© Springer-Verlag London Ltd
Virtual Reality (2000) 5:8-22

VoxColliDe: Voxel Collision Detection for Virtual Environments

S. M. Lock, D. P. M. Wills

Department of Computer Science, University of Hull, UK

Abstract: Collision detection is fundamental in achieving *natural* dynamics in virtual environments, but current algorithms are too slow, causing a major bottleneck in processing and hindering the building of interactive simulation environments. This paper provides an overview of the collision detection problem and current attempted solutions. A voxel-based approach to rigid-body collision detection is presented, with its potential high performance explained.

Voxel collision detection takes place on a pair-wise basis, involving two additional representations of a polygonal object, a Voxmap and a Point Shell. These are constructed in a pre-processing step and allow fast collision detection through a simple look-up reference of points into voxels. Collision performance depends upon the number of points in the shell, and can trade accuracy for speed. A range of *pruning* techniques, needed to cut down the number of objects undergoing collision testing, are reviewed and implemented. These allow most effective use of the voxel collision detection algorithm in multi-body simulations, such as virtual environments.

Performance evaluations demonstrate the voxel collision detection algorithm's ability to achieve interactive rates (above 20 Hz) for both high precision pair-wise collision tests, and for large numbers of objects in multi-body environments. The voxel collision detection algorithm is suitable for parallel, hardware implementation. This provides the potential for great enhancements to already extremely high performance, rendering the voxel-based approach to collision detection all the more promising.

Keywords: Collision detection; Presence; Virtual environments; Voxel

1. Introduction

Collision detection is crucial in the development of realistic, interactive simulation environments. *Natural* dynamics, based on the properties of real objects and real world physics, are needed to produce simulations that convince users of their *presence* in environments. Unfortunately, most current collision detection algorithms are too computationally expensive, resulting in slow and inconsistent frame rates, with an adverse effect on feelings of presence and user performance.

This paper presents an approach to determining collisions based upon *voxel collision detection* [1,2], that uses pre-processed representations of polygonal objects (polytopes) in order to fundamentally simplify the collision detection process. This results in high and consistent detection rates, appropriate for interactive virtual environments containing many complex objects. This method has been successfully applied since 1996 for detecting collisions in our Virtual Environments Knee Arthroscopy Training System [3].

Section 2 presents the need for collision detection in order to produce natural dynamics in virtual environments. The collision detection problem is explained in Section 3, with attempted solutions from the literature reviewed in Section 4. The voxel-based approach to collision detection is explained in Section 5, including descriptions of additional representations required, the collision detection algorithm itself and important performance factors. Section 6 reviews pruning techniques needed in multi-body simulation environments to make most effective use of the collision detection algorithm, and describes the four techniques implemented. The simulation environment used for evaluation is described in Section 7, and performance evaluation results presented and discussed in Section 8. Section 9 finishes with some conclusions and future work.

2. Natural Dynamics and Collision Detection in Virtual Environments

Virtual environments promise great potential as a new communication medium for human-computer interaction, with a broad range of application areas [4]. In order to achieve this potential, the *presence* [5,6], the sense of *being* that users' feel through their interactions with the environment and its perceived realism, must be maximised.

Many existing virtual environments suffer because of a limited ability to interact with objects in the environment, and a lack of realistic behaviour exhibited by objects. Such failures of the environment to exhibit natural dynamics, compatible with users' cognitive models [7–9] of the world in which they are immersed, decreases the ability of the user to suspend disbelief during interaction, and decreases performance.

A virtual environment may be populated by objects that users can only look at, not pick up and manipulate, or that do not behave in physically satisfying ways; a ball released from a virtual hand does not fall and bounce around the environment naturally. These examples highlight two characteristics of objects which we take for granted in the real world, but must be built in to virtual worlds – *solidness* and the ability to *interact* with things. Solidness is a central characteristic of real objects, and must be enforced in virtual environments, preventing objects penetrating one another, and rendering the simulation more natural and believable. Also, user interaction with objects requires detection of contacts between the user and objects. Collision detection algorithms are required to enforce both of these properties.

Collision detection must also be achieved at real-time rates, requiring high and steady frame rates, with low response latency [4,10]. Efficiency is critical in virtual environments, otherwise their interactive nature is lost [11], reducing presence, with the potential of giving users a feeling of motion sickness [12,13], and decreasing their performance [14].

A fast and accurate collision detection algorithm is therefore a fundamental component of any complex, interactive virtual environment. Unfortunately, such efficient algorithms are rare at present, with collision detection causing a major bottleneck in processing, and hindering the development of interactive virtual environments.

3. The Collision Detection Problem

Collision detection has been studied extensively in robotics and computational geometry, but none of these algorithms is adequate for virtual environments [15]. Robotics algorithms have concentrated on collision-free path planning between obstacles, with object motion expressed as a closed form function of time [15]. This is inappropriate for virtual environments because objects' motion is generally constrained both by the dynamics of the system and by external forces originated from users. Computational geometry has focused on theoretically efficient intersection detection algorithms, mostly restricted to static problems, which are too complex and slow to produce real-time performance. Researchers in the field of virtual environments have therefore been required to dramatically increase the efficiency of existing interference algorithms, or to develop efficient algorithms from scratch.

Collision detection in a multi-body environment typically proceeds in two phases, a *narrow phase* and a *broad phase* [16,17]. The narrow phase involves a *pair-processing* collision detection algorithm, which determines whether two objects are interpenetrating and reports all collision points. The broad phase involves cutting down, or *pruning*, the number of pairs of objects that must undergo pair-processing collision testing. This is needed to avoid the potential quadratic rise in processing cost, O(n^2), as the number of dynamic objects, n, in the environment increases.

The *N-body* collision detection problem of virtual environments is thus heavily dependent upon a very efficient two-body (or *pair-processing*) collision detection algorithm [18] which must provide high and almost constant detection rates, even when an application's geometric characteristics change [17]. Algorithms should have good scalability, and their accuracy should be appropriate for the application and users' perceptual requirements.

Polygonal, rigid body models are currently the most common geometric representation used for interactive computer graphics, and are also used by most (polygon-polygon) collision detection algorithms. A basic polygonal collision detection algorithm, such as that proposed by Moore and Wilhelms [19], is very computationally expensive and therefore slow. This is because many special cases exist, requiring complex processing. Many authors have attempted to improve collision detection performance through a variety of techniques.

4. Solutions: Improvements and Alternative Representations

The main techniques used by authors to improve upon a basic polygonal collision detection algorithm are back-face culling, spatial decomposition hierarchies, and multiple levels-of-detail and closest feature tracking.

Attempted Improvements

Back-face culling [20] decreases the number of polygons undergoing collision testing for a pair of objects, by eliminating polygons whose face normals are in the opposite direction to the direction of object motion, defined by one or more relative-velocity vectors.

Spatial decomposition approaches use a hierarchical, tree representation of polygonal objects, allowing localisation of collision areas before performing collision testing. Palmer and Grimsdale [21] use an octree-based sphere tree, localising the collision area by testing bounding spheres at each level in the tree, finally using polygon-polygon collision testing to determine any points of contact. Hubbard [17] uses a tighter-fitting sphere tree based on the medial-axis surface of polygonal objects. This more optimal sphere fitting requires a large amount of pre-processing time, but reduces the number of levels required in the tree, achieving more efficient localisation, and thus collision testing. Hubbard [16,17,22,23] adopts a *time-critical* approach to collision detection that trades accuracy for speed by approximating collision location using the tree spheres, rather than polygons, traversing the tree to the deepest possible level in the time available. Others have used oriented bounding box (OBB) trees [24,25] rather than bounding spheres, for localisation.

An alternative approach to spatial decomposition, which achieves effective localisation of contacts, is through the use of representations with multiple levels-of-detail [18,20,26]. Vanecek and colleages aim to produce a boundary-based representation that inherently provides access to local detail, but with a face-retrieval access complexity independent of the overall boundary complexity. This is an attempt to avoid the problem of spatial decomposition hierarchies, which work well for lower complexity objects but fail to be efficient for complex objects, due to retaining a look-up complexity that is a function of the overall boundary complexity. Complex objects have deeper trees, and an associated increase in localisation time. Vanecek's proposed solution [18, 20, 26] involves the use of Locally Resolvable B-reps (LRB-reps), multi-resolution geometric representations (or *wrappers*), and extended BSP tree data structures. The BSP tree addresses the spatial sorting required to localise areas of interest quickly, with the LRB-reps addressing the complexity of the boundary detail. Lazy evaluation is used to postpone or completely eliminate the unnecessary detail. This solution, directly aimed at the N-body collision detection problem, shows an alternative and promising approach to spatial decomposition.

A number of collision detection algorithms have been developed by researchers at the University of North Carolina (UNC), based around a fast distance computation algorithm for convex polytopes by Lin and Canny [27]. The algorithm is used to detect the exact contact state between a pair of closest features, with the Euclidean distance between them calculated in order to detect collisions. This approach is well suited to dynamic environments, as the algorithm can exploit inter-frame coherence, with the closest features changing only infrequently between objects. The closest feature tracking approach requires pre-computation of Voronoi regions in order to find closest feature pairs. The closest feature-pair tracking algorithm is used by I-COLLIDE [15] with an extension for non-convex objects, by Lin and Manocha [28,29] who extend it for use with convex and non-convex curved objects, and by Ponamgi et al. [30] who use an octree-based hierarchy for contact detection between concave objects. The closest feature-pair tracking algorithm is currently one of the fastest polygonal collision detection methods.

An Alternative Geometric Representation for Collision Detection

The attempted improvements discussed in the previous subsection all use boundary descriptions of objects (B-reps or LRB-reps), in the form of polygonal models or hierarchical/multiple levels-of-detail descriptions thereof, in order to represent objects for collision detection. All techniques mentioned, with the notable exception of closest feature-pair tracking, make use of polygonal (polygon-polygon) collision detection algorithms at some stage of collision processing. A large amount of time and effort has been spent devising often quite ingenious methods of cutting down the amount of work to be done by the polygonal collision algorithm, generally through efficient localisation of collision areas, in order to squeeze reasonable interactive collision update rates from the algorithm. The fact remains that the underlying polygonal collision detection algorithm is inefficient, and thus inappropriate for virtual environment applications which demand high update rates for complex objects.

The underlying polygonal representation appears to be inappropriate for efficient collision testing, leading to the inevitable question: do alternative object representations exist that are fundamentally better suited for efficient collision testing, and the requirements of interactive N-body simulations? The present authors believe such an alternative representation does exist, allowing a simple collision detection algorithm to be implemented that supports fast, consistent and accurate collision detection. The remainder of this paper describes and explains the voxel-based approach to collision detection.

5. Voxel-Based Collision Detection

The voxel-based approach to collision detection exploits the current trend of the increasing relative cheapness of memory compared to processor performance. Rather than placing a heavy burden on processing demand, as is the case with polygonal collision detection algorithms, the voxel-based approach uses alternative, pre-processed representations of polygonal objects. These allow very fast and simple contact point determination, but with the additional memory demand required to store the representations.

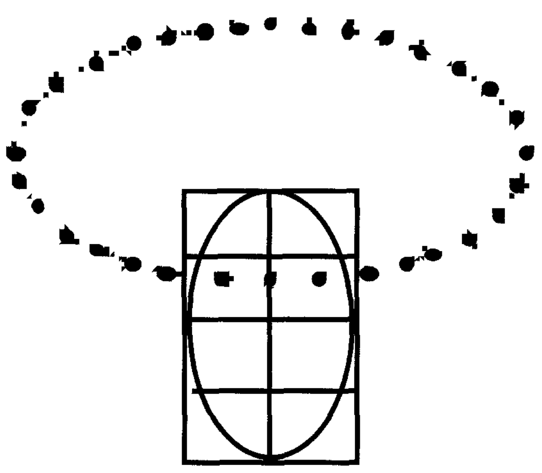


Fig. 1. A two-dimensional diagram of voxel collision detection.

The voxel collision detection algorithm developed uses two additional representations of objects, a Voxel map (or Voxmap) – a volume representation of the object consisting of *filled* or *empty* voxels – and a Point Shell – a surface representation of the object consisting of an even distribution of surface points. Objects are free to move around the environment, with both their Voxmap and Point Shell representations rotated and translated along with them. Collision detection proceeds on a pair-wise basis, using the Point Shell of one object to perform a look-up of points into the Voxmap voxels of the second object. A collision point is detected when a Shell point lies inside a *filled* voxel. Figure 1 shows a two-dimensional example of a voxel collision.

The Voxmap

The Voxmap is the name given to the volume representation of an object. It is a three-dimensional, regular array of cubic voxels, sometimes known as *cuberilles* [31]. Voxmaps are constructed for individual objects rather than entire environments, and allow the object interior and surface to be distinguished from the surrounding empty and uninteresting volume. This is done by assigning every voxel as *filled* or *empty*, depending on whether it is

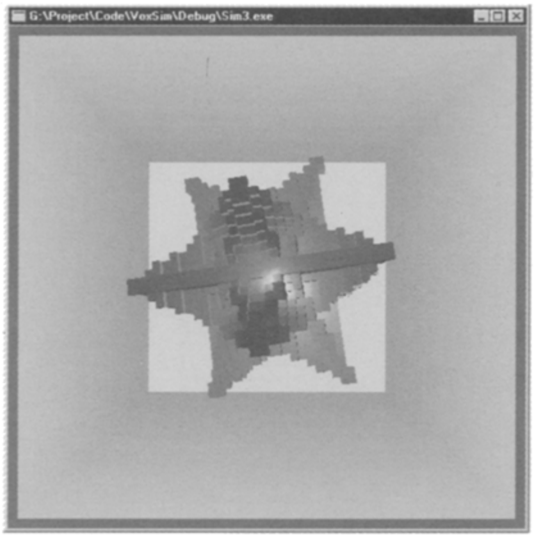
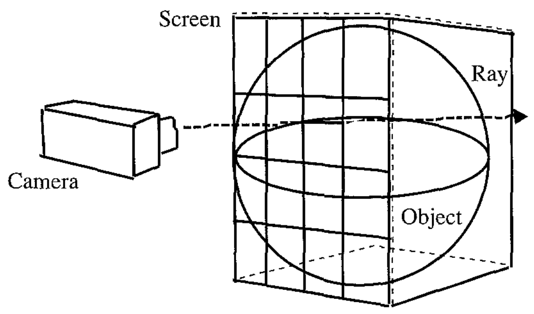


Fig. 2. The Voxmap of a stelated dodecahedron.

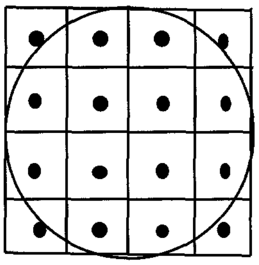
inside or outside the object. A Voxmap for a concave object is shown in Fig. 2.

Two voxelisation strategies were considered, 3D scan-conversion followed by seed-fill, or a ray casting approach. Three-dimensional scan-conversion algorithms are developments of 2-D scan-conversion algorithms used to convert polygon (floating-point) data to pixel (integer) data for 2D graphical display. The 3D algorithms [32] determine the state of every voxel inside the three-dimensional space of the tightest fitting bounding-box around the polygonal object, divided up at the desired granularity of the Voxmap. The surface voxels in this three-dimensional space are identified and filled, producing a surface representation, followed by the application of a seed-fill algorithm to allocate the remainder of the voxels as either filled or empty. Although this scan-conversion strategy works well, it is less easily extended to handle alternative input geometry representations, and is more conceptually complex than the ray casting strategy implemented.

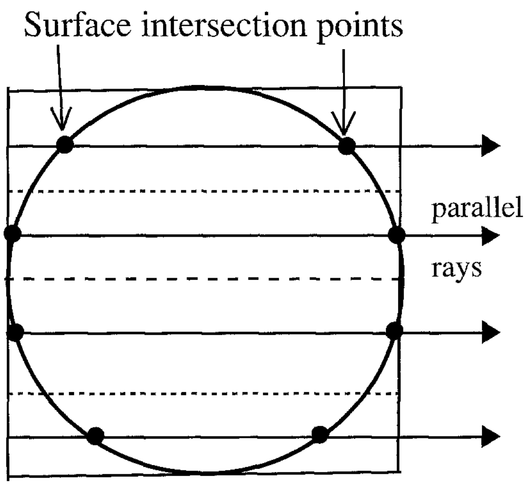
The ray casting strategy of voxelisation developed is simple and intuitive, although the authors can find no mention of this approach in the literature. It is a straightforward development of the graphics ray casting technique [33], or single shot ray tracing [34], that uses an orthogonal projection and calculates all surface intersection points along every ray, but omits any lighting/shading model. As in a ray caster [33,34], rays are cast through every pixel of a virtual *screen*, but in this case the screen is the two-dimensional surface of one side of the tightest-fitting axis-aligned bounding box around the polygonal object. The screen is divided at the desired resolution of the Voxmap, and rays are cast in parallel through each pixel (also parallel to the x, y, or z-axis), calculating, sorting and storing the values of all intersections along a ray (see Fig. 3). Pairs of intersection points along each ray represent the transition from outside-to-inside and inside-to-outside the object, with a guaranteed even number



a) Three-dimensional view of ray casting Voxeliser.



b) Front view of rays passing through centre of screen pixels.



c) Side view showing ray intersection points with the sphere.

Fig. 3. Ray casting voxeliser.

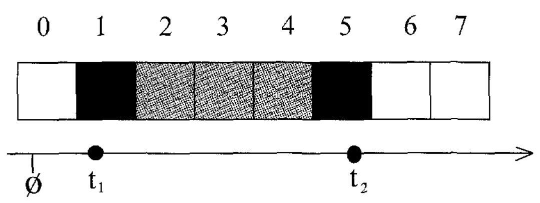


Fig. 4. The third-dimension of the Voxmap is added by filling between pairs of surface voxels. t_1 and t_2 are a pair of surface intersection points. Surface voxels are black, and the other filled voxels are shaded grey.

of intersection points along each ray. This is achieved by ray-polygon intersection being detected only if a ray passes inside a triangle, and not through its edges or vertices. The Voxmap is constructed by converting all pairs of intersection points along each ray from their floating-point values to their integer, surface voxel values. The third dimension is then added to the Voxmap by filling all voxels between and including the surface voxel pairs, as seen in Fig. 4. The Voxmap is stored as a three-dimensional array of Boolean variables, requiring only two bits of memory to record the status of each voxel.

The Voxeliser uses raw triangle format (RAW) as the input format for polygonal objects. When used in conjunction with a three-dimensional file format translator, such as Crossroads 3D [35], this allows a wide range of file formats (such as 3DStudio (3DS), AutoCAD (DXF), and VRML V1.0 (WRL)) to be converted to raw triangle format, making the Voxeliser as general as possible.

The Voxeliser allows both convex and concave objects to be voxelised. This means that concave objects may be represented for collision detection without the need for a hierarchical decomposition into convex parts, with its associated increase in computational expense for both pre-processing and collision testing.

The ray casting strategy is conceptually simple and has the advantage that the Voxeliser may be readily extended to accept alternative object representation types, such as algebraic surfaces (e.g. spheres, cones, cylinders, torii, etc.). Quite simply, any representation for which ray-surface intersections can be determined may be voxelised using this approach.

The Point Shell

The Point Shell is the name given to the second, surface representation of an object. It consists of a number of points evenly distributed over the object's surface, and is used to perform a look-up into the Voxmap of another object during collision testing. The accuracy of the voxel collision detection algorithm is dependent on the fact that no voxel is allowed to completely penetrate the Point Shell of an object without detection. If such penetration were possible, then the accuracy of the algorithm would be rendered indeterminate. The points in the Shell must be distributed so that in a worst-case scenario, undetected penetration is still impossible. This could be achieved by using a very high density of evenly distributed surface points. Unfortunately, the performance of the collision detection algorithm is inversely proportional to the number of points in the Shell. The optimisation problem posed is to find the smallest number of points distributed over a surface that prevent a voxel of a given size and any orientation from completely penetrating the surface without touching any points. The distance between points, and hence their distribution, should be approximately even.

The solution implemented uses the centre points of the surface voxels of a second Voxmap of the object, which has been voxelised at a different resolution to the first. The surface voxel representation was used because of the pre-imposed regular organisation of voxels, resulting in an almost even distribution of points provided by surface voxel centre points. The solution is based on the fact that for an object constructed from voxels of size D, the largest possible square gap between surface voxel centres is of size $\sqrt{2}$D. Thus any voxel of size $\sqrt{2}$D or above will not be able to penetrate the shell without detection (see Fig. 5). The second Voxmap, from

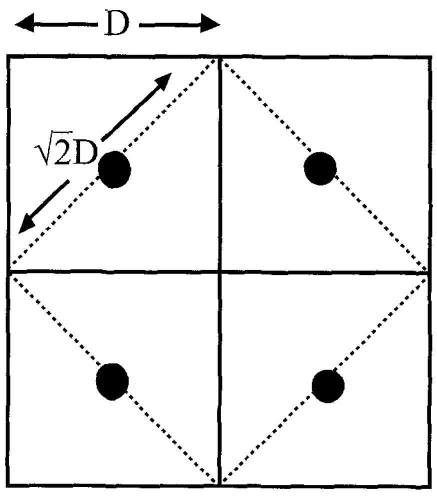


Fig. 5. Square gap of size 2D between voxel centres (for voxels of size D).

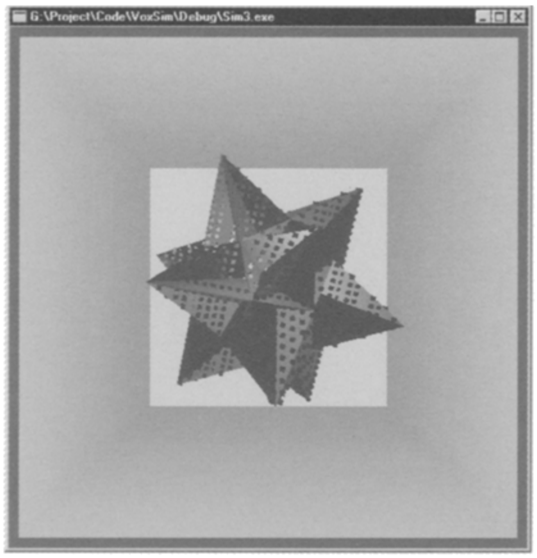


Fig. 6. Point shell superimposed on a stellated dodecahedron.

which the Point Shell for the object is obtained, is therefore constructed at the desired resolution multiplied by $\sqrt{2}$. The surface voxels of this Voxmap are identified, using an evaluation of voxels' 26 neighbours, and their centre point values stored in a list of Point Shell points for the object.

This strategy distributes points over the surface of the object sufficiently densely to prevent any voxel completely penetrating the Shell undetected, whilst avoiding the potential increase in collision processing from an excessive number of points. The Point Shell of a stellated dodecahedron, superimposed on its polygonal representation, is shown in Fig. 6.

Collision Detection: Algorithm and Performance Factors

The voxel collision detection algorithm is a fast and efficient *pair-processing* algorithm that is able to detect collisions to a user-specified level of accuracy. The algorithm sits at the heart of the proposed solution to the N-body collision detection problem, and resolves the pair-processing weakness of polygonal collision detection algorithms.

The algorithm allows complete freedom of orientation and position within an environment by rotating and translating both object Voxmaps and Point Shells along with their polygonal representation, used for graphical display. Collision detection proceeds on a pair-wise basis, using the Point Shell of one object and the Voxmap of the other, determining whether interpenetration has occurred and identifying all points of contact. The algorithm works by transforming the Point Shell of one object into the voxel space of the second object. Every Shell point is checked against the Voxmap voxels of the other object. If a point is contained within the Voxmap and the voxel is filled, then a collision has occurred at that point. All such points are checked, identifying all collision points and voxels, along with their *world* co-ordinate locations. Collisions between pairs of concave, stellated dodecahedrons and convex, icosahedrons are shown in Fig. 7. Point Shells superimposed on polygonal representations and Voxmaps are shown.

The voxel collision detection algorithm is fast owing to its simplicity. The complexity inherent in polygon-polygon collision detection is eliminated through the use of the pre-processed Voxmap and Point Shell representations. This allows the pair-processing collision detection problem to be reduced to a simple look-up of Shell points into Voxmap voxels.

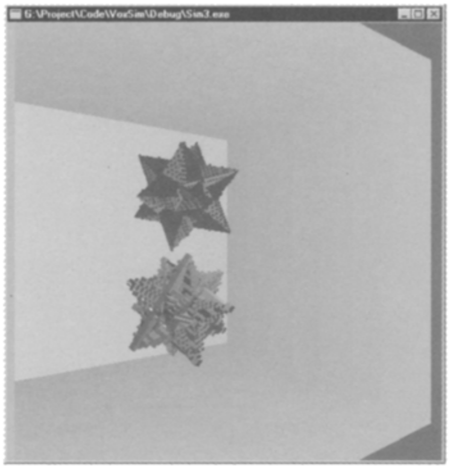
The speed of the collision detection algorithm varies linearly with the number of points in the Point Shell. It is, theoretically, independent of the number of voxels in the Voxmaps, but increasing the resolution of Voxmaps does affect the speed of the algorithm. This is because of the corresponding increase in Point Shell density required to prevent undetected voxel penetration, as previously explained.

The accuracy of the algorithm depends upon the size of the voxels used. The minimum attainable accuracy corresponds to the dimensions of the voxels. To achieve optimal performance, all object Voxmaps in the environment should contain voxels of the same dimensions, with appropriate Point Shell density. This can be achieved by identifying the desired minimum collision accuracy for an application, allowing users to trade accuracy for speed as required.

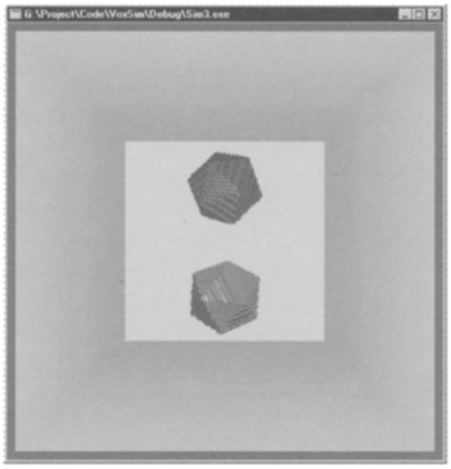
The reporting of all contact points facilitates the calculation of collision response. This allows application of both analytic and penalty collision response methods. For details of collision response methods readers are referred to: [19, 36–40].

6. N-Body Pruning

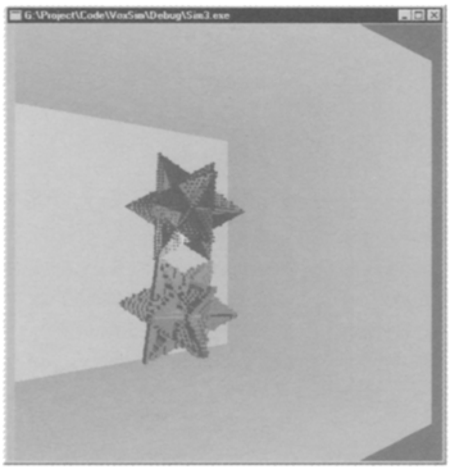
Although a fast, efficient pair-processing collision detection algorithm is a necessary component in



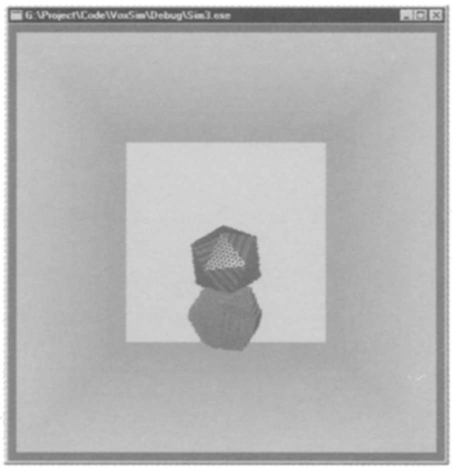
(a) Concave objects before collision.



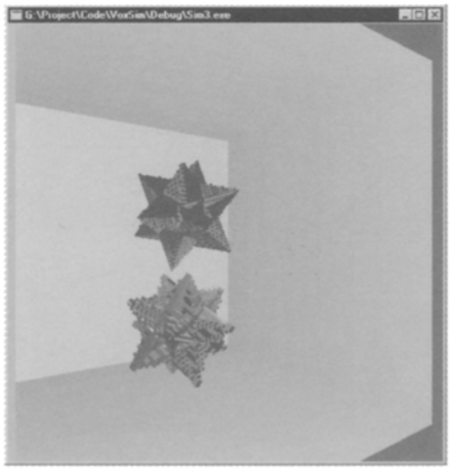
(d) Convex objects before collision.



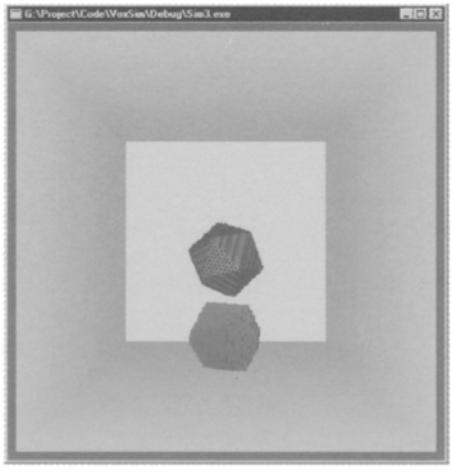
(b) Concave objects during collision.



(e) Convex objects colliding.



(c) Concave objects after collision.



(f) Convex objects after collision.

Fig. 7. Concave and convex objects colliding.

achieving high detection rates, it is *not* sufficient to solve the N-body collision detection problem alone. Rather pruning techniques are required to reduce the number of pairs of objects to which the algorithm must be applied. These techniques are needed in order to avoid the potential quadratic increase in processing cost, O(n^2), as the number of dynamic objects in the environment, n, increases. This makes most effective use of the pair-processing collision detection algorithm.

Main Pruning Technique Types

The majority of pruning techniques found in the collision detection literature can be divided into three main approach types:

- spatial decomposition
- bounding volumes
- acceleration/velocity bounds

Spatial decomposition approaches partition environment space into a number, or hierarchy of regions, and include techniques such as octrees [19], BSP trees [41], and Witnesses [38]. They prune by only comparing pairs of objects for collisions that are in the same region and are therefore close together. They suffer from a high computational expense required to choose good partition sizes, needed for optimal pruning, and through the need to frequently update these partitions as objects [15].

Bounding volumes, such as spheres or boxes, are frequently used as rapid rejection tests of trivial cases. These bounding objects provide a surrogate representation of objects, with very fast pair-overlap testing allowing rapid rejection of pairs that are not very close together. The pair-processing collision detection algorithm is only applied to pairs of objects whose bounding volumes overlap. Very efficient bounding volume overlap techniques have been developed that allow the high geometric coherence of dynamic environments to be exploited (e.g. [15,42]).

Acceleration/velocity bounds are constraints placed on the acceleration/velocity of objects in order to allow some prediction of their future [16,22,29]. They rely on these bounds to prune object pairs which are far apart, moving apart, or with low velocity and maximum relative acceleration, and thus will not collide in the next (few) simulation frames.

Techniques Implemented

Four pruning techniques were chosen for implementation, based on their suitability for use with the voxel collision detection algorithm and their efficiency. These were:

- Bounding spheres;
- Fast axis-aligned bounding boxes (Fast AABBs);
- Coherent axis-aligned bounding boxes (Coherent AABBs);
- Priority Queue.

All four techniques make use of bounding volumes, and the Priority Queue imposes acceleration/velocity bounds on objects.

Bounding Volumes

The two types of bounding volumes used, bounding spheres and static, axis-aligned bounding boxes (AABBs) were chosen for the ease with which overlaps can be determined, and their suitability for dynamic environments. Both are defined by a centre location and radius, with AABBs also requiring six extent values, a maximum and minimum for each dimension (x, y and z). Recalculation of position is simply achieved by a single vector-vector addition, with the additional update of extents for AABBs requiring three additions and three subtractions of the radius value from the centre location x, y, and z values.

A tight fit is ensured by assigning bounding volume radii as half the length of the line joining the two most distant vertices of an object, and the centre as the midpoint of this line. Bounding volume fitting is shown in Fig. 8, along with the fact that

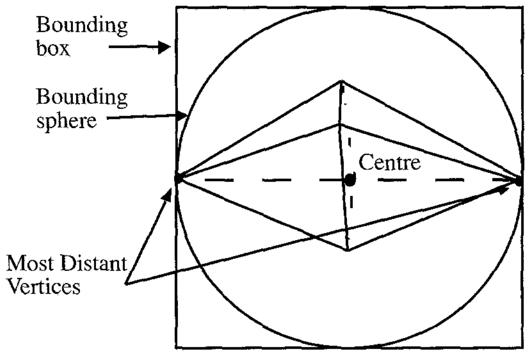


Fig. 8. Tight bounding sphere and axis-aligned bounding box fitting.

bounding spheres will always achieve a slightly tighter fit than AABBs.

Dynamically resized AABBs achieve a tighter fit, especially for long thin objects. They were not considered for implementation on the basis of Cohen et al.'s findings [15], that the additional recalculation of minima and maxima as objects move outweighs the benefit of their tighter fit.

Centralised Bounding Volume Table

The voxel collision detection algorithm, pruning techniques and simulation environment (described below) were developed using object-oriented (OO) approaches, and implemented in Visual C++. Palmer and Grimsdale [21] highlight an efficiency problem associated with OO approaches for collision detection algorithms. All geometric data for objects is encapsulated in OO systems, and therefore is hidden from all other objects. The interactive nature of collision detection requires extensive inter-object communication, which incurs "serious processing overheads" [21, p. 105] for an OO system. In order to minimise these overheads a centralised Bounding Volume Table was developed, based on the solution proposed by Palmer and Grimsdale [21], which reduces inter-object communication to a minimum. Objects communicate with their global table entry, updating it as they move, and the table entries are used to determine overlap status. Only when an overlap occurs is any inter-object communication required.

Bounding Volume Overlap Testing

A simple bounding sphere overlap test is implemented. For a pair of spheres, if the distance between their centres is greater than (or equal to) the sum of their radii, then they overlap. Axis-aligned bounding box overlap was computed for both Fast and Coherent AABBs using *dimension reduction* [15,42]. This approach is based on the fact that if two boxes collide in 3-space, their orthogonal projections onto x, y and z-axes must also all overlap.

Fast and Coherent implementations both proceed via three steps: update bounding boxes, sort the minimum and maximum co-ordinates of each dimension and determine which boxes overlap. The second and third steps can be carried out simultaneously.

Fast AABB overlap testing reduces the computation time required from the $O(n^2)$ time for a naive algorithm, comparing all pairs of n objects, to $O(n\log n + k)$ time, where k is the number of pair-wise overlaps [42]. The implementation uses an adaptation of Wagner and Fusco's [43] *Fast n-dimension overlap testing*, which achieves further performance gains through minimising the number of extents required for testing.

Coherent AABB overlap testing exploits coherence, avoiding the need to start pruning from scratch from each frame. Rather the sorted list from the previous frame is kept, which is nearly in correct order for coherent environments, using a sort algorithm that is most efficient for nearly sorted values, such as insertion sort [44]. Exploiting coherence can reduce computation time to $O(n + k)$.

Priority Queue

A collision priority queuing scheme was developed, based on the scheduling scheme algorithm proposed in Lin and Manocha [29]. This uses knowledge of objects' maximum acceleration and velocity in order to calculate a lower bound on time to collision, establishing a priority queue based on this lower bound. A development was the use of bounding spheres for separation distance calculation, and all pair-wise collision tests.

Priority queuing is an iterative process that continually inserts and *pops* object pairs off a sorted queue based on their *wakeup* times, an approximate time to collision. An *active pair* is maintained which is a lower bound on the soonest possible collision time, with no more collision processing required until its wakeup time is exceeded. As the approximate time to collision is a lower bound, no collision can be missed.

7. Simulation Environment

A simple, cubic simulation environment of one-hundred unit side length was developed for evaluation of collision detection performance, and for demonstration purposes. Within this volume, objects are free to move and to collide, both with each other and with the *walls*.

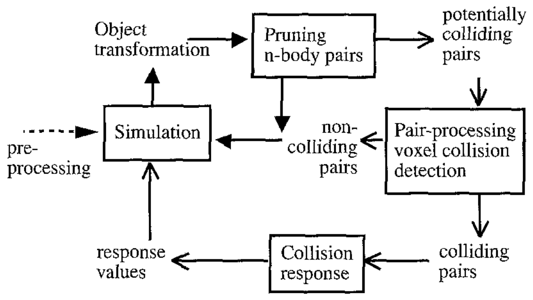


Fig. 9. Architecture of the simulation environment.

A subset of rigid body properties was implemented, sufficient to support dynamics and collision testing. Object motion was produced using Newtonian mechanics and the Euler integration method. Two types of collision detection were implemented, a simple bounding volume overlap test for object-wall collisions, and the four pruning techniques described in the previous section combined with the voxel collision detection algorithm for object-object collisions. Simple collision response was achieved by inverting the component(s) of velocity perpendicular to the wall normal(s) for object-wall collisions, and exchanging velocities and inverting rotation direction for object-object collisions. Figure 9 shows the architecture of the simulation environment.

8. Performance Evaluation

The performance of the voxel collision detection algorithm was evaluated, both alone and when combined with n-body pruning techniques. All tests were performed using the simulation environment described above, recording mean CPU times to millisecond accuracy for three simulation runs of one thousand frames for each result sample value. All simulations were run on a PC with a Pentium 166 MHz processor.

Simulation timings include calculation of objects' new positions, orientations and velocities, update of bounding volumes, all pruning tests, object-wall and object-object collision detection and response calculations. Some minor initialisation costs may be excluded, and as all tests aim to record collision detection cost, were run without graphics, and thus rendering times are **not** included. The density of objects, defined as the percentage of the simulation environment volume occupied by objects, was maintained at 2.5%. The test object was a simple, irregular convex polytope, consisting of 36 faces, and is shown in Fig. 10. It was chosen to increase similarity with performance testing of I-COLLIDE [15], and has an insignificant effect on results owing to the independence of collision performance from the polygonal representation of an object.

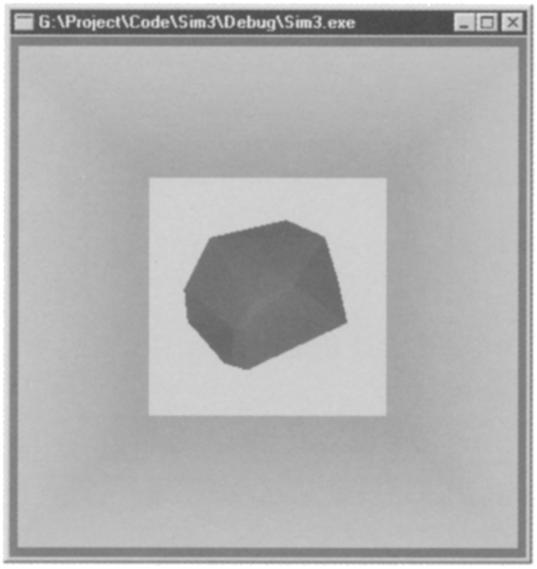


Fig. 10. Simple, irregular 36-faced object.

Pair-Processing Performance

Two tests were performed to evaluate the performance of the voxel collision detection algorithm alone. Figure 11 shows the results of increasing collision accuracy using Point Shells of appropriate density. The size of the Voxmap is constant, whilst its resolution is increased, decreasing the voxel size, and therefore increasing the number of voxels. The graph shows the collision update rate achieved in Hertz (i.e. frames per second, where a frame is one step of the simulation), for 12 sample points taken for between 20 and 200,000 voxels. The graph demonstrates that collision detection performance is inversely proportional to the accuracy of collision detection.

The second experiment was designed to test whether collision detection performance varies linearly with the number of points in an object's Point Shell, and is independent of the number of Voxmap voxels. Figure 12 shows collision update in Hertz for 12 Point Shells, including between 20 and 10,000 points, tested for collisions against a Voxmap containing a fixed number of voxels

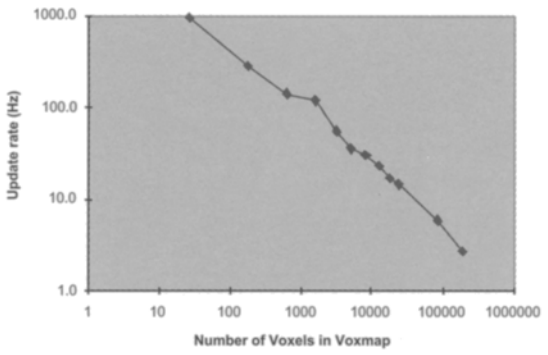


Fig. 11. Update rate versus number of voxels (logarithmic axes).

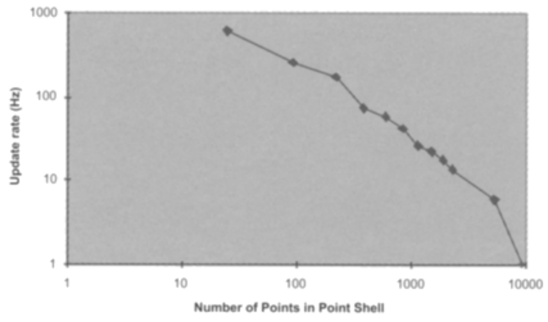


Fig. 12. Update rate versus number of points in point shell (logarithmic axes).

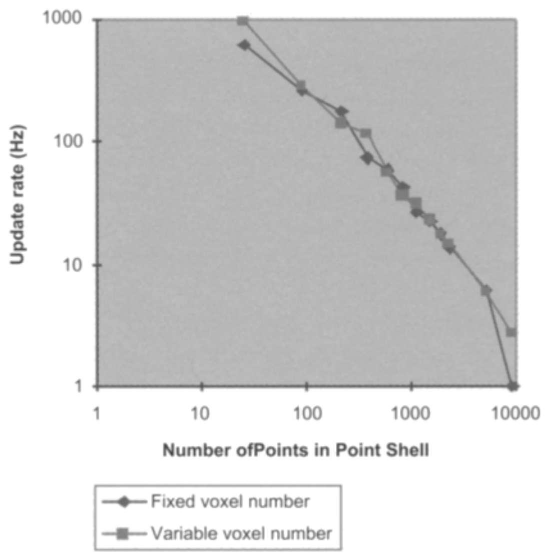


Fig. 13. Comparison of update rates for a fixed versus a varied number of Voxmap voxels (logarithmic axes).

(12,672). The graph demonstrates, with the notable exception of the last point, the expected linear relationship between collision detection performance and the number of points in an object's Point Shell. The independence of collision performance from Voxmap resolution is seen by the similarity of these two sets of results, highlighted by their direct comparison in Fig. 13.

The mean time per collision point look-up values in Table 1 show an almost constant look-up time of 0.03 milliseconds per point. The only exceptions are for small numbers of Shell points (25 and 92), where initialisation costs included in this time significantly affect the mean value, and for a very large number of points (9337) where a lack of free memory resulted in disc swapping and an associated reduction in performance (as demonstrated by the increased gradient to the final point in Figure 12). This constant time per point look-up is the fundamental determinant of voxel collision detection performance, with any improvements in hardware or software producing direct performance gains. Results show collision performance to be high, achieving an update rate of approximately 33Hz for 1000 Shell points, with all collision points reported.

Table 1. Performance results for varying the number of Shell points.

Number of Shell points	Update rate (Hz)	Time per collision point look-up (msec.)
25	618.4	0.064
92	261.0	0.042
219	173.0	0.026
384	73.4	0.035
606	58.7	0.028
840	41.7	0.029
1150	26.7	0.033
1509	22.5	0.029
1909	17.7	0.030
2314	13.8	0.031
5321	6.0	0.031
9337	1.0	0.105

N-Body Performance

This second set of experiments was designed to test performance of the voxel collision detection

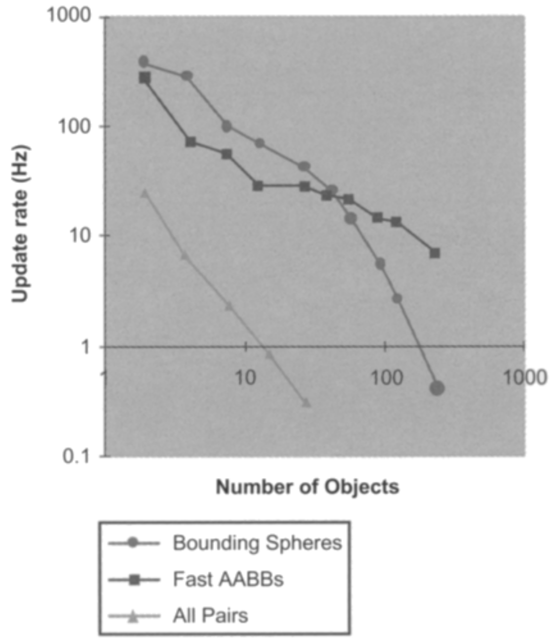


Fig. 14(a). Update rate versus object number – part one (logarithmic axes)

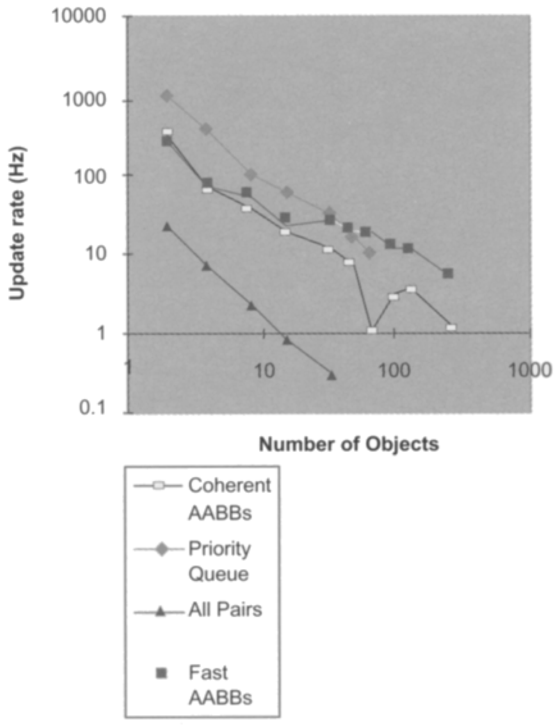


Fig. 14(b). Update rate versus object number – part two (logarithmic axes).

algorithm in combination with Bounding Sphere, Fast and Coherent AABB, and Priority Queue pruning techniques. Collision accuracy was kept constant for all tests (at a minimum of one *world* unit), maintaining environment density at 2.5% and giving all objects a random linear momentum of up to 10% of their approximate radius per frame. Collision update rates (in Hertz) are shown in Fig. 14 as the number of objects in the environment is increased from 2 to 256 with ten sample points taken.

Figure 14(a) compares Bounding Sphere and Fast AABB performance, against a base-line provided by *all-pair* processing where no pruning method was used. Both pruning methods achieve a significant improvement of at least an order of magnitude on this base-line. Bounding Spheres show the higher initial performance, achieving 99.5Hz for eight objects compared to 57.4Hz for Fast AABBs. By 64 objects Fast AABB performance overtakes that of Bounding Spheres, achieving 19.2 versus 12.9Hz. From then on Bounding Sphere performance drops off dramatically compared to a steady decline in Fast AABB performance.

Figure 14(b) compares Coherent AABB performance and the Priority Queue against the *all-pairs* base-line. Fast AABB performance is included as an aid for comparison with Fig. 14(a). The Priority Queue achieves the highest initial performance with an update rate of in excess of 1000Hz. for two objects, with a steady performance reduction exhibited as the number of objects rises. The high memory demand required for queue maintenance meant that only eight sample points were taken. Coherent AABB and Fast AABB performance is initially similar, but beyond 32 objects Coherent AABB performance becomes relatively poorer, with an anomalous result seen for 64 objects. The lower performance of the coherent approach was unexpected, suggesting the additional book-keeping required outweighs the benefits of exploiting coherence, although further investigation is needed.

These results show that interactive update rates of in excess of 20Hz can be achieved for up to 64 independent objects with high collision accuracy on a standard PC. By simply halving the collision accuracy, resulting in a barely perceivable difference, collision performance could be increased by a theoretical maximum of eight times, but in practise will be at least doubled. Such results are very promising for the voxel-based approach.

Discussion of Results

The voxel collision detection algorithm can achieve real-time performance, i.e. almost constant detection rates above 20Hz, for both high collision accuracy cases using high resolution Voxmaps, and for large numbers of objects at a lower resolution. Thus changes in an application's geometric characteristics are easily dealt with both between and within simulation runs. The experimental results indicate excellent performance, but are difficult to compare with those of polygonal collision detection algorithms. A suitable comparison might be to equate the number of faces of a polygonal object with the number of points in an object's Point Shell. Such a comparison is most likely to hold for more complex polygonal objects with hundreds or thousands of faces. The present results would correspond to update rates of approximately 300Hz for 100 faces, 67Hz for 500 faces and 33Hz for 1000 faces, for pair-wise collision tests. These figures compare very favourably with results from polygonal collision detection algorithm, particularly when it is considered that these results hold for both convex and concave objects, with simulations run on a standard PC.

9. Conclusions

The voxel-based approach to collision detection is particularly suited to complex multi-body simulation environments, such as virtual environments, due to its speed and simplicity. The voxel collision detection algorithm, that sits at the heart of collision processing, is a very efficient two-body collision detection algorithm. It achieves this efficiency through use of pre-processed Voxmap and Point Shell representations, which allow run-time collision detection to be fundamentally simplified to a look-up procedure. Fast collision detection can be achieved to a user-defined level of accuracy, allowing changes in an application's geometric characteristics to be readily dealt with. When combined with appropriate pruning techniques the voxel-based approach can achieve real-time, multi-body collision detection for large numbers of dynamic objects.

The potential exists to greatly enhance already high collision performance through the parallel, hardware implementation of the voxel collision detection algorithm, rendering the voxel-based approach to collision detection all the more promising.

References

1. Logan IP, Wills DPM, Avis NJ, Mohsen AMMA, Sherman KP. Virtual environment knee arthroscopy training system. Society for Computer Simulation, Simulation Series 1996; 28(4): 17–22
2. McNeely WA, Puterbaugh KD, Troy JJ. Six degree-of-freedom haptic rendering using voxel sampling. In: Computer Graphics Proceedings, Siggraph'99,8–13 August 1999; 401–408
3. Ward JW, Wills DPM, Sherman KP, Mohsen AMMA. The development of an arthroscopic surgical simulator with haptic feedback. Future Generation Computer Systems 1998; 14: 243–251
4. Ellis SR. What are virtual environments. IEEE Computer Graphics and Applications 1994; 14(1): 17–22
5. Heeter C. Being there: the subjective experience of presence. Presence: Teleoperators and Virtual Environments 1992; 1(2): 262–271
6. Hendrix C, Barfield W. Presence in virtual environments as a function of visual display parameters. Presence: Teleoperators and Virtual Environments 1996; 5(2): 274–289
7. Johnson-Laird PN. Mental models. In: Foundations of cognitive science. Posner MI, ed. Cambridge University Press, 1983; 469–493
8. Johnson-Laird PN. The computer and the mind. Cambridge, MA: Harvard University Press, 1988
9. Norman DA. The psychology of everyday things. Basic Books, New York, 1988
10. Barfield W, Hendrix C, Bjourneseth O, Kaczmarek KA, Lotens W. Comparison of human sensory capabilities with technical specifications of virtual environment equipment. Presence 1995; 4(4): 329–356
11. Zeltner D. Autonomy, interaction and presence. Presence 1992; 1(1): 127
12. Kennedy RS, Lane NE, Lilienthal MG, Berbaum KS, Hettinger LJ. Profile analysis of simulator sickness symptoms: application to virtual environment systems. Presence 1992, 1(3): 293–301
13. Hettinger LJ, Riccio GE. Visually induced motion sickness in virtual environments. Presence 1992; 1(3): 306–310
14. Pausch R, Crea T, Conway M. A literature survey for virtual environments: military flight simulator visual systems and simulator sickness. Presence 1992; 1(3): 344–363
15. Cohen JD, Lin MC, Manocha D, Ponamgi MK. I-COLLIDE: an interactive and exact collision detection system for large-scale environments. In: Proceedings of ACM International 3D Graphics Conference 1995; 189–196
16. Hubbard PM. Collision detection for interactive graphics applications. IEEE Transactions on Visualization and Computer Graphics 1995b; 1(3): 218–230
17. Hubbard PM. Approximating polyhedra with spheres for time-critical collision detection. ACM Transactions on Graphics 1996; 15(3): 179–210
18. Jiang H, Vanecek G. Jr. N-body collision detection based on lazy evaluation. Extended abstract, Department of Computer Science, University of Purdue, W. Lafayette; presented at SIVE 1995
19. Moore M, Wilhelms J. Collision detection and response for computer animation. Computer Graphics 1988; 22(4): 289–298

20. Vanecek G. Jr. Back-face culling applied to collision detection of polyhedra. The Journal of Visualization and Computer Animation 1994; 5: 55–63

21. Palmer IJ, Grimsdale RL. Collision detection for animation using sphere-trees. Computer Graphics Forum 1995; 14(2): 105–116

22. Hubbard PM. Interactive collision detection. Technical Report, Department of Computer Science, Brown University. In: Proceedings of the 1993 IEEE Symposium on Research Frontiers in Virtual Reality, October 1993; 24–31

23. Hubbard PM. Real-time collision detection and time-critical computing. Workshop on Simulation and Interaction in Virtual Environments, July 1995a; 92–96

24. Gottschalk S, Lin MC, Manocha D. OBB-tree: a hierarchical structure for rapid interference detection. Technical Report TR96-013, Department of Computer Science, University of N. Carolina, Chapel Hill. In: Proceedings of ACM SIGGRAPH '96; 171–180

25. Hudson TC, Lin MC, Cohen J, Gottschalk S, Manocha D. V-COLLIDE: accelerated collision detection for VRML. In: Proceedings of VRML97, 24–26 February 1997. Monterey, CA: ACM Press; 119–125. Available from URL: http://www.cs.unc.edu/~geom/collide.html

26. Vanecek G. Jr, Gonzalez-Ochoa C. Representing complex objects in collision detection. SIVE 95, Ewa City, IA. Available from URL: www.cs.uiowa.edu/ncremer/sive95.html

27. Lin MC, Canny J. Efficient collision detection for animation. Proceedings of the Third Eurographics Workshop on Animation and Simulation 1991, Cambridge, UK.

28. Lin MC, Manocha D. Efficient contact determination between geometric models. Technical Report TR94-024, Department of Computer Science, University of North Carolina, Berkley, NC, 1994

29. Lin MC, Manocha D. Fast interference detection between geometric models. The Visual Computer 1995; 11: 542–561

30. Ponamgi MK, Manocha D, Lin MC. Incremental algorithms for collision detection between solid models. In: Proceedings of ACM/Siggraph Symposium on Solid Modeling 1995; 293–304

31. Jones MW. The production of volume data from triangular meshes using voxelisation. Proc. of EuroGraphics '96, Computer Graphics Forum 1996; 15(5): 311–318

32. Kaufman A. An algorithm for 3D scan-conversion of polygons. In: Proceedings of Eurographics '87, Amsterdam, Netherlands; 197–208

33. Glassner AS. An introduction to ray-tracing. London: Academic Press, 1989

34. Watt A. 3D computer graphics. Suffolk, UK: Addison-Wesley, 1993

35. Rule K. Crossroads 3D. Homepage and freeware download at URL: http://www.europa.com/~keithr/crossroads/, 1997

36. Baraff D. Issues in computing contact forces for non-penetrating rigid bodies. Algorithmica 1993; 10: 292–353

37. Baraff D. Fast contact force computation for non-penetrating rigid bodies. In: Computer Graphics Proceedings, SIGGRAPH '94, 24–29 July 1994; 23–34–38

38. Baraff D. Curved surfaces and coherence for non-penetrating rigid body simulation. Computer Graphics 1990; 24(4): 19–28

39. Hahn JK. Realistic animation of rigid bodies. Computer Graphics 1988; 22(4): 299–308

40. Kammat VV. A survey of techniques for simulation of dynamic collision detection and response. Comput. & Graphics 1993; 17(4): 379–385

41. Thibault WC, Naylor FB. Set operations on polyhedra using binary space partitioning trees. Computer Graphics 1987; 21(4): 153–162

42. Baraff D. An introduction to physically based modelling: rigid body simulation II – non-penetration constraints. SIGGRAPH '95 & '97 Course Notes on Physically Based Modelling 1997. Available from URL: http://www.cs.cmu.edu/afs/cs/user/baraff/www/pbm/pbm.html

43. Wagner L, Fusco M. Fast n-dimension extent overlap testing. In: Graphics gems III. Kirk D. ed. Boston, MA: Academic Press, 1991; 240–243 and 527–533

44. Sedgewick R. Algorithms in C++. Reading, MA: Addison-Wesley, 1992; 93–105

Correspondence and offprint requests to: *D. P. M. Wills, Department of Computer Science, University of Hull, Hull, HU6 7RX, UK. Email: d.p.wills@dcs.hull.ac.uk*