BucketTree: Improving Collision Detection Between Deformable Objects

Fabio Ganovelli [*]
John Dingliana [†]
Carol O'Sullivan [‡]

Abstract

In recent years, thanks to the increasing computational power available, real time computer animation has naturally evolved to model more complex and computationally expensive scenes. Consequently, all the problems concerning physical modelling need further research to tackle these new requirements, especially the problem of collision detection for deformable objects. Most existing solutions cannot not be trivially extended, because they are strongly based on the assumption that the shape of the object is fixed. In this paper we propose a general approach to reduce the cost of collision detection between deformable objects explicitly represented, regardless of the specific geometrical and physical manner in which they are modelled.

Keywords: Deformable Objects, Collision Detection, Hierarchical Models, Real-time Animation

1 Introduction

Virtual Reality (VR) applications allow users to enter a computer-generated virtual world and interact with graphical objects and virtual agents with a sense of reality. Such systems may be either immersive, or desktop based. One thing they have in common is a requirement for extremely high and constant frame-rates. Physical interactions such as touching, hitting and throwing are usually triggered by collision. The more objects in the environment, and the more complex these objects are, the higher the burden on the engine that powers the animation, and hence the greater the need for extremely rapid collision detection. Increasing performance of V.R. is often driven by hardware developments, as in [28], but significant opportunities exist to increase performance via algorithmic improvement. In this paper, we present a simple technique for improving the progressive refinement level of collision detection between deformable objects. A hierarchical model, such as an octree, is often used to localize collision testing to certain regions in space, thus reducing the number

of costly calculations needed to identify interfering object parts. We associate an octree of axis aligned bounding boxes with each object, where the primitives composing the surface of the object reside in the leaves of the tree. We refer to the bounding boxes associated with the leaves of the tree as *buckets*. Our method provides a means of keeping this hierarchy efficiently and dynamically updated for deformable objects.

The rest of the paper proceeds as follows: an overview of previous approaches is presented in Section 2, where the concept of hybrid collision detection is explored. Section 3 presents our approach,*BucketTree*, and in Section 4, we briefly compare it with similar approaches. Results and plans for future work are discussed in Section 5 and Section 6 respectively.

2 Collision Detection

Hybrid collision detection [17] refers to any collision detection method which first performs approximate tests to identify interfering objects in the entire workspace, and then performs more accurate tests to identify the object parts causing interference. [12] and [4] also propose hybrid algorithms for collision detection. The former refers to the two levels of the algorithm as the *broad phase*, where approximate intersections are detected, and the *narrow phase*, where exact collision detection is performed. Such an approach is essential for acceptable collision detection performance. The narrow phase itself may also consist of several levels of intersection testing between two objects at increasing levels of accuracy, the last of which may be fully accurate. We shall refer to these as the *exact level*, and the *progressive refinement levels*.

2.1 Broad Phase Collision Detection

When animating more than two objects, the most obvious problem that arises is the $O(N^2)$ problem of detecting collisions between all N objects. The aim of the broad-phase of a collision detection algorithm is to quickly eliminate objects which could not possibly be intersecting. 4-dimensional structures called space-time bounds have been used in [12] and [2] to provide a conservative estimate of where an object may be in the future. The fourth

[*] Visual Computing Group IEI-CNR Pisa ganovell@iei.pi.cnr.it
[†] Image Synthesis Group TCD Dublin John.Dingliana@cs.tcd.ie
[‡] Image Synthesis Group TCD Dublin Carol.Osullivan@cs.tcd.ie

dimension represents time. Overlaps of these bounds trigger the narrow phase. Using the space-time bounds, attention is focused on the objects that are likely to collide, and those far away can be ignored.

In [4] multiple object pairs are "pruned" using bounding boxes. Overlapping bounding boxes then trigger the narrow phase of the algorithm. Their "Sweep and Prune" algorithm orthogonally projects axis-aligned bounding boxes of all objects onto the x, y and z-axes. This results in intervals, of which overlaps in all three dimensions indicate overlaps of the corresponding bounding boxes. Because of coherence, the relative positions of objects will not change significantly between frames, so insertion sort is used to keep the interval lists sorted, which runs in almost linear time for almost-sorted lists. The algorithm has been built into a general collision detection package I-Collide, which is freely available on the World Wide Web.

2.2 Narrow Phase: The Exact Level

Any collision detection algorithm depends on the technique used to model the objects, and the data structure used to represent that model. The narrow phase, where exact collision detection is performed, depends greatly on the object representation scheme used. Much of the previous work on collision detection techniques has concentrated on detecting collisions between convex polytopes. Such approaches fall into two broad categories: Feature-based methods, and Simplex-based methods. Feature-based methods concentrate on the inter-relations between the vertices, edges and faces of two polytopes, i.e. their features. The main goal of such algorithms is to detect whether two polytopes are touching or not. The most significant of these algorithms are the Lin-Canny algorithm [20, 19], and the V-Clip (Voronoi-Clip) feature-based algorithm [22] which it inspired.

Simplex-based algorithms treat a polytope as the convex hull of a point set. Operations are then performed on simplices defined by subsets of these points. The first of such algorithms was presented in [16] and is commonly referred to as GJK. The main strength of this algorithm is that, in addition to detecting whether two objects have collided or not, it can also return a measure of interpenetration. [27] improved upon GJK by exploiting coherence, and [3] developed it further to produce the algorithm which is known as Enhanced GJK.

The above algorithms may be extended to cater for non-convex polytopes by using hierarchies of convex components. Although this technique works well for slightly non-convex objects, it becomes very inefficient as the level of non-convexity increases. Therefore, these techniques are very useful for situations where a small number of convex, or slightly non-convex objects are interacting in real-time, but in other situations techniques based on hierarchical representations are much more suitable. The algorithms also depend on the object being rigid, and are hence

unsuitable for collision detection between deformable objects.

Implicit Surfaces, where a scalar field function is used to define the shape of an object [1, 14, 34], are commonly used to model objects which deform, split or blend. Seeds may be used to produce a set of polygons that fit the surface at run time [5]. A method of producing piecewise contact allows collision detection and its subsequent response [7]. However, the polygonisation methods used are still too slow for true real-time performance, and other problems arise when handling volume preservation upon collision, and unwanted blending.

Deformable objects are often modelled solidly as a soup of solid primitives, such as tetrahedra. This approach is particularly prevalent in the field of virtual surgery. Collision detection and response for such objects is usually with a surgical instrument, such as a scalpel, which typically has a trivial topology [21]. No satisfactory real-time techniques exist for fast collision detection between arbitrarily complex deformable objects modeled in this way. In this paper, we concentrate on the problem of collision detection between the surfaces of objects explicitly represented by solid primitives.

2.3 Narrow Phase: Progressive Refinement Levels

The progressive refinement levels of the narrow phase of a collision detection algorithm are often based on using bounding volumes and spatial decomposition techniques in a hierarchical manner. Hierarchical methods have the advantage that as a result of simple tests at a given point in the object hierarchies, branches below a particular node can be identified as irrelevant to the current search and so pruned from the search.

Trees of bounding volumes are used, each level approximating the object. This is a form of Level Of Detail (LOD) representation of the object. This differs from the polygonal levels of detail used in multiresolution methods for faster rendering of complex objects, or surfaces such as mountainous terrain [9, 10, 11, 29]. In such techniques, the aim is to render an approximation that is as visually similar to the original model as possible. LODs for collision detection are always conservative approximations to the object, and the choice of volume is usually based on the speed of their intersection tests. More recently emphasis has been placed on their ability to approximate the geometry of the bounded object. The following hierarchies have been used:

- AABB-trees [6]. Axis Aligned Bounding Boxes are used, the advantage of these being their ease of computation and overlap testing.

- Octrees [32, 17] Octrees are built by recursively sub-dividing the volume containing an object into

eight octants, and retaining only those octants that contain some part of the original object as nodes in the tree. Such a data structure is simple to produce automatically, and lends itself to efficient and elegant recursive algorithms. The disadvantage of this approach is that each level of the hierarchy does not fit the underlying object very tightly.

- Sphere Trees [13, 25, 26]. The main advantages of using spheres are that they are rotationally invariant, making them very fast to update, and it is very simple to test for distances between them, and test for overlaps. The disadvantage is that spheres do not approximate certain types of objects very efficiently. Hubbard attempts to improve upon this by building first a medial axis surface, which is like a skeleton representation of an object, and then placing the spheres upon this to provide a tighter-fitting approximation to the object. [24] also developed a method of tightly fitting spheres to an object. In [15], the nature of the sphere tree is exploited to gracefully degrade collision handling in time-critical animation systems.

- C-trees consist of a mixture of convex polyhedra and spheres [35]. This has the advantage of choosing primitives which best approximate the enclosed object, but a major drawback is that the hierarchy must be created by hand, and cannot be produced automatically. A similar approach is taken in [28].

- OBB-trees [8]. These hierarchies consist of tightly fitting Oriented Bounding Boxes. It is claimed that using an algorithm based on a separating axis, it can accurately detect all the contacts between large complex geometries at interactive rates. However, it is admitted that other methods are very good at performing fast rejection tests, and a disadvantage of OBB-trees over Sphere trees is that they are slower to update. A similar approach is taken in [18], who use hierarchies of k-DOPs, or discrete orientation polytopes, which are convex polytopes whose facets are determined by half spaces whose outward normals come from a small fixed set of k orientations. Again, they implement it with a small number of highly complex objects, for the purposes of haptic force-feedback. If there are a large number of objects between which fast rejection or acceptance is needed, the update time needed for these approximations is likely to add an unacceptable additional burden.

- ShellTrees [30]. These trees consist of oriented bounding boxes and spherical shells, which enclose curved surfaces such as Bezier patches and NURBS.

For rigid bodies, these data-structures may be pre-computed, as their shape will not change throughout the animation. The same transformations that are applied to the object can simply be applied to the hierarchy whenever a broad-phase collision is detected. With deformable objects, the hierarchical approximation must be updated at each frame, due to the constantly changing shape and/or topology of the objects. This is a computationally expensive process.

3 Octree and bucket strategy

As previously stated, most hierarchical approaches for collision detection between rigid bodies use structures, for example hierarchies of bounding volumes, that have to be computed in a preprocessing phase and that are strictly connected with the shape of the objects, which is a constant of the system.

Generally speaking, the quality of a strategy that uses bounding volumes is influenced by two factors:

- how well the bounding volume approximates the object

- how much computation is required to detect the overlapping between two bounding volumes

Note that in the context of highly deformable objects, the first item is almost meaningless, since the kind of bounding volume which best approximates an object could be the worst after a few iterations. Consequently, we decide to use bounding volumes that are the easiest ones to check for overlapping: Axis Aligned Bounding Boxes.

For the sake of generality, we do not make any assumption as to how the objects are physically and geometrically modeled. We simply think of an object as a soup of *primitives* freely moving in the scene. Primitives could be, for example, vertices of polygons or the polygons themselves; the only requirement is that a total order among primitives can be defined in the x, y and z directions.

Basically, we use an *octree* where the root is associated with the axis aligned bounding box of the object, and where each leaf contains the set of primitives which are inside the corresponding box (we use the term *bucket* to refer to the box associated with a leaf). At each time step the coordinates of the bounding box are updated and each primitive is placed in the right bucket. The collision test between two objects is done recursively testing pairs of nodes. When two non-leaf nodes overlap, the children of the one with smaller volume is tested with the node with bigger volume; if only one is a leaf, it is tested against the children of the non-leaf node; if two leaf nodes overlap, the two sets of primitives to test for exact collision are in the respective buckets. In the latter case, the rest of the collision process depends on how the objects are modeled. Remember, our intent is to reduce the number of tests between surface elements of the object regardless

of their representation, i.e. to improve the broad phase of the collision detection process. If the object's surface is a triangular mesh, for example, this algorithm returns the triangles of both surfaces that have to be tested.

3.1 Keeping each primitive in the correct bucket

An octree with l levels contains 8^l leaves each one corresponding to a bucket. If the dimensions of the bounding box are S_x, S_y and S_z, then the dimensions of a bucket are $s_x = \frac{S_x}{2^l}$, $s_y = \frac{S_y}{2^l}$ and $s_z = \frac{S_z}{2^l}$, and the buckets are indexed with a triple $(n_x, n_y, n_z) : 0 \le n_x, n_y, n_z < 2^l$. Consequently, if p_x is the x position of a primitive in space and B_x is the minimum x position of the bounding box, the index of the correct bucket for this primitive is:

$$n_x(p) = (p_x - B_x)/s_x \qquad (1)$$

Clearly the same holds for $n_y(p)$ and $n_z(p)$. We propose two different ways of assigning each primitive to the correct bucket, which we term **Algorithm 1** and **Algorithm 2** respectively:

Algorithm 1 : This is the brute force solution: at each step during the simulation, we process each primitive and assign it to the appropriate bucket. The cost for each assignment is given by the operations required by equation 1: if m is the number of primitives, each step requires $3m$ subtractions, $3m$ divisions and $3m$ floor operations.
Please note that no information on the positions of primitives at any previous step is used by this algorithm.

Algorithm 2: If we assume *frame to frame coherency*[1], we can take advantage of the fact that most of the primitives stay in the same bucket between two consecutive steps, thus avoiding the use of equation 1 to compute the bucket position.
We use three arrays X, Y and Z, each one storing all the primitives, and keep these arrays ordered on the respective coordinates using the *insertion sort* algorithm [31], which runs in average $O(n)$ time when frame to frame coherency is guaranteed. It is obvious that:

$$\begin{aligned} if \quad & n_x(X[i]) = n_x(X[j]) = b, \ i < j \\ then \quad & n_x(X[u]) = b \ \forall i < u < j \end{aligned} \qquad (2)$$

$$\begin{aligned} if \quad & n_x(X[i]) = b \ and \ n_x(X[i+1]) \ne b \\ then \quad & n_x(X[i+1]) = b+1 \end{aligned} \qquad (3)$$

equation (2) shows that each list can be subdivided into a series of intervals so that all the primitives in the same interval have equal n_x; equation (3) shows that we can record

---

[1]Frame to frame coherency means that the order of the primitives along the x, y and z axes at time t is almost valid also at time $t + \Delta t$
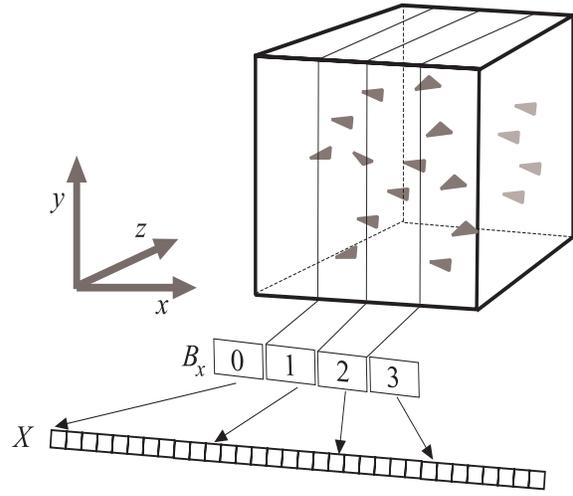

Figure 1: A representation of the data structure for a two levels octree regarding to x direction

the position of such intervals with an integer index of their first element. Hence we can use an array of integers B_x with size 2^l, where $B_x[k]$ contains the position of the first primitive p in X for which $n_x(p) = k$ (see Figure 1). At each step, we only update the array B_x in average constant time with the following procedure:

```
UpdateBucketPointers(x)
for i = 0..2^l
  pos := B_x[i];
  threshold := i * bucket_size.x;
  if( X[pos].x > threshold )
  then
   do
    pos := pos - 1;
   while( (pos >= 0) and ( X[pos].x > threshold ))
  else
   do
    pos := pos + 1
   while( (pos < 2^l - 1) and (X[pos]).x <= threshold)
  B_x[i] := pos;
```

This procedure simply checks if the position $B_x[i]$ still points to the first primitive (the one with minimum x) in the bucket i. If this is not true, it moves $B_x[i]$ backwards or forwards depending on whether the value of the primitive in $X[B_x[i]]$ is greater or smaller than the current *threshold*, i.e. the minimum x value of the bucket.

Clearly, Y and Z are similarly handled.

3.2 Propagating cell occupancy information

Since each node of the octree is statically associated with a portion of the space occupied by the bounding

box of the object, it may happen that some subtrees are empty. To avoid visiting empty subtrees during the collision detection process, we propagate the information about cell occupancy, starting from the leaves up to the root of the octree. A temporal mark tm_i is associated with each node i of the octree as follows: at the end of each step, with the buckets properly updated, we set $tm_i = simulationstepnumber$ only for the ones containing at least one primitive. Starting from these buckets, we propagate the temporal mark from the bottom up. Obviously, the mark of the nodes corresponding to empty subtrees will not be updated. Therefore, when we visit node i, if tm_i is equal to the current simulation step number, then the subtree with its root in i is not empty.

4 Comparison with other approaches

We have seen that there are few approaches suitable for collision detection between deformable objects in real time. Up to now, the most effective has probably been the one based on AABB trees (cited in section 2.2). This approach, like our one, uses a hierarchy of bounding boxes: it proceeds recursively, starting from the bounding box of the whole object and partitioning the primitives into two sets, separated by a plane orthogonal to the longest side of the bounding box: for each of the two sets the corresponding bounding box is computed.

All the bounding boxes in the hierarchy are axis aligned with respect to the local object coordinate system, which coincides with the global coordinate system at the beginning of the simulation. This means that when the object is undergoing a rotation, the boxes are no longer aligned with the global axis. Hence, the overlapping tests are related to boxes freely oriented in space. When an object is undergoing a deformation, the boxes are resized to fit the primitives contained.

This approach is very efficient under two conditions: firstly, the number of primitives has to be known a priori as well as the topology of the object. This means that no cut or fusion operations can be done without rebuilding the whole tree. Secondly, the behaviour of the object has to be expressed in terms of a rigid body and a deformation component, as proposed in [33], because the refitting procedure is based on deformation expressed in the local coordinate system.
In our approach, no information about topology is required, which makes it suitable for cut and fusion operations; the overlapping between boxes requires only 6 comparisons, because they are always aligned with the axes of the global coordinate system.

On the other hand, the drawback of our approach is that the primitives are not uniformly distributed among the leaves of the tree, or the nodes at the same level. For ex-

ample, if an object is composed of m primitives, where $m - 1$ are very close to each other and 1 is far away, we could have $m - 1$ primitives concentrated in a single bucket, which makes the whole approach useless.

5 Results

Our intention was to make as few assumptions as possible about the geometry and topology of the objects, or about the way in which their physical behavior is modeled. We therefore test the algorithm by simply modeling an object as a *soup* of primitives. At each iteration, a main direction and velocity for the object is randomly chosen and stored in a vector v (the velocity is $|v|$). Each primitive p of the object moves in a direction, v_p *close* to v, i.e. such that the angle formed by v and v_p is less than a given value α (see Figure 2). The choice of α controls the degree of frame to frame coherency: setting $\alpha = 0$, all the primitives move in the same direction (rigid body motion), hence their order does not change along any axis between two consecutive steps; increasing α we give them more freedom to move.

Table 3 shows a comparison between the time required to keep the octree updated for Algorithm **1** (compute the right bucket for each primitive) and Algorithm **2** (use insertion sort) with or without frame to frame coherency. Note that Algorithm **1** is not affected by frame to frame coherency, simply because it does not use any order relation between primitives, while Algorithm **2** works better when frame to frame coherency is introduced. Furthermore, though less intuitive, it exhibits almost the same performance than **Algorithm 1** also without up until to 15000 primiteves. This is due to the fact that, as previously stated, it keeps updated the structure without perform any multiplication or division to place the primitives in the correct bucket.

Figure 4 shows the influences of the number of octree levels on the performance of the algorithm. Observe that up until 4 levels (4096 buckets), the time for updating the structure grows linearly, because the time for processing the buckets is almost negligible w.r.t. the time for processing primitives. From the fifth level upwards, the exponential factor due to the octree significantly degrades performance. Note that the number of operations depends linearly on the number of primitives and exponentially on the number of levels.

Table 1 shows the times for collision detection, i.e. the time needed to visit the octrees and to determine pairs of overlapping leaves. We do not differentiate between algorithms **1** and **2** since the detection process is identical for both. All the tests are performed on a Pentium II 300Mhz - 64Mb.
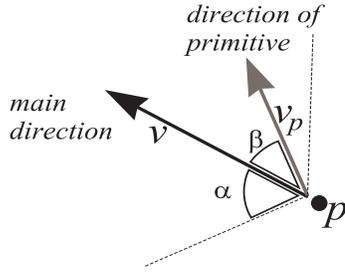
Figure 2: The angle β, formed by the direction of primitive *p* and the main direction of the object, is limited by α, to ensure frame to frame coherency
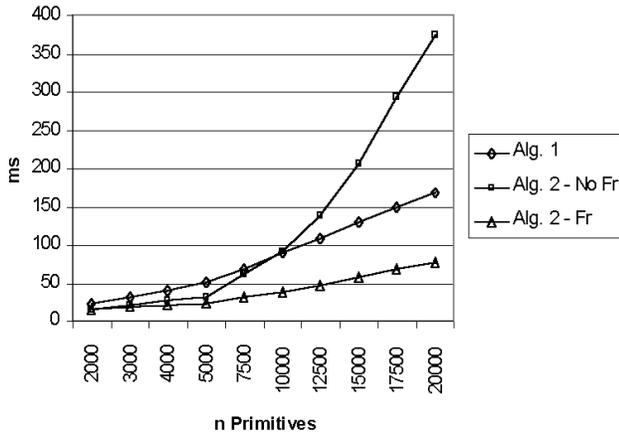

Figure 3: Comparison between algorithms 1 and 2 with or without frame to frame coherency. The number of levels of the octree is set to 4
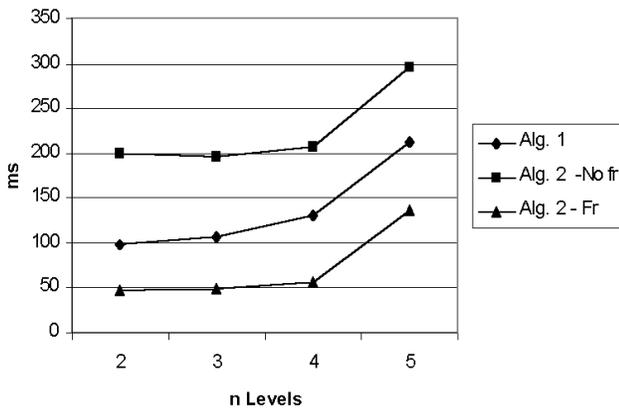

Figure 4: Degradation of performance on increasing of the number of levels. The number of particles per object is set to 15000

n L	t
3	95
4	325
5	684

Table 1: **n L**: number of levels; *t* time in milliseconds for detecting 1000 collision between buckets
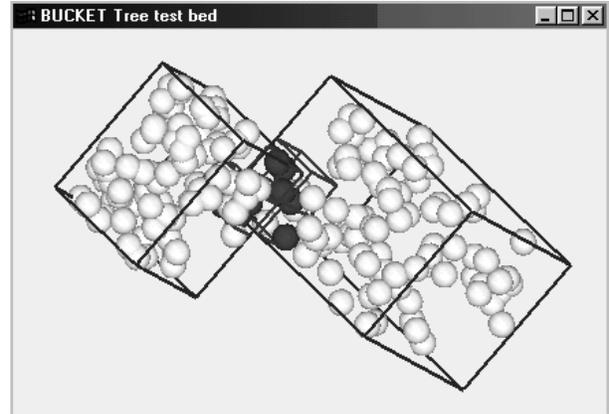

Figure 5: A snapshot of the algorithm test bed while running. The primitives in the two buckets colliding are rendered in black. The bounding boxes of the two objects are showed. Please note that the boxes are axis aligned, while the camera is moved for the sake of visibility

6 Conclusions and Future Work

This paper presents a simple and effective technique for the progressive refinement phase of collision detection between deformable objects explicitly represented. Since we did not make any assumptions about the model, the approach is very general and can therefore be adopted in a variety of situations. To witness its adaptability, we are current developing a C++ template library where the template is just the object class *class_primitive*, implementing both **Algorithm 1** and **2**. Further research is needed to extend this approach to the problem of self intersection of the surface, that always arises for highly deformable objects or for objects that can be cut. The latter case needs special attention, since a cut in an object gives rise to two adjacent and opposite surfaces. We plan to extend BucketTree to cope with such situations, using the information about vicinity that we have when two primitives reside in the same bucket, and to introduce it into our model for deformable objects [23].

References

[1] J. Blinn. A generalization of algebraic surface drawing. 1(3):235–256, 1982.

[2] S. Cameron. Collision detection by four–dimensional intersection testing. *IEEE Transaction on Robotics and Automation*, 6(3):291–302, June 1990.

[3] S.A. Cameron. Enhancing GJK: Computing minimum penetration distances between convex polyhedra. pages 3112–3117, 1997.

[4] J. D. Cohen, M. C. Lin, D. Manocha, and M. K. Ponamgi. I-COLLIDE: An interactive and exact collision detection system for large-scale environments. In Pat Hanrahan and Jim Winget, editors, *1995 Symposium on Interactive 3D Graphics*, pages 189–196. ACM SIGGRAPH, April 1995. ISBN 0-89791-736-7.

[5] Mathieu Desbrun, Nicolas Tsingos, and Marie-Paule Gascuel. Adaptive sampling of implicit surfaces for interactive modeling and animation. In *Implicit Surfaces '95*, April 1995.

[6] Van Den Bergen G. Efficient collision detection of complex deformable models using aabb trees. *Journal of Graphics Tools*, 2(4):1–13, 1998.

[7] M.-P. Gascuel. An implicit formulation for precise contact modeling between flexible solids. *Computer Graphics (SIGGRAPH '93 Proceedings)*, 27:313–320, 1993.

[8] Stefan Gottschalk, Ming Lin, and Dinesh Manocha. OBB-Tree: A hierarchical structure for rapid interference detection. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 171–180. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[9] H. Hoppe. Progressive meshes. In *ACM Computer Graphics Proc., Annual Conference Series, (Sigraph '96)*, pages 99–108, 1996.

[10] Hugues Hoppe. View-dependent refinement of progressive meshes. In *ACM Computer Graphics Proc., Annual Conference Series, (Siggraph '97)*, 1997. 189-198.

[11] Hugues Hoppe. Efficient implementation of progressive meshes. *Computer & Graphics*, 22(1):27–36, 1998.

[12] P. M. Hubbard. Collision detection for interactive graphics applications. *IEEE Transactions on Visualization and Computer Graphics*, 1(3):218–230, September 1995. ISSN 1077-2626.

[13] Philip M. Hubbard. Approximating polyhedra with spheres for time-critical collision detection. *ACM Transactions on Graphics*, 15(3):179–210, July 1996.

[14] Nishimura H Hirai M Kawai T Kawata T Shirakawa I and Omura K. Object modeling by distribution function and a method of image generation. volume 68, pages 718–725, 1995.

[15] Dingliana J. and O'Sullivan C. Graceful degradation of collision handling in physically based animation. *Computer Graphics Forum(Eurographics 2000 Proceedings) (to appear)*.

[16] E.G.Gilbert D.W.Johnson S.S. Keerthy. A fast procedure for computing the distance between complex objects in three-dimensional space. 4(2):193–203, 1988.

[17] Kitamura Y. Takemura H. Ahuja N. Kishino. Efficient collision detection among objects in arbitrary motion using multiple shape representations. volume 1, pages 390–396, 1994.

[18] James T. Klosowski, Joseph S. B. Mitchell, Henry Sowizral, and Karel Zikan. Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):21–36, January 1998.

[19] M. C. Lin. *Efficient Collision Detection for Animation and Robotics*. PhD thesis, University of California, Berkeley, March 1994.

[20] M.C. Canny J.F. Lin. Efficient algorithms for incremental distance computation. pages 1008–1014.

[21] J.C. Lombardo, M.P.Gascuel, and F.Neyret. Realtime collision detection for virtual surgery. In *Proceedings of Computer Animation '99*, pages 33–39, May 1999.

[22] Brian Mirtich. VClip: Fast and robust polyhedral collision detection. *ACM Transactions on Graphics*, 17(3):177–208, July 1998.

[23] F. Ganovelli P. Cignoni C. Montani and R. Scopigno. A multiresolution model for soft objects supporting interactive cuts and lacerations. *Computer Graphics Forum(Eurographics 2000 Proceedings) (to appear)*.

[24] J. O'Rourke and N. Badler. Decomposition of three-dimensional objects into spheres. *IEEE Trans. On Pattern Analysis and Machine Intelligence*, PAM-1:295–305, 417, July 1979.

[25] R.L. Palmer, I.J. Grimsdale. Collision detection for animation using sphere-trees. volume 14, pages 105–116, 1995.

[26] S. Quinlan. Efficient distance computation between non-convex objects. In Edna Straub and Regina Spencer Sipple, editors, *Proceedings of the International Conference on Robotics and Automation. Volume 4*, pages 3324–3330, Los Alamitos, CA, USA, May 1994. IEEE Computer Society Press.

[27] Rich Rabbitz. Fast collision detection of moving convex polyhedra. In Paul Heckbert, editor, *Graphics Gems IV*, pages 83–109. Academic Press, Boston, 1994.

[28] J. Rohlf and J. Helman. IRIS performer: A high performance multiprocessing toolkit for real-time 3D graphics. *Proceedings of SIGGRAPH'94*, page 381, 1994.

[29] J. Rossignac and P. Borrel. Multi-resolution 3D approximation for rendering complex scenes. In B. Falcidieno and T.L. Kunii, editors, *Geometric Modeling in Computer Graphics*, pages 455–465. Springer Verlag, 1993.

[30] M. Lin S. Krishnan, A. Pattekar and D. Manocha. Spherical shell: A higher order bounding volume for fast proximity queries. In David C. Evans and Rusell J. Athay, editors, *WAFR Proceedings*, pages 287–296, March 1998.

[31] H. Samet. *The design and Analysis of Spatial Data Structures*. Addison Wesley, Reading, MA, 1990.

[32] H. Sammet and R. Webber. Hierarchical data structures and algorithms for computer graphics. 8(3):48–68, 1988.

[33] Demetri Terzopoulos and Kurt Fleischer. Deformable models. *The Visual Computer*, 4(6):306–331, December 1988.

[34] Geoff Wyvill, Craig McPheeters, and Brian Wyvill. Data structure for soft objects. *The Visual Computer*, 2(4):227–234, 1986.

[35] K. Youn, J.H. Wohn. Realtime collision detection for virtual reality applications. pages 18–22, 1993.