Purdue University
Purdue e-Pubs

Computer Science Technical Reports | Department of Computer Science

1990

A Data Structure for Analyzing Collisions of Moving Objects

George Vanecek

Report Number:
90-986

Vanecek, George, "A Data Structure for Analyzing Collisions of Moving Objects" (1990). *Computer Science Technical Reports.* Paper 838.
http://docs.lib.purdue.edu/cstech/838

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

A DATA STRUCTURE FOR ANALYZING
COLLISIONS OF MOVING OBJECTS

George Vanecek, Jr.

CSD TR-986
June 1990

A Data Structure for Analyzing Collisions of Moving Objects

George Vaněček, Jr.[1]
Department of Computer Science
Purdue University
West Lafayette, IN 47907

June 18, 1990

Abstract

Computer systems that simulate dynamic systems of bodies require the detection and analysis of collisions between moving bodies. For a restricted class of parameterized objects, collision detection is relatively simple. However, when a simulation system uses a solid modeler to obtain objects with arbitrarily complex topology, detecting collisions efficiently between two moving objects becomes difficult.

This paper presents a new data structure for analyzing the spatial relationship of two objects. Given the objects as boundary representations (B-reps), a spatial index is constructed for the B-reps which allows fast edge and vertex classifications to be performed. This data structure solves the collision detection problem efficiently, and uses, moreover, purely solid modeling techniques.

[1] This work has been supported in part by NSF Grant CCR-86-19817 to Purdue University.

1 Introduction

Dynamic simulation systems are being developed to model the motion of rigid bodies according to Newtonian physics. Such systems have use in the engineering analysis of mechanisms in animation, automatic assembly, and in motion planning. The ability to perform realistic simulations necessitates the detection and analysis of the collisions between objects. When the time and place of collisions can be predicted a priori, time-varying forcing functions can be added to prevent interpenetration and mimic the effects of collision. However, in case the objects have complex boundaries, collisions cannot usually be predicted, and must be detected instead in an automatic fashion. How difficult it is to do this depends greatly on the class of objects the simulations allow.

When only the motion of bodies is of interest, then all computer modeled objects can be simulated. For this, only an object's centroid, mass, density, and inertial information is required. For instance, the simulation of the spin of a satellite or the tumble of an asteroid assumes the absence of other possibly colliding bodies. However, when multiple objects are simulated simultaneously, and the objects could interfere with each other, the topology and geometry of an object becomes pertinent, and detecting collisions becomes necessary. Once a collision is detected, the collision has to be analyzed and its point(s) of contact and intensity have to be determined. For collisions of sufficient impact, impulse vectors have to be computed. However, resolving the type of collision is not that simple. Depending on the relative velocities of the respective objects, the objects may be in stable contact, or be sliding across each other. In these cases of temporary contact, the objects may touch at several distinct regions, for example two legs of a robot resting on a floor.

The easiest approach is to restrict the allowed class of objects. The simplest is the class of all balls (i.e, solids with spherical boundaries). Each object is represented simply by the position of the centroid and the radius. A collision between two balls occurs when the distance between their two centroids is equal to the sum of their radii.

A slightly more difficult approach is to allow only a small well chosen set of parameterized convex objects, for instance, the class of all spheres, and blocks. This class of objects is rich enough to conduct many reasonable simulations as was the case in the original Newton system[4, 5]. With this class, some difficulty arises from the introduction of objects containing edges and vertices, since vertex/face, vertex/edge, edge/edge, edge/face, and face/face contacts have to be taken into account. However, because the topology of the parameterized primitives can be known a priori, the objects do not have to be represented explicitly in a boundary representation. The various conditions for determining the contacts can be enumerated from the relative positions and orientations.

The hardest approach is to allow arbitrary polyhedral solids. Such solids are readily obtained from any solid modeler capable of synthesizing complex boundaries, for example by regularized set operations [9, 6]. Due to the complex topology, the boundaries of the objects have to be represented explicitly to allow local analysis of the boundary.

In this paper, the objects are solids with boundaries consisting of planar faces. They are considered to be rigid objects with no large-scale deformation occurring during collisions. The positions and orientations of the objects are assumed to be specified by a simulation system, and the time steps are assumed to be sufficiently small so that only small interpenetration occurs in

relation to the size of the objects.

We are concerned with the problem of detecting and analyzing the spatial relationship between two objects. Given that a simulation system specifies the positions and orientations of two objects, we wish to report whether the two solids are separate, interpenetrate, or touch. If they touch we wish to specify in addition the points of contact and the tangent planes of each contact point. If they penetrate, the simulation system converges to the proper time of collision by successively dividing the time interval during which the collision took place. This can be done either by always dividing the time interval in half, or by first estimating the collision time and dividing the time interval as appropriate. Here, we also wish to provide an estimate of the collision time.

In Section 2 we introduce a new data structure called the B-rep index which allows an efficient collision detection algorithm. Section 3 covers the collision detection algorithm. The contact analysis algorithm is presented in Section 4, and a discussion of our methods ends this paper in Section 5.

2 B-rep Index

Already during a fairly simple simulation, pairs of solids are checked for contact perhaps thousands of times. Given that a pair of solids is known to touch, determining what parts of it do and how, needs to be done efficiently and robustly. Although the boundary representations for each solid are readily available for analysis, finding the points of contact is a difficult undertaking. What is missing is a volume-based access to the topological entities (i.e., the edges, vertices and faces) of the boundary representation. Given a point or a line in space, it should be easy and precise to determine where that point or the line lies in relation to the solid. For this reason, we introduce an index for a boundary representation that provides a volume-based access to its entities.

A *B-rep index*, $T(s)$, for solid s, is a volume-based, ternary tree data structure that provides access to the topological entities of a boundary-based data structures. Each node n in the tree represents a non-empty, open region $R(n)$ of E^3. Each internal node n references a plane $P(n)$ that intersects the region $R(n)$. The three subtrees of n represent the subregions of region $R(n)$ lying *above* $P(n)$, *on* $P(n)$, and *below* $P(n)$ respectively, where above is by convention in the direction of the plane's normal. Thus a region $R(n)$ is defined by the intersection of the planes and half-spaces on the path from the root down to node n.

If $R(n)$ is a d-dimensional region, then the regions above and below $P(n)$ are also d-dimensional, but the region on $P(n)$ is $(d-1)$-dimensional. Considering the path from the root to a node n that represents a k-dimensional region, for $0 \leq k \leq 3$, the path contains exactly $d-k$ nodes with dimension one less than their parents.

An index is a geometric structure with its planes uniquely specifying the various topological entities of the B-rep. A vertex is defined as the intersection of three mutually intersecting planes. A point is coincident with the vertex when it lies on each of the three planes. When a vertex is adjacent to more than three faces, and therefore more than three planes, then three of the planes are chosen that maximize the absolute value of the 3-by-3 determinant formed by the unit normals of the planes, ties broken arbitrarily. An edge is defined by four planes, two given by the two adjacent faces of the edges, and two transversal planes passing through each of the two vertices of

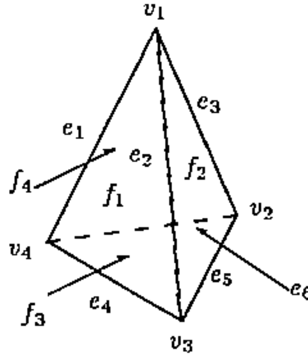

Figure 1: A tetrahedron with labeled vertices, edges, and faces.

the edge. In the case that the two adjacent faces of the edge are coplanar, the plane of the faces and an auxiliary plane perpendicular to it is used. A point is on the edge if it lies on the first two planes, and is contained between the other two planes. A convex face is defined by the plane containing the face, and one plane for each coface of the bordering edges. Thus a point is in the face if it lies on the first plane and is contained below all the other planes. The planes defining the vertices, edges, and faces of a solid can be organized in a tree structure for which a point lying on any entity follows a single path from the root to that entity.

Given a boundary representation, the B-rep index is created in three phases as follows:

1. Create an index tree T by processing first all the faces, then the edges, and finally the vertices. At this point, it is not yet possible to refer back from the tree nodes to the corresponding areas on the solid's boundary in the B-rep. During this step, all the concave faces of the boundary representation are partitioned into convex regions.

2. Insert all the face, edge and vertex nodes of the B-rep data structure into T, thus attaching the index to the boundary representation.

3. Topologically reduce the partitioned faces by merging adjacent collinear edges and adjacent coplanar faces. Afterwards, update the B-rep index with the topological changes.

Figure 1 shows a tetrahedron with its vertices, edges and faces labeled. Figure 2 shows the B-rep index for the tetrahedron. For convex objects, such as this tetrahedron, there is only a single region inside the solid at the end of the rightmost path, and the left branches (i.e., above P_i) of all nodes lead to regions outside the solid. This is the direct result of only choosing partitioning planes that contain faces of the solid, which is called autopartitioning [7]. For nonconvex objects autopartitions yield trees with more than one inside regions.

2.1 Creating the Index Tree

We now describe the three steps of creating the B-rep index in greater detail. Specifically, we now explain how to create the index tree, a process requiring three phases.

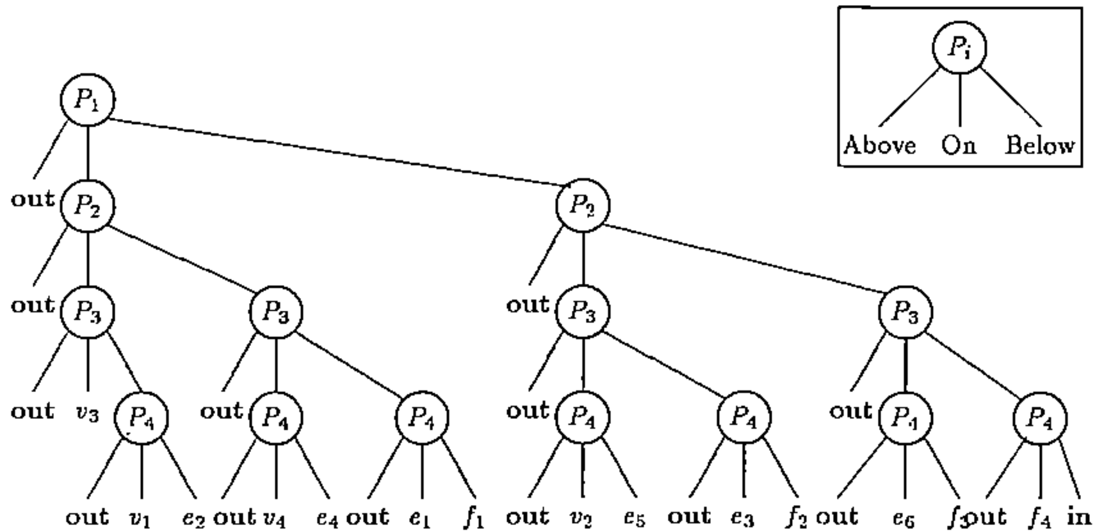

Figure 2: The B-rep index for the tetrahedron shown above. Here P_i correspond to the planes of face f_i.

In the first phase we are given a set F of faces bounding the solid. F is processed recursively as follows:

1. If F is a singleton then create an interior node with no descendants. The left leaf is labeled INSIDE, the middle leaf is labeled ON, and the right leaf is labeled OUTSIDE. If the face is not convex, it is split into convex regions. Skip all remaining steps.
2. If F is not a singleton, select a face f in F, and create an interior node n with the left, middle, and right descendants referred to as Above(n), On(n), and Below(n) respectively. Label the middle descendent as ON. Also label n with the oriented plane P in which f lies.
3. Remove from F the face f and all other faces coplanar to f. If any of the removed faces are not convex, they are split into convex regions.
4. Of the remaining faces, split any faces that cross P into regions lying entierly above and below P, and partition the faces into two sets F_a and F_b according to whether they are above or below P.
5. Recursively process the face sets F_a and F_b resulting in the subtrees Above(n) and Below(n), respectively.

In the second phase of Step 1, this tree is refined by processing all the edges of the solid. The processing of the faces in Step 1 partitioned all concave faces into convex faces, and this necessarily added some new (i.e., pseudo) edges and vertices. These have to be processed as well.

With E, the set of edges, and T, the tree created in the previous phase, start at the root of T, and process E recursively as follows:

1. If E is empty then skip Step 2.

2. Let n be the internal node currently visited. Split the edges of E by plane $P(n)$ and partition the resulting edges into E_a, E_o, and E_b. Process E_a recursively in the subtree Above(n), and E_b in Below(n). If E_o is non-empty, then create and assign a new subtree On(n) as follows:

   (a) If E_o is a singleton then create an interior node with the left, middle, and right children labeled as INSIDE, ON, and OUTSIDE respectively. Skip all remaining steps.

   (b) For a nonsingleton E_o, select an edge e in E_o and create an interior node m.

   (c) Assign $P(m)$ the plane that contains e and that is perpendicular to $P(n)$. Orient $P(m)$ so that the face adjacent to e lying in $P(n)$ lies below $P(m)$.

   (d) Delete from E_o all edges lying on $P(m)$ (i.e., collinear with e).

   (e) Split all remaining edges of E_o by $P(m)$ and partition the resulting edges into the edges according to which ones are above or below $P(m)$. Call the sets E_a and E_b respectively.

   (f) Recursively process (i.e, from Step (a)) the edge sets E_a and E_b resulting in the subtrees Above(m) and Below(m) respectively.

In the third phase of Step 1, the vertices are processed recursively in a similar fashion to the edges. When vertices fall on a 1-dimensional region a third plane is chosen to separate the vertices. The third plane is taken to be perpendicular to the line which is the 1-dimensional region. For brevity, the details of this third phase are omitted.

2.2 Attaching the Index to the B-rep

We have now the index and the B-rep from which it was created. The two data structures are now interconnected in Step 2 of the algorithm by inserting every topological entity of the B-rep as a leaf in the proper place in the index. This is done as follows. For every entity x (i.e., vertex, edge or face), traverse the index tree from the root down to a leaf, and replace the leaf with x. Refer to Figure 2 for an example. As a result of inserting the entities, all 2-dimensional leaf regions labeled INSIDE or ON get assigned a face, all 1-dimensional leaf regions labeled INSIDE or ON get assigned an edge, and all 0-dimensional regions labeled ON get assigned a vertex.

2.3 Reducing the B-rep and the Index

The index creation phase necessarily fragmented the faces of the original B-rep into convex regions by introducing pseudo edges and pseudo vertices. Step 3 of the algorithm recreates the maximally connected faces of the original B-rep by removing all pseudo entities while preserving topological consistency. The reduction step proceeds by first removing all pseudo edges and then removing all pseudo vertices. Pseudo edges are removed and their adjacent coplanar faces are merged into a single face. Pseudo vertices are removed and their adjacent collinear edges are merged. This topological reduction is a simpler problem than a general topological reduction algorithm of [11].

The reduction of the B-rep does not effect the B-rep index. This leaves pointers in the index to topological entities that are no longer a part of the B-rep. Consequently, the index has to updated. Pseudo entities remaining in the index have to be replaced by the corresponding entities that replaced them in the B-Rep. To do this update, a single pass through all the leaves of the index suffices.

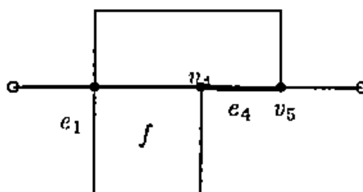

Figure 3: Line segment/solid classification example.

2.4 Point and Line Segment Classification

With the aid of the B-rep index it is possible to easily classify a point or a line segment according to its spatial relationship with a solid. Let s be the solid, and $T(s)$ its B-rep index. The classification of a point p in relation to s determines whether the point is inside, or outside s, or, if it lies on the boundary of s, it determines the vertex, edge or face it lies on. The classification of a line segment bounded by points (p, q) yields a partitioning of the segment into points and segments that have a uniform classification in relation to s.

As an example of line segment classification, consider a line segment lying partially on the face f of another solid as shown in Figure 3. The classification of the line segment results in the sequence

$$[\textbf{out}, (e_1, q), f, (v_4, p_4), e_4, (v_5, p_5), \textbf{out}].$$

From left to right as seen in the figure, the line segment start outside the solid, crosses edge e_1 at point q, enters face f, then passes vertex v_4 at point p_4, and so on.

Prior classification, the point or the line segment is checked against the box extent of s to determine its possible position outside s. Only when the point or any part of the line segment lie inside the extent are they checked against the index. The checking is performed by the aid of point/solid and line/solid classification routines described next.

Point/Solid Classification A point p is classified by traversing the B-rep index of s starting from the root and returning the leaf, which is either a topological entity of s, or the flags INSIDE or OUTSIDE. The traversal proceeds at each node n by checking p against the oriented plane $P(n)$ and proceeding down the appropriate subtree of n. The decision of which way to proceed is based on the distance of p from $P(n)$. A tolerance $\epsilon > 0$ is used to determine when p is on $P(n)$. Here, ϵ is the minimum separation distance below which collisions occur.

Line/Solid Classification The edge, specified by its endpoints (p, q) is classified by recursively traversing the B-rep index of s. For each leaf node reached, the topological entity or the inside/outside label is simply returned. At each internal node n visited, the points p and q are checked against the oriented plane $P(n)$. If the segment lies completely above, on, or below $P(n)$, the classification proceeds down the appropriate subtree. If, however, the segment

Number of V/E/F	Index Nodes	Index Height	Average Height	Time (sec)
Sphere				
58	58	16	7.8	1.7
134	135	19	9.4	4.5
242	245	18	9.9	7.7
382	463	18	10.4	15.6
1562	2309	24	13.0	104.0
Torus				
64	110	9	6.1	1.9
144	299	11	7.3	8.0
256	435	14	8.6	12.5
400	1019	19	10.2	39.0
1600	4721	27	12.6	233.0

Table 1: A sphere and a torus with various number of faces.

(p,q) crosses $P(n)$, it is split into (p,r), r, and (r,q), r is classified using the point/solid classification using n as the root of the tree, and the two segments (p,r), and (r,q) are classified recursively. The results of all three subtrees are then combined, compressed and returned.

Each of the three results is a sequence of point or line segment classifications. After combining the three results, adjacent classifications that are the same are compressed into one. For example, the result

$$[\ldots, \mathbf{out}, \mathbf{out}, (e, p'), \ldots]$$

is compressed to

$$[\ldots, \mathbf{out}, (e, p'), \ldots].$$

2.5 Examples and Timings

The B-rep-index creation algorithm has been implemented in Common Lisp. The code was added to ProtoSolid [10], a solid modeler integrated with the Newton simulation system. The entire simulation system runs on two Symbolics 3620 Lisp machines, and uses an Personal Iris workstation for its user-interface and movie record/playback facility.

To illustrate the sizes of the B-rep-index trees, spheres and torii with various number of faces have been created. Table 1 shows the number of vertices, edges and faces, the number of internal nodes in the index, the height of the index, the average height of the index, and the time in seconds to create the index.

The average height of the tree is important to achieve efficiency, because the average classification cost of a point with respect to a solid is linear in the average height of the B-rep index.

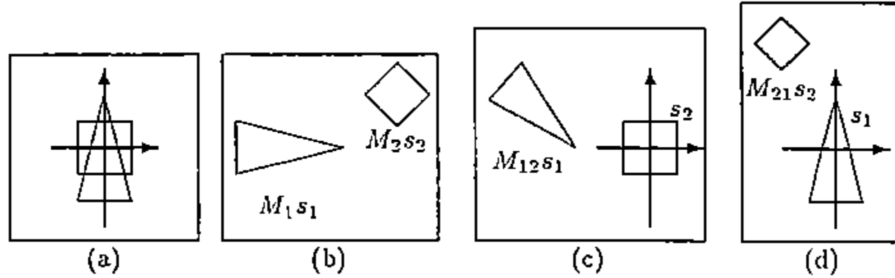

Figure 4: a) Solids s_1 and s_2 with respect to local coordinates. b) s_1 and s_2 placed with respect the global frame of reference. c) s_1 mapped into the local s_2 frame of reference. d) s_2 mapped into the local s_1 frame of reference.

Thus, the classification of n points by a B-rep index of average height H requires $O(nH)$ steps. As an example, 100 points can be classified with respect to a torus with 200 faces (i.e., 1600 total entities) on average using only 1260 dot products and comparisons total. The cost of a line segment classification depends on the number of internal nodes that split the line segment, and is somewhat higher.

3 Detecting Collisions

Consider two solids s_1 and s_2 and their positions and orientations at some time. Solids are created such that their center of mass is at the origin of their local coordinate space. The position of solid s_i with respect to a global frame of reference is specified by the vector t_i, and the orientation by a 3-by-3 rotation matrix in a 4-by-4 matrix R_i. The position and the orientation yields a 4-by-4 transformation matrix M_i that maps the solid from its local frame to the global reference frame, via

$$M_i = R_i \cdot T_i,$$

where T_i is the translational matrix given by t_i. Solid s_i can be mapped into the local configuration space of solid s_j by a transformation given by

$$M_{ij} = M_i \cdot M_j^{-1} = R_j^{\mathrm{T}} \cdot R_i \cdot T_{ij},$$

with T_{ij} being the translation $t_i - t_j$. (See the illustration in Figure 4).

With the two M_{ij} matrices, it is possible to check the vertices and edges of $M_{ij} \cdot s_i$ against the B-rep index of s_j. Using the point/solid and the line/solid classification routines, the contact points between s_1 and s_2 are obtained. The vertices, and then the edges of s_1 are checked against s_2. Similarly, the vertices, and then the edges of s_2 are checked against s_1.

For the purpose of correctly assessing the physical consequence of a collision, the Newton simulation system requires the following topological and geometrical information for each point of contact between the solids s_1 and s_2:

x_1 the vertex, edge, or face of s_1 in contact with s_2.

x_2 the vertex, edge, or face of s_2 in contact with s_1.

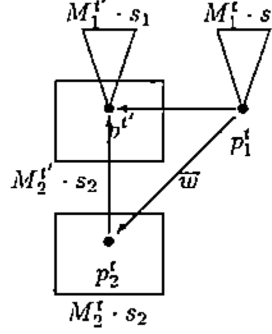

Figure 5: Solids s_1 and s_2 in global reference frame at times t and t'. Here $\overline{w}$ is the relative velocity vector at point $p^{t'}$ for solid s_2.

p_2 the point of contact with respect to the local coordinate frame of s_2.

The points of contact between the two solids are found and retained. Since each contact point may be reported twice with slightly different points, only one is retained. This occurs because each solid is checked against the other. If during the checking of the vertices and the edges a penetration is detected, the detection terminates, and the solids are reported to penetrate. After the checking, if no contact points are detected and no penetration occurred, the solids are reported as separate.

Consider now that the two solids penetrate each other. This means that the time step was too large, and a smaller time step must be found. Providing only the information that the solids either penetrate, touch, or are separate, the simulation system must converge to the proper time by successively halving the time interval. This approach is used by others, namely [1]. Depending on the tolerance ϵ, and on the complexity of the solids, this can be a time consuming process. The collision detection algorithm can improve the performance greatly by providing an estimate for the time of collision.

Let t be the time of the previous frame, and let t' be the current time with the detected penetration. Furthermore, let M_i^t be the translation matrix that maps s_i to the global reference frame at time t. Figure 5 shows the positions of the solids at t and t'. If $p^{t'}$ is some point in the global space and on the inside of solid s_2 at time t', the relative velocity of solid s_1 in relation to s_2 at point $p^{t'}$ is

$$\overline{w} = p_2^t - p_1^t,$$

where $p_i^t = M_i^t \cdot (M_i^{t'})^{-1} p_i$. To eliminate the penetrating point of s_1 from s_2, the point has to be pulled out of s_2 in the direction $-\overline{w}$. How far depends on the boundary of s_2 in that direction. The distance can be determined by classifying a line segment in the local frame of s_2 between the points $a = (M_2^t)^{-1} \cdot p_2^t$ and $b = (M_2^t)^{-1} \cdot p_1^t$. Since a is known to be inside, and b is known to be outside s_2, the classification must produce some interval of length d, for $0 < d < |a - b|$ that lies inside s_2. Therefore, the time for which point a was on the boundary of s_2 is

$$t + \frac{d(t' - t)}{|a - b|}.$$

This estimate assumes that the time step $t' - t$ is small enough so that the true motion of point p

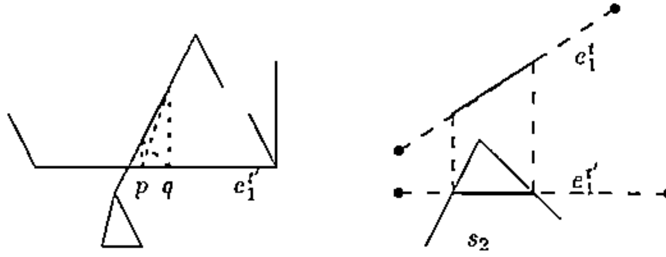

Figure 6: Edge e_1^t outside solid s_2 at time t penetrates the solid at time t' through points p and q.

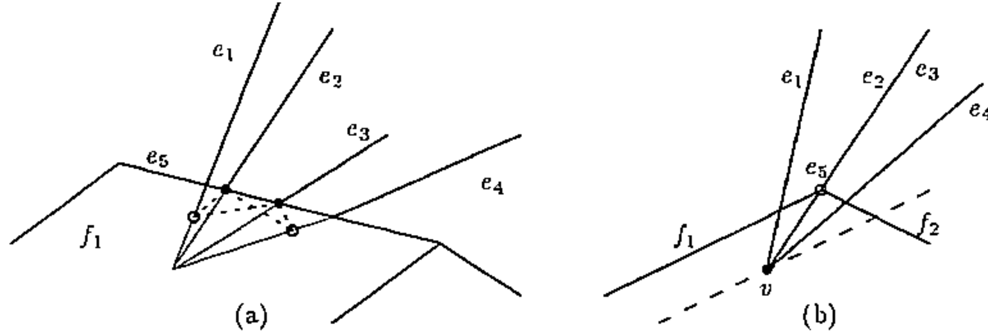

Figure 7: a) Vertex v penetrates face f_1 to a depth of ϵ, at a distance 2ϵ from the edge e_5. b) A side view of this configuration. The dashed line marks ϵ distance from f_1.

is nearly linear.

Now consider an edge segment of s_1 penetrating s_2, as shown in Figure 6. The position of the penetrating edge can be linearly interpolated between the time interval t and t'. Given β, as $0 < \beta < 1$, the edges position at time $t + \beta(t' - t)$ can be derived and checked against s_2. A recursive binary search on β can yield the time at which the edges touches the boundary of s_2.

The estimated time of a collision is thus computed by successively deriving the time that each penetrating vertex and edge of either solid clears the other solid, and taking the smallest such time.

4 Contact Analysis

When contact points are found and no penetration occurs, the contact points are analyzed and for each point a plane of tangency and its normal direction is determined. However, as the next example illustrates, not all contact points should be reported to Newton. Consider a case in which the apex of a cone penetrates another solid by less than the assumed tolerance ϵ, as shown in Figure 7. Had the penetration of the cone been perpendicular to one of the faces, then only the vertex/face contact point would have been found. Since, however, the cone penetrated at a large incline, six contact points have been found, namely

$$(v/f_1, p_1), (e_1/e_5, p_2), (e_2/e_5, p_3), (e_3/e_5, p4), (e_4/e_5, p_5), (v/e_5, p_6).$$

These points can be distinguished from other contact points in that any two are less than 4ϵ apart.

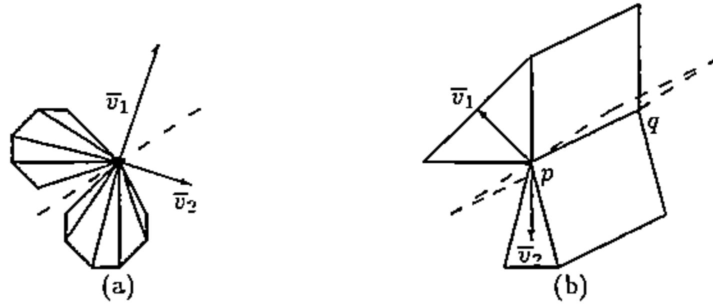

Figure 8: Finding the tangent plane (dashed lines) for the vertex/vertex and the collinear edge/edge cases.

We call such a set of contact points a *cluster*. We first partition all contact points into clusters, and then reduce each cluster to a single contact point. To accommodate the partitioning, we impose a mininum distance of α on the separation of nonincident topological entities of solids, where typically $\alpha = 100\epsilon$. Thus, for example, edges must be at least α in length, and vertices cannot be closer than α to other faces. This is a reasonable restriction that allows us to determine the clusters without having to separate several fused clusters.

Properly reducing a cluster to a single contact point is still an open problem. In a cluster, the entities of one solid are all connected through topological adjacency. It is therefore possible to determine the patches on the boundaries of the two solids covered by the cluster and identify the main topological entity on each. For instance, the example given in Figure 7 is a vertex/edge contact (i.e., vertex v touches edge e_5). It is also possible to compute the relative velocity at the center of the cluster and separate the solids by the minimum distance which would reduce the cluster to a single contact point. It is however just as easy and appropriate to assign each type of contact a priority and simply choose the contact with the highest priority. We have adopted the latter approach in our implementation.

The priority of the contact point in a cluster is based on the ability to compute the tangent plane at that point. As such, a vertex/face (also face/vertex) contact point has the highest priority. The others, in highest to lowest order are an edge/edge, an edge/vertex (also vertex/edge) and a vertex/vertex contact points. Thus, as an example, the cluster of Figure 7 is reduced to a vertex/face contact point (i.e., $(v/f_1, p_1)$).

Once a single contact point is determined for each cluster, a tangent plane $\overline{\pi}$ must be determined. The tangent plane's normal vector corresponds to the impulse vector needed by Newton to evaluate the collision. As a convention, $\overline{\pi}$ is computed with respect to the local frame of s_2, and points towards the inside of s_2 at the contact point. The different cases are now listed given that x/y means that x is on s_1 and y is on s_2:

Vertex/Face The vector $\overline{\pi}$ is the complement of the normal vector of the face since the normal is pointing away from s_2.

Face/Vertex The vector $\overline{\pi}$ is the normal vector of the face after it is mapped into s_2's object space, using R_1 and R_2.

Edge/Edge Given edges e_1 and e_2, e_1 is mapped into s_2's object space as e'_1, and the angle between e'_1 and e_2 is computed. If the edges are transversal (i.e, $|e'_1 \cdot e_2| < 1$), $\overline{n}$ is either $e'_1 \times e_2$ or $e_2 \times e'_1$ depending on which one points towards s_2. If the edges are collinear, the average of the invectors of each solid yield $\overline{v}_1$ and $\overline{v}_2$ (see Figure 8(b)). Using the two endpoints p and q of the edge, the tangent plane is obtained which passes through the three points $(p, q, p + \overline{v}_1 + \overline{v}_2)$.

Vertex/Vertex This case is similar to the collinear edge/edge case. For each vertex, average out the normal vectors of the incident faces, yielding vectors $\overline{v}_1$ and $\overline{v}_2$ (see Figure 8(a)). From these two vectors, compute the tangent plane that passes through the three points

$$(p, p + \overline{v}_1 \times \overline{v}_2, p + \overline{v}_1 + \overline{v}_2).$$

Edge/Vertex, Vertex/Edge Similarly to the above indeterminate cases, a vector is obtained from each solid, and the tangent plane is found that contains the edge and the average of the two obtained vectors.

The edge/face, face/edge, and face/face contacts are ignored. They are implicitly handled by their bordering vertex/x and x/vertex contacts.

5 Discussion

The idea for the B-rep-index data structure grew out of the work on grid generation [12] using binary space partition (BSP) trees [3, 8]. The first implementation of the collision detection was based on BSP trees. Although it was possible to detect points of contact and even compute the tangent planes, the method was not robust because the BSP tree could not resolve clusters. Moreover, the approach could not accommodate temporal coherence, that is, the ability to track contact points over time. The ability to identify the topological entities at the contact points was missing.

The B-rep Index was indirectly inspired by the cut-trees of Dobkin and Edelsbrunner [2]. It was straight forward to extend the BSP tree to the index and attach it to the B-rep.

The B-rep index has been successfully implemented and is currently used in the Newton system. However, several problems remain. The first problem concerns the choice of splitting planes. Choosing the faces (and therefore the planes), or the edges, in different order results in different trees, possibly with different average heights. Various heuristics have been used for binary space partition trees to reduce the height of the trees [8]. In our implementation, we use a hybrid method that uses regular decomposition when a region contains a large number of faces, and an autopartition decomposition described here when the region contains only a few faces. Empirical evidence shows that this hybrid method reduces the height of the tree greatly.

The second problem concerns the reduction of the index size. The creation of the index fragments the boundary representation. Since the boundary now contains unnecessary pseudo entities, it is reduced to its minimal form. The index, however, does not reduce in size. Ways of reducing the index need to be investigated.

Acknowledgements

The implementation of the collision detection code and the integration of Newton and ProtoSolid has been made possible by the equal efforts of Bill Bouma whose programming efforts support the Newton project at Purdue. In addition to Bill, Jim Cramer at Cornell has cooperated closely with us and offered many helpful suggestions on the physics of collisions and the internals of Newton.

References

[1] D. Baraff. Analytical methods for dynamic simulation of non-penetrating rigid bodies. *Computer Graphics (Proc. of SIGGRAPH '88)*, 23(3):223–231, July 1989.

[2] D. P. Dobkin and H. Edelsbrunner. Space searching for intersecting objects. *Journal of Algorithms*, 8:348–361, 1987.

[3] H. Fuchs, Z. M. Kedem, and B. F. Naylor. On visible surface generation by a priori tree structures. *Conf. Proc. of SIGGRAPH '80*, 14(3):124–133, July 1980.

[4] C. M. Hoffmann and J. E. Hopcroft. Simulation of physical systems from geometric models. *IEEE Journal of Robotics and Automation*, RA–3(3):194–206, June 1987.

[5] C. M. Hoffmann and J. E. Hopcroft. Model generation and modification for dynamic systems from geometric data. *CAD Based Programming for Sensory Robots*, F50:481–492, 1988.

[6] M. Karasick. *On the Representation and Manipulation of Rigid Solids.* PhD thesis, McGill University, 1988.

[7] M. S. Paterson and F. F. Yao. Binary partitions with applications to hidden-surface removal and solid modeling. In *Proceedings of ACM on Computational Geometry*, pages 23–32. ACM, 1989.

[8] W. C. Thibault and B. F. Naylor. Set operations on polyhedra using binary space partitioning trees. *ACM Computer Graphics SIGGRAPH '87*, 21(4):153–162, July 1987.

[9] G. Vaněček Jr. Obtaining boundaries with respect: A simple approach to performing set operations on polyhedra. CAPO Report CER-89-25, Purdue University, Department of Computer Science, West Lafayette, IN 47907, November 1989.

[10] G. Vaněček Jr. Protosolid: An inside look. CAPO Report CER-89-26, Purdue University, Department of Computer Science, West Lafayette, IN 47907, November 1989.

[11] G. Vaněček Jr. Building solid models from polygonal data. CAPO Report CER-90-20, Purdue University, Department of Computer Science, West Lafayette, IN 47907, May 1990.

[12] G. Vaněček Jr. Towards automatic grid generation using binary space partitions trees. CAPO Report CER-90-6, Purdue University, Department of Computer Science, West Lafayette, IN 47907, January 1990.