EUROGRAPHICS 2001 / A. Chalmers and T.-M. Rhyne
(Guest Editors)

*Volume 20 (2001), Number 3*

# Accurate and Fast Proximity Queries Between Polyhedra Using Convex Surface Decomposition†

Stephen A. Ehmann and Ming C. Lin

Department of Computer Science, University of North Carolina, Chapel Hill, U. S. A.

**Abstract**

*The need to perform fast and accurate proximity queries arises frequently in physically-based modeling, simulation, animation, real-time interaction within a virtual environment, and game dynamics. The set of proximity queries include intersection detection, tolerance verification, exact and approximate minimum distance computation, and (disjoint) contact determination. Specialized data structures and algorithms have often been designed to perform each type of query separately. We present a unified approach to perform any of these queries seamlessly for general, rigid polyhedral objects with boundary representations which are orientable 2-manifolds. The proposed method involves a hierarchical data structure built upon a surface decomposition of the models. Furthermore, the incremental query algorithm takes advantage of coherence between successive frames. It has been applied to complex benchmarks and compares very favorably with earlier algorithms and systems.*

## 1. Introduction

Proximity queries are fundamental in a wide range of domains such as simulated environments, virtual prototyping, and robotics. The set of applications include virtual environments, computer games, tolerance verification, simulation-based design, and motion planning.

Proximity queries have been extensively studied, and several specialized algorithms have been proposed. These algorithms differ based on model representations, the type of queries, and the characteristics of the simulated environments [20]. Some of the commonly used algorithms are based on spatial data structures or bounding volume hierarchies. They cannot efficiently compute contact regions or perform disjoint contact determination, while our algorithm can.

We present a novel approach that can handle multiple types of queries of varying difficulty for general polyhedral models. Our goal is to provide a unified algorithmic framework within which many types of queries can be answered without using specialized data structures and costly postprocessing, such as is the case with many earlier approaches. The types of queries we address are:

- **Intersection Detection:** returns a boolean value indicating whether a pair of polyhedra is intersecting or not.
- **Tolerance Verification:** given a tolerance, returns a boolean value indicating whether the minimum distance between a pair of polyhedra is below the tolerance or not. This type of query is a generalization of *intersection detection* which has a tolerance of zero.
- **Exact Minimum Distance:** given a tolerance, returns the minimum separation distance between a pair of polyhedra *if* the distance is less than the tolerance.
- **Approximate Minimum Distance:** given an absolute error, a relative error, and a tolerance, returns a distance value such that the *exact* minimum distance between a pair of polyhedra is within either of the given error bounds *and* the returned distance value is less than the tolerance.
- **Disjoint Contact Determination:** given a tolerance, returns a set of pairs of features such that each pair represents a local minimum in the distance function between a pair of polyhedra *and* the distance at the local minimum is less than the tolerance. From these pairs of features, distances, nearest points, and contact normals can be computed yielding a simple geometric description of a contact region that can be used for collision response [1, 4, 25].

These queries can be further grouped into three classes based upon the generalizations given. We will call them: tolerance, distance, and contact queries.

† Supported in part by ARO DAAG55-98-1-0322, NSF EIA-9806027, NSF DMI-9900157, NSF IIS-9821067 and Intel

### 1.1. Main Contributions

We present a family of algorithms for efficient proximity query of general rigid polyhedra, based on convex surface decomposition. Our approach proceeds in three phases. First, a convex surface decomposition of a given polyhedron is computed. A hierarchy is created out of the resulting convex patches. Each node of the hierarchy is a convex hull which bounds its children. At the leaves are the convex hulls of the convex patches yielded by the decomposition. A pair of convex hulls is tested using a modified Lin-Canny closest feature algorithm [22]. A pair of hierarchies can be efficiently queried by recursive descent. Our key contributions are:

- A unified framework for proximity queries of varying levels of generality.
- An efficient disjoint contact determination algorithm for computing pairs of nearest features and local minima in the distance function between polyhedra. These are useful in dynamics for computing collision response subject to non-penetration constraints.
- The ability to perform proximity queries for solids in addition to their boundary surfaces. Our algorithm also supports surfaces with boundary.
- An integrated generalization of query accelerations.

The algorithm has been implemented and analyzed using various benchmarks. Experiments were performed on different shapes and combinatorial complexity of objects along with several algorithmic variations. We observe excellent performance for our algorithm, along with an indication that our acceleration techniques offer substantial improvements.

The next section surveys previous work. Section 3 gives an outline of our algorithm. Convex surface decomposition is described in section 4. Hierarchy construction is discussed in section 5. Sections 6 and 7 describe in detail the queries and query acceleration techniques. Section 8 compares the performance of our prototype implementation against other publicly available packages.

## 2. Previous Work

Proximity queries have been widely studied in many fields. Many algorithms have been proposed for various model representations. They are either approximate or exact, and answer different subsets of proximity queries. Similarly, convex decomposition and hierarchy construction have also been investigated in computer graphics, computational geometry, and solid modeling. Next, we examine earlier work in these areas.

### 2.1. Convex Decomposition

The problem of computing what we call a convex solid decomposition of a general polyhedron $P$, has been extensively investigated [3, 6, 8, 28, 29] and is known to have output of size $O(n^2)$ where $n$ is the number of boundary elements of $P$. The definition of this decomposition is:

$$P = \bigcup_i C_i$$

$$\forall i, j : i \neq j \quad C_i \cap C_j = \emptyset \tag{1}$$

where the $C_i$ are solid convex pieces.

Another type of decomposition is a convex surface decomposition. This has been addressed by [7, 9] and the definition is:

$$boundary(P) = \bigcup_i c_i$$

$$\forall i, j : i \neq j \quad c_i \cap c_j = \emptyset$$

$$c_i \subseteq boundary(C_i) \tag{2}$$

where the $c_i$ are convex surface patches that lie entirely on the surface (boundary) of their convex hull $C_i = \mathbf{CH}(c_i)$ which is the corresponding convex piece. This type of decomposition is of complexity $O(r)$ where $r$ is the number of reflex edges in the surface (boundary) of $P$. This also works for surfaces with boundary (non-solids) since no notion of inside or outside is required. The problem of finding the minimum number of patches is proven to be NP-Complete in [7], so they investigate various approximation algorithms to find a small number of patches. Some of our decomposition algorithms are derived and extended from theirs.

### 2.2. Hierarchy Construction

Bounding volume hierarchies (BVHs) have proven efficient tools in the computation of proximity information. The property that they satisfy is that each node of the hierarchy bounds all the geometry in its child subtrees. There are two main approaches to build BVHs: top-down or bottom-up. Top-down construction involves recursive splitting and is fast and simple to implement [15, 21]. Bottom-up, on the other hand, involves grouping primitives hierarchically and is often computationally more expensive [2]. For the most part, binary trees are constructed although trees of higher degree are possible. Analytical evidence against using degrees higher than two is presented in [18].

### 2.3. Proximity Query

#### 2.3.1. Convex Polyhedra

Much of the prior work on proximity queries has focused on algorithms for solid convex polyhedra. A number of algorithms with good asymptotic performance have been proposed, including an $O(log^2 n)$ algorithm by [10]. This elegant approach is difficult to implement robustly in 3D, however. Good theoretical and practical approaches based on the linear complexity of the linear programming problem

are known [24, 30]. Minkowski difference and convex optimization techniques are used in [14] to compute the distance between convex polyhedra by finding the closest point between the Minkowski polytope and the origin. In applications involving rigid motion, geometric coherence has been exploited to design algorithms for convex polyhedra based on either traversing features using locality or convex optimization [5, 22, 23, 26]. These algorithms exploit spatial and temporal coherence between successive queries, and work well in practice. Multiresolution representations have been used to further speed up queries between convex polytopes [11, 16]. For a more thorough discussion of convex polyhedral proximity query, see [12]. We use a minimum distance computation algorithm based on "Voronoi marching" as a subroutine to test the proximity of a pair of convex polyhedra. It is powerful and general in that it yields answers to all five proximity query types for convex polyhedra, with just one query.

#### 2.3.2. General Polygonal Models

Bounding volume hierarchies are presently regarded as one of the most general methods for performing proximity queries between general polyhedra. Specifically, sphere trees, AABB trees, OBB trees, and k-dops have been used for fast intersection detection queries [2, 15, 17, 18, 27]. Swept sphere volumes (SSVs) have been used for intersection detection, tolerance verification, and exact and approximate minimum distance queries [21]. These algorithms do not assume connectivity or convexity of the input, thus their queries are purely based on boundary representations and cannot discriminate between the case of one solid entirely within another and disjoint solids. With the exception of [4], most of them do not perform the contact analysis required for physically-based simulation.

## 3. Algorithm Overview

We assume, for simplicity, and without loss of generality, that a polyhedron's boundary is composed of triangles. Our algorithm consists of three major components. The first two are preprocessing and the last is the query, happening at runtime. Next, a brief overview of each is given.

### 3.1. Convex Surface Decomposition

The first phase is the decomposition of a general (possibly non-convex) polyhedron's *surface* into a small collection of convex patches. We have developed an approach based on graph searching methods [7]. A seed point is used to start a graph search which includes faces into the current convex patch if certain criteria are met. The resulting decomposition is similar to Eqn. 2. This is in contrast to a convex *solid* decomposition. A convex piece is formed from a convex patch by taking its convex hull. More details, along with our experiences, are given in section 4.

### 3.2. Hierarchy Construction

Based on the decomposition, a BVH is built with the convex pieces as its leaves. We focus on the top-down construction of binary bounding volume hierarchies. There are various splitting axes and rules that we could use. Typically, BVHs have primitives at the leaves that are triangles and internal nodes which are certain simple bounding volumes. Since we use convex hull bounding volumes, a convex polyhedron is the only type of geometry stored in our BVH. BVHs constructed using our algorithm have substantially fewer levels and are tighter fitting than other BVHs. Furthermore, we have a potential for pruning relationships between any internal node and its leaves. More details on the hierarchy construction in general are given in section 5.

### 3.3. Query

Without loss of generality, the proximity of two models is queried. Given that the geometry in the hierarchy is convex, geometric locality is exploited to speed up queries in a unified manner. A recursive traversal algorithm is used to test a pair of hierarchies. Due to the structure of our BVHs, we only require a test between a pair of convex polyhedra. More details on the query algorithm in general, along with possible acceleration techniques, are given in sections 6 and 7. In section 8, we benchmark various queries with and without optimizations.

## 4. Convex Surface Decomposition

We consider an orientable polyhedral surface $S$. If it is closed, then it is the surface of some polyhedral solid $P$: $S = boundary(P)$. A constraint is added to Eqn. 2 to decompose a polyhedral surface $S$:

$$S \cap C_i = c_i \tag{3}$$

That is, $S$ can only meet the surface of any convex piece ($C_i$) in exactly the convex patch ($c_i$) used to create the piece. If $S$ is closed, the pieces are contained entirely within $P$ but their union does not necessarily equal $P$. Whether it is closed or not, $S$ is a subset of the union of the boundaries of the pieces: $S \subseteq \bigcup_i boundary(C_i)$. This is important in order to ensure correctness of the query.

The algorithm that we use to create a decomposition like this is based on exploring face relationships as a patch is being created. It uses a graph search and an incremental convex hull algorithm. We have noticed that there are many factors that determine the number and structure of the patches that are created. The interaction between a decomposition method and the query performance is complex, difficult to characterize in practice, and highly dependent on the environments and contact scenarios. Therefore, we take as a good decomposition for now, one whose collection of pieces takes up a small amount of memory.

We extend the surface decomposition algorithms of [7] to handle the additional non-intersection constraint (Eqn. 3), and propose a better "seeding" technique.

The graph of a polyhedral surface is simply its edges and vertices. The *dual* of this graph is a graph with the same

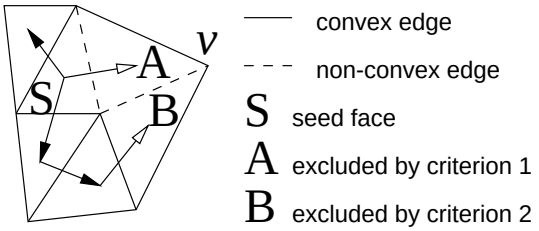


**Figure 1:** *Creation of a convex patch*

edges but with the roles of vertices and faces reversed. A depth-first search (DFS) or breadth-first search (BFS) is used to traverse the dual graph of $S$, while a convex hull $C$, is maintained as faces are added to the current convex patch $c$. When the search terminates, $c$ is the collection of faces of $S$ that were added during the search. The convex hull $C$ is the convex piece corresponding to $c$. In general, when the search enters a candidate face $f$ (which is triangular), there is a vertex $v$, on the opposite side from the edge $e$, of entry. The addition of $v$ to $C$ is identical to the addition of $f$ to $c$. Not every face visited by the graph search can be included however.

To meet the required constraints, the criteria by which to add $v$ to $C$ to get $C'$ are:

1. $e$ is not a reflex edge
2. no faces from $S$ that are in $c$ are visible from $v$
3. the addition of $v$ to $C$ does not cause $C'$ to intersect with any part of $S$ not in $c$

which if satisfied, allow the addition of $v$ to $C$ and $f$ to $c$. Figure 1 shows a search that starts at a face **S** and forms a patch. Face **A** cannot be included because the search crosses a reflex edge to get to it (first criterion violated). Face **B** cannot be included because **S** (already in the patch) is visible to $v$ (second criterion violated).

## 5. Top-Down Hierarchy Construction

To create a binary hierarchy top-down, a recursive splitting of the list of primitives is done. Recall that the primitives in this context are convex pieces. Algorithm 1 demonstrates the process. First, the recursion descends, splitting as it goes. As the recursion unwinds, the bounding volumes are created and face sharing and classification are computed.

### 5.1. Splitting

There are many ways to split a set of primitives into two groups. We assign as a representative point to each primitive its center of mass (COM). The covariance matrix of the convex hull of the COMs is computed. Using eigen-analysis, the maximum spread direction is found and this becomes the splitting axis. Each COM is projected onto the splitting axis. The projected COMs are then sorted along it. The problem has now been reduced to a sorted list of real values. There

**Algorithm 1:** Construct_Hierarchy( L )

**Input:** list L of primitives

```
1.  if L is of length 1 then
2.      return New_Leaf_Node( L )
3.  Split( L, L1, L2 )
4.  H1 = Construct_Hierarchy( L1 )
5.  H2 = Construct_Hierarchy( L2 )
6.  H = New_Internal_Node( H1, H2 )
7.  CH = Compute_Convex_Hull( H1, H2 )
8.  for all the faces in CH do
9.      if the face is in H1 then
10.         store a pointer to the H1 face in H
11.     else if the face is in H2 then
12.         store a pointer to the H2 face in H
13.     else
14.         create a copy of the CH face in H
15. return H
```

are many possible ways to go about splitting the values into two groups. We considered the following:

- **Median:** split at the median value.
- **Midpoint:** split at the midpoint of the first and last value.
- **Mean:** split at the mean of the values.

Of course there are many other methods that can be used. An optimal splitting strategy for top-down construction of BVHs remains an open research issue.

### 5.2. Face Sharing and Classification

In Algorithm 1, lines 8-14 compute the face sharing between levels. If a face is shared by more than one convex polyhedron, only one copy of it exists. This cuts down the memory requirements. The face adjacency information must still be copied since the face's neighbors could be different for each convex polyhedron it resides on.

Each face is given one of three classifications. These classifications are the face relationships between a bounding volume at an internal node and the convex pieces at its leaves. They are important for making the query efficient. The classifications listed in order of decreasing pruning power are:

1. **ORIGINAL:** faces from the original surface.
2. **CONTAINED:** faces created when converting patches to pieces by convex hull. According to the definition of convex surface decomposition that we use, all these faces are contained in the original solid if the surface is closed.
3. **FREE:** faces created when constructing the hierarchy. These are created in line 14 of Algorithm 1. They are free in the sense that they do not belong to any of the convex pieces.

These classifications are propagated up the hierarchy as far as they can go due to the face sharing, which is desirable.

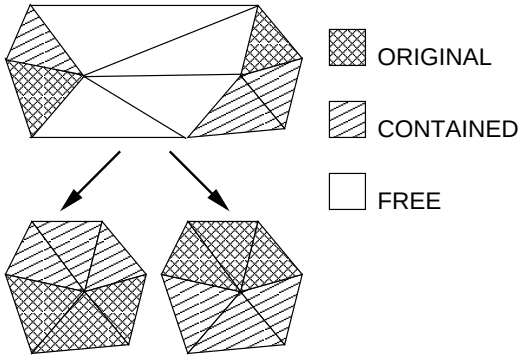


**Figure 2:** *Hierarchy with face classifications*

Thus, every face in every convex polyhedron in the hierarchy has one of the classifications. See Figures 2, 9, and 10. Each lower dimensional feature (vertex or edge) must also have a classification. No vertices were created during either the decomposition or the hierarchy construction. Therefore, all vertices are implicitly treated as **ORIGINAL**. We consider an edge to be a part of both faces it is adjacent to. Thus, we classify an edge the same as its neighboring face of highest pruning power. These classifications are used in the query, which is described in the next section.

## 6. Hierarchical Query

In this section, the runtime query algorithm is explained in detail. First, the core routine that performs a proximity test between a pair of convex polyhedra is described. Then, we describe an algorithm for traversing a pair of hierarchies. A basic traversal is outlined, followed by an elaboration for each type of query.

### 6.1. Atomic Query Using Voronoi Marching

We use an iterative distance minimization algorithm based on Voronoi marching (VM) [11] to perform atomic queries between pairs of convex polyhedra. VM is a technique that performs surface walking on a pair of convex polyhedra using their external Voronoi regions to find the pair of nearest features between them. From this pair, all the information that is needed for all types of queries (tolerance, distance, and contact) can be derived. If intersection is detected, the distance is considered to be negative. The test is done on convex polyhedral *solids*. Thus, even if the boundary of one is contained entirely within the other, intersection is still detected. One issue relating to this subroutine is its initialization. We propose a simple yet effective directional lookup table scheme [12] for cases without high temporal coherence. If temporal coherence is present, the previous closest features for a pair of convex polyhedra are used to initialize where possible.

### 6.2. Hierarchical Traversal

Algorithm 2 is the generic traversal algorithm used to query a pair of polyhedra by using their hierarchies. Here, we will only focus on the three fundamental types of queries: tolerance, distance, and contact. The traversal has different pruning conditions and different conditions under which to find an answer for the different types of query. In general, not all the features on the bounding volumes at the internal nodes are **FREE**. If both of the nearest features are non-**FREE** (conditions at line 3 and line 7 are both false), then the recursion terminates, and an answer can be given for the pair of sub-hierarchies. Each query type is examined next.

**Algorithm 2:** Traverse_Hierarchies( H1, H2 )

**Input:** the hierarchies, H1 and H2, to traverse

```
1.  Perform VM test on the roots of H1 and H2 to get
        nearest features f1 and f2
2.  if not pruned then
3.      if f1 is FREE then
4.          Traverse_Hierarchies( H1.left_child, H2 )
5.          if query is resolved then return answer
6.          Traverse_Hierarchies( H1.right_child, H2 )
7.      else if f2 is FREE then
8.          Traverse_Hierarchies( H1, H2.left_child )
9.          if query is resolved then return answer
10.         Traverse_Hierarchies( H1, H2.right_child )
11. return answer
```

#### 6.2.1. Tolerance Query

Tolerance queries involve returning a boolean answer, given a tolerance, indicating whether two polyhedra have a minimum distance less than the tolerance. The tolerance must be greater or equal to zero. If it is zero, then the query is the same as intersection detection. The recursion is pruned for this type of query if the distance yielded by the VM subroutine is greater than the tolerance. If this is the case, we know that no pairs in the sub-hierarchies can possibly be closer. The answer returned would be *false*. If the minimum distance is less than the tolerance and both of the nearest features are non-**FREE**, then the polyhedra failed the tolerance, and *true* is returned. Otherwise, recursion is performed.

#### 6.2.2. Distance Query

Given an absolute error $\varepsilon_a$, a relative error $\varepsilon_r$, and a tolerance, distance queries return a distance that is less than the tolerance and is within the given error of the exact minimum distance. If the error is zero, then exact minimum distance is computed. If the tolerance is zero, then the query is equivalent to intersection detection. Each recursive call receives as a parameter the minimum distance, $D_{min}$, found so far. The recursion is pruned for this type of query if the distance yielded by the VM subroutine, $d$, is greater than the tolerance or if no pairs in the sub-hierarchies can affect the distance. If $d+\varepsilon_a > D_{min}$ and $d \times (1+\varepsilon_r) > D_{min}$, then there is no need to recur. The recursion has reached the base case if both of the nearest features are non-**FREE**. If this is the case

and the nearest features are intersecting, then a negative distance is returned since the polyhedra are intersecting and the distance cannot be further minimized. Otherwise, $d$ is compared to $D_{min}$. The new minimum distance $d$ is assigned to $D_{min}$ if $d + \varepsilon_a < D_{min}$ or $d \times (1 + \varepsilon_r) < D_{min}$. On the other hand, if at least one of the nearest features is **FREE**, then recursion is performed and $D_{min}$ is passed on.

#### 6.2.3. Contact Query

Given a tolerance, contact queries record pairs of features at local minima in the distance function between two polyhedra. This is the most general type of query. Its recursive structure is identical to that of the exact minimum distance query. At each recursive invocation, the VM subroutine is performed and the nearest features and distance are evaluated. The recursion is pruned if the distance is greater than the tolerance. The recursion has reached the base case if both of the nearest features are non-**FREE**. If this is the case and the nearest features are intersecting, then the entire query terminates and reports intersection since contacts are only computed when the polyhedra are disjoint. Otherwise, the nearest feature pair is added to the contact list. On the other hand, if at least one of the nearest features is **FREE**, then recursion is performed.

## 7. Query Acceleration

In this section, we describe three techniques for accelerating hierarchical queries. We present them in order of increasing sophistication. An evaluation of them is given in section 8. They are derived from some related work but extended and generalized for our framework. Each exploits one of two types of coherence: spatial or temporal. The former type refers to the coherence that is inherent in the spatial structure of a *single* polyhedron. Since it is rigid, this type of coherence exists for any query. The latter type is coherence for a pair across queries. In a sense, it refers to the changes in relative positions and orientations of two polyhedra.

### 7.1. Piece Caching

Piece caching is an idea similar to triangle caching in [21] which exploits temporal coherence. It can be used for tolerance and distance queries since they only require a single answer. The idea is to remember the pair of convex polyhedra that answered the last query. If there was not much relative movement between the last query and the current one, it is highly likely that the same pair can yield some valuable information that can speed up the query. The pair of nearest features found in the previous query can be used to initialize the atomic query for the cached pair of convex pieces.

For tolerance queries, the pair from the previous query is not stored if the polyhedra are sufficiently distant. Otherwise, the previous query failed and the pair where failure was detected is tested before any other pair. If the current query fails on it, we are done. Otherwise, an ordinary query is performed.

For distance queries, the pair that yielded the minimum distance from the previous query is tested first. If the pair of nearest features are both non-**FREE**, then the minimum distance is initialized to be the distance computed by the test. This does not entirely answer the query, but provides for increased pruning due to $D_{min}$ being smaller. Otherwise, $D_{min}$ is initialized to infinity and the ordinary query is performed.

### 7.2. Priority Directed Search

This type of optimization is derived from [21] and takes advantage of spatial coherence within the geometry of a single object and can be used even when temporal coherence is not present. It speeds up distance queries by honing in on the minimum distance more quickly. The idea is to always process next, the pair of sub-hierarchies whose root bounding volumes have the smallest distance. A priority queue of outstanding pairs of sub-hierarchies is maintained with the highest priority going to the pair with the minimum distance between its roots' bounding volumes.

This acceleration technique counts on the fact that it is likely that a small distance can be found between sub-hierarchies whose roots have a small distance between their bounding volumes. Thus, the priority directed search tends to first explore the recursion branches that are most promising, with the hope of lowering $D_{min}$ and pruning recursion branches that would otherwise have been taken.

To illustrate how priority directed search works, assume that the pair at the head of the queue is to be tested. It is removed from the queue and the two root bounding volumes are tested using the subroutine for convex polyhedra. If the test prunes away any more tests between the two sub-hierarchies, or if both of the nearest features are non-**FREE**, we are done (since the recursion branch terminated). Otherwise, one or both of the nearest features is **FREE**. It is decided which sub-hierarchy to descend on, and the two recursive pairs are inserted into the queue. If the queue is too full to accommodate the second pair (it can always accommodate the first due to the removal of the head), then the second pair is immediately resolved recursively like would be done without priority directed search.

### 7.3. Generalized Front Tracking

Our most powerful and sophisticated method is called *generalized front tracking* and is a query acceleration technique that exploits temporal coherence. It can be applied to all types of queries. Front tracking is a term originally coined by [19] for caching a *single* bounding volume with respect to *one* hierarchy during a hierarchical traversal. Generalized front tracking extends this idea to caching multiple pairs of BVs.

The main idea is that when two objects are in close proximity, we would like to avoid having to traverse the upper levels of their BVHs each time. If the roots of sub-hierarchies where important events happened in a query could be remembered, then these spots in the traversal can

be used as starting points for future traversals. In our case, pairs in the recursion found to intersect are important.

A *front* refers to the leaves of a special recursion tree that is built and modified as queries progress. This tree is akin to the bounding volume test tree (BVTT) that was used as an analysis tool in [21]. The BVTT represents the hierarchy of tests performed during a query. Each node in the BVTT corresponds to a single test between a pair of BVs. Each node in the BVTT represents a pair of convex polyhedra from the hierarchies which either has no children (it is a leaf), or exactly two children. If a pair exists in the tree, then one of the following conditions holds for it:

- it was intersecting the last time it was tested
- its sibling was intersecting the last time and this pair exists to ensure that its parent has two children
- neither it nor its sibling are intersecting. Since the front is only raised one level at a time, the front has not yet been raised to its parent

The front is used to initialize multiple recursion branches, whose overall effect is identical to starting at the root of the two hierarchies. Each item in the front can also possibly store a pair of features used to initialize the atomic query subroutine for its pair of convex polyhedra.

The issue of how to modify the BVTT is explained next. As the recursion deepens, the tree grows by the creation of new children. This is termed "dropping" the front and normally occurs when the polyhedra move closer together. Usually, when the polyhedra move apart, the front must be "raised". We only raise it one level per query since raising it recursively incurs the cost of additional bounding volume tests.

Generalized front tracking, as we have presented it here, works well across different types of queries. As an example, if intersection is queried, followed by contact, the front generated by the intersection query is a good starting point for the contact query.

## 8. Implementation and Experiments

We implemented our algorithms including the various acceleration techniques using C++. The source code is available to the public at our project website:

http://www.cs.unc.edu/∼geom/SWIFT++/

We used the publicly available QHULL package to perform convex hull computations. We also created our own rigid body dynamics simulator which was used to prepare benchmarks and verify the overall correctness of the implementation. We used an OpenGL front-end to visualize the various phases of our algorithm.

### 8.1. Benchmark Design

For performance comparison, we created 6 scenarios (see Figures 3– 8):

1. a pair of interlocked tori, one bent, the other wrinkled
2. a fixed cup and a spoon being dropped into the cup
3. a fixed spiral peg and three large bent tori moving on it
4. a fixed spiral peg and three small bent tori moving on it
5. a fixed upper teeth model and a lower teeth model that shifts laterally
6. a fixed upper teeth model and a lower teeth model that opens and closes.

For the first four, we generated sequences of transformations by running a dynamic simulation with non-penetration constraints. For the last two, we generated transformations procedurally. Each of the tori is composed of 2000 triangles, and the spiral of 8000. The cup has 8000 triangles and the spoon consists of 84. The teeth are scanned models, and have over 40,000 triangles each. Please refer to the Proceedings CD and the project website for video clips of our simulations.

Each scenario was run and queries were performed for intersection detection, approximate distance computation (10% relative error), and exact distance computation, all using our implementation of convex hull (CH) bounding volume hierarchies. For intersection detection, we also tested BVHs composed of OBBs [15] and 18-dops [18]. For distance computation, we also tested SSV BVHs [21]. Implementations of all three types of BVHs are publicly available as RAPID, QuickCD, and PQP respectively.

### 8.2. Results and Analysis

All timings were performed on a 450 MHz Intel PIII. Timings are given in milliseconds and correspond to average query time. We only show timings for our algorithm with the **Midpoint** splitting method since all the splitting methods yielded similar results.

Intersection detection results are reported in Table 1. Our implementation, using CH BVHs, was run with all the combinations of piece caching (PC) and generalized front tracking (FT) optimizations. Since our benchmarks do not include much intersection, the PC results do not yield much additional insight, therefore we do not show them. BVHs using OBBs and 18-dops were tested using identical input data. The 18-dop BVHs were tested with and without the front tracking (FT) optimization available in QuickCD. Since QuickCD is unable to handle scenes with multiple moving objects, those entries in the table are denoted "N/A".

The performance of approximate distance (10% relative error) and exact distance computation are very similar, so we just report the exact distance results in Table 2. Our implementation was run with all combinations of PC, FT, and priority directed search (PD). SSV BVHs were also tested using identical input data, once with the optimization of triangle caching (TC), and once with the optimization of priority directed search (PD), each available in PQP.

OBB BVHs are well-suited to intersection detection. They perform better than ours on some of the benchmarks. However, they are very inefficient at distance computation [21],

| Algorithm | Interlocked Tori | Cup and Spoon | Spiral Peg Large Tori | Spiral Peg Small Tori | Teeth Shifting | Teeth Open and Close |
|---|---|---|---|---|---|---|
| OBB | 2.711 | 0.8761 | 0.276 | 0.459 | 0.514 | 0.841 |
| 18-dop | 1.609 | 0.584 | N/A | N/A | 24.69 | 17.79 |
| 18-dop (FT) | 7.857 | 0.660 | N/A | N/A | 91.67 | 20.65 |
| CH | 1.510 | 0.840 | 9.865 | 1.410 | 43.42 | 14.32 |
| CH (FT) | 0.696 | 0.505 | 3.505 | 0.966 | 0.177 | 4.190 |

**Table 1:** *Performance of Various Algorithms on Intersection Detection.*

| Algorithm | Interlocked Tori | Cup and Spoon | Spiral Peg Large Tori | Spiral Peg Small Tori | Teeth Shifting | Teeth Open and Close |
|---|---|---|---|---|---|---|
| SSV (TC) | 7.492 | 3.931 | 34.91 | 36.61 | 0.604 | 1.733 |
| SSV (PD) | 11.16 | 3.814 | 43.16 | 47.67 | 1.780 | 3.432 |
| CH | 4.424 | 1.728 | 11.81 | 2.720 | 54.73 | 20.23 |
| CH (PC) | 2.452 | 1.234 | 10.99 | 1.910 | 11.53 | 10.43 |
| CH (PD) | 2.562 | 1.284 | 11.88 | 2.009 | 7.152 | 9.937 |
| CH (FT) | 2.414 | 1.058 | 4.938 | 1.572 | 0.184 | 7.980 |
| CH (PC,PD) | 2.498 | 1.266 | 11.78 | 1.969 | 1.466 | 9.285 |
| CH (PC,FT) | 2.444 | 1.070 | 4.975 | 1.599 | 0.114 | 8.090 |
| CH (PD,FT) | 1.602 | 0.923 | 5.105 | 1.513 | 5.529 | 7.740 |
| CH (PC,PD,FT) | 1.612 | 0.919 | 5.113 | 1.526 | 5.653 | 7.957 |

**Table 2:** *Performance of Various Algorithms on Exact Distance Computation.*

and cannot be used to do contact determination without additional post-processing which can be costly. Each of our bounding volume tests yields much more information than either the OBB or the 18-dop overlap test, allowing us to answer the more complex queries without additional computation. Furthermore, generalized front tracking works well in parallel close proximity situations as do OBBs, as can be seen in the shifting teeth benchmark.

Our algorithm outperforms the 18-dop implementation on every benchmark. The 18-dop implementation performs better without FT. We conjecture that this could be due to the fact that its front tracking technique was designed to handle a complex static scene with a single relatively simpler dynamic object. Our generalized front tracking, on the other hand, gives a performance gain on all of the benchmarks, thus it is applicable to various types of environments.

For exact distance computation, SSV BVHs are outperformed by our algorithm consistently in four of the benchmarks. In the shifting teeth benchmark, generalized front tracking once again gives a noticeable speedup and is the best overall. Our distance computation in general is very efficient, partly due to the large convex grouping at the leaves of the hierarchy, and partly due to the VM subroutine which is quite efficient at computing distance. All of the query optimizations yield improvements in all the benchmarks. Generalized front tracking combined with priority directed search is the best combination in the first two benchmarks while front tracking alone is the best in the others.

## 9. Conclusion

We have presented an unified algorithmic framework that provides multiple types of proximity queries between general polyhedral solids. The three phases of the algorithm were discussed and the resulting performance evaluated. A new algorithm for disjoint contact determination was presented, as well as some query acceleration techniques.

There is still open research in each of the areas. For convex surface decomposition, other heuristics for starting the surface searching may yield performance gains. Optimal hierarchy construction remains a hard problem due to the vast difference among models. A combination of bottom-up and top-down construction may yield better hierarchies than a purely top-down approach, at a reasonable cost. Finally, the memory requirements of our algorithm are higher than most commonly used bounding volume hierarchies and it may be possible to perform lazy evaluation with almost no degradation in speed, rather than storing the entire hierarchies in memory during precomputation.

Perhaps a multiresolution hierarchy can be built directly on top of the original surface for general polyhedra. The extension of this method to deformable models is an exciting challenge as well.

For more details as well as additional results, please refer to the companion technical report [13].

## References


1. D. Baraff. *Dynamic Simulation of Non-Penetrating Rigid Body Simulation.* PhD thesis, Cornell Univ., 1992. 1
2. G. Barequet, B. Chazelle, L. Guibas, J. Mitchell, and A. Tal. BOXTREE: A hierarchical representation of surfaces in 3D. *Proc. Eurographics*, 1996. 2, 3
3. C. Bajaj and T. K. Dey. Convex decomposition of polyhedra and robustness. *SIAM J. Comp.*, 21:339–364, 1992. 2
4. W. Bouma and G. Vanecek, Modeling contacts in a physically based simulation. *Proc. Solid Modeling*, 409-419, 1993. 1, 3
5. S. Cameron. Enhancing GJK: Computing minimum and penetration distance between convex polyhedra. *Proc. IEEE ICRA*, 3112–3117, 1997. 3
6. B. Chazelle. Convex partitions of polyhedra: a lower bound and worst-case optimal algorithm. *SIAM J. Comp.*, 13:488–507, 1984. 2
7. B. Chazelle, D. Dobkin, N. Shouraboura, and A. Tal. Strategies for polyhedral surface decomposition: An experimental study. *Comp. Geom. Theory Appl.*, 7:327–342, 1997. 2, 3
8. B. Chazelle and L. Palios. Triangulating a non-convex polytope. *Discrete Comp. Geom.*, 5:505–526, 1990. 2
9. B. Chazelle and L. Palios. Decomposing the boundary of a nonconvex polyhedron. *Algorithmica*, 17:245–265, 1997. 2
10. D. P. Dobkin and D. G. Kirkpatrick. Determining the separation of preprocessed polyhedra – a unified approach. *Proc. 17th Intl. Colloq. Automata Lang. Program.*, volume 443 of *Lecture Notes Comp. Sci.*, 400–413. Springer-Verlag, 1990. 2
11. S. A. Ehmann and M. C. Lin. Accelerated proximity queries between convex polyhedra using multi-level Voronoi marching. *Proc. IEEE IROS*, 2000. 3, 5
12. S. A. Ehmann and M. C. Lin. Accelerated proximity queries between convex polyhedra using multi-level Voronoi marching. Tech. Report TR00-026, Department of Computer Science, Univ. of North Carolina, 2000. 3, 5
13. S. A. Ehmann and M. C. Lin. Accurate and fast proximity queries between polyhedra using convex surface decomposition. Tech. Report TR01-012, Department of Computer Science, Univ. of North Carolina, 2001. 9
14. 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*, 4:193–203, 1988. 3
15. S. Gottschalk, M. Lin, and D. Manocha. OBB-Tree: A hierarchical structure for rapid interference detection. *Proc. ACM SIGGRAPH*, 171–180, 1996. 2, 3, 7
16. L. Guibas, D. Hsu, and L. Zhang. *H-Walk*: Hierarchical distance computation for moving convex bodies. *Proc. ACM Comp. Geom.*, 1999. 3
17. P. M. Hubbard. Interactive collision detection. *Proc. of IEEE Symposium on Research Frontiers in VR*, 1993. 3
18. J. Klosowski, M. Held, J. S. B. Mitchell, K. Zikan, and H. Sowizral. Efficient collision detection using bounding volume hierarchies of *k*-DOPs. *IEEE TVCG*, 4(1):21–36, 1998. 2, 3, 7
19. J. Klosowski. *Efficient Collision Detection for Interactive 3D Graphics and Virtual Environments.* PhD thesis, State Univ. of New York at Stony Brook, 1998. 6
20. M. Lin and S. Gottschalk, *Collision detection between geometric models: a survey*. *Proc. IMA Conf. on Mathematics of Surfaces*, 37–56, 1998. 1
21. E. Larsen, S. Gottschalk, M. Lin, and D. Manocha. Fast proximity queries with swept sphere volumes. Tech. Report TR99-018, Department of Computer Science, Univ. of North Carolina, 1999. 2, 3, 6, 7
22. M. C. Lin and J. F. Canny. Efficient algorithms for incremental distance computation. In *IEEE Conference on Robotics and Automation*, pages 1008–1014, 1991. 2, 3
23. M. C. Lin. *Efficient Collision Detection for Animation and Robotics.* PhD thesis, Univ. California, Berkeley, 1993. 3
24. N. Megiddo. Linear-time algorithms for linear programming in $R^3$ and related problems. *SIAM J. Computing*, 12:759–776, 1983. 3
25. B. Mirtich and J. Canny. Impulse-based simulation of rigid bodies. *Proc. ACM I3D*, 1995. 1
26. B. Mirtich. V-Clip: Fast and robust polyhedral collision detection. *ACM ToG*, 17(3):177–208, July 1998. 3
27. S. Quinlan. Efficient distance computation between non-convex objects. *Proc. IEEE ICRA*, 3324–3329, 1994. 3
28. A. Rappoport. The extended convex differences tree (ECDT) representation for N-dimensional polyhedra. *Int. Journ. on Comp. Geom. & Apps.*, 1(3):227–241, 1991. 2
29. J. Ruppert and R. Seidel. On the difficulty of triangulating three-dimensional non-convex polyhedra. *Discrete Comp. Geom.*, 7:227–253, 1992. 2
30. R. Seidel. Linear programming and convex hulls made easy. *Proc. ACM Comp. Geom.*, 211–215, 1990. 3

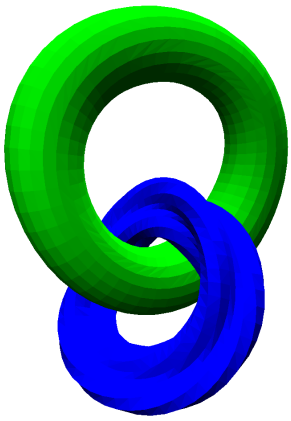

**Figure 3:** *Two interlocked tori*

**Figure 4:** *A cup and a spoon*

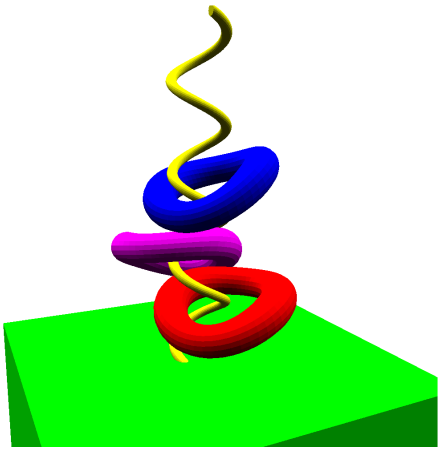

**Figure 5:** *Spiral peg and three large bent tori*

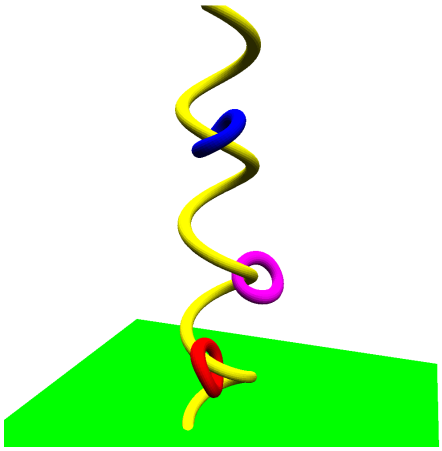

**Figure 6:** *Spiral peg and three small bent tori*

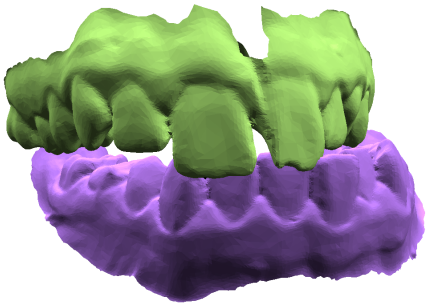

**Figure 7:** *Scanned teeth model (over 40k polygons) shifting laterally*

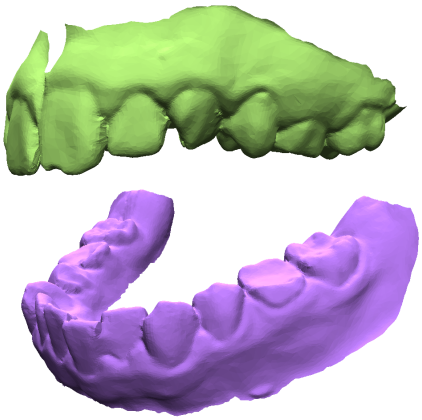

**Figure 8:** *Scanned teeth model (over 40k polygons) opening and closing*

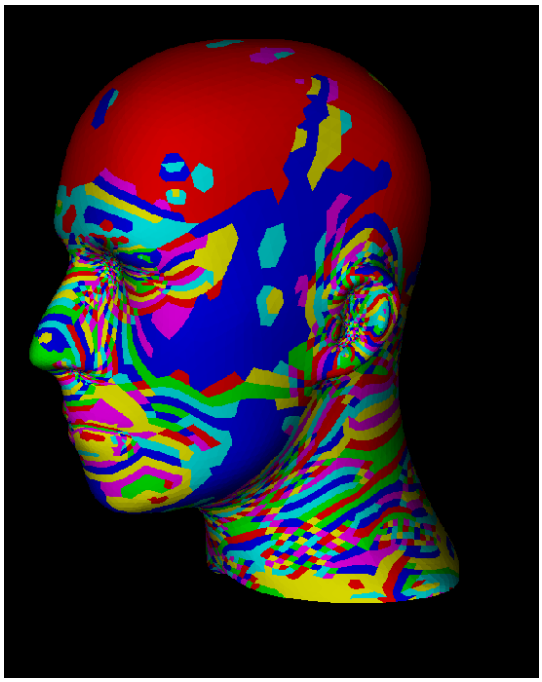

**Figure 9:** *The surface decomposition of a head composed of over 20k triangles. The different colors are randomly assigned to the different patches. Regions of higher non-convexity result in more patches like in the ear and eye. The top of the head is fairly convex, yielding few patches.*

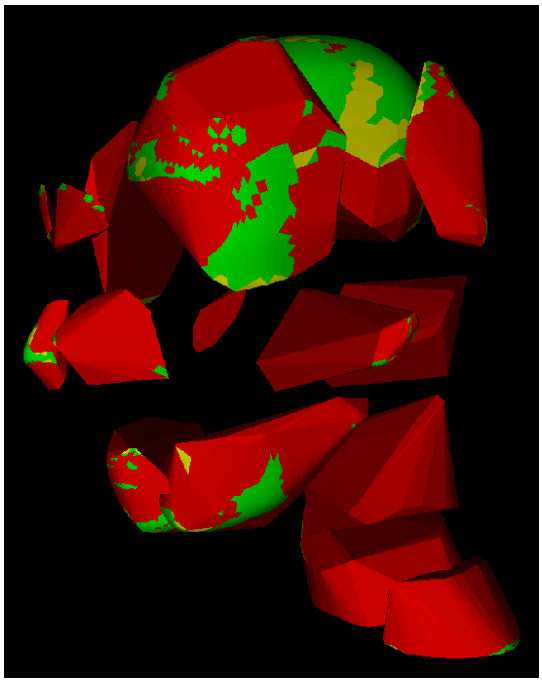

**Figure 10:** *A level of the hierarchy built for the head model. The convex hulls have been translated outwards for easier visualization. Green denotes **ORIGINAL** faces, yellow is for **CONTAINED** faces, and the red faces are **FREE**.*