsilon \rightarrow 0$, we get a finer approximation of the curved surface and better the convergence of the Gauss Newton routines. However, a smaller ϵ increases the number of features in the resulting polytope. Though polyhedral collision detection is an expected constant time algorithm at each step, the overall performance of this algorithm is governed by the total number of feature pairs traversed by the algorithm. The latter is dependent on motion and the resolution of the approximation. Consequently, a very small ϵ can slow down the overall algorithm. In our applications, we have chosen ϵ as a function of the dimension of a simple bounding box used to bound S_A. In particular, let l be dimension of the smallest cube, enclosing S_A. We have chosen $\epsilon = \delta l$, where $.01 \leq \delta \leq .05$. This has worked well in the examples we have tested so far.

The algorithm is similar for surfaces represented algebraically. Objects which can be represented as a union of convex surfaces, we use the hierarchical representation and the algorithm highlighted above on the leaf nodes.

4.3.3 Non-Convex Curved Objects

In this section we outline the algorithm for non-convex surfaces which cannot be represented as a union of convex surfaces. A common example is a torus and even a model of a teapot described by B-spline surfaces comes in this category. The approach highlighted in terms of ϵ-polytopes is not applicable to non-convex surfaces, as the resulting polytope is non-convex and its convex decomposition would result in many convex polytopes (of the order of $O(1/\epsilon)$).

Given two B-spline non-convex surface surfaces, we decompose them into a series of Bézier surfaces. After decomposition we use a hierarchical representation for the non-convex surface. The height of the resulting tree is two. Each leaf node corresponds to the Bézier surface. The nodes at the first level of the tree correspond to the convex hull of the control polytope of each Bézier surface. The root of the tree represents the union of the convex hull of all these polytopes. An example of such a hierarchical representation is shown in Fig.4.5 for a torus. The torus is composed of biquadratic Bézier surfaces, shown at the leaf nodes. The convex polytopes at the first level are the convex hull of control polytopes of each Bézier surface and the root is a convex hull of the union of all control points.

The algorithm proceeds on the hierarchical representation, as explained in Section 3. However, each leaf node is a Bézier surface. The fact that each surface is non-convex implies that the closest points of the polytopes may not be a good approximation to the closest points on the surface. Moreover, two such surfaces

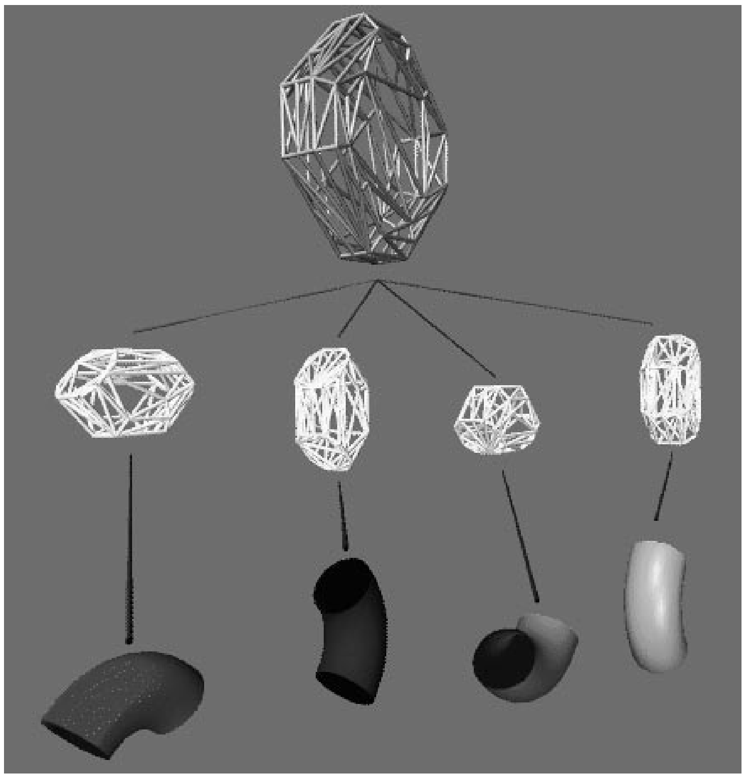


Figure 4.5: Hierarchical representation of a torus composed of Bézier surfaces

can have more than one closest feature at any time. As a result, local optimization methods highlighted in the previous sections may not work for the non-convex objects.

The problem of collision detection between two Bézier surfaces is solved by finding all the solutions to the equations (4.5) and (4.7). A real solution in the domain to those equations implies a geometric collision and a precise contact between the models. The algebraic method based on resultants and eigenvalues is used to find all the solutions to the equations (4.5) and (4.7) [55]. This global root finder is used when the control polytopes of two Bézier surfaces collide. At that instant the two surfaces may or may not have a geometric contact. It is possible that all the solutions to these equations are complex. The set of equations in (4.5) represents an overconstrained system and may have no solution in the complex domain as well. However, we apply the algebraic method to the first four equations in (4.5) and compute all the solutions.

The total number of solutions of a system of equations is bounded by the Bezout bound. The resultant method computes all these solutions. As the objects move, we update the coefficients of these equations based on the rigid motion. We obtain a new set of equations corresponding to (4.5) and (4.7), whose coefficients are slightly different as compared to the previous set. All the roots of the new set of equations are updated using Newton's method. The previous set of roots are used as initial guesses. The overall approach is like *homotopy methods* [64], This procedure represents an algebraic analog of the geometric coherence exploited in the earlier section.

As the two objects move closer to each other, the imaginary components of some of the roots start decreasing. Finally, a real collision occurs, when the imaginary component of one of the roots becomes zero.

We do not have to track all the paths corresponding to the total number of solutions. After a few time steps we only keep track of the solutions whose imaginary components are decreasing. This is based on the fact that the number of closest points is much lower than the Bezout bound of the equations.

Chapter 5

Interference Tests for Multiple Objects

In a typical environment, there are moving objects and stationary obstacles as well. Assume that there are N objects moving around in an environment of M stationary obstacles. Each of N moving objects is also considered as a moving obstacle to the other moving objects. Keeping track of the distance for $\binom{N}{2} + NM$ pairs of objects at all time can be quite time consuming, especially in a large environment. In order to avoid unnecessary computations and to speed up the run time, we present two methods: one assumes the knowledge of maximum acceleration and velocity, the other purely exploits the spatial arrangement without any other information to reduce the number of pairwise interference tests. For the scheduling scheme in which we assume the dynamics (velocities, accelerations, etc.) is known, we rely on the fact that only the object pairs which have a small separation are likely to have an impact within the next few time instances, and those object pairs which are far apart from each other cannot possibly come to interfere with each other until certain time. For spatial tests to reduce the number of pairwise comparisons, we assume the environment is rather sparse and the objects move in such a way that the geometric coherence can be preserved, i.e. the assumption that the motion is essentially continuous in time domain.

5.1 Scheduling Scheme

The algorithm maintains a queue (implemented as a heap) of all pairs of objects that might collide (e.g. a pair of objects which are rigidly attached to each other will not appear in the queue). They are sorted by lower bound on time to collision; with the one most likely to collide (i.e. the one that has the smallest approximate time to collision) appearing at the top of the heap. The approximation is a lower bound on the time to collision, so no collisions are missed. Non-convex objects, which are represented as hierarchy trees as described in Chapter 4, are treated as single objects from the point of view of the queue. That is, only the roots of the hierarchy trees are paired with other objects in the queue.

The algorithm first has to compute the initial separation and the possible collision time among all pairs of objects and the obstacles, assuming that the magnitude of relative initial velocity, relative maximum acceleration and velocity limits among them are given. After initialization, at each step it only computes the closest feature pair and the distance between one object pair of our interests, i.e. the pair of objects which are most likely to collide first; meanwhile we ignore the other object pairs until one of them is about to collide. Basically, the algorithm puts all the object pairs to sleep until the clock reaches the "wakeup" time for the first pair on top of the heap. Wakeup time W_i for each object pair P_i is defined as

$$W_i = t_w^i + t_0$$

where t_w^i is the lower bound on the time to collision for each pair P_i for most situations (with exceptions described in the next sub-section) and t_0 is the current time.

5.1.1 Bounding Time to Collision

Given a trajectory that each moving object will travel, we can determine the exact collision time. Please refer to [17] for more details. If the path that each object travels is not known in advance, then we can calculate a lower bound on collision time. This lower bound on collision time is calculated adaptively to speed up the performance of dynamic collision detection.

Let a_{max} be an upper bound on the relative acceleration between any two *points* on any pair of objects. The bound a_{max} can be easily obtained from bounds on the relative absolute linear $\vec{a}_{lin}$ and relative rotational accelerations $\vec{a}_{rot}$ and relative rotational velocities $\vec{\omega}_r$ of the bodies and their diameters: $\vec{a}_{max} = \vec{a}_{lin} + \vec{a}_{rot} \times \vec{r} + \vec{\omega}_r \times (\vec{\omega}_r \times \vec{r})$ where r is the vector difference between the centers of mass of two bodies. Let d be the initial separation for a given pair of objects, and v_i (where $\vec{v}_i = \vec{v}_{lin} + \vec{\omega}_r \times \vec{r}$) the initial relative velocity of closest points on these objects. Then we can bound the time t_c to collision as

$$t_c = \frac{\sqrt{v_i^2 + 2a_{max}d} \Leftrightarrow v_i}{a_{max}} \tag{5.1}$$

This is the minimum safe time that is added to the current time to give the wakeup time for this pair of objects. To avoid a "Zeno's paradox" condition where smaller and smaller times are added and the collision is never reached, we must add a lower bound to the time increment. So rather than just adding t_c as derived above, we added $t_w = \max(t_c, t_{min})$, where t_{min} is a constant (say 1 mSec or $\frac{1}{FrameRate}$) which determines the effective time resolution of the calculation.

As a side note here, we would like to mention the fact that since we can calculate the lower bound on collision time adaptively, we can give a fairly good estimate of exact collision time to the precision in magnitude of t_{min}. In addition, since the lower bound on time to collision is calculated adaptively for the object most likely to collide first, it is impossible for the algorithm to fail to detect an interpenetration.

This can be done by modifying t_{min}, the effective time resolution, and the user defined safety tolerance ϵ according to the environment so as to avoid the case where one object collides with the other between time steps. ϵ is used because the polyhedra may be actually shrunk by ϵ amount to approximate the actual object. Therefore, the collision should be declared when the distance between two convex polytopes is less than 2ϵ. This is done to ensure that we can always report a collision or near-misses.

5.1.2 The Overall Approach

Our scheme for dynamic simulation using our distance computation algorithm is an iterative process which continuously inserts and deletes the object pairs from a heap according to their approximate time to collision, as the objects move in a dynamic environment.

It is assumed that there is a function $A_i(t)$ given for each object, which returns the object's pose at time t, as a 4x4 matrix. Initially, all poses are determined at $t = 0$, and the distance calculation algorithm in Chapter 3 is run on all pairs of objects that might collide. The pairs are inserted into the heap according to their approximate time to collision.

Then the first pair of objects is pulled off the queue. Its closest feature pair at $t = 0$ will be available, and the distance measuring algorithm from Chapter refdist is run. If a collision has occurred and been reported, then the pair is re-inserted in the queue with a minimum time increment, t_{min}. If not, a new lower bound on time-to-collision for that pair is computed, and the pair is re-inserted in the queue. This process is repeated until the wakeup time of the head of the queue exceeds the simulation time.

Note that the lower bound on collision time is calculated adaptively for the pair most likely to collide. Therefore, no collision can be missed. We will not need to worry about those sleeping pairs (which will not collide before their wake-up time), until the clock reaches the wake-up time W_i for each pair P_i.

This scheme described above can take care of all object pairs efficiently so that the distant object pairs wake up much less frequently. Thus, it reduces the run time in a significant way.

If no upper bounds on the velocity and acceleration can be assumed, in the next section we propose algorithms which impose a bounding box hierarchy on each object in the environment to reduce the naive bound of $\begin{pmatrix} N \\ 2 \end{pmatrix}$ pairwise comparisons for a dynamic environment of n objects. Once the object pairs' bounding boxes already collide, then we can apply the algorithm described in Chapter 3 to perform collision detection and to find the exact contact points if applicable.

5.2 Sweep & Sort and Interval Tree

In the three-dimensional workspace, if two bodies collide then their projections down to the lower-dimensional $x \Leftrightarrow y, y \Leftrightarrow z, x \Leftrightarrow z$ hyperplanes must overlap as well. Therefore, if we can efficiently *update* their overlapping status in each axis or in a 2-dimensional plane, we can easily eliminate the object pairs which are definitely not in contact with each other. In order to quickly determine all object pairs overlapping in the lower dimensions, we impose a virtual bounding box hierarchy on each body.

5.2.1 Using Bounding Volumes

To compute exact collision contacts, using a bounding volume for an interference test is not sufficient, but it is rather efficient for eliminating those object pairs which are of no immediate interests from the point of collision *detection*. The bounding box can either be spherical or rectangular (even elliptical), depending on the application and the environment. We prefer spherical and rectangular volume due to their simplicity and suitability in our application.

Consider an environment where most of objects are elongated and only a few objects (probably just the robot manipulators in most situations) are moving, then rectangular bounding boxes are preferable. In a more dynamic environment like a vibrating parts feeder where all objects are rather "fat" [68] and bouncing around, then spherical bounding boxes are more desirable. If the objects are concave or articulated, then a subpart-hierarchical bounding box representation (similar to subpart-hierarchical tree representation, with each node storing a bounding box) should be employed. The reasons for using each type of the bounding volumes are as follows.

Using a spherical bounding volume, we can precompute the box during the preprocessing step. At each time step, we only need to update the center of each spherical volume and get the minimum and maximum $x, y, z-$coordinates almost instantaneously by subtracting the measurement of radius from the coordinates of the center. This involves only one vector-matrix multiplication and six simple arithmetic operations (3 addition and 3 subtraction). However, if the objects are rather oblong, then a sphere is a rather poor bounding volume to use.

Therefore, a rectangular bounding box emerges to be a better choice for elogated objects. To impose a virtual rectangular bounding volume on an object rotating and translating in space involves a recomputation of the rectangular bounding volume. Recomputing a rectangular bounding volume is done by updating the maximum and minimum $x, y, z-$coordinates at each time instance. This is a simple procedure which can be done at constant time for each body. We can update the "min" and "max" by the following approaches:

Since the bounding boxes are convex, the maximum and minimum in the $x, y, z-$ coordinates must be the coordinates of vertices. We can use a modified *vertex-face* routine from the incremental distance computation algorithm described in Chapter 3. We can set up 6 imaginary boundary walls, each of these walls is located at the maximal and minimal $x, y, z-$ coordinates possible in the environment. Given the previous bounding volume, we can update each vertex of the bounding volume by performing only half of modified *vertex-face* test, since all the vertices are always in the Voronoi regions of these boundary walls. We first find the nearest point on the boundary wall to the previous vertex (after motion transformation) on the bounding volume, then we verify if the nearest point on the boundary wall lies inside of the Voronoi region of the previous vertex. If so, the previous vertex is still the extremal point (minimum or maximum in x, y, or $z-$axis). When a constraint is violated by the nearest point on the face (wall) to the previous extremal vertex, the next feature returned will be the neighboring *vertex*, instead of the neighboring edge. This is a simple modification to the point-vertex applicability criterion routine. It still preserves the properties of locality and coherence well. This approach of recomputing bounding volume dynamically involves 6 *point-vertex* applicability tests.

Another simple method can be used based on the property of convexity. At each time step, all we need to do is to check if the current minimum (or maximum) vertex in $x-$(or $y, z-$)coordinate still has the smallest (or largest) $x-$(or $y, z-$)coordinate values in comparison to its neighboring vertices. By performing this verification process recursively, we can recompute the bounding boxes at expected constant rate. Once again, we are exploiting the temporal and geometric coherence and the locality of convex polytopes. We only update the bounding volumes for the moving objects. Therefore, if *all* objects are moving around (especially in a rather unpredictable way), it's hard to preserve the coherence well and the update overhead may slow down the overall computation.

But, we can speed up the performance of this approach by realizing that we are only interested in one coordinate value of vertices for each update, say x coordinate while updating the minimum or maximum value in x-axis. Therefore, there is no need to transform the other coordinates, say y and z values, in updating the x extremal vertices during the comparison with neighboring vertices. Therefore, we only need to perform 24 vector-vector multiplications. (24 comes from 6 updates in minimum and maximum in $x, y, z-$ coordinates and each update involves 4 vector-vector multiplications, assuming each vertex has 3 neighboring vertices.)

For concave and articulated bodies, we need to use a hierarchical bounding box structure, i.e. a tree of bounding boxes. Before the top level bounding boxes collide, there is no need to impose a bounding volume on each subpart or each link. Once the collision occurs between the parent bounding boxes, then we compute the bounding boxes for each child (subpart or linkage). At last we would like to briefly mention that in order to report "near-misses", we should "grow" the bounding boxes by a small amount to ensure that we perform the exact collision detection algorithm when two objects are about to collide, *not* after they collide.

Given the details of computing bounding volume dynamically, we will present two approaches which use "sort and sweep" techniques and a geometric data structure to quickly perform intersection test on the real intervals to reduce the number of pairwise comparison.

5.2.2 One-Dimensional Sort and Sweep

In computational geometry, there are several algorithms which can solve the overlapping problem for d-dimensional bounding boxes in $O(nlog^{d-1}n + s)$ time where s is the number of pairwise overlaps [31, 47, 80]. This bound can be improved using coherence.

Let a one-dimensional bounding box be $[b, e]$ where b and e are the real numbers representing the beginning and ending points. To determine all pairs of overlapping intervals given a list of n intervals, we need to verify for all pairs i and j if $b_i \in [b_j, e_j]$ or $b_j \in [b_i, e_i]$, $1 \leq i < j \leq n$. This can be solved by first sorting a list of all b_i and e_i values, from the lowest to the highest. Then, the list is traversed to find all the intervals which overlap. The sorting process takes $O(nlogn)$ and $O(n)$ to sweep through a sorted list and $O(s)$ to output each overlap where s is the number of overlap.

For a sparse and dynamic environment, we do not anticipate each body to make a relatively large movement between time steps, thus the sorted list should not change much. Consequently the last sorted list would be a good starting point to continue. To sort a "nearly sorted" list by *bubble sort* or *insertion sort* can be done in $O(n + e)$ where e is the number of exchanges.

All we need to do now is to keep track of "status change", i.e. from overlapping in the last time step to non-overlapping in the current time step and vise versa. We keep a list of overlapping intervals at all time and update it whenever there is a status change. This can be done in $O(n + e_x + e_y + e_z)$ time, where e_x, e_y, e_z are the number of exchanges along the x, y, z-coordinate. Though the update can be done in linear time, e_x, e_y, e_z can be $O(n^2)$ with an extremely small constant. Therefore, the *expected* run time is linear in the total number of vertices.

To use this approach in a three-dimensional workspace, we pre-sort the minimum and maximum values of each object along the $x, y, z-$axis (as if we are imposing a virtual bounding box hierarchy on each body), sweep through each nearly sorted list every time step and update the list of overlapping intervals as we mentioned before. If the environment is sparse and the motions between time frames are "smooth", we expect the extra effort to check for collision will be negligible. This "pre-filtering" process to eliminate the pairs of objects *not* likely to collide will run essentially in linear time. A similar approach has been mentioned by Baraff in [48].

This approach is especially suitable for an environment where only a few objects are moving while most of objects are stationary, e.g. a virtual walk-through environment.

5.2.3 Interval Tree for 2D Intersection Tests

Another approach is to extend the one-dimensional sorting and sweeping technique to higher dimensional space. However, as mentioned earlier, the time bound will be worse than $O(n)$ for two or three-dimensional sort and sweep due to the difficulty to make a tree structure dynamic and flexible for quick insertion and deletion of a higher dimensional boxes. Nevertheless, for more dense or special environments (such as a mobile robot moving around in a room cluttered with moving obstacles, such as people), it is more efficient to use an interval tree for 2-dimensional intersection tests to reduce the number of pairwise checks for overlapping. We can significantly reduce the extra effort in verifying the exchanges checked by the one-dimensional sort and sweep. Here we will briefly describe the data structure of an interval tree and how we use it for intersection test of 2-dimensional rectangular boxes.

An interval tree is actually a *range tree* properly annotated at the nodes for fast search of real intervals. Assume that n intervals are given, as $[b_1, e_1], \cdots, [b_n, e_n]$ where b_i and e_i are the endpoints of the interval as defined above. The range tree is constructed by first sorting all the endpoints into a list $(x_1, \cdots, x_m)$ in ascending order, where $m \leq 2n$. Then, we construct the range tree top-down by splitting the sort list L into the left subtree L_l and the right subtree L_r, where $L_l = (x_1, \cdots, x_p)$ and $L_r = (x_{p+1}, \cdots, x_m)$. The root has the split value $\frac{x_p + x_{p+1}}{2}$. We construct the subtrees within each subtree recursively in this fashion till each leave contains only an endpoint. The construction of the range tree for n intervals takes $O(nlogn)$ time. After we construct the range tree, we further link all nodes containing stored intervals in a doubly linked list and annotate each node if it or any of its descendant contains stored intervals. The embellished tree is called the *interval tree.*

We can use the interval tree for static query, as well as for the rectangle intersection problem. To check for rectangle intersection using the sweep algorithm: we take a sweeping line parallel to the y axis and sweep in increasing x direction, and look for overlapping y intervals. As we sweep across the x axis, y intervals appears or disappear. Whenever there is an appearing y interval, we check to see if the new interval intersects the old set of intervals stored in the interval tree, report all intervals it intersects as rectangle intersection, and add the new interval to the tree.

Each query of interval intersection takes $O(logn + k)$ time where k is the number of reported intersection and n is the number of intervals. Therefore, reporting intersection among n rectangles can be done in $O(nlogn + K)$ where K is the total number of intersecting rectangles.

5.3 Other Approaches

Here we will also briefly mention a few different approaches which can be used in other environments or applications.

5.3.1 BSP-Trees and Octrees

One of the commonly used tree structure is BSP-tree (binary space partitioning tree) to speed up intersection tests in CSG (constructive solid geometry) [83]. This approach construct a tree from separating planes at each node recursively. It partitions each object into groups of parts which are close together in binary space. When, the separation planes are chosen to be aligned with the coordinate axes, then a BSP tree becomes more or less like an octree.

One can think of an octree as tree of cubes within cubes. But, the size of the cube varies depending on the number of objects occupying that region. A sparsely populated region is covered by one large cube, while a densely occupied region is divided into more smaller cubes. Each cube can be divided into 8 smaller cubes if necessary. So, each node in the tree has 8 children (leaves).

Another modified version of BSP-Tree proposed by Vanecek [37] is a multi-dimensional space partitioning tree called *Brep-Index*. This tree structure is used for collision detection [10] between moving objects in a system called *Proxima* developed at Prudue University.

The problem with tree structures is similar to that of using 2-d interval tree that its update (insertion and deletion) is inflexible and cumbersome, especially for a large tree. The overhead of insertion and deletion of a node in a tree can easily dominate the run time, especially when a collision occurs. The tree structures also cannot capture the temporal and spatial coherence well.

5.3.2 Uniform Spatial Subdivision

We can divide the space into unit cells (or volumes) and place each object (or bounding box) in some cell(s) [68]. To check for collisions, we have to examine the cell(s) occupied by each box to verify if the cell(s) is(are) shared by other objects. But, it is difficult to set a near-optimal size for each cell and it requires tremendous amount of allocated memory. If the size of the cell is not properly chosen, the computation can be rather expensive. For an environment where almost all objects are of uniform size, like a vibrating parts feeder bowl or molecular modeling [85, 68], this is a rather ideal algorithm, especially to run on a parallel-computing machine. In fact, Overmars has shown that using a hash table to look up an enetry, we can use a data structure of O(n) storage space to perform the point location queries in constant time [68].

5.4 Applications in Dynamic Simulation and Virtual Environment

The algorithms presented in this chapter have been utilized in dynamic simulation as well as in a walk-through environment. These applications attest for the practicality of the algorithms and the importance of the problem natures.

The algorithm described in Sec. 5.1 and the distance computation algorithm described in Chapter 3 have been used in the dynamics simulator written by Mirtich

[62]. It reduces the frequency of the collision checks significantly and helps to speed up the calculations of the dynamic simulator considerably. Our vision of this dynamic simulator is the ability to to simulate thousands of small mechanical parts on a vibrating parts feeder in real time. Looking at our current progress, we believe such a goal is attainable using the collision detection algorithms described in this thesis.

At the same time, the algorithm described in Sec. 5.2.2 is currently under testing to be eventually integrated into a Walk-Through environment developed at the University of North Carolina, Chapel Hill on their in-house Pixel Plane machine [24]. In a virtual world like "walk-through environment" where a human needs to interact with his/her surrounding, it is important that the computer can simulate the interactions of the human participants with the passively or actively changing environment. Since the usual models of "walk-through" are rather complex and may have thousands of objects in the world, an algorithm as described in Sec. 5.2.2 becomes an essential component to generate a realism of motions.

Chapter 6

An Opportunistic Global Path Planner

The algorithm described in Chapter 3 is a key part of our general planning algorithm presented in this chapter. This path planning algorithm creates an one-dimensional roadmap of the free space of a robot by tracing out curves of maximal clearance from obstacles. We use the distance computation algorithm to incrementally compute distances between the robot pieces and the nearby obstacles. From there we can easily compute gradients of the distance function in configuration space, and thereby find the direction of the maximal clearance curves.

This is done by first finding the pairs of closest features between the robot and the obstacles, and then keeping track of these closest pairs incrementally by calls to this algorithm. The curves traced out by this algorithm are in fact maximally clear of the obstacles. As mentioned earlier, once a pair of initialization features in the vicinity of the actual closest pair is found, the algorithm takes a very short time (usually constant) to find the actual closest pair of features. Given the closest features, it is straight forward to compute the gradient of the distance function in configuration space which is what we need to trace the skeleton curves.

In this chapter, we will describe an opportunistic global path planner which uses the opportunistic local method (Chapter 3) to build up the one-dimensional skeleton (or freeway) and global computation to find the critical points where linking curves (or bridges) are constructed.

6.1 Background

There have been two major approaches to motion planning for manipulators, (i) local methods, such as artificial potential field methods [50], which are usually fast but are not guaranteed to find a path, and (ii) global methods, like the first Roadmap Algorithm [18], which is guaranteed to find a path but may spend a long time doing it. In this chapter, we present an algorithm which has characteristics of both. Our method is an incremental construction of a skeleton of free-space. Like the potential field methods, the curves of this skeleton locally maximizes a certain potential function that varies with distance from obstacles. Like the Roadmap Algorithm, the skeleton, computed incrementally, is eventually guaranteed to contain a path between two configurations if one exists. The size of the skeleton in the worst case, is comparable with the worst-case size of the roadmap.

Unlike the local methods, our algorithm never gets trapped in local extremal points. Unlike the Roadmap Algorithm, our incremental algorithm can take advantage of a non-worst-case environment. The complexity of the roadmap came from the need to take recursive slices through configuration space. In our incremental algorithm, slices are only taken when an initial search fails and there is a "bridge" through free space linking two "channels". The new algorithm is no longer recursive because bridges can be computed directly by hill-climbing . The bridges are built near "interesting" critical points and inflection points. The conditions for a bridge are quite strict. Possible candidate critical points can be locally checked before a slice is taken. We expect few slices to be required in typical environments.

In fact, we can make a stronger statement about completeness of the algorithm. The skeleton that the algorithm computes eventually contains paths that are homotopic to all paths in free space. Thus, once we have computed slices through all the bridges, we have a complete description of free-space for the purposes of path planning. Of course, if we only want to find a path joining two given points, we stop the algorithm as soon as it has found a path.

The tracing of individual skeleton curves is a simple enough task that we expect that it could be done in real time on the robot's control hardware, as in other artificial potential field algorithms. However, since the robot may have to backtrack to pass across a bridge, it does not seem worthwhile to do this during the search.

For those readers already familiar with the Roadmap Algorithm, the following description may help with understanding of the new method: If the configuration space is $\mathbb{R}^k$, then we can construct a hypersurface in $\mathbb{R}^{k+1}$ which is the graph of the potential function, i.e. if $P(x_1, \ldots, x_k)$ is the potential, the hypersurface is the set of all points of the form $(x_1, \ldots, x_k, P(x_1, \ldots, x_k))$. The skeleton we define here is a subset of a roadmap (in the sense of [18]) of this hypersurface.

This work builds on a considerable volume of work in both global motion planning methods [18] [54], [73], [78], and local planners, [50]. Our method shares a common theme with the work of Barraquand and Latombe [6] in that it attempts to use a local potential field planner for speed with some procedure for escaping local maxima. But whereas Barraquand and Latombe's method is a local method made global, we have taken a global method (the Roadmap Algorithm) and found a local opportunistic way to compute it.

Although our starting point was completely different, there are some other similarities with [6]. Our "freeways" resemble the valleys intuitively described in [6]. But the main difference between our method and the method in [6] is that we have a guaranteed (and reasonably efficient) method of escaping local potential extremal points and that our potential function is computed in the configuration space.

The chapter is organized as follows: Section 6.2 contains a simple and general description of roadmaps. The description deliberately ignores details of things like the distance function used, because the algorithm can work with almost any function. Section 6.3 gives some particulars of the application of artificial potential fields. Section 6.4 describes our incremental algorithm, first for robots with two degrees of freedom, then for three degrees of freedom. Section 6.5 gives the proof of completeness for this algorithm.

6.2 A Maximum Clearance Roadmap Algorithm

We denote the space of all configurations of the robot as CS. For example, for a rotary joint robot with k joints, the configuration space CS is $\mathbb{R}^k$, the set of all joint angle tuples $(\theta_1, \ldots, \theta_k)$. The set of configurations where the robot overlaps some obstacle is the configuration space obstacle CO, and the complement of CO is the set of free (non-overlapping) configurations FP. As described in [18], FP is bounded by algebraic hypersurfaces in the parameters t_i after the standard substitution $t_i = \tan(\frac{\theta_i}{2})$. This result is needed for the complexity bounds in [18] but we will not need it here.

A roadmap is a one-dimensional subset of FP that is guaranteed to be connected within each connected component of FP. Roadmaps are described in some detail in [18] where it is shown that they can be computed in time $O(n^k \log n(d^{O(n^2)}))$ for a robot with k degrees of freedom, and where free space is defined by n polynomial constraints of degree d [14]. But n^k may still be too large for many applications, and in many cases the free space is much simpler than its worst case complexity, which is $O(n^k)$. We would like to exploit this simplicity to the maximum extent possible. The results of [6] suggest that in practice free space is usually much simpler than the worst case bounds. What we will describe is a method aimed at getting a minimal description of the connectivity of a particular free space. The original description of roadmaps is quite technical and intricate. In this paper, we give a less formal and hopefully more intuitive description.

6.2.1 Definitions

Suppose CS has coordinates $x_1, \ldots, x_k$. A slice $CS|_v$ is a slice by the hyperplane $x_1 = v$. Similarly, slicing FP with the same hyperplane gives a set denoted $FP|_v$. The algorithm is based on the key notion of a channel which we define next:

A *channel-slice* of free space FP is a connected component of some slice $FP|_v$.

The term channel-slice is used because these sets are precursors to channels. To construct a channel from channel slices, we vary v over some interval. As we do this, for most values of v, all that happens is that the connected components of $FP|_v$ change shape continuously. As v increases, there are however a finite number of values of v, called *critical values*, at which there is some topological change. Some events are not significant for us, such as where the topology of a component of the cross-section changes, but there are four important events: As v increases a connected component of $FP|_v$ may appear or disappear, or several components may join, or a single component may split into several. The points where joins or splits occur are called *interesting critical points*. We define a channel as a maximal connected union of cross sections that contains no image of interesting critical points. We use the notation $FP|_{(a,b)}$ to mean the subset of FP where $x_1 \in (a,b) \subset \mathbb{R}$.

A *channel* through FP is a connected component of $FP_{(a,b)}$ containing no splits or joins, and (maximality) which is not contained in a connected component of $FP_{(c,d)}$ containing no splits or joins, for (c,d) a proper superset of (a,b). See Fig. 6.1 for an example of channels.

The property of no splits or joins can be stated in another way. A maximal connected set $C|_{(a,b)} \subset FP|_{(a,b)}$ is a channel if every subset $C|_{(e,f)}$ is connected for $(e,f) \subset (a,b)$.

6.2.2 The General Roadmap

Now to the heart of the method. A roadmap has two components:

(i) Freeways (called silhouette curves in [18]) and

(ii) Bridges (called linking curves in [18]).

A freeway is a connected one-dimensional subset of a channel that forms a backbone for the channel. The key properties of a freeway are that it should span the channel, and be continuable into adjacent channels. A freeway *spans* a channel if its range of x_1 values is the same as the channels, i.e. a freeway for the channel $C|_{(a,b)}$ must have points with all x_1 coordinates in the range (a,b). A freeway is *continuable* if it meets

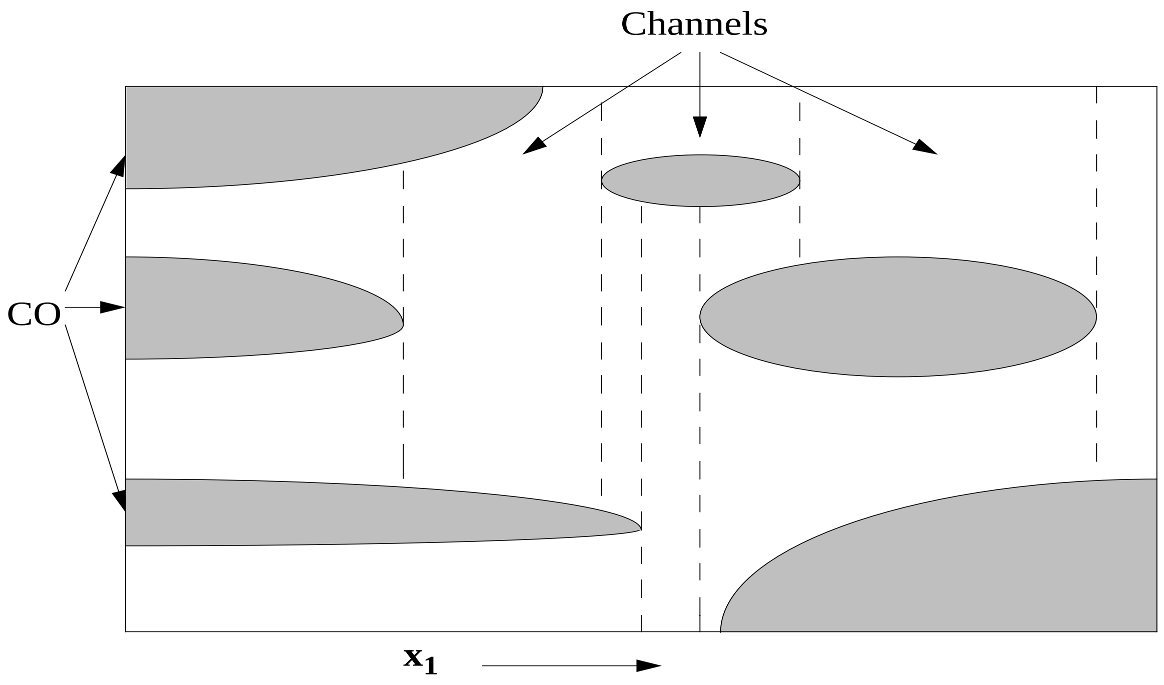


Figure 6.1: A schematized 2-d configuration space and the partition of free space into x_1-channels.

another freeway at its endpoints. i.e. if $C|_{(a,b)}$ and $C'|_{(b,c)}$ are two adjacent channels, the b endpoint of a freeway of $C|_{(a,b)}$ should meet an endpoint of a freeway of $C'|_{(b,c)}$. (Technically, since the intervals are open, the word "endpoint" should be replaced by "limit point")

In general, when a specific method of computing freeway curves is chosen, there may be several freeways within one channel. For example, in the rest of this chapter, freeways are defined using artificial potential functions which are directly proportional to distance from obstacles. In this case each freeway is the locus of local maxima in potential within slices $FP|_v$ of FP as v varies. This locus itself may have some critical points, but as we shall see, the freeway curves can be extended easily past them. Since there may be several local potential maxima within a slice, we may have several disjoint freeway curves within a single channel, but with our incremental roadmap construction, this is perfectly OK.

Now to bridges. A bridge is a one-dimensional set which links freeways from channels that have just joined, or are about to split (as v increases). Suppose two channels C_1 and C_2 have joined into a single channel C_3, as shown in Fig. 6.2. We know that the freeways of C_1 and C_2 will continue into two freeway curves in C_3. These freeways within C_3 are not guaranteed to connect. However, we do know that by definition C_3 is connected in the slice slice $x_1 = v$ through the critical point, so we add linking curves from the critical point to *some* freeway point in each of C_1 and C_2. It does not matter which freeway point, because the freeway curves inside the channels C_1 and C_2 must be connected within each channel, as we show in Sec. 6.5. By adding bridges, we guarantee that whenever two channels meet (some points on) their freeways are connected.

Once we can show that whenever channels meet, their freeways do also (via bridges), we have shown that the roadmap, which is the union of freeways and bridges, is connected. The proof of this very intuitive result is a simple inductive argument on the (finite number of) channels, given in Sec. 6.5.

The basic structure of the general Roadmap Algorithm follows:

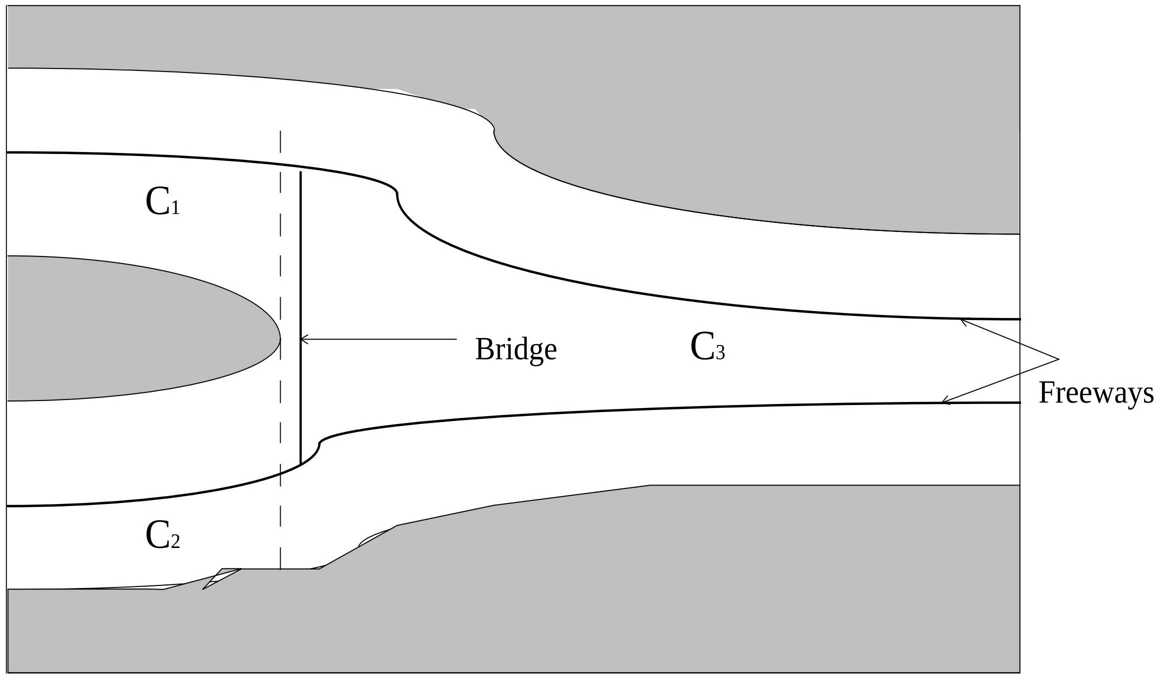


Figure 6.2: Two channels C_1 and C_2 joining the channel C_3, and a bridge curve in C_3.

1. Start tracing a freeway curve from the start configuration, and also from the goal.

2. If the curves leading from the start and goal are not connected, enumerate a split or join point, and add a bridge curve "near" the split or join (x_1-coordinate of the slice slightly greater than that of the joint point for a join, slightly less for a split).

3. Find all the points on the bridge curve that lie on other freeways, and trace from these freeways. Go to step (2).

The algorithm terminates at step (2) when either the start and goal are connected, in which case the algorithm signals success and returns a connecting path, or if it runs out of split and join points, in which case there is no path connecting the start and goal. This description is quite abstract, but in later sections we will give detailed description of the approach in two- and three-dimensional configuration spaces.

Three things distinguish our new algorithm from the previous Roadmap Algorithm. The most important is that the new algorithm is not recursive. Step 2 involves adding a bridge curve which is two pieces of curve found by hill-climbing on the potential. In the original roadmap algorithm, linking curves had to be defined recursively, because it is not possible to hill-climb to a maximum with an algebraic curve. Another difference is that the freeways do not necessarily lie near the boundary of free space as they did in [18]. In our present implementation we are in fact using maximum clearance freeways. But the most important difference is that we now only enumerate *true* split or join points. For a robot with k degrees of freedom and an environment of complexity n, it can be shown that there are at most $O(n^{(k-1)})$ potential split or join points. (Please refer to Sec. proof.planner for the proof on the upper bound for the maximum number of interesting critical points.) But many experiments with implemented planners in recent years have shown that the number of true splits or joins in typical configuration spaces is much lower. In our new algorithm, we can make a purely local test on a potential split or join point to see if it is really qualified. The vast majority of candidates will not be, so we expect far fewer than $O(n^{(k-1)})$ bridges to be required.

Definition

A point p in $\mathbb{R}^{k+1}$ is an *interesting critical point* if for every neighborhood U of p, one of the following holds:

(i) The intersection $U \cap x_1^{-1}(x_1(p) + \epsilon)$ consists of several connected components for all sufficiently small ϵ. This is a generalized split point.

(ii) The intersection $U \cap x_1^{-1}(x_1(p) - \epsilon)$ consists of several components for all sufficiently small ϵ. This is a generalized join point.

We will assume the environment is generic, i.e. there is no special topology such that a small perturbation will change the clearance of the paths. This is true for almost all practical situations: most obstacles have a reasonably large interior space that a small perturbation will not affect much of the obstacle configuration space. Based on the transversality condition of general position assumptions in [18], the interesting critical points can be computed as follows. Let S be the set of equations used to calculate the critical points. The set S is defined by inequalities, and its boundary is a union of surfaces of various dimensions. Let S_α be such a surface; it will be defined as the intersection of several configuration space constraint surfaces. Each of these is given by an equation of the form $f_i = 0$. To find the critical points of such a surface w.r.t. the function $x_1(.)$, we first define a polynomial g as follows:

$$g = \sum_{i=1}^{l} f_i^2 \tag{6.1}$$

and then solve the system

$$g = \epsilon, \quad \frac{\partial}{\partial x_2} g = 0 \quad \cdots \quad \frac{\partial}{\partial x_k} g = 0 \tag{6.2}$$

where l is the number of equations which are zero on S_α, the $x_2, \ldots, x_k$ are coordinates which are orthogonal to x_1, and ϵ is an infinitesimal that is used to simplify the computation (see [18]).

It can be shown [21] that the solutions of interest can be recovered from the lowest degree coefficient in ϵ of the resultant of this system. This normally involves computing a symbolic determinant which is a polynomial in ϵ [57]. But a more practical approach is to recover only the lowest coefficient in ϵ by using straight line program representations and differentiating [33].

To enumerate all the interesting critical points is computationally expensive, since we have to solve $O(n^{(k-1)})$ systems of non-linear equations. Thus, we also plan to experiment with randomly chosen slice values in some bounded ranges, alternating with slices taken at true split or join points. The rationale for this is that in practice the "range" of slice values over which a bridge joins two freeways is typically quite large. There is a good probability of finding a value in this range by using random values. Occasionally there will be a wide range of slice values for a particular bridge, but many irrelevant split and join points may be enumerated with values outside this range. To make sure we do not make such easy problems harder than they should be, our implementation alternates slices taken near true split and join points with slices taken at random x_1 values.

6.3 Defining the Distance Function

The idea of our approach is to construct a potential field which repels the point robot in configuration space away from the obstacles. Given a goal position and a description of its environment, a manipulator will move along a "maximum potential" path in an "artificial potential field". The position to be reached represents a critical point that will be linked by the *bridge* to the nearest maximum, and the obstacles represent repulsive surfaces for the manipulator parts.

Let CO denote the obstacles, and x the position in $\mathbb{R}^k$. The artificial potential field $U_{art}(x)$ induces an artificial repulsion from the surface of the obstacles. $U_{art}(x)$ is a non-negative function whose value tends to zero as any part of the robot approaches an obstacle. One of the classical analytical potential fields is the Euclidean distance function.

Using the shortest distance to an obstacle O, we have proposed the following potential field $U_{art}(x)$:

$$U_{art}(x) = \min_{ij}(D(O_i, L_j(x)))$$

where $D(O_i, L_j(x))$ is the shortest Euclidean distance between an obstacle O_i and the link L_j when the robot is at configuration x. $D(O_i, L_j(x))$ is obtained by a local method for fast computation of distance between convex polyhedra in Chapter 3.

Notice that the proposed $U_{art}(x)$ is not a continuously differentiable function as in many potential field methods. $U_{art}(x)$ is *piecewise* continuous and differentiable. This is perfectly all right for the application in our Roadmap algorithm. In fact it will be a lower envelope of smooth functions. This is all the better because it means that local maxima that do not occur where the function is smooth are all the more sharply defined. The graph of the distance function certainly has a *stratification* into a finite number of smooth pieces [87]. Its maxima will be the union of certain local maxima of these smooth pieces. So we can still use the system of equations defined earlier to find them.

With this scheme, a manipulator moves in such a way to maximize the artificial potential field $U_{art}(x)$. But like any local method, just following one curve of such maxima is not guaranteed to reach the goal. Thus, the need for bridges.

6.4 Algorithm Details

The algorithm takes as input a geometric description of the robot links and obstacles as convex polyhedra or unions of convex polyhedra. It also takes the initial and goal configurations, and the kinematic description of the robot, say via Denavit-Hartenberg parameters. The output is a path between the initial and goal configurations represented as a sequence of closely spaced points (more closely than the C-space distance to the nearest obstacle at that point), assuming such a path exists. If there is no path, the algorithm will eventually discover that, and output "NO PATH".

The potential function is a map $U_{art} : CS \rightarrow \mathbb{R}$. The graph of the function is a surface in $CS \times \mathbb{R}$. Let u and v denote two coordinate axes, the Roadmap

Algorithm fixes v and then follows the extremal points in direction u as the value of v varies. But, the new algorithm differs from the original roadmap algorithm[18] in the following respects:

- It does not always construct the entire roadmap
- In the new algorithm, $v = x_i$, where x_i is one of the CS coordinates while $u = h$, where h is the height of the potential function. Yet, in the original, $v = x_i$ and $u = x_j$ where x_i and x_j are *both* CS coordinates.
- The original Roadmap algorithm fixes the x_i coordinate and follows extremal points (maxima, minima and saddles) in x_j, to generate the silhouette curves. On the other hand, the new algorithm fixes x_i, and follows only *maxima* in h.
- The new algorithm is not recursive. Recursion was necessary in the original because there is no single algebraic curve that connects an arbitrary point to an extremum in u. But the new algorithm uses numerical hill-climbing which has no such limitation.

6.4.1 Freeways and Bridges

A roadmap has two major components – freeways and bridges. They are generated as following:

Freeway Tracing is done by tracking the locus of local maxima in distance within each slice normal to the sweeping direction. Small steps are made in the sweep direction, and the local maxima recomputed numerically. Freeway tracing continues in both directions along the freeway until it terminates in one of two ways:

(a) The freeway runs into an inflection point, a point where the curve tangent becomes orthogonal to the sweep direction. It is always possible to continue past these points by adding a bridge.

(b) The freeway runs into an obstacle. This is a normal termination, and the tracing simply stops and the algorithm backtracks.

Bridges begin always at inflection points or critical points and terminate always at freeway points within the same slice. The algorithm simply follows the gradient of the potential function from the start point within the slice until it reaches a local maximum, which must be a freeway point.

Enumeration of Critical Points

Critical points are calculated as in Section 2. But most of these critical points will not be interesting. We can check locally among all the critical points to see if they qualify to be a "split" or "join". This test checks if the point has a neighborhood that is "saddle-like". It is based on the orientations of the CSpace boundary surface normals at the critical point.

Random Slicing

Optionally, the user may wish to add roadmaps of randomly chosen slices, rather than calculating many critical points (or rather than calculating them at all, but then of course, completeness will be lost). This *is* a recursive procedure, and involves choosing a v value at random, making a recursive call to the algorithm on this v-slice.

Random slicing may also be used within the slice itself, and so on, which leads to a depth-k recursion tree. If this is done, some search heuristics must be added to guide the choice of where in the tree the next slice (and hence the next child) should be added. The heuristic also needs to trade off random slicing and critical point enumeration. The goal of the heuristic is to enumerate enough random slices that the algorithm will have a good chance of success on "easy" environments (intuitively where there are large passages between channels) without having to explore too many critical points. Yet it should still find its way around a difficult environment using the critical slices without having wasted most of its time taking random slices.

Given the general outline of our algorithm, we will now give an instantiation on 2-D and a detailed description of how it can be applied to 3-D.

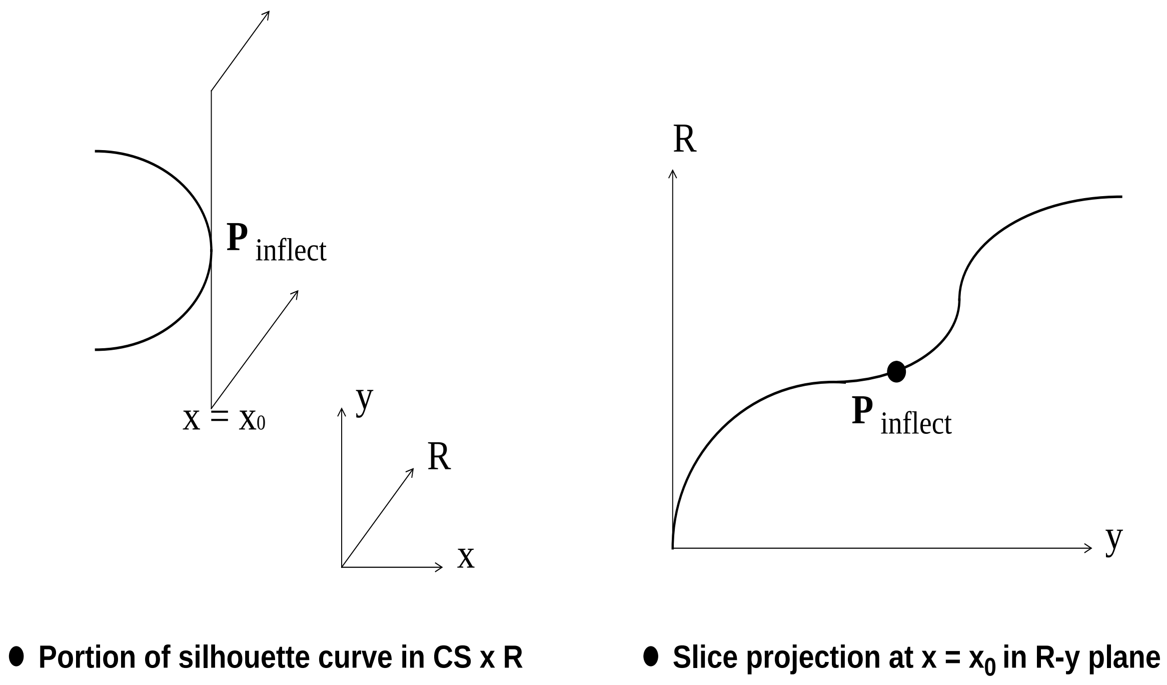

P inflect
x = x0
y
R
x
R
P inflect
y
Portion of silhouette curve in CS x R
Slice projection at x = x0 in R-y plane

Figure 6.3: A pictorial example of an inflection point in $CS \times \mathbb{R}$ vs. its view in $\mathbb{R} \times y$ at the slice $x = x_0$

6.4.2 Two-Dimensional Workspace

Starting from the initial position $p_{init} \in CS$, we first fix one of the axes of CS and then take the x coordinate of a slice to be the x coordinate of p_{init}. Then we search this slice to find the nearest local maximum. (This local maximum is a freeway point.) Next, we build a *bridge* between the point p_{init} and this local maximum. At the same time, we begin tracing a freeway curve from the goal. If the goal is not on the maximum contour of the potential field, then we must build a *bridge* to link it to the nearest local maximum. Afterwards, we trace the locus of this local maximum as x varies until we reach an endpoint. If the current position p_{loc} on the curve is the goal G, then we can terminate the procedure. Otherwise, we must verify whether p_{loc} is a "dead-end" or an inflection point of the slice $x = x_0$. (See Fig. 6.3.) If p_{loc} is a point of inflection, then we can continue the curve by taking a slice at the neighborhood of the inflection point and hill-climbing along the gradient direction *near* the inflection point. This search necessarily takes us to another local maximum.

Fig. 6.4 demonstrates how the algorithm works in 2-d CS. This diagram is a projection of a constructed potential field in CS x $\mathbb{R}$ onto the x-y plane of the 2-d CS. The shaded area is the CO in the configuration space. The solid curves represent the contour of maximum potential, while the dashed curves represent the minima. Furthermore, the path generated by our planner is indicated by arrows. In addition, the vertical lines symbolize channel slices through the interesting critical points and inflection points. When this procedure has been taken to its conclusion and both endpoints of the freeway terminate at dead-ends, then at this point it is necessary to take a slice at some value of x. Our planner generates several random x-values for slices (at a uniformly spaced distribution along the span of the freeway), interweaving them with an enumeration of all the interesting critical points. If after a specified number of random values, our planner fails to find a connecting path to a nearby local maximum, then it will take a slice through an interesting critical point. Each slice, being 1-dimensional, itself forms the bridge curve (or a piece of it does). We call this procedure repeatedly until we reach the goal position G or have enumerated all the interesting critical points.

The algorithm is described schematically below:

• Algorithm

```
Procedure FindGoal (Environment, $p_{init}$, G)

    if  ($p_{init} \neq G$)
        then Explore($p_{init}$) and Explore($G$)
        else return(FoundGoal);
    even := false;
    While ( CritPtRemain and NotOnSkeleton(G) ) do
        if (even)
            then x := Random (x-range)
            else x := x-coord(next-crit-pt());
        TakeSlice(x);
```

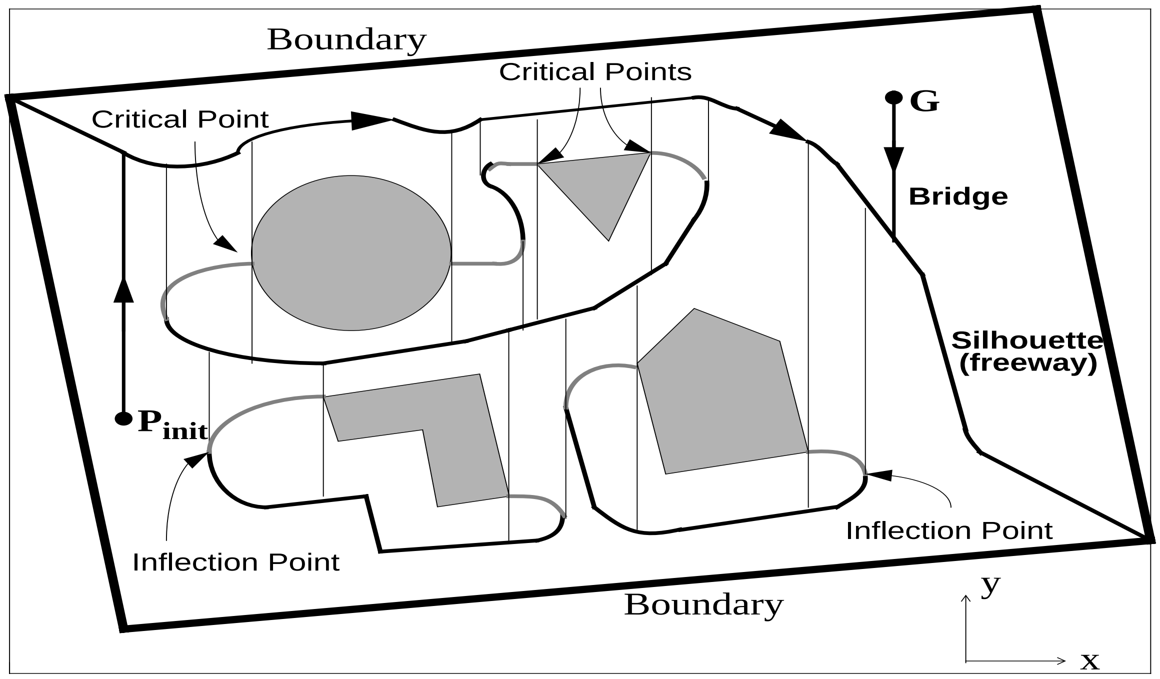


Figure 6.4: An example of the algorithm in the 2-d workspace

```
        even := not even;
    end(while);

End(FindGoal);


Function Explore(p)
% Trace out a curve from p

    q := search-up&down(p);
    % To move up & down only in $y$, using gradient near p
    if new(q) then
    % new() checks if q is already on the curve
        begin(if)
        <e1,e2> := trace(q);
        % trace out the curve from q, output two end points
        if inflection(e1) then Explore(e1);
        if inflection(e2) then Explore(e2);
        % inflection(p) checks if p is an inflection point
        end(if);

End(Explore);

Function TakeSlice(x-coordinate(p))
% This function generates all points on the slice and explore
% all the maxima on the slice.

    old-pt := find-pt(x-coordinate);
    % find-pt() find all the points on the x-coordinate.
    % It moves up&down until reaches another maximum.
    new-pt := null;
    For (each pt in the old-pt) do
        <up,down> := search-up&down(pt);
        % <up,down> is a pair of points of 0,1,or2 pts
        new-pt := new-pt + <up,down>;
        For (each pt in the new-pt) do
            Explore(pt);

End(TakeSlice);
```

6.4.3 Three-Dimensional Workspace

For a three-dimensional workspace, the construction is quite similar. Starting from the initial position p_{init} and the goal G, we first fix one axis, X. We trace from the start point to a local maximum of distance within the Y-Z plane containing the start point. Then we follow this local maximum by taking steps in X. If this curve terminates in an inflection point, we can always reach another maximum by following the direction of the potential gradient *just beyond* the inflection point in X. Eventually, though, we expect to terminate by running into an obstacle.

When we wish to enumerate a critical point, the bridge curve is the same as the first segment that we used to get from p_{init} onto the freeway. That is, we trace from the critical point along the direction of the gradient within the current Y-Z slice. There will be two directions outward from the critical point along which the distance increases. We follow both of these, which gives us a bridge curve linking freeways of two distinct channels.

If we decide to use random slicing, we select a slice $FP|_x$ normal to the x-axis and call the algorithm of the last section on that slice. We require it to produce a roadmap containing any freeway points that we have found so far that lie in this slice. This algorithm itself may take random slices, so we need to limit the total number of random slices taken before we enumerate the next interesting critical point (in 3-D), so that random slicing does not dominate the running time.

6.4.4 Path Optimization

After the solution path is obtained, we plan to smooth it by the classical principles of variational calculus, i.e. to solve a classical two points boundary value problem. Basically we minimize the potential which is a function of both distance and smoothness to find a *locally* optimal (smooth) path.

Let s be the arc that is the path refined from a sequence of points between a and b in space, r be the shortest Euclidean distance between the point robot and the obstacle, and κ be the curvature of the path at each point. The cost function for path optimization that we want to minimize is:

$$f(s, r, \kappa) = \int_a^b (\frac{A}{r^2} + B\kappa^2) ds$$

where r, κ, and s are functions of a point P_i in a given point sequence, and A, B are adjustment constants. Taking the gradient of this function with respect to each point P_i gives us the direction of an improved, *locally* optimal path.

This can be done in the following manner: given a sequence of points $(P_1, P_2, \cdots, P_k)$, we want to minimize the cost function

$$g(P_i) = \sum_i \frac{A}{r(P_i)^2}|\Delta S_i| + B\kappa(P_i)^2|\Delta S_i| \tag{6.3}$$

where ΔS_i and $\kappa(P_i)$ are defined as:

$$\Delta S_i = \frac{P_{i+1} - P_{i-1}}{2}$$

$$\kappa(P_i) = \frac{\angle P_{i-1}, P_i, P_{i+1}}{|\Delta S_i|}$$

Now, taking the gradient w.r.t. P_i, we have

$$\nabla g(P_i) = \sum_i \frac{-2A}{r(P_i)^3}\nabla r(P_i)|\Delta S_i| + 2B\kappa(P_i)\nabla\kappa(P_i)|\Delta S_i| \tag{6.4}$$

The most expensive procedure in computing the above gradient is to compute the distance at each point. By using the incremental distance calculation algorithm described in Chapter 3, we can compute the distance between the robot and the closest obstacle in constant time. Since we have to do this computation for a sequence of points, the computation time for each iteration to smooth the curve traced out by our planner is linear in the total number of points in a given sequence. After several iterations of computing the gradient of the summation in Eqn.6.4, the solution path will eventually be smooth and *locally* optimal.

6.5 Proof of Completeness for an Opportunistic Global Path Planner

Careful completeness proofs for the roadmap algorithm are given in [18] and [19]. These proofs apply with very slight modification to the roadmap algorithm that we describe in this paper. The roadmap of [18] is the set of extremal points in a certain direction in free space. Therefore it hugs the boundary of free space. The roadmap described in this paper follows extrema of the *distance function*, and therefore stays well clear of obstacles (except at critical points). But in fact the two are very similar if we think of the *graph of the distance function* in $\mathbb{R}^n$. This is a surface S in $\mathbb{R}^{(n+1)}$ and if we follow the extrema of distance on this surface, the roadmap of this paper is exactly a roadmap in the sense of [18] and [19].

The silhouette curves of [18] correspond to the freeway curves of this paper, and the linking curves correspond to bridges. Recall the basic property required of roadmaps:

Definition

A subset of R of a set S satisfies the *roadmap condition* if every connected component of S contains a single connected component of R.

For this definition to be useful, there is an additional requirement that any point in S can "easily" reach a point on the roadmap.

There is one minor optimization that we take advantage of in this paper. That is to trace only *maxima* of the distance function, rather than both maxima and minima. This can also be applied to the original roadmap.

For those readers not familiar with the earlier papers, we give here an informal sketch of the completeness proof. We need some notation first.

Let S denote the surface in $\mathbb{R}^{(n+1)}$ which is the graph of the distance function. S is an n-dimensional set and is semi-algebraic if configuration space is suitably parametrized. This simply means that it can be defined as a boolean combination of inequalities which are polynomials in the configuration space parameters.

One of the coordinates in configuration space $\mathbb{R}^n$ becomes the *sweep direction*. Let this direction be x_1. Almost any direction in CS will work, and heuristics can be used to pick a direction which should be good for a particular application. When we take slices of the distance surface S, they are taken normal to the x_1 coordinate, so $S|_a$ means $S \cap (x_1 = a)$. Also, for a point p in $\mathbb{R}^{(n+1)}$, $x_1(p)$ is the x_1-coordinate of p.

The other coordinate we are interested in is the distance itself, which we think of as the height of the distance surface. So for a point p in $\mathbb{R}^{(n+1)}$, $h(p)$ is the value of the distance at this configuration.

For this paper, we use a slightly different definition of silhouettes, taking only local maxima into account. We will assume henceforth that the configuration space is bounded in every coordinate. This is certainly always the case for any practical robot working in a confined environment, such as industrial robot manipulators, mobile robots, etc. If it is not bounded, there are technical ways to reduce to a bounded problem, see for example [14]. The set of free configurations is also assumed to be closed. The closed and bounded assumptions ensure that the distance function will attain locally maximal values on every connected component of free space.

A *silhouette point* is a locally maximal point of the function $h(.)$ on some slice $S|_a$ of S. The *silhouette* $\Sigma(S)$ of S is the set of all such points for all a.

The key properties of the silhouette are ([18], [19]):

(i) Within each slice of S, each connected component of $S|_a$ must contain at least one silhouette point.

(ii) The silhouette should be one-dimensional.

(iii) The critical points of the silhouette w.r.t the function $x_1(.)$ should include the critical points of the set S.

Clearly, using local maxima will satisfy property (i). This is true simply because a continuous function (in this case, a distance function with the value zero on the boundary and positive values in the interior) has a local maximum in a compact set. For property (ii) we require that the directions x_1 and h be "generic" (see the earlier papers). This is easily done by picking a general x_1, but h may not be generic a priori. However, rather than the true distance h, we assume that the distance plus a very small linear combination of the other coordinates is used. This linear combination can be arbitrarily small, and we assume that it is small enough that it does not significantly affect the clearance of silhouette points.

For property (iii), we depart somewhat from the original definition. The critical points of the silhouette curves that we have traced can be discovered during the tracing process (they are the points where the curve tangent becomes orthogonal to x_1). But we need to find all (or a sufficient set of) critical points to ensure completeness. All critical points do indeed lie on silhouette curves, but since our algorithm is incremental, we may not discover these other curves unless we encounter points on them. So we need a systematic way to enumerate the critical points of S, since these will serve as starting points for tracing the silhouette curves that we need for completeness.

In fact, not all critical points of S are required. There is a subset of them called *interesting critical points* that are sufficient for our purpose. Intuitively, the interesting critical points are the split or join points in higher dimensions. They can be defined as follows:

Definition

A point p in $\mathbb{R}^{k+1}$ is an *interesting critical point* if for every neighborhood U of p, one of the following holds:

(i) The intersection $U \cap x_1^{-1}(x_1(p) + \epsilon)$ consists of several connected components for all sufficiently small ϵ. This is a generalized split point.

(ii) The intersection $U \cap x_1^{-1}(x_1(p) - \epsilon)$ consists of several components for all sufficiently small ϵ. This is a generalized join point.

From the definition above, it follows that as we sweep the plane $x_1 = a$ through S, the number of connected components of $S|_a$ changes only when the plane passes though interesting critical points.

Definition

Now we can define the roadmap of the surface S. The roadmap $R(S)$ is defined as follows: Let $P_C(S)$ be the set of interesting critical points of $x_1(.)$ on S, $P_C(\Sigma)$ be the set of critical points of $x_1(.)$ on the silhouette, and P_C the union of these two. The roadmap is then:

$$R(S) = \Sigma(S) \cup (\bigcup_{p \in P_c} L(p)) \tag{6.5}$$

That is, the roadmap of S is the union of the silhouette $\Sigma(S)$ and various linking curves $L(p)$. The linking curves join critical points of S or Σ to other silhouette points.

The new roadmap algorithm has an advantage over the original in that it is not restricted to algebraic curve segments. This is because the original was formulated to give precise algorithmic bounds on the planning problem, whereas the new algorithm approximates the silhouette by tracing. Tracing is just as easy for many types of non-algebraic curves as for algebraic ones.

This allows us to do linking in a single step, whereas algebraic linking curves must be defined recursively. We generate linking curves in the present context by simply fixing the x_1 coordinate and hill-climbing to a local maximum in $h(.)$. The curve traced out by the hill-climbing procedure starts at the critical point and ends at a local maximum (which will be a silhouette point) of the distance within the same x_1 slice. Thus it forms a linking curve to the silhouette. If we are at an interesting critical point, there will be two opposing directions (both normal to x_1) along which the distance function increases. Tracing in both directions links the critical point to silhouette points on both channels that meet at that critical point.

Theorem $R(S)$ satisfies the roadmap condition.

Proof Let $a_1, \ldots, a_m$ be the x_1-coordinates of the critical points P_C, and assume the a_i's are arranged in ascending order. The proof is by induction on the a_i's.

Our inductive hypothesis is that the roadmap condition holds to the "left" of a_{i-1}. That is, we assume that $R(S)|_{\leq a_{i-1}} = R(S) \cap x_1^{-l}(x_1 \leq a_{i-1})$ satisfies the roadmap condition as a subset of $S|_{\leq a_{i-1}}$.

The base case is $x_1 = a_1$. If we have chosen a general direction x_1, the set $S|_{a_1}$ consists of a single point which will also be part of the roadmap.

For the inductive step we start with some basic results from Chapter 2 in [87], which state that we can smoothly deform or retract a manifold (or union of manifolds like the surface S) in the absence of critical points. In this case, it implies that the set $S|_{<a_i}$ can be smoothly retracted onto $S|_{\leq a_{i-1}}$, because the interval (a_{i-1}, a_i) is free of critical values. There is also a retraction which retracts $R(S)|_{<a_i}$ onto $R(S)|_{\leq a_{i-1}}$. These retractions imply that there are no topological changes in $R(S)$ or S in the interval (a_{i-1}, a_i), and if $R(S)|_{\leq a_{i-1}}$ satisfies the roadmap condition, then so does $R(S)|_{<a_i}$.

So all that remains is the transition from $R(S)|_{<a_i}$ to $R(S)|_{\leq a_i}$. Let p_i be the critical point whose x_1 coordinate is a_i. The roadmap condition holds for $R(S)|_{<a_i}$, i.e. each component of $S|_{<a_i}$ contains a single component of $R(S)|_{<a_i}$. The only way for the condition to fail as x_1 increases to a_i is if the number of silhouette curve components increases, i.e. when p_i is a critical point of the silhouette, or if the number of connected components of S decreases, which happens when p_i is a join point. Let us consider these cases in turn:

If p_i is a critical point of the silhouette, the tangent to the silhouette at p_i is normal to x_1. By assumption, a new component of the silhouette appeared at p_i as x_1 increased to a_i. This means that in the slice $x_1 = a_i - \epsilon$ (for ϵ small enough) there is no local maximum in the neighborhood of p_i. On the other hand, there must be a local maximum of distance in this slice, which we can find by hill-climbing. So to link such a critical point, we move by ϵ in the $-x_1$ direction (or its projection on S so that we remain on the surface) to a nearby point q_i. Then we hill climb from q_i in the slice $x_1 = a_i - \epsilon$ until we reach a local maximum, which will be a silhouette point. This pair of curves links p_i to the existing roadmap, and so our inductive hypothesis is proved for $R(S)|_{\leq a_i}$.

At join points, though, the linking curve will join p_i to a silhouette point in each of the two channels which meet at p_i. If these two channels are in fact separate connected components of $S|_{<a_i}$, the linking curve will join their respective roadmaps. Those roadmaps are by hypothesis connected within each connected component of $S|_{<a_i}$. Thus the union of these roadmaps and the linking curve is a single connected curve within the connected component of $S|_{\leq a_i}$ which contains p_i. Thus we have proved the inductive hypothesis for a_i if p_i is a join point. $\square$

We have proved that $R(S)$ satisfies the roadmap condition. And it is easy to link arbitrary points in free-space with the roadmap. To do this we simply fix x_1 and hill-climb from the given point using the distance function. Thus our algorithm is complete for finding collision-free paths.

Note that we do not need to construct configuration space explicitly to compute the roadmap. Instead it suffices to be able to compute the interesting critical points, and to be able to compute the distance function and its gradient. This should not surprise the reader familiar with differential topology. Morse theory has already shown us that the topology of manifolds can be completely characterized by looking locally at critical points.

6.6 Complexity Bound

Since our planner probably does not need to explore all the critical points, this bound can be reduced by finding only those *interesting* critical points where adding a bridge helps to reach the goal. If n is the number of obstacle features (faces, edges, vertices) in the environment and the configuration space is $\mathbb{R}^k$, then the number of "interesting critical points" is at most $O((2d)^k n^{(k-1)})$. As mentioned earlier, the algorithm is no longer recursive in calculating the critical points and linking curves (bridges) as in [18], the complexity bound calculated in [18] does not apply here.

6.7 Geometric Relations between Critical Points and Contact Constraints

Let n be the number of obstacle features in the environment and the robot has constant complexity. Free space FP is bordered by $O(n)$ constraint surfaces. Each constraint surface corresponds to an elementary contact, either face-vertex or edge-edge, between a feature of the robot and a feature of the environment. Other types of contacts are called non-elementary, and can be viewed as multiple elementary contacts at the same point, e.g. vertex-edge. They correspond to intersections of constraint surfaces in configuration space.

Definition

An *elementary contact* is a local contact defined by a single equation. It corresponds to a constraint surface in configuration space. For example, face-vertex or edge-edge.

Definition

A *non-elementary contact* is a local contact defined by two or more equations. It corresponds to an intersection or conjunction of two or more constraint surfaces in configuration space. For example, vertex-edge or vertex-vertex. There are $O(n)$ of non-elementary contacts if the robot has constant complexity.

We can represent CO in disjunctive form:

$$CO = \Big(\bigvee_{\substack{e_i \in edges(obstacles) \\ f_j \in faces(robot)}} O_{e_i,f_j} \Big) \vee \Big(\bigvee_{\substack{e_j \in edges(robot) \\ f_i \in faces(obstacles)}} O_{e_j,f_i} \Big)$$

where O_{e_i,f_j} is an overlap predicate for possible contact of an edge and a face. See [18] for the definition of O_{e_i,f_j}. For a fixed robot complexity, the number of branches for the disjunctive tree grows linearly w.r.t. the environment complexity. Each O_{e_i,f_j} has constant size, if the polyhedron is preprocessed (described in Chapter 3). Each clause, O_{e_i,f_j}, is a conjunction of inequalities. This disjunctive tree structure is useful for computing the maximum number of critical points by combinatorics. The interesting critical points (which correspond to the non-elementary contacts) occur when two or more constraint surfaces lie under one clause.

Using the disjunctive tree structure, we can calculate the upper bound for the maximum number of critical points by combinatorial means (by counting the number of systems of equations we must solve to find all the critical points). Generically, at most k surfaces intersect in k dimensions. For a robot with k degrees of freedom and an environment of complexity n, (i.e. n is the number of feature constraints between robot and obstacles) the number of critical points is

$$(2d)^k \begin{pmatrix} n+k \\ k \end{pmatrix} \quad = \quad O((2d)^k n^k)$$

where d is the maximum degree of constraint polynomial equations. This is an upper bound on the number of critical points from [61] and [84]. Therefore, for a given robot (with fixed complexity), there are *at most* $O((2d)^k n^k)$ critical points in terms of n, where k is the dimension of configuration space and n is the number of obstacle features. (NOTE: We only use the above argument to prove the upper bound, *not* to calculate critical points in this fashion.)

These $O((2d)^k n^k)$ intersection points fall into two categories: (a) All the contacts are elementary; (b) one or more contacts are non-elementary. When all contacts are elementary, i.e. all contact points are distinct on the object, free space in a neighborhood of the intersection point is homeomorphic to the intersection of k half-spaces (one side of a constraint surface), and forms a cone. This type of intersection point cannot be a split or join point, and does not require a bridge. However if one or more contacts are non-elementary, then the intersection point is a potential split or join point. But because the $O(n)$ non-elementary contact surfaces have codimension ≥ 2, there are only $O(n^{(k-1)})$ systems of equations that define critical points of type (b), and therefore at most $O((2d)^k n^{k-1})$ possible points. Interesting critical points may be either intersection points, and we have seen that there are $O((2d)^k n^{(k-1)})$ candidates; or they may lie on higher dimensional intersection surfaces, but these are certainly defined by fewer than k equations, and the number of possible critical points is not more than $O((2d)^k n^{(k-1)})$ [84], [61]. Therefore, the number of interesting critical points is at most $O((2d)^k n^{(k-1)})$.

6.8 Brief Discussion

By following the maxima of a well-designed potential field, and taking slice projections through critical points and at random values, our approach builds incrementally an obstacle-avoiding path to guide a robot toward the desired goal, by using the distance computation algorithm described in Chapter 3. The techniques proposed in this chapter provide the planner with a systematic way to escape from these local maxima that have been a long standing problem with using the potential field approach in robot motion planning.

This path planning algorithm, computed from local information about the geometry of configuration space by using the incremental distance computation techniques, requires no expensive precomputation steps as in most global methods developed thus far. In a two dimensional space, this method is comparable with using a Voronoi Diagram for path planning. In three-dimensional space, however, our method is more efficient than computing hyperbolic surfaces for the Voronoi diagram method. In the worst case, it should run at least as fast as the original roadmap algorithm. But, it should run faster in almost all practical cases.

Chapter 7

Conclusions

7.1 Summary

A new algorithm for computing the Euclidean distance between two polyhedra has been presented in Chapter 3. It utilizes the convexity of polyhedra to establish three important applicability criteria for tracking the closest feature pair. With subdivision techniques to reduce the size of coboundaries/boundaries when appropriate, its expected runtime is *constant time* in updating the closest feature pair. If the previous closest features have not been provided, it is typically sublinear in the total number of vertices, but linear for cylindrical objects. Besides its efficiency and simplicity, it is also complete — it is guaranteed to find the closest feature pair if the objects are separated; it gives an error message to indicate collision (when the distance $\leq \epsilon$, the user defined safety tolerance) and returns the contacting feature pair if they are just touching or intersecting.

The methodology described can be employed in distance calculations, collision detection, motion planning, and other robotics problems. In Chapter 4, this algorithm is extended for dynamic collision detection between nonconvex objects by using subpart hierarchical tree representation. In Chapter 4, we also presented an extension of the expected constant time distance computation algorithm to contact determination between convex objects with curved surfaces and boundary. Since the distance computation algorithm described in Chapter 3 runs in expected constant time once initialized, the algorithm is extremely useful in reducing the error by increasing the resolution of polytope approximations for convex objects with smooth curved surfaces. Refining the approximation to reduce error will no longer have a detrimental side effect in running time.

In Chapter 5, we proposed techniques to reduce $\begin{pmatrix} N \\ 2 \end{pmatrix}$ pairwise intersection tests for a large environment of n moving objects. One of them is to use the priority queue (implemented as a heap) sorted by the lower bound on time to collsion; the other is to use the sweeping and sorting techniques and geometric data structures along with hierarchical bounding boxes to eliminate the objects pairs which are definitely not in the vicinity of each other. These algorithms work well in practice and help to achieve almost real time performance for most environments.

One of our applications is path planning in the presence of obstacles described in Chapter 6. That algorithm traces out the skeleton curves which are loci of maxima of a distance function. This is done by first finding the pairs of closest features between the robot and the obstacles, and then keeping track of these closest pairs incrementally by calls to our incremental distance computation algorithm. The curves traced out by this algorithm are in fact maximally clear of the obstacles. It is very computationally economical and efficient to use our expected constant time distance calculation for the purpose of computing the gradient of the distance function.

7.2 Future Work

The chapters in this thesis address work which has been studied in robotics, computational geometry and computer graphics for several years. The treatment of these topics was far from exhaustive. There is no shortage of open problems awaiting for new ideas and innovative solutions.

7.2.1 Overlap Detection for Convex Polyhedra

The core of the collision detection algorithm is built upon the concepts of Voronoi regions for convex polyhedra. As mentioned earlier in Chapter 3, the Voronoi regions form a partition of space outside the polyhedron. But the algorithm can possibly run into a cyclic loop when interpenetration occurs, if no special care is taken to prohibit such events. Hence, if the polyhedra can overlap, it is important that we add a module which detects interpenetration when it occurs as well.

• **Pseudo Voronoi Regions:**

This small module can be constructed based upon similar ideas of space partitioning to the *interior* space of the convex polyhedra. The partitioning does not have to form the exact Voronoi regions since we are not interested in knowing the closest features between two interpenetrating polyhedra but only to detect such a case. A close approximation with simple calculation can provide the necessary tools for detecting overlapping.

This is done by barycentric partition of the interior of a polyhedron. We first calculate the centroid of each convex polyhedron, which is the weighted average, and then construct the constraint planes of each face to the centroid of the polyhedron. These interior constraint planes of a face F are the hyperplanes passing through the centroid and each edge in F's boundary and the hyperplane containing the face F. If all the faces are equi-distant from the centroid, these hyperplanes form the exact Voronoi diagrams for the interior of the polyhedron. Otherwise, they will provide a reasonable space partition for the purpose of detecting interpenetration.

The data structure of these interior *pseudo* Voronoi regions is very much like the exterior Voronoi regions described in Chapter 3. Each region associated with each face has $e + 1$ hyperplanes defining it, where e is the number of edges in the face's boundary. Each hyperplane has a pointer directing to a neighboring region where the algorithm will step to next, if the constraint imposed by this hyperplane is violated. In addition, a *type* field is added in the data structure of a Voronoi cell to indicate whether it is an interior or exterior Voronoi region.

The exact Voronoi regions for the interior space of the polyhedron can also be constructed by computing all the equi-distant bisectors of all facets. But, such an elaborate precomputation is not necessary unless we are also interested in computing the degree of interpenetration for constructing some type of penalty functions in collision response or motion planning.

As two convex polyhedral objects move and pass through each other, the change of feature pairs also indicate such a motion since the pointers associated with each constraint plane keep track of the *pseudo* closest features between two objects during penetration phase as well.

• **Other Applications**

In addition, by appending this small module to the expected constant time algorithm described in Chapter 3, we can speed up the run time whenever one candidate feature lies beneath another candidate feature (this scenario corresponds to local minima of distance function). We no longer need the linear search to enumerate all features on one polyhedron to find the closest feature on one polyhedron to the other. By traveling through the interior space of two convex objects, we can step out of the local minima at the faster pace. This can be simply done by replacing the linear time search by the overlap detection module described here, whenever a feature lies beneath another face.

However, to ensure convergence, we will need to construct the exact Voronoi regions of the interior space or use special cases analysis to guarantee that each switch of feature necessarily decreases the distance between two objects.

7.2.2 Intersection Test for Concave Objects

• **New Approaches for Collision Detection between Non-convex Objects**

Currently, we are also investigating other possibilities of solving the intersection problem for non-convex objects. Though sub-part hierarchical tree representation is an intuitive and natural approach, we have run into the problem of extending this approach for any general input of models. This is due to the fact that there is no optimal convex decomposition algorithm, but near-optimal algorithm [23]. A simple convex decomposition does not necessarily give us the decomposition characterizing the geometry structure of the object. Thus, it is hard to exploit the strength of this methodology.

Another solution is to use the concept of "destructive solid geometry", that is to represent each non-convex object by boolean operations of

$$C = A - \sum_i B_i$$

where A and B_i's are convex and C is the non-convex object. By examining the overlap between A_1 of the object C_1 and B_{2_i} of the object C_2 (where B_{2_i} represents one of the notches or concavities of C_2), we have the relevant information for the actual contact location within the concavities. Again, we only have to keep track of the local closest feature pairs within the concavities of our interests, once the convex hulls of two non-convex objects intersect. We plan to use the fact that certain object types, especially cylinders, have a bounded number of closest feature pairs.

In addition, we can possibly combine the priority queue and sweeping and sorting techniques described in Chapter 5 with the basic algorithm for nonconvex objects to further reduce the number of intersection tests needed inside of the concavities.

- **Automated Generation of the Sub-Part Hierarchical Representation**

To support the simulation, geometric modeling to represent three-dimensional objects is necessary for collision detection and dynamics computation. Furthermore, to automate the process of generating the hierarchical representation will be extremely useful in handling complex, non-convex objects in a computer generated simulation.

Our collision detection algorithm for non-convex objects uses the sub-part hierarchical tree representation to model each non-convex object. If the objects are generated by hand, this representation is easy to use. However, if we are given a set of faces describing the non-convex object without any special format, we can use a convex decomposition module which does not necessarily give us an optimal convex decomposition of the non-convex objects, since the problem itself is NP-complete [52, 67, 22, 76]. Therefore, applying our sub-part hierarchical tree representation directly to the output of a non-optimal convex decomposition routine may not yield satisfactory results for the purpose of collision detection, especially if we have numerous number of convex pieces. One possibility is to establish a criterion for grouping nodes, e.g. $maximize \frac{volume(A \cup B)}{volume(conv(A \cup B))}$, where $conv(P)$ represents the convex hull of P.

So far, in our simulation, we have to build the sub-part hierarchical tree by both visual inspection of the data and extracting the information by hand, in order to take advantage of the geometry of the original object. This is a rather painful and time consuming process.

One proposed solution would be to generate all models by a Computer Aided Geometric Design tool which has only union as a primitive and all primitive volumes are convex. However, to make a general system for collision detection, we still have to face the problem when we are given only the face information without any special format to extract the geometry information to construct the sub-part hierarchical tree. This is an interesting and challenging geometry and modeling problem.

7.2.3 Collision Detection for Deformable objects

One of most challenging problems is collision detection for objects of deformable materials or with time varying surfaces. In terms of complexity, deformable objects offer us the greatest challenge for the problem of contact determination. Our algorithms presented in this thesis work extremely well with rigid bodies. Whereas using deformable models, it is very difficult to predict where concavity and degeneracy may be introduced due to external forces, and the shape of objects can be changed in all possible ways. Therefore, the problem of collision detection becomes rather complex and almost impossible to solve interactively.

Some work has been done for time-dependent parametric surfaces [45]. Herzen and etc. use a hierarchical algorithm which first finds the potential collision over large volumes and then refines the solution to smaller volumes. The bound on these volumes are derived from the derivatives with respect to time and the parameters of the surfaces. Though it is reasonably robust and applicable for deforming time-dependent surfaces, it cannot guarantee to detect collisions for surfaces with unbounded derivatives. In many interactive environments, the derivatives of the surface with respect to time are not obtainable and this approach cannot work under such a circumstance.

Another commonly used method is to model the deformable objects by finite element methods and nodal analysis [70]. However, this approach is computationally too expensive. Unless we use specialized parallel machines to simulate the motions, its speed is not satisfactory.

Baraff and Witkin use polyhedral approximation to handle the deformable objects [5] for the purpose of collision detection. If the model is very refined, this approach may yield reasonable results; however, even with the use of coherence based on a culling step to reduce the number of the polygon-polygon intersection tests, the resulting algorithm still takes longer than linear time. For low resolution of the polyhedral model, the visual effect generated by this algorithm would be very disconcerting (the viewer may see intersecting facets frequently) and the numerical solution will be rather poor for the purpose of simulating dynamics and robust integration.

Snyder and etc. [81] present a general collision detection algorithm for any type of surface by using interval arithmetics, optimization routines, and many other numerical methods. Although the algorithm can be used for a large class of various models, it is extremely slow. As Snyder admitted, it cannot be used for real time simulation. However, these days it is not acceptable to spend hundreds of hours generating a few seconds of simulation. One of the important factors as we mention in the introduction is speed, i.e. how quickly the algorithm can generate results.

Our algorithm is especially tailored for rigid bodies, since the Voronoi regions are constructed based on the properties of convex polyhedra. For linear deformation where the objects remain convex, we can transform the precomputed Voronoi regions for the purpose of collision detection between flexible objects. But, in most situations, the deformable object may become locally non-convex and the algorithm described in Chapter 3 cannot be easily modified to efficiently handle such a contact during the deformation phase. However, it is possible that we use the algorithms described in this thesis as a top level module until the impacts occurs or external forces are applied to the deformable objects, for the purpose of eliminating collision checks. But, a fast and exact method is still needed. This leads to a new area of exciting research.

7.2.4 Collision Response

In a physically-based dynamic simulation, there are two major components to a successful and realistic system display: collision detection and collision response. Given that we have understood the problem of collision detection reasonably well for the rigid objects, another related problem is to resolve the difficulty of simulating the physics of real world objects: Contact forces must be calculated and applied until separation finally occurs; in addition, objects' velocities must be adjusted during the contact course in response to the impacts. All these processes are considered as a part of dynamics response to the collision following the basic laws of physics.

An automated dynamics generator allows the user to interact with a changing environment. Such an environment can arise because of active motion (locomotion) or passive motion (riding a vehicle). Furthermore, modeling physics of dynamics tightly couples interaction with force feedback between human participants in the virtual world and guided agent (i.e. graphic images slaved to other human participants); it also tightly couples among human participants and the force feedback devices. It contributes to realistic portrayal of autonomous movements of all virtual objects in the synthetic environments.

Most dynamic simulators [4, 7, 25, 88, 89] make simplification of models in simulating the physics of translating and rotating objects, and mostly on frictionless impacts. Recently, Keller applied Routh's frictional impact equations [75] to a few simplified cases with numerous assumptions [49]. Wang and Mason also characterize frictional impacts for the two-dimensional impacts [86]. Currently Mirtich and Canny are investigating a better approach to model the three-dimensional frictional impact, which will be extremely useful for a general-purpose dynamics simulator in computer generated virtual environment or robotics simulation for manufacturing purposes.

Many open problems are still left to be addressed. The goal of research in this area would be to simulate the dynamics (the geometric component, the physics module, the numerical integrator, and motion control) in real time.

Bibliography

[1] L. E. Andersson and N. F. Steward. Maximal distance for robotic simulation: the convex case. *Journal of Optimization Theory and Applications*, 57(No. 2):215–222, 1988.

[2] C. Bajaj and T. Dey. Convex decomposition of polyhedra and robustness. *SIAM J. of Comput.*, (No. 2):339–364, 1992.

[3] D. Baraff. Curved surfaces and coherence for non-penetrating rigid body simulation. *ACM Computer Graphics*, 24(4):19–28, 1990.

[4] D. Baraff. Issues on computing contact forces for non-penetrating rigid bodies. *Algorithmica*, Vol. 10(2-4):292–352, Aug/Sept/Oct 1993.

[5] David Baraff and Andrew Witkin. Dynamic simulation of non-penetrating flexible bodies. *Computer Graphics*, 26(2):303–308, July 1992.

[6] J. Barraquand, B. Langlois, and J-C. Latombe. Robot motion planning with many degrees of freedom and dynamic constraints. In *Proceedings 5th Int. Symp Robotics Research*, Tokyo, Japan, 1989.

[7] R. Barzel and A. Barr. A modeling system based on dynamic constraints. *ACM Computer Graphics*, 22(4):31–39, 1988.

[8] D. N. Bernstein. The number of roots of a system of equations. *Funktsional'nyi Analiz i Ego Prilozheniya*, 9(3):1–4, Jul-Sep 1975.

[9] J. E. Bobrow. A direct minimization approach for obtaining distance between convex polyhedra. *International Journal of Robotics Research*, 8(No. 3):65–67, 1989.

[10] W. Bouma and Jr. G. Vanecek. Collision detection and analysis in a physically based simulation. In *Proceeding of the Seconde Eurographics Workshop on Animation and Simulation*, 1992. Vienna Austria.

[11] J. W. Boyse. Interference detection among solids and surfaces. *Comm. ACM*, 22(1):3–9, 1979.

[12] S. A. Cameron. A study of the clash detection problem in robotics. *Proc. IEEE ICRA*, pages pp. 488–493, 1985.

[13] S. A. Cameron and R. K. Culley. Determining the minimum translational distance between two convex polyhedra. *Proc. IEEE ICRA*, pages pp. 591–596, 1986.

[14] J. Canny. Computing roadmaps of general semi-algebraic sets. In *AAECC-91*, pages 94–107, 1991.

[15] J. Canny and B. Donald. Simplified voronoi diagrams. *Discret Comput. Geometry*, pages 219–236, 1988.

[16] J. Canny and I. Emiris. An efficient algorithm for the sparse mixed resultant. In G. Cohen, T. Mora, and O. Moreno, editors, *Proc. 10th Intern. Symp. on Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, pages 89–104. Springer Verlag, May 1993. Lect. Notes in Comp. Science 263.

[17] J. F. Canny. Collision detection for moving polyhedra. *IEEE Trans. PAMI*, 8:pp. 200–209, 1986.

[18] J. F. Canny. *The Complexity of Robot Motion Planning.* MIT Press, Cambridge, MA, 1988.

[19] J. F. Canny. Constructing roadmaps of semi-algebraic sets I: Completeness. *Artificial Intelligence*, 37:203–222, 1988.

[20] J. F. Canny and M. C. Lin. An opportunistic global path planner. *Algorithmica, Special Issue on Computational Robotics*, Vol. 10(2-4):102–120, Aug/Sept/Oct 1993.

[21] J.F. Canny. Generalized characteristic polynomials. *Journal of Symbolic Computation*, 9(3):241–250, 1990.

[22] B. Chazelle. Convex partitions of polyhedra: A lower bound and worst-case optimal algorithm. *SIAM J. Comput.*, 13:488–507, 1984.

[23] B. Chazelle and L. Palios. Triangulating a non-convex polytope. *Discrete & Comput. Geom.*, (5):505–526, 1990.

[24] J. Cohen, M. Lin, D. Manocha, and M. Ponamgi. Exact collision detection for interactive, large-scaled environments. Tech Report #TR94-005, 1994. University of North Carolina, Chapel Hill.

[25] James F. Cremer and A. James Stewart. The architecture of newton, a general-purpose dynamics simulator. In *IEEE Int. Conf. on Robotics and Automation*, pages 1806–1811, May 1989.

[26] D. P. Dobkin and D. G. Kirkpatrick. A linear algorithm for determining the separation of convex polyhedra. *J. Algorithms*, 6(3):pp. 381–392, 1985.

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

[28] B. R. Donald. Motion planning with six degrees of freedom. Master's thesis, MIT Artificial Intelligence Lab., 1984. AI-TR-791.

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

[30] M. E. Dyer. Linear algorithms for two and three-variable linear programs. *SIAM J. on Computing*, 13:pp. 31–45, 1984.

[31] H. Edelsbrunner. A new approach to rectangle intersections, part i&ii. *Intern. J. Computer Math.*, 13:pp. 209–229, 1983.

[32] H. Edelsbrunner. *Algorithms in Combinatorial Geometry.* Springer-Verlag, Berlin, Heidelberg, New York, London, Paris, Tokyo, 1988.

[33] I. Emiris and J. Canny. A general approach to removing degeneracies. In *IEEE FOCS*, pages 405–413, 1991.

[34] S. F. Fahlman. A planning system for robot construction tasks. *Artifical Intellignece*, 5:pp. 1–49, 1974.

[35] G. Farin. *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide.* Academic Press Inc., 1990.

[36] D. Filip, R. Magedson, and R. Markot. Surface algorithms using bounds on derivatives. *Computer Aided Geometric Design*, 3:295–311, 1986.

[37] Jr. G. Vanecek. Brep-index: A multi-dimensional space partitioning tree. *ACM/SIGGRAPH Symposium on Solid Modeling Foundations and CAD Applications*, pages 35–44, 1991. Austin Texas.

[38] E. G. Gilbert and C. P. Foo. Computing the distance between general convex objects in three dimensional space. *IEEE Trans. Robotics Automat.*, 6(1), 1990.

[39] E. G. Gilbert and S. M. Hong. A new algorithm for detecting the collision of moving objects. *Proc. IEEE ICRA*, pages pp. 8–14, 1989.

[40] E. G. Gilbert and D. W. Johnson. Distance functions and their application to robot path planning in the presence of obstacles. *IEEE J. Robotics Automat.*, RA-1:pp. 21–30, 1985.

[41] E. G. Gilbert, D. W. Johnson, and S. S. Keerthi. A fast procedure for computing the distance between objects in three-dimensional space. *IEEE J. Robotics and Automation*, vol RA-4:pp. 193–203, 1988.

[42] G.H. Golub and C.F. Van Loan. *Matrix Computations.* John Hopkins Press, Baltimore, 1989.

[43] J. K. Hahn. Realistic animation of rigid bodies. *ACM Computer Graphics*, 22(4):pp. 299–308, 1988.

[44] Mark Hall and Joe Warren. Adaptive polygonalization of implicitly defined surfaces. *IEEE Computer Graphics and Applications*, 10(6):33–42, November 1990.

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

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

[47] J.E. Hopcroft, J.T. Schwartz, and M. Sharir. Efficient detection of intersections among spheres. *The International Journal of Robotics Research*, 2(4):77–80, 1983.

[48] Kass, Witkin, Baraff, and Barr. An introduction to physically based modeling. Course Notes 60, 1993.

[49] J. B. Keller. Impact with friction. *Journal of Applied Mathematics*, Vol. 53, March 1986.

[50] O. Khatib. Real-time obstable avoidance for manipulators and mobile robots. *IJRR*, 5(1):90–98, 1986.

[51] J.C. Latombe. *Robot Motion Planning.* Kluwer Academic Publishers, 1991.

[52] A. Lingas. The power of non-rectilinear holes. In *Proc. 9th Internat. Colloq. Automata Lang. Program.*, volume 140 of *Lecture Notes in Computer Science*, pages 369–383. Springer-Verlag, 1982.

[53] T. Lozano-Pérez and M. Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *Comm. ACM*, 22(10):pp. 560–570, 1979.

[54] T. Lozano-Pérez and M. Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *Comm. ACM*, 22(10):560–570, 1979.

[55] D. Manocha. *Algebraic and Numeric Techniques for Modeling and Robotics*. PhD thesis, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, May 1992.

[56] D. Manocha. Solving polynomial systems for curve, surface and solid modeling. In *ACM/SIGGRAPH Symposium on Solid Modeling*, pages 169–178, 1993. Revised version to appear in IEEE Computer Graphics and Applications.

[57] D. Manocha and J. F. Canny. Efficient teniques for multipolynomial resultant algorithms. *Proceedings of ISSAC'91*, 1991. Bonn, Germany.

[58] N. Megiddo. Linear-time algorithms for linear programming in $\mathbb{R}^3$ and related problems. *SIAM J. Computing*, 12:pp. 759–776, 1983.

[59] N. Megiddo. Linear programming in linear time when the dimension is fixed. *Jour. ACM*, 31:pp. 114–127, 1984.

[60] N. Megiddo and A. Tamir. Linear time algorithms for some separable quadratic programming problems. *Operations Research letters*, 13(4):203–211, 1993.

[61] J. Milnor. On the betti numbers of real varieties. *Proc. Amer. Math. Soc.*, 15:275–280, 1964.

[62] B. Mirtich and J. Canny. Impusle-based, real time dynamic simulation. Submitted to ACM SIGGRAPH, 1994. University of California, Berkeley.

[63] M. Moore and J. Wilhelms. Collision detection and response for computer animation. *ACM Computer Graphics*, 22(4):pp. 289–298, 1988.

[64] A. P. Morgan. Polynomial continuation and its relationship to the symbolic reduction of polynomial systems. In *Symbolic and Numerical Computation for Artificial Intelligence*, pages 23–45, 1992.

[65] M. Orlowski. The computation of the distance between polyhedra in 3-space. Presented SIAM Conf. on Geometric Modeling and Robotics, 1985. Albany, NY.

[66] J. O'Rourke, C.-B Chien, T. Olson, and D. Naddor. A new linear algorithm for intersecting convex polygons. *Computer Graphics and Image Processing*, 19:384–391, 1982.

[67] J. O'Rourke and K. J. Supowit. Some NP-hard polygon decomposition problems. *IEEE Trans. Inform. Theory*, IT-30:181–190, 1983.

[68] M. H. Overmars. Point location in fat subdivisions. *Inform. Proc. Lett.*, 44:261–265, 1992.

[69] A. Pentland. Computational complexity versus simulated environment. *Computer Graphics*, 22(2):185–192, 1990.

[70] A. Pentland and J. Williams. Good vibrations: Modal dynamics for graphics and animation. *Computer Graphics*, 23(3):185–192, 1990.

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

[72] W. E. Red. Minimum distances for robot task simulation. *Robotics*, 1:pp. 231–238, 1983.

[73] J. Reif. *Complexity of the Mover's Problem and Generalizations*, chapter 11, pages pp. 267–281. Ablex publishing corp., New Jersey, 1987. edited by J.T. Schwartz and M. Sharir and J. Hopcroft.

[74] D. F. Rogers. *Procedural Elements for Computer Graphics.* McGraw-Hill Book Company, 1985.

[75] Edward J. Routh. *Elementary Rigid Dynamics.* 1905.

[76] J. Ruppert and R. Seidel. On the difficulty of tetrahedralizing 3-dimensional non-convex polyhedra. In *Proc. of the Fifth Annual Symposium on Computational Geometry*, pages 380–392. ACM, 1989.

[77] N. Sancheti and S. Keerthi. Computation of certain measures of proximity between convex polytopes: A complexity viewpoint. *Proceedings of IEEE ICRA'92*, 3:2508–2513, May 1992.

[78] J.T. Schwartz and M. Sharir. *On the 'Piano Movers' Problem, II. General Techniques for Computing Topological Properties of Real Algebraic Manifolds*, chapter 5, pages 154–186. Ablex publishing corp., New Jersey, 1987.

[79] R. Seidel. Linear programming and convex hulls made easy. In *Proc. 6th Ann. ACM Conf. on Computational Geometry*, pages 211–215, Berkeley, California, 1990.

[80] H. W. Six and D. Wood. Counting and reporting intersections of d-ranges. *IEEE Trans. on Computers*, C-31(No. 3), March 1982.

[81] J. Snyder, A. Woodbury, K Fleischer, B. Currin, and A. Barr. Interval methods for multi-point collisions between time-dependent curved surfaces. *Computer Graphics, Proceedings of ACM SIGGRAPH'93*, pages 321–334, August 1993.

[82] D. Sturman. A discussion on the development of motion control systems. In *SigGraph Course Notes: Computer Animation: 3-D Motion Specification and Control*, number 10, 1987.

[83] W. Thibault and B. Naylor. Set operations on polyhedra using binary space partitioning trees. *Computer Graphics – SIGGRAPH'87*, (4), 1987.

[84] R. Thom. Sur l'homologie des varietes algebriques reelles. *Differential and Combinatorial Topology*, pages 255–265, 1965.

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

[86] Yu Wang and Matthew T. Mason. Modeling impact dynamics for robotic operations. In *IEEE Int. Conf. on Robotics and Automation*, pages 678–685, May 1987.

[87] C. G. Gibson K. Wirthmuller and A. A. du Plessis E. J. N. Looijenga. *Topological Stability of Smooth Mappings*. Springer-Verlag, Berlin . Heidelberg . New York, 1976.

[88] Andrew Witkin, Michael Gleicher, and William Welch. Interactive dynamics. *Computer Graphics*, 24(2):11–22, March 1990.

[89] Andrew Witkin and William Welch. Fast animation and control of nonrigid structures. *Computer Graphics*, 24(4):243–252, August 1990.

[90] P. Wolfe. Finding the nearest points in a polytope. *Math. Programming*, 11:pp. 128–149, 1976.

[91] K. Zikan and W. D. Curtis. Intersection and separations of polytopes. *Boeing Computer Services Technology*, BCSTECH-93-031, 1993. A note on collision and interference detection.

Appendix A

Calculating the Nearest Points between Two Features

In this appendix, we will described the equations which the implementation of the distance computation algorithm described in Chapter 3 is based upon.

(I) VERTEX-VERTEX: The nearest points are just the vertices.

(II) VERTEX-EDGE: Let the vertex be $V = (V_x, V_y, V_z, 1)$ and the edge E have head $H_E = (H_x, H_y, H_z, 1)$, tail $T_E = (T_x, T_y, T_z, 1)$, and $\vec{e} = H_E - T_E = (E_x, E_y, E_z, 0)$. Then, the nearest point P_E on the edge E to the vertex V can be found by:

$$P_E = T_E + min(1, max(0, \frac{(V - T_E) \cdot \vec{e}}{\mid \vec{e} \mid^2}))\vec{e} \tag{A.1}$$

(III) VERTEX-FACE: Let the vertex be $V = (V_x, V_y, V_z, 1)$; If we use a normalized unit face outward normal vector, that is the normal $n = (a, b, c)$ of the face has the magnitude of 1 and $N_F = (n, -d) = (a, b, c, -d)$ and $-d$ is the *signed* distance of the face F from the origin, and the vertex V defined as above. We define a new vector quantity $\vec{n}_F$ by $\vec{n}_F = (n, 0)$. The nearest point P_F on F to V can be simply expressed as:

$$P_F = V - (V \cdot N_F)\vec{n}_F \tag{A.2}$$

(IV) EDGE-EDGE: Let H_1 and T_1 be the head and tail of the edge E_1 respectively. And H_2 and T_2 be the head and tail of the edge E_2 as well. Vectors $\vec{e}_1$ and $\vec{e}_2$ are defined as $\vec{e}_1 = H_1 - T_1$ and $\vec{e}_2 = H_2 - T_2$. We can find for the nearest point pair P_1 and P_2 on E_1 and E_2 by the following:

$$\begin{aligned} P_1 &= H_1 + s(T_1 - H_1) = H_1 - s\vec{e}_1 \\ P_2 &= H_2 + u(T_2 - H_2) = H_2 - u\vec{e}_2 \end{aligned} \tag{A.3}$$

where s and u are scalar values parameterized between 0 and 1 to indicate the relative location of P_1 and P_2 on the edges E_1 and E_2. Let $\vec{n} = P_1 - P_2$ and $\mid \vec{n} \mid$ is the shortest distance between the two edges E_1 and E_2. Since $\vec{n}$ must be orthogonal to the vectors $\vec{e}_1$ and $\vec{e}_2$, we have:

$$\begin{aligned} \vec{n} \cdot \vec{e}_1 &= (P_1 - P_2) \cdot \vec{e}_1 = 0 \\ \vec{n} \cdot \vec{e}_2 &= (P_1 - P_2) \cdot \vec{e}_2 = 0 \end{aligned} \tag{A.4}$$

By substituting Eqn.(A.3) into these equations (A.4), we can solve for s and u:

$$\begin{aligned} s &= \frac{(\vec{e}_1 \cdot \vec{e}_2)[(H_1 - H_2) \cdot \vec{e}_2] - (\vec{e}_2 \cdot \vec{e}_2)[(H_1 - H_2) \cdot \vec{e}_1]}{det} \\ u &= \frac{(\vec{e}_1 \cdot \vec{e}_1)[(H_1 - H_2) \cdot \vec{e}_2] - (\vec{e}_1 \cdot \vec{e}_2)[(H_1 - H_2) \cdot \vec{e}_1]}{det} \end{aligned} \tag{A.5}$$

where $det = ((\vec{e}_1 \cdot \vec{e}_2) * (\vec{e}_1 \cdot \vec{e}_2)) - (\vec{e}_1 \cdot \vec{e}_1) * (\vec{e}_2 \cdot \vec{e}_2)$. However, to make sure P_1 and P_2 lie on the edges E_1 and E_2, s and u are truncated to the range [0,1] which gives the correct nearest point pair (P_1, P_2)

(V) EDGE-FACE: Degenerate, we don't compute them explicitly. Please see the pseudo code in Appendix B for the detailed treatment.

(VI) FACE-FACE: Degenerate, we don't compute them explicitly. Please see the pseudo code in Appendix B for the detailed treatment.

Appendix B

Pseudo Code of the Distance Algorithm

PART I - Data Structure

```
type VEC {
  REAL X                 %% X-coordinate
  REAL Y                 %% Y-coordinate
  REAL Z                 %% Z-coordinate
  REAL W                 %% scaling factor
}

type VERTEX {
  REAL X, Y, Z, W;
  FEATURE *edges;        %%  pointer to its coboundary - a list of edges
  CELL *cell;            %% vertex's Voronoi region
};

type EDGE {
  VERTEX *H, *T;         %% the head and tail of this edge
  FACE *fright, *fleft;  %% the right and left face of this winged edge
  VEC vector;            %% unit vector representing this edge
  CELL *cell;           %% edge's Voronoi region
};

type FACE {
  FEATURE *verts;       %% list of vertices on the face
  FEATURE *edges;       %% list of edges bounding the face
  VEC norm;             %% face's unit outward normal
  CELL *cell;           %% face's PRISM, NOT including the plane of face
  POLYHEDRON *cobnd;    %% the polyhedron containing the FACE
};

type FEATURE {
  union{                %% features are union of
    VERTEX *v;          %% vertices,
    EDGE *e;            %% edges,
    FACE *f;            %% and faces
  };
  FEATURE *next;        %% pointer to next feature
};

struct CELL {
  VEC cplane;           %% one constraint plane of a Voronoi region:
  %%%% cplane.X * X + cplane.Y * Y + cplane.Z * Z + cplane.W = 0 %%%%
  PTR *neighbr;         %% ptr to next feature if this app test fails
  CELL *next;           %% if there are more planes in this V. region
};

type POLYHEDRON {
  FEATURE *verts;       %% all its vertices
  FEATURE *edges;       %% all its edges
  FEATURE *faces;       %% all its faces
  CELL *cells;          %% all the Voronoi regions assoc. with features
  VEC pos;              %% its current location vector
  VEC rot;              %% its current direction vector
};
```

PART II - Algorithm

```
%%% vector or vertex operation: dot product
PROCEDURE vdot (v1, v2)
  RETURN (v1.X*v2.X + v1.Y*v2.Y + v1.Z*v2.Z + v1.W*v2.W)

%%% vector operation: cross product
PROCEDURE vcross (v1, v2)
  RETURN (v1.Y*v2.Z-v1.Z*v2.Y, v1.Z*v2.X-v1.X*v2.Z, v1.X*v2.Y-v1.Y*v2.X)

%%% vector operation: triple product
PROCEDURE triple(v1, v2, v3)
  RETURN (vdot(vcross(v1, v2, v3))

%%% distance function: it tests for the type of features in order
%%% to calculate the distance between them.  It takes in 2 features
%%% and returns the distance between them.  Since it is rather simple,
%%% we only document its functionality and input here.
PROCEDURE dist(feat1, feat2)

%%% Given 2 features "feat1" and "feat2", this routine finds the
%%% nearest point of one feature to another:
PROCEDURE nearest-pt(feat1, feat2)

%%% Given 2 faces "Fa" and "Fb", it find the closest vertex
%%% or edges in Fb's boundary to the plane containing Fa
PROCEDURE closestToFplane(Fa, Fb)

%%% Given an edge E and a face F, it finds the closest vertex
%%% or edge in the boundary of the face F
PROCEDURE closestToE(E, F)
```

```
%%% Given 2 faces "Fa" and "Fb", it find the pair of edges
%%% closest in distance between 2 given faces
PROCEDURE closest-edges(Fa, Fb)

%%% Given 2 faces, it determines if the projection of Fa down
%%% to Fb overlaps with Fb.  This can be implemented with best
%%% known bound O(N+M) by marching along the boundary of Fa and
%%% Fb to find the intersecting edges thus the overlap polygon,
%%% where N and M is the number of vertices of Fa and Fb.
PROCEDURE overlap(Fa, Fb)

%%% This is the linear time routine used to find the closest
%%% feature on one "polyhedron" to a given feature "feat"
PROCEDURE find-closest(feat, polyhedron)

%%% Point-Cell Applicability Condition:
%%% This routine returns TRUE if P satisfies all applicability
%%% constraints of the Voronoi cell, "Cell"; it returns the
%%% neighboring feature whose constraint is violated the most
%%% if "P" fails at least one constraint of "Cell".
PROCEDURE point-cell-checkp (P, Cell)
  min = 0
  NBR = NULL
  while (NOT(Cell.cplane = NULL)) Do
        test = vdot(P, Cell.cplane)
        if (test < min)
           then
               min = test
               NBR = Cell.neighbor
        Cell = Cell.next
RETURN NBR

%%% Point-Face Applicability Condition
PROCEDURE point-face-checkp (P, F)
  NBR =  point-cell-checkp (P, F.cell)
  if (NBR = NULL)
     then if vdot(P, F.norm) > 0
```

```
             then RETURN(NBR)
             else RETURN(find-closest(P, F.cobnd))
     else RETURN(NBR)

%% This procedure returns TRUE if Ea lies within the
%% prismatic region swept out by Fb along its face normal
%% direction, FALSE otherwise.
PROCEDURE E-FPrism(E, F)
  min = 0
  max = length(Ea)
  for (cell = Fb.cell till cell=NULL; cell = cell.next)
      norm = vdot(Ea.vector,cell.cplane)
      %% Ea points inward of the hyperplane
      if (norm > 0)
         %% compute the relative inclusion factor
         then K =  vdot(Ea.H, cell.cplane) / norm
              if (K < max) {
                 then max = K
                      if (min > max) RETURN(FALSE) }
         %% Ea points outward from the hyperplane
         else if (norm < 0)
                 %% compute the relative inclusion factor
                 then K = vdot(Ea.T, cell.cplane) / norm
                      if (K > min) {
                         min = K
                         if (max < min) RETURN(FALSE)}
                 %% norm = 0 if the edge Ea and Ei are parallel
                 else if vdot(Ea.H, cell.cplane) < 0 RETURN(FALSE)
  RETURN(TRUE)


%%% Vertex-Vertex case:
PROCEDURE vertex-vertex (Va, Vb)
  NBRb = point-cell-checkp (Va, Vb.cell)
  if (NBRb = NULL)
     then NBRa = point-cell-checkp (Vb, Va.cell))
          if (NBRa = NULL)
```

```
            then RETURN (Va, Vb, dist(Va,Vb))
            else close-feat-checkp (NBRa, Vb)
     else close-feat-checkp (Va, NBRb)

%%% Vertex-Edge case:
PROCEDURE vertex-edge (Va, Eb)
  NBRb = point-cell-checkp(Va, Eb.cell)
  if (NBRb = NULL)
     then NBRa = point-cell-checkp(nearest-pt(Va, Eb), Va.cell)
          if (NBRa = NULL)
            then RETURN (Va, Eb, dist(Va,Eb))
            else close-feat-checkp (NBRa, Eb)
     else close-feat-checkp (Va, NBRb)

%%% Vertex-Face case:
PROCEDURE vertex-face (Va, Fb)
  NBRb = point-face-checkp (Va,Fb)
  if (NBRb = NULL)
     then NBRa = point-cell-checkp (nearest-pt(Va, Fb), Va.cell)
          if (NBRa = NULL)
            then RETURN (Va, Fb, dist(Va,Fb))
            else close-feat-checkp (NBRa, Fb)
     else close-feat-checkp (Va, NBRb)

%%% Edge-Edge case:
PROCEDURE edge-edge (Ea, Eb)
  NBRb = point-cell-checkp (nearest-pt(Eb, Ea), Eb.cell)
  if (NBRb = NULL)
     then NBRa = point-cell-checkp (nearest-pt(Ea, Eb), Ea.cell))
          if (NBRa = NULL)
            then RETURN (Ea, Eb, dist(Ea,Eb))
            else close-feat-checkp (NBRa, Eb)
     else close-feat-checkp (Ea, NBRb)

%%% Edge-Face case:
PROCEDURE edge-face (Ea Fb)
  if (vdot(Ea.H, Fb.norm) = vdot(Ea.T, Fb.norm))
```

```
     then if (E-FPrism(Ea, Fb.cell) # NULL)
             then if (vdot(Ea.H, Fb.norm) > 0)
                     then cp_right = triple(E.fright.norm, Ea.vector, Fb.norm)
                          cp_left = triple(E.vector, F.fleft.norm, Fb.norm)
                          if (cp_right >= 0)
                             then if (cp_left >= 0)
                                  then RETURN (Ea, Fb, dist(Ea,Fb))
                                  else close-feat-checkp (Ea.fleft, Fb)
                             else close-feat-checkp (Ea.fright, Fb)
                     else close-feat-checkp (Ea, find-closest(Ea, Fb.cobnd))
             else close-feat-checkp (Ea, closestToE(Ea, Fb))
     else if (sign(vdot(Ea.H, Fb.norm)) # sign(vdot(Ea.T, Fb.norm)))
             then close-feat-checkp (Ea, closestToE(Ea, Fb))
             else if (dist(Ea.H, Fb) < dist(Ea.T, Fb))
             %% dist returns unsigned magnitude
                     then sub-edge-face(Ea.H, Ea, Fb)
                     else sub-edge-face(Ea.T, Ea, Fb)

%%% Sub-Edge-Face case:
PROCEDURE sub-edge-face (Ve, E, F)
  NBRb = point-cell-checkp (Ve, F.cell)
  if (NBRb = NULL)
     then if (vdot(Ve, F.norm) > 0)
             then NBRa = point-cell-checkp (nearest-pt(Ve, F), Ve.cell)
                  if (NBRa = NULL)
                     then RETURN (Ve, F, dist(Ve,F))
                     else close-feat-checkp (NBRa, F)
             else close-feat-checkp (Ve, find-closest(Ve, F.cobnd))
     else close-feat-checkp (E, closestToE(E, F))

%%% Face-Face-case:
PROCEDURE face-face (Fa Fb)
  if (abs(vdot(Fa.norm, Fb.norm)) = 1) %% check if Fa and Fb parallel
     then if (overlap(Fa, Fb))
             then if (vdot(Va, Fb.norm) > 0)
                     then if (vdot(Vb, Fa.norm) > 0)
                             then RETURN (Fa, Fb, dist(Fa,Fb))
```

```
                              else close-feat-checkp(find-closest(Fb, Fa.cobnd),Fb)
                      else close-feat-checkp (Fa, find-closest(Fa, Fb.cobnd))
                  else close-feat-checkp ((closest-edges(Fa, Fb))
      else NBRa = closestToFplane(Fb, Fa)
           if (type(NBRa) = VERTEX)      %% check if NBRa is a vertex
              then NBRb = point-face-checkp (NBRa, Fb)
                   if (NBRb = NULL)
                      then close-feat-checkp (NBRa, Fb)
                      else sub-face-face(Fa, Fb)
              else if (E-FPrism(NBRa, Fb.cell))
                      then if (vdot(NBRa.H, Fb.norm) > 0)
                              then close-feat-checkp (NBRa, Fb)
                              else close-feat-checkp
                                        (NBRa, find-closest(NBRa, Fb.cobnd))
                      else sub-face-face(Fa, Fb)

%%% Sub-Face-Face:
PROCEDURE sub-face-face(Fa, Fb)
  NBRb = closestToFplane(Fa, Fb)
  if (type(NBRb) = VERTEX)              %% Is NBRb a vertex?
     then NBRa = point-face-checkp (NBRb, Fa)
          if (NBRa = NULL)
             then close-feat-checkp (Fa, NBRb)
             else close-feat-checkp (closest-edges(Fa,Fb))
     else if (E-FPrism(NBRb, Fa.cell))
             then if (vdot(NBRb.H, Fa.norm) > 0)
                     then close-feat-checkp (Fa, NBRb)
                     else close-feat-checkp
                               ((find-closest(NBRb, Fa.cobnd), NBRb)
             else close-feat-checkp (closest-edges(Fa,Fb))

%%% The main routine
PROCEDURE close-feat-checkp (feat1, feat2)
  case (type(feat1), type(feat2))
    (VERTEX, VERTEX) RETURN vertex-vertex(feat1, feat2)
    (VERTEX, EDGE)   RETURN vertex-edge(feat1, feat2)
    (VERTEX, FACE)   RETURN vertex-face(feat1, feat2)
  (EDGE, VERTEX)   RETURN reverse(vertex-edge(feat2, feat1)
  (EDGE, EDGE)     RETURN edge-edge(feat1, feat2)
  (EDGE, FACE)     RETURN edge-face(feat1 feat2)
  (FACE, VERTEX)   RETURN reverse(vertex-face(feat2, feat1))
  (FACE, EDGE)     RETURN reverse(edge-face(feat2, feat1))
  (FACE, FACE)     RETURN face-face(feat1, feat2)
%% To check if two objects collide by their minimum separation
if (dist(feat1,feat2) ~ 0)
   then PRINT ("Collision")
```