TR−C−0103

Towards the Realization of Real−Time Collision Detection

アンドリュー　スミス　　北村　喜文
Andrew Smith　　Yoshifumi KITAMURA

1994　12．20

ATR通信システム研究所

Towards the Realization of Real-Time Collision Detection

Andrew Smith and Yoshifumi Kitamura
Artificial Intelligence Department
ATR Communication Systems Research Laboratories

December 20, 1994

Contents

List of Figures

Preface

This technical report consists of 4 chapters which together describe my research activities during my stay at the Communication Systems Research Laboratory of the Advanced Telecommunications Research Institute from October of 1993 to December of 1994. The main focus of my research at ATR was on efficient algorithms for collision detection, and the flow of my research was as follows. Upon my arrival at ATR, work on efficient collision detection for polyhedral objects had been going on for about 6 months. This original algorithm works by precomputing octrees for polyhedral objects, updating these objects' octrees at each time step and checking for interference of objects' updated octrees to localize possible collisions among the faces of the objects. This algorithm was shown to be quite efficient compared with the standard polyhedral collision detection method, where all combinations of different objects' faces must be tested for collision, but was still not effective for real-time collision detection in a virtual environment. Thus, my original project was to implement this algorithm on a parallel computer for real-time performance.

In starting work on parallelization of the original collision detection algorithm, I discovered various optimizations to the serial algorithm which made it roughly 5 times faster. These optimizations are described in the first chapter entitled "Optimization of Octree-based Collision Detection." Another problem with the original algorithm was that it had to move the octrees of objects at each time-step, which was quite time consuming. Thus, I investigated the problem of octree motion and discovered that not much research had been previously done on efficient algorithms for octree motion. Against this lack of research, I formulated various optimizations to the octree motion algorithm being used for collision detection, and these are described in the second chapter entitled "Efficient Algorithms for Octree Motion."

After implementing these optimizations to collision detection and octree motion, the collision detection was quite a bit more efficient but, unfortunately, still not entirely suitable for real-time applications. Thus, I formulated a new, more efficient collision detection algorithm. This algorithm was similar to the optimized collision detection algorithm (described in "Optimization of Octree Based Collision Detection"—the optimizations described in this chapter form the basis of the new algorithm) except that it eliminated the precomputation and update of octrees; instead, the algorithm used bounding boxes and octree-like spatial subdivision directly on the polyhedral faces to find the colliding faces. This new algorithm was shown by experiments to be more efficient than the original collision detection algorithm, and was suitable for real-time applications. This new algorithm is described in the third chapter entitled "A Simple and Efficient Method for Accurate Collision Detection Among Deformable Polyhedral Objects in Arbitrary Motion." Finally, another version of this new algorithm, which does not miss collisions between time instants by testing for intersection of faces' swept space between time instants, was implemented. This version of the new algorithm was quite a bit slower (due to the necessity of computing interference between swept spaces) and did not have real-time performance; thus, parallelization of this algorithm was done and the last chapter, entitled "Parallelization of Collision Detection for Real-time Performance," describes the parallel algorithms and experimental results. These four chapters together describe the research that I conducted while at ATR.

Chapter 1

Optimization of Octree-Based Collision Detection

1.1 Introduction

The collision detection algorithm in [KTK94] was shown to be much more computationally efficient than the standard, polyhedral method. However, in terms of getting real-time performance, it is still not sufficient. In this short chapter, four important optimizations to the original algorithm of [KTK94] are presented.

1.2 Original Collision Detection Algorithm

The algorithm is used to determine the colliding faces, if any, of different objects in the world. The details of the collision detection algorithm are given in [KTK94]. However, briefly, the algorithm proceeds as follows. All objects in the world are represented using two object representations: a polyhedral representation (the main representation), and the octree representation (the auxiliary representation–used for efficient collision detection). All objects (both the polyhedral representation and octree representation) are moved to new positions at discrete time steps, and collisions are checked for at these times. Thus, at each time step the collision detection proceeds as a series of course-to-fine steps:

1. For every object, compute the bounding box of the polyhedral representation of the object. For every pair of objects, determine whether the bounding boxes of the objects overlap. Any object whose bounding box doesn't overlap any other object's bounding box does not collide and does not need to be checked by the following steps.

2. Rebuild the octrees of the remaining objects (i.e., move the octrees). Then, search the rebuilt octrees for interference (i.e., when two or more objects have black nodes at the same location in the rebuilt octrees). Any object whose rebuilt octree doesn't interfere with any other object's rebuilt octree does not collide and does not need to be checked by the following steps.

3. For all interfering black nodes and all objects whose rebuilt octrees contain those black nodes (found in step 2), find the faces from the polyhedral representation of the object that intersect the interfering black node; do this by checking all faces of the object for intersection with the interfering black node. Then, for each interfering black node find all possible pairs of faces (where the faces are from different objects) from the faces found to be intersecting that black node.

4. For each pair of faces found in step 3, sweep each face the area that it moves up to the current timestep (i.e., the area it sweeps as it moves from the previous timestep to the current). Then, form the convex hull of each swept area and determine if the two convex hulls intersect. If the convex hulls intersect then the two faces are considered to have collided, otherwise not.

1.3 Optimizations

1.3.1 Use of Face Octrees Instead of Volumetric Octrees

The original algorithm uses standard, volumetric octrees. In this type of octree, any node that is completely contained in (internal node) or intersects a face of (face node) an object is included in the octree of the object.

However, for collision detection purposes, collisions will never be detected for internal nodes and thus it is wasted computation to move and check interferences for them. Thus, this optimization is to use octrees which only contain face nodes—called face octrees.

1.3.2 Intersection of Octree Black Nodes with Overlap Regions of Bounding Boxes

In the original algorithm, if an object's bounding box overlaps with other objects' bounding boxes then the octree for the object is completely rebuilt and checked for interference with other objects' rebuilt octrees (step 2). However, the only black nodes in an object's octree that could possibly cause interference are the ones which are contained in or intersect the regions of overlap of the object's bounding box with other objects' bounding boxes. Thus, this optimization changes step 2 to update only part of an object's octree: the part that is contained in or intersects the regions of overlap of the object's bounding box with other objects' bounding boxes.

1.3.3 Progressive Refinement of Octrees in Searching for Interfering Black Nodes

In step 2 of the original algorithm, the octrees of objects are first completely rebuilt and then checked for interference. However, this does not take advantage of the hierarchical nature of the octree in order to quickly (i.e., with minimal rebuilding of octrees) localize interfering black nodes. This optimization essentially interleaves the rebuilding of octrees with the checking for interfering black nodes, progressively refining parts of octrees found to be interfering with other objects' octrees until the level of resolution is reached. With this optimization, instead of completely rebuilding the octrees and only then checking for interfering black nodes, octrees are partially rebuilt to a certain level (starting from the highest level–root node) and then, the partially rebuilt octrees are checked for interference; only interfering black nodes (i.e., nodes that contain or intersect with black nodes from two or more objects' octrees) in the partially rebuilt octrees are rebuilt to the next lower level.

1.3.4 Association of Polyhedral Faces with Octree Black Nodes

In step 3 of the original algorithm, all faces of an object must be checked for intersection with an interfering black node. This is extremely computationally expensive, especially if the object has thousands of faces. This optimization greatly reduces the number of faces that must be checked for intersection with an interfering black node. This is done by associating (using pointers) the black nodes of the source octrees with the polyhedral faces that are contained inside of or intersect with them. Then, when the source octrees are rebuilt (i.e., moved to reflect the new position of the objects) these pointers are copied to the new rebuilt octree; a black node in a rebuilt octree copies all pointers from every source octree black node that is inside of or intersects with that black node. Finally, in step 3 instead of checking an interfering black node for intersection with all faces of the objects which contain that black node only the faces pointed to by the pointers for that interfering black node need to be checked.

Chapter 2

Efficient Algorithms For Octree Motion

Abstract

This chapter presents efficient algorithms for updating moving octrees. The first algorithm works for octrees undergoing both translation and rotation motion; it works efficiently by compacting source octrees into a smaller set of cubes (not necessarily standard octree cubes) as a precomputation step, and by using a fast, exact cube/cube intersection test between source octree cubes and target octree cubes. A parallel version of the algorithm is also described. Finally, the chapter presents an efficient algorithm for the more limited case of octree translation only. Experimental results are given to show the efficiency of the algorithms in comparison to competing algorithms. In addition to being fast, the algorithms presented are also space efficient in that they can produce target octrees in the linear octree representation.

2.1 Introduction

Octrees are commonly used in computer graphics and robot path planning applications as an auxiliary object representation to speed up spatial access to the parts of the main object representation. For example, in collision detection octrees of objects can be searched to localize quickly interference between objects [KTAK94]. In speeding up the spatial access to the parts of the main object representation, the octree is immensely helpful. The problem is that when an object moves the octree for that object must be updated to reflect the new location, which is not as straightforward as for other object representations such as boundary representations and is in general very computationally intensive. Not much research has been done on algorithms for arbitrary octree motion [WA87, HO87].

This chapter deals with the problem of efficiently updating moving octrees [SKK94]. The chapter starts by considering the general problem of arbitrary motion of octrees (i.e., both translation and rotation) and takes as a basis the arbitrary octree motion algorithm described in [WA87]. The chapter describes a new, more efficient octree arbitrary motion algorithm and provides data showing that the algorithm runs 3 to 4 times faster than [WA87]. The new algorithm has two important features which allow it to run efficiently. The first feature is based on the fact that the computational cost of octree motion is proportional to the number of black nodes; thus, the chapter shows how, as precomputation, an octree can be compacted into a smaller list of nonoverlapping cubes (usually about half as many cubes). The second feature is the use of a fast cube/cube intersection test which is specialized for efficient moving of octrees. A parallel version of the arbitrary motion algorithm will also be presented and experimental results will be given to show it's effectiveness. Finally, the chapter considers the more limited problem of octree translation only and describes a simple method which can be used to translate octrees most efficiently. It is important to note that, in addition to being fast, the algorithms presented in this chapter are also space efficient; this is because they can be used for octrees represented in the linear octree representation.

2.2 Octree Shape Representation

2.2.1 Basic Representation Scheme/Linear Representation

The octree is a volumetric, hierarchical shape representation scheme. The octree represents the shape of an object by recursively subdividing cubes into 8 smaller cubes (octants), starting from a single large root cube representing the entire world space. A cube is labelled black (white) if it is completely contained within the object (free space); otherwise, the node is labelled gray. Cubes at the highest level of resolution (smallest cubes) are called voxels and, since there can be no gray voxels, are labelled black or white depending on application specific rules. An example of an octree is shown in figure 2.1.

Since an explicit pointer-based octree storage scheme can be prohibitively expensive in terms of memory requirements, more compact linear encodings of octrees have been invented. An example of this is the DF-representation for octrees [Man88]. Essentially, this scheme stores an octree by listing consecutively the octree nodes encountered on performing a preorder traversal of the octree, where the alphabet used is "(", (gray node), "B" (black node), and "W" (white node). Since there are only three characters in the alphabet, two bits per node are sufficient for storing the octree. As an example, the example octree of figure 2.1 has the DF-representation "(B(BWBBBWWWBWBWB(B(BWBBBWBWBBB
WBW".

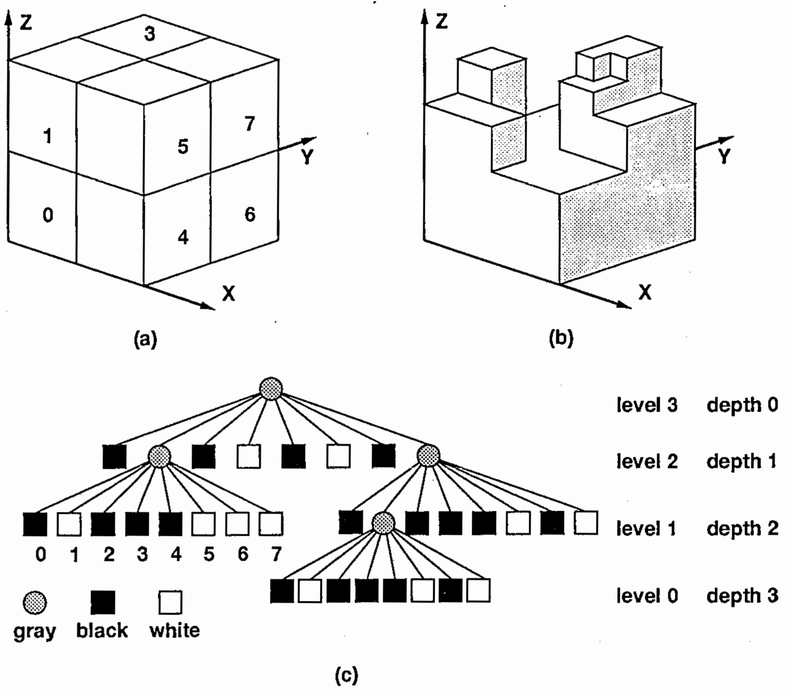


Figure 2.1: The octree shape representation. (a) The ordering of octants (b) An example octree (c) Pointer-based representation of the example octree.

2.2.2 Octree Motion Basic Algorithm

The basic algorithm for moving an octree [WA87] (for both translation only and arbitrary motion) is to apply the motion transformation matrix to each black cube in the octree to be moved (called the source octree) separately (for a series of motions, the same source octree is always used and only the motion matrix is changed; this prevents digitization errors from continually increasing) and to test recursively, starting from the largest cube, for intersections between the transformed black cubes and the standard, upright (i.e., faces parallel to the standard euclidean coordinate axes) cubes of the new octree to be created (called the target octree). Standard cubes in the target octree are labelled white, gray, or black depending on whether they don't intersect with a transformed black cube, intersect partially with a transformed black cube, or are completely inside of a transformed black cube. Target voxels are labelled black or white depending on application specific rules. After a transformed source black node is tested for intersection with a target octree gray node, the gray node is tested to see if its 8 children are all labelled black or all labelled white; if so then the children are erased and the gray node is labelled the same as the children were (this is called condensing the octree). This basic algorithm is illustrated in figure 2.2 (to simplify the figure, we illustrate the algorithm for the 2D case, called a quadtree; the octree algorithm works in an analogous way). The following sections will describe efficient variations of this basic algorithm.

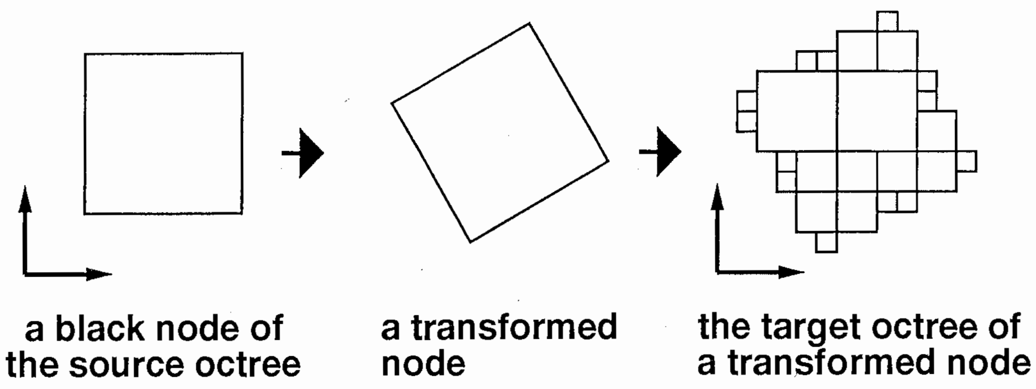

a black node of the source octree
a transformed node
the target octree of a transformed node

Figure 2.2: The octree motion basic algorithm illustrated for the 2D case (quadtree).

2.3 Compaction of Octrees

Note the fact that the computational cost of octree motion is proportional to the number of black nodes in the source octree. Also, since motion always starts from the same source octree much precomputation on the source octree can be done to speed up the octree motion. In particular, it is not even necessary to store the octree directly as an octree. Thus an optimization to the basic algorithm is, as a precomputation step, to compact the octrees to be moved into the smallest set of nonoverlapping cubes. The cubes in this compacted set are not restricted to being the standard cubes of an octree (e.g., they don't necessarily have side lengths which are powers of 2), but they completely cover the black area of the octree.

Conceptually, the smallest set of nonoverlapping cubes which completely cover an octree's black area can be found by examining all combinations of integer side length cubes which are inside the root cube. Practically, however, this is intractable. Here, we merely wish to demonstrate the utility of octree compaction for octree motion. So for demonstration purposes we used the following approximate algorithm; this algorithm is not guaranteed to find the strictly smallest set of nonoverlapping cubes but it does generally find a set that contains about half as many cubes as there were black cubes in the original source octree.

The algorithm works as follows. First, the bounding box of the octree is found; this is the axis-aligned parallelpiped inside the root cube which just encloses all of the black cubes of the octree. Next, all cubes which have integer side length and which are contained inside the bounding box at integer-valued vertices are enumerated in order from larger to smaller cubes (the ordering of same sized cubes does not matter). The enumerated cubes are then scanned from larger cubes to smaller cubes. Each scanned cube is tested for intersection with the white cubes in the octree and the cubes that don't intersect any white cubes (i.e., that are completely inside the black area of the octree) are put into a new list. This new list is scanned, again from larger cubes to smaller cubes, and the scanned cubes which don't intersect any cubes already in the output list (this is initialized to contain no cubes) are added to the output list. After this, the output list will contain a list of cubes which are nonoverlapping and which completely cover the black area of the octree; usually, there will be fewer cubes than there are black cubes in the octree. Note that this basic algorithm could be easily combined with a random or genetic algorithm component to get better compaction. For example, an extra step could be added which randomly changed the order of the enumerated cubes and then ran the algorithm again; this could be done for some number of steps and the smallest set found could be used. For our experiments (see section 2.6), we scan the list of enumerated cubes n times, where n is the number of enumerated cubes, starting from a different cube for each scan (but still going from larger cubes to smaller cubes—thus, not all cubes are always scanned).

2.4 Arbitrary Octree Motion

2.4.1 Problems with Related Work

For arbitrary octree motion, the transformed source cubes are not necessarily upright and so there is no simple intersection test. [WA87] claims that performing an accurate intersection test between the transformed source cubes and the target octree standard cubes is too computationally expensive and that an approximate intersection test between the circumscribed spheres of the transformed source cubes and the non-voxel target octree standard cubes will allow most efficient arbitrary motion of octrees (remember that target voxels are handled by application specific rules—thus, the created target octree is not approximate). There are two problems with this, however. First, [WA87] claims to be testing for intersection between the circumscribed spheres of the transformed source cubes and the target octree standard cubes. However, the mathematical test that is actually described to perform this intersection test is in fact geometrically an intersection test between the bounding boxes of the circumscribed spheres of the transformed source cubes and the target octree standard cubes; thus, it is doubly approximate. Second, [WA87] claims that using an approximate intersection test is the efficient way to move octrees; however, an important result of this chapter is that using the exact cube/cube intersection test described in this chapter greatly reduces the total number of such cube/cube intersection tests (the exact test eliminates more target cubes from further consideration earlier on) which need to be performed and allows the algorithm to run approximately 40% faster.

2.4.2 Exact Source Cube/Target Cube Intersection Test

The exact source cube/target cube intersection test requires the positions (i.e., center point and eight corner points) of each cube before motion (i.e., their upright positions) and both the motion transformation matrix and the inverse motion transformation matrix. In the algorithm, any transformations done on the source cube use the motion transformation matrix and any transformations done on the target cube use the inverse transformation matrix. The test will return one of three possibilities: intersection, no intersection, or complete intersection (this will be returned if the target cube is completely inside the source cube). The test is a series of coarse-to-fine steps as follows:

1. Determine the smaller of the two cubes. If the two cubes are the same size then the source cube is considered to be the smaller cube. Also, determine the radius of the circumscribed and inscribed spheres of the smaller cube.

2. Transform the center point of the smaller cube and determine if either its circumscribed sphere or inscribed sphere (these will be centered at the transformed center point and have radiuses as calculated in step 1) intersect with the upright version of the larger cube[1]. If the circumscribed sphere does not intersect, then the two cubes definitely do not intersect. If the inscribed sphere does intersect, then the two cubes definitely do intersect (however, if the smaller cube is the target cube, then even if intersection is detected continue to the next step to check for complete intersection). Otherwise, continue to the next step.

3. Next, transform the eight corner points of the smaller cube and check to see if any of them are contained within the upright version of the larger cube. If the smaller cube is the target cube then check to see if all of the eight transformed points are contained within the upright version of the larger cube; if so then return complete intersection (if not all eight points are inside, but one or more is inside then return intersection). If not, Stop and report intersection after finding the first such corner point inside. Otherwise, continue on to the next step.

4. Transform the eight corner points of the larger cube and check to see if any of them are contained within the upright version of the smaller cube. Stop and report intersection after finding the first such corner point inside. Otherwise, continue to the next step.

5. Now, because we have gotten this far we know that the two cubes are intersecting if and only if each cube has at least one edge (i.e., one of the edges of its faces) intersecting a face of the other cube. Determine the edges of the transformed version of the smaller cube. Then, test each edge against every face of the upright version of the larger cube to see if there is a face for which the edge is on the outside of the face. If there does exist such a face, then the edge cannot be intersecting with the other cube. If there is no such face, then the edge might be intersecting the other cube so store it in a list of edges. If, after checking all edges, there are no edges in the list then the two cubes definitely do no intersect. Otherwise, pass the list of edges onto the next step.

6. For each edge in the list of edges and for each face plane of the upright cube that it intersected, find the intersection point of the edge with that plane. If the intersection point is inside the face of the upright cube, then there is definitely intersection. If not then continue with the next face (for the current edge) or the next edge (from the list). If all such edges and faces are checked without finding any intersection points on a face, then there is no intersection. This concludes the test.

Note that optimizations to this can be made for efficient octree motion. For example, the circumscribed and inscribed radiuses can be precomputed since there are only a finite number of them. Also, even though a source or target cube might need to be tested for intersection with many other cubes, its center point and eight corner points only need to be transformed once.

[1]The actual sphere/cube intersection test used is described on page 335 of [Gla90]; however, we optimize this test by noting that it does not need to be called twice separately for the circumscribed and inscribed spheres—since both have the same center point the calculation of d_{min} is the same for both and thus only an extra conditional is needed at the end for the extra sphere.

2.4.3 Arbitrary Motion Efficient Algorithm

The new algorithm works by recursively traversing (in preorder) the target octree down to the level of resolution (starting from the root and with all source black cubes, which are gotten from octree compaction or by simply listing the original source black cubes) and testing the traversed target nodes for intersection (using the exact source cube/target cube intersection test described in section 2.4.2) with the source cubes. A target node is only tested for intersection with the source cubes found to be intersecting with its parent node (i.e., the source cubes are passed down recursively from the root node to the target nodes with which they intersect). A non-voxel target node determines its color (black, white, or gray) depending on its intersections with the source cubes passed to it (black if the cube/cube intersection test returns complete intersection for one or more of the source cubes, white if the cube/cube intersection test returns no intersection for all source cubes, and gray if the cube/cube intersection test returns intersection for one or more source cubes). However, a non-voxel target node that initially determines itself to be gray in this way waits for its children to determine their colors before finally determining its own color; if the children are all white then the non-voxel target node sets itself to be colored white and if the children are all black then the non-voxel target node sets itself to be colored black (this is condensing the target octree). Target octree voxels are tested for intersection with source cubes using an application specific rule and determine themselves to be black or white depending on whether this rule returns intersection or no intersection.

To create the target octree in the DF-representation, each target node, upon determining its color, writes the symbol for its color (i.e., "(", "B", or "W") to the current location in an array (the current location is initialized to be element 1 of the array) and increments the current location to point to the next location in the array. However, a gray node, before incrementing the current location and recursing to its children, saves the current location; if it later changes itself to be white or black (due to condensing) it sets the current location to be the saved current location, sets the current location in the array to be its new color symbol ("W" or "B"), and increments the current location. After the target octree has been completely traversed in this way, the array will contain the DF-representation of the target octree.

2.4.4 Parallel Algorithm

The algorithm can be fairly easily parallelized, due to the many independent cube/cube intersection tests which are involved. Before the parallel algorithm is run, a precomputation step is run to divide the source black cubes evenly among the processors. In other words, if there are N processors then processor i will receive source black cubes i,$i+N$,$i+2N$,... After the precomputation step, each processor runs the serial algorithm on the source black cubes that it was assigned and creates a partial target octree. After all processors create a partial target octree, one of the processors creates the target octree by performing a union on all of the partial target octrees.

2.5 Octree Translation Only

For translation only, the transformed source cubes are axis-aligned. Thus, the test for intersection between a source and target cube is simply testing the source cube against the six face planes of the target cube to see if it is completely outside of one of them (i.e., this is just like checking bounding boxes for intersection). This is the conventional algorithm but, unfortunately, it doesn't take into account the fact that many of the target cubes' faces share the same face plane and so there are many redundant tests of source cubes against the same face planes. The most efficient way to perform octree translation is thus to test the source cubes against the face planes only once, storing the results, and then combining the results to create the target octree.

In particular, the main idea is to perform binary space subdivision in each of the x, y, and z dimensions separately for each source cube and then combine these results and add them to the target octree. In other words, successively divide the one dimensional space in half, starting from the space which extends from the minimum to maximum extent of the dimension, and determine on which side of the division the source cube lies—the side(s) on which the source cube lies are further subdivided and this continues to the level of resolution of the target octree. After the x, y, and z dimensions have been separately subdivided and compared against the source cube (for all source cubes), these results are combined by traversing the target octree starting from it's root (and down to the level of resolution) and determining whether the source cubes overlap any of the target cubes traversed (using the test described in the previous paragraph); however, the determination of which side of a face plane a source cube is on does not actually have to be calculated, but rather can be looked up from the results of the binary space subdivision. This method minimizes the number of source cube/face plane comparisons that must be done and results in a large speedup over the conventional approach (see section 2.6). To obtain the octree in the DF-representation, the algorithm for arbitrary motion (described in section

2.4.3) is used except that instead of using the cube/cube intersection test the binary space subdivision results are looked up to determine if there is intersection between source cubes and target cubes.

2.6 Experiments

We implemented the algorithm and a test environment on a Silicon Graphics 4D/340VGX, which is a shared-memory multiprocessor with four 33 MHZ MIPS R2000A/R3000 processors. We first did experiments for the arbitrary octree motion algorithms. All time measurements are for the time taken to create completely the target octree. As the application specific rule for target voxels, we determine that a target voxel intersects a source node if the center point of the target voxel (inverse transformed) is inside of the upright version of the source node; this is the rule that was used by [WA87][2] and we use it here for direct comparison with our algorithm[3]. As the test, we moved a space shuttle object (resolution level 5 source octree with 863 black nodes—458 black cubes after being compacted with the octree compaction algorithm described in section 2.3) with both translation and rotation motion for a number of cycles; at each cycle we measured the time that it took to create the target octree. Using this test, we compared the algorithm of [WA87] against a version of [WA87] which uses the exact source cube/target cube intersection test described above (i.e., other than that the algorithm is the same as [WA87]—note that these implementations both represent octrees using an explicit pointer-based representation). Then, we compared our proposed efficient arbitrary motion algorithm, with all features (i.e., compaction of octrees, use of linear octree representation, etc.), against [WA87]. We also implemented the parallel version of the proposed efficient algorithm and performed the test using 2, 3, and 4 CPUs. Figure 2.5 shows the space shuttle test object. The results of the experiments for the arbitrary octree motion algorithms can all be seen in figure 2.3. Also, at the last cycle (cycle 39) of the experiment the Weng and Ahuja algorithm performed 54007 cube/cube intersection tests while the same algorithm with the exact cube/cube intersection test performed only 43223 such tests; the numbers for the other cycles were similar.

Finally, we compare the conventional octree translation algorithm against our proposed efficient translation algorithm (both described in section 2.5). As the application specific rule for target voxels here, we determine intersection if any part of a target voxel intersects a source node. As the test, we use the same test as for the arbitrary motion test without the rotation component (i.e., move the space shuttle with the same translation component). Once again, at each cycle we measured the time that it took to create the target octree. Note that we tested the proposed efficient translation algorithm both with and without octree compaction; the result from without compaction shows that the proposed method truly is efficient (and not due to just the compaction). The results of the experiments for octree translation algorithms can be seen in figure 2.4.

2.7 Discussion

As can be seen from the figures, the arbitrary motion algorithm is quite efficient in comparison to the algorithm of [WA87]. Figure 2.3 shows that, as we stated previously, the algorithm of [WA87] works approximately 40% faster when it uses the exact cube/cube intersection test. Even better, our new algorithm runs nearly 4 times faster than [WA87]. In addition, the parallel algorithm achieved reasonable speedups. The parallel algorithm (with four processors) achieves about an eight times speedup over the algorithm of [WA87]. In addition, the optimized translation algorithm performs about 3 times better than the basic algorithm; note also that the translation algorithm is much faster than the arbitrary motion algorithm (for the same object and same translation, but without the rotation)—thus, if motion is only translation then big performance gains can be had by using the translation only algorithm instead of the arbitrary motion algorithm.

[HO87] also describes an algorithm for updating octrees undergoing arbitrary motion; it works similar to [WA87], except that it avoids condensing octrees by comparing traversed target nodes for intersection with both the black and white nodes of the source octree (i.e., if a target node intersects only black source nodes or only white source nodes then the node is known definitely to be black or white—no condensing is necessary). We did not implement this algorithm in order to compare it to ours. This is because, even allowing for speedups due to faster computer hardware, our algorithm performs better for similar sized octrees (i.e., compared to the performance figures given in [HO87], our algorithm performs more than 115 times better, and just better hardware most likely cannot make up for this).

[2]Because of this specific rule, when testing for intersection between a source cube and a non-voxel target cube it suffices to test for intersection between the source cube and a shrunken version of the non-voxel target cube which just contains all the center points of the voxels in the non-voxel target cube. This is a cube which has the same center point as the non-voxel target cube but whose side length is one less. This rule specific optimization is used in [WA87] and we also use it, but it is not generally applicable.

[3]Note, however, that to insure correctness in collision detection and robot path planning the rule must be to label a target voxel black if any part of it intersects with any transformed source node—our algorithm can easily and efficiently adapt to this rule whereas [WA87] cannot.

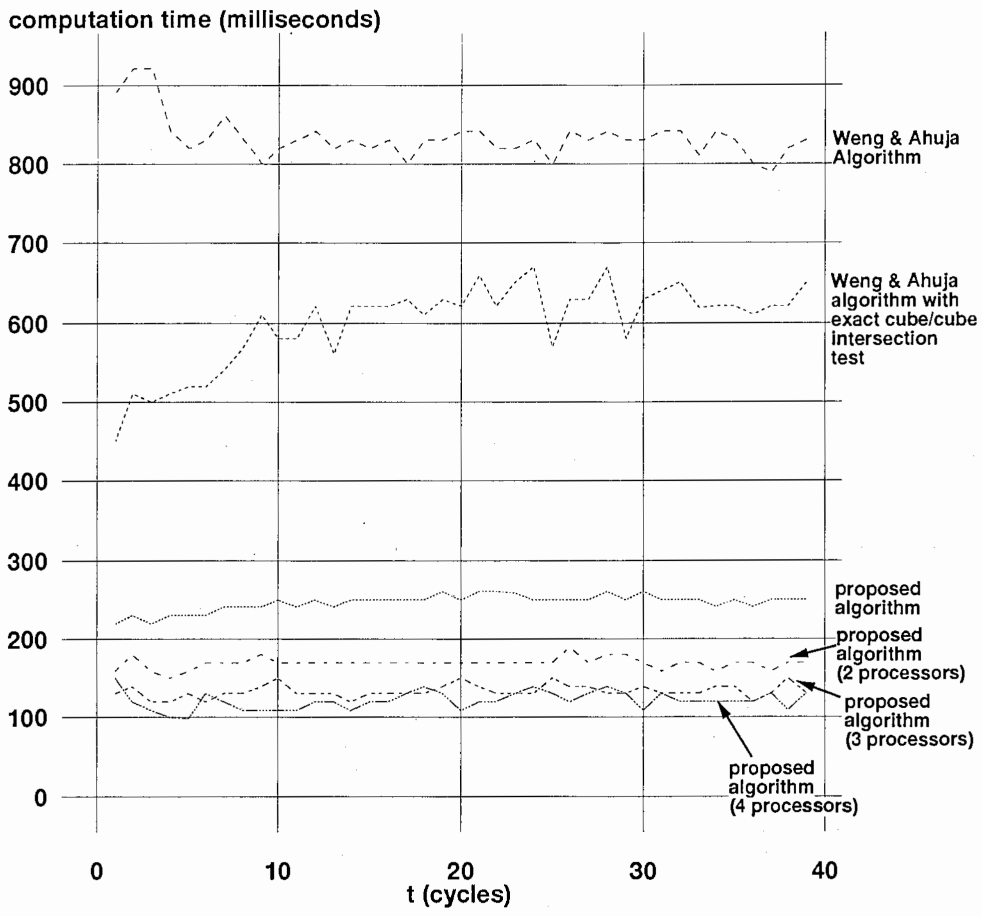


Figure 2.3: Results from the tests done for the arbitrary motion algorithms.

A final important characteristic of the new arbitrary motion algorithm is that it can optimize geometric search using octrees. In other words, in many applications the octree serves merely as an auxiliary object representation to some main representation which is actually visualized (e.g., boundary representation, constructive solid geometry). In this case the octree is used to speed up spatial access to the parts of the main representation. In this kind of an application, it is not always necessary to update completely (i.e., to voxel level) the octree for an object, but rather only until the necessary spatial access operation is complete. For example, in collision detection using octrees, if a non-voxel target node is found to be intersecting only source black cubes from one object then it is not necessary to check the child nodes of that target node (because there can be no collisions anywhere within that target node—only one object's source black cubes are contained within it). The new arbitrary motion of this chapter can easily handle this situation, whereas [WA87] cannot because it must traverse the target octree many times for each source black cube separately.

2.8 Conclusion

In this chapter, we have presented efficient algorithms for updating octrees undergoing both arbitrary motion and translation only motion. The arbitrary motion algorithm achieves efficiency by using a fast, exact cube/cube intersection test and by compaction of octrees as a precomputation step. An efficient parallel version of the arbitrary motion algorithm was also presented. For translation only, efficiency is achieved by testing source black cubes against face planes in the target octree only once and then combining the results to create the target octree. Both the arbitrary motion algorithm and the translation only algorithm can be used for octrees represented in the linear DF-representation.

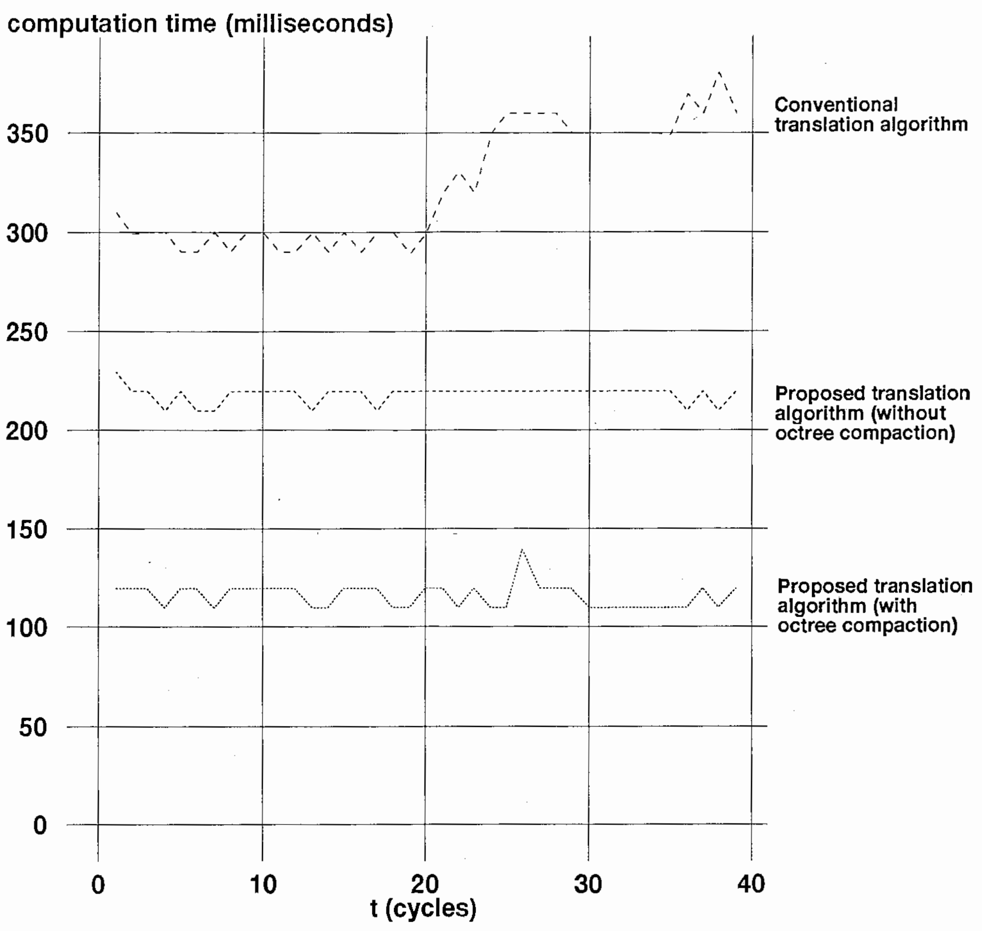


Figure 2.4: Results from the tests done for the translation motion only algorithms.

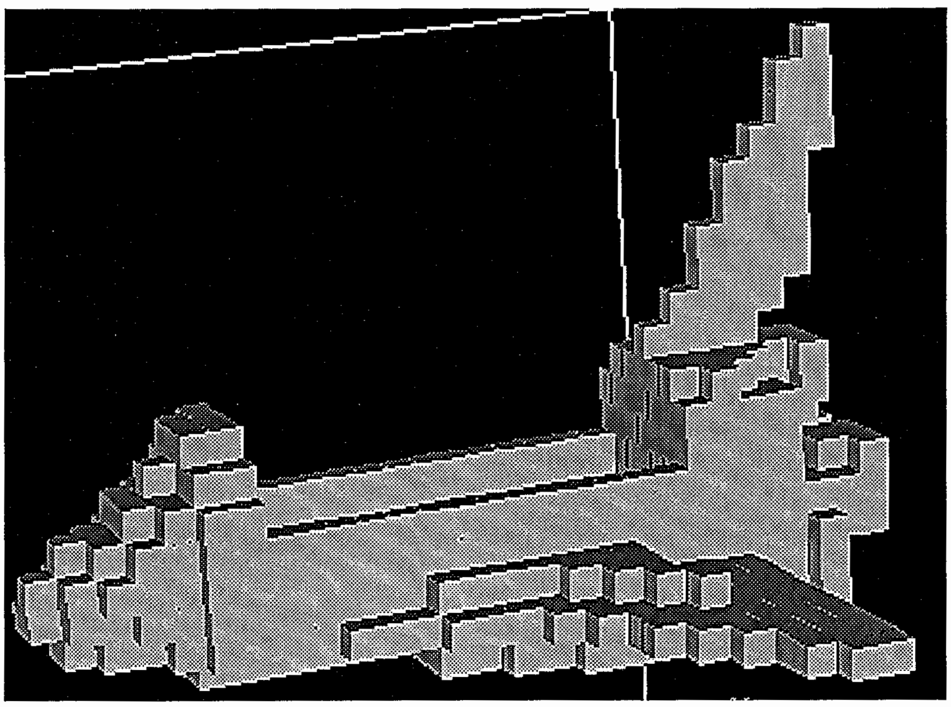


Figure 2.5: The space shuttle experimental object.

Chapter 3

A Simple and Efficient Method for Accurate Collision Detection Among Deformable Polyhedral Objects in Arbitrary Motion

Abstract

We propose an accurate collision detection algorithm for use in virtual reality applications. The algorithm works for three-dimensional graphical environments where multiple objects, represented as polyhedra (boundary representation), are undergoing arbitrary motion (translation and rotation). The algorithm can be used directly for both convex and concave objects and objects can be deformed (non-rigid) during motion. The algorithm works efficiently by first reducing the number of face pairs that need to be checked accurately for interference by first localizing possible collision regions using bounding box and spatial subdivision techniques; face pairs that remain after this pruning stage are then accurately checked for interference. The algorithm is efficient, simple to implement, and does not require any memory intensive auxiliary data structures to be precomputed and updated. Since polyhedral shape representation is one of the most common shape representation schemes, this algorithm should be useful to a wide audience. Performance results are given to show the efficiency of the proposed method.

3.1 Introduction

In a virtual environment, we can simulate various kinds of physical phenomena. An important example of this is being able to determine when moving objects collide; this is called the "collision detection problem." It is vitally important to be able to update a virtual environment at real-time rates to engender realism for a user. Unfortunately, current collision detection algorithms, if used, are an enormous bottleneck and make real-time update impossible [Pen90, Hah88]. The difficulty of collision detection for polyhedral objects can be seen by examining the basic, naive way of performing it. The basic method works by performing static intersection tests at discrete time instants; the time interval between tests is assumed small enough so that collisions are not missed. Then, interference among polyhedral objects at a time instant is detected by testing all combinations of faces and edges for the presence of an edge of one object piercing the face of another object; if such an edge-face pair exists then there is a collision [Boy79]. The average time complexity for this test (for n objects) is $O(n^2EF)$, where E and F are the number of edges and faces in the average object. As can be seen from this complexity figure, the problem lies in the necessity of having to perform such a large number of computationally expensive intersection tests at every time instant, where the number of such tests increases quadratically as the number and complexity of objects increase. For anything more than a simple world with a few objects of a few hundred faces each, this method is untenable in terms of maintaining real-time performance.

The main problem with the basic, naive collision detection method is that it requires such a large number of computationally expensive edge-face intersection checks. In an actual virtual world, the number of edge-face pairs that intersect at any time instant is a small percentage of the total number of possible pairs (in fact, much of the time there are no intersections). Thus, it is desirable to have a collision detection algorithm which checks a number of edge-face pairs proportional to the number that actually intersect. In this chapter, we present an algorithm that realizes this and can be used for general (i.e., the environment can contain both convex and concave objects), deformable polyhedral objects undergoing arbitrary motion [SKTK95].

The remainder of the chapter is organized as follows. The next section discusses other research efforts towards efficient collision detection. After that, the details of our collision detection algorithm are described. Next, experiments carried out using our algorithm are described, and performance results, showing the efficiency of the approach, are given. Finally, the last section concludes the chapter.

3.2 Efficient Collision Detection Approaches

There is much literature devoted to efficient collision detection approaches and this section discusses this research. The first subsection simply describes other approaches to efficient collision detection. Then, the last two subsections evaluate these other approaches, describing the problems with them which make them not entirely suitable for practical, large-scale virtual environments and how our algorithm addresses these problems.

3.2.1 Related Collision Detection Research

Much research on collision detection for polyhedra aims to drastically reduce the number of edge-face pairs that need to be checked for intersection. A common first step in many collision detection routines is an approximate bounding region (usually an axis-aligned box or a sphere) overlap test to quickly eliminate many objects as not interfering. An extension of this idea is to use a hierarchy of bounding regions to localize collision regions quickly [Hah88]. Related methods use octrees and voxel sets. [GSF94] stores a voxel data structure with each object, with pointers from voxels to polyhedra faces that intersect them. Collision is localized by testing for intersection of voxels between two objects. [KTK94] stores an octree for each object and, at each time instant, checks the interference of objects' updated octrees; face pairs from inside of interfering octree nodes are then checked for collision. Other voxel and octree methods include [MW88, Tur89, ZPOM93, SH92, Hay86].

Another method for collision detection involves keeping track of the distance between each pair of objects in the world; if the distance between a pair goes below some small threshold then the pair has collided. A noteworthy use of this idea for collision detection of rigid, convex objects is [LMC94], where coherence of objects between time instants (i.e., object positions change only slightly) and the property of convex polyhedra are used to detect collisions among objects in roughly constant time per object pair. Other research which uses this distance based approach include [GJK88, Qui94].

Briefly, some other approaches to collision detection are as follows. [BV91] uses a data structure called a "BRep-Index" (an extension of the well-known BSP tree) for quick spatial access of a polyhedron in order to localize contact regions between two objects. [Bar90] finds separating planes for pairs of objects; using object coherence, these separating planes are cached and then checked at succeeding time instants to yield a quick reply of non-collision most of the time. [SF91] uses ideas from the z-buffer visible surface algorithm to perform interference detection through rasterization. [Van94] uses back-face culling to remove roughly half of the faces of objects from being checked for detailed interference; the basic idea is that polygons of a moving object which do not face in the general direction of motion cannot possibly collide. [FHA90] uses a scheduling scheme, whereby object pairs are sorted by distance and only close objects are checked at each time instance. [Hub93] uses four dimensional space-time bounds to determine the earliest time that each pair of objects could collide and does not check the pair until then. [Pen90] models objects as superquadrics and shows how collision detection can be done efficiently using the inside/outside function of a superquadric. For coarse collision detection, [FPB94] stores bounding regions of objects in a stack of 2D structures similar to quadtrees (to reduce memory use) and uses only bit manipulations to add or delete objects to this (to reduce computation).

Finally, our algorithm uses ideas from methods for localized set operations on polyhedra [MT83, FK85]. These methods attempt to perform efficiently set operations, such as intersection, union, etc., on polyhedra by localizing the regions where faces are using spatial subdivision techniques; a set operation for a face then only needs to be done against the other faces inside the region that the face is in. As a particular example, the idea of intersecting faces with overlap regions of bounding boxes in order to localize the interference region of two objects was first described in [Mar72] and we use this idea effectively in our algorithm.

3.2.2 Evaluation

We evaluate the above algorithms on the basis of four properties of a collision detection algorithm necessary for effective use in a practical, large-scale virtual environment inhabited by humans. These are the ability to handle deformable (non-rigid) objects, the ability to handle concave objects, not using excessive amounts of memory for storing auxiliary data structures, and having better than $O(n^2)$ complexity for n objects in the world. None of the algorithms surveyed in the previous subsection has all four properties and some do not even have one of them. Our algorithm can satisfy all four of these properties.

Deformable Objects

In a virtual environment inhabited by humans, it is very important to be able to perform collision detection for objects which deform during motion. For example, in physical-based simulations forces between colliding objects are determined and the colliding objects are then deformed based on these forces. In general, a user should be allowed to deform objects in a virtual environment, which necessitates collision detection for deformable objects. Many of the above algorithms require precomputation and computationally expensive updating of auxiliary data structures (e.g., octrees, voxel sets, BRep-indices, etc.) for each object. This limits their usefulness because it means that objects are essentially limited to being rigid; this is because when an object deforms, its auxiliary data structures must be recomputed and this is usually an expensive operation. Our collision detection algorithm handles deformable objects.

Auxiliary Data Structures

In addition to being expensive to recompute, storing auxiliary data structures for each object can take up considerable memory. This limits the number of objects for which such algorithms can be effectively used. Our algorithm does not require any auxiliary data structures beyond simple bounding boxes and arrays.

Concave Objects

Another problem is that some of the above collision detection algorithms require objects to be convex [Bar90, LMC94, GJK88, Qui94]. However, it is clear that most objects of interest in the real-world are concave and a virtual environment, to be useful, should allow concave objects. To solve this problem, the above authors argue that a concave object can be modeled as a collection of convex pieces. While this can in fact be done for any concave object, it adds many fictitious elements (i.e., vertices, edges, etc.) to an object. In addition, breaking a concave object up into convex pieces means that the one object becomes many objects; unfortunately, this greatly worsens the complexity problem described in the next section (because each convex piece of the concave object must be treated as a separate object for the purposes of collision detection). Most importantly, however, any algorithm that requires objects to be convex or unions of convex pieces cannot be used to detect collisions for deformable objects; this is because, in general, deformations of an object easily lead to concavities. Our algorithm deals directly with concave objects in the same way as convex ones, with no extra computational overhead.

Complexity

The $O(n^2)$ complexity problem becomes apparent for large-scale virtual environments. [Pen90] discusses problems due to computational complexity in computer-simulated graphical environments and notes that collision detection is one such problem for which, in order to simulate realistically complex worlds, algorithms which scale linearly or better with problem size are needed. To understand the problem concretely, consider a collision detection algorithm that takes 1 millisecond per pair of objects. While for very small environments this algorithm is extremely fast, the algorithm is impractical for large-scale environments. For example, for an environment with just 50 objects 1225 pairwise checks between objects must be done, taking more than a second of computation; in this example, real-time performance cannot be maintained for environments with more than 14 objects (being able to compute something in 100 milliseconds or less is considered to be real-time performance [CMN83]). All of the distance based approaches [LMC94, GJK88, Qui94] and many of the others [Bar90, Pen90, GSF94] suffer from this complexity problem. In our experiments, we did use a bounding box test among objects which is $O(n^2)$ for n objects. However, the bounding box test between two objects is extremely fast and thus should not become a bottleneck unless there are many objects in the environment; for such an environment, however, the problem can be easily solved by using a bounding box check with better complexity ([Kir92] describes such a method) or by skipping the bounding box stage altogether and going directly to the face octree spatial subdivision stage which is $O(n)$ for n objects.

Other

A few other minor problems with the surveyed algorithms are as follows. Some of these algorithms [Tur89, Pen90] cannot be used for polyhedra, which limits their usefulness for current graphical applications where polyhedra dominate as the object representation. Some of the algorithms do not provide accurate collision detection (i.e., identify exactly which objects are interfering and which faces of the objects interfere— [KTK94] describes how this is useful for operator assistance) among objects [SF91, Hub93, FHA90, FPB94]. While most of the algorithms described above are clearly improvements over the basic, naive collision detection algorithm, none of them provide a solution to the problem that is as general, efficient, and simple as ours. The details of our collision detection algorithm are presented next.

3.2.3 Proposed Algorithm

Our proposed algorithm is an extension of the methods for localized set operations for use in collision detection. In particular, we extend the ideas in [MT83, FK85, Mar72] to a world with multiple objects; these papers describe algorithms for 2 objects but never precisely explain how to extend their algorithms efficiently to handle multiple objects (thus, direct use of these algorithms requires $O(n^2)$ complexity for n objects). In addition, these algorithms, in testing for intersection between a face and an axis-aligned box (while performing spatial subdivision), advocate using an approximate test between the bounding box of the face and the axis-aligned box; however, in developing our algorithm we found that using the exact intersection test described in a later section gave better performance (because it reduces the number of edge-face pairs even more, without much of an added computational cost). Also, these algorithms are used for performing static set operations; we show how they can be used for collision detection in a dynamic environment with multiple moving objects. Finally, we provide empirical evidence to show the efficiency of the proposed algorithm. The next section describes the details of our proposed collision detection algorithm.

3.3 Collision Detection Algorithm

3.3.1 Assumptions

All objects in the world are modeled as polyhedron (boundary representation). The faces of a polyhedron are assumed to be triangular patches without any loss of generality of range of representation. Objects can be concave or convex. The objects are undergoing motion which is not predetermined (e.g., a user can move his graphical hand in a sequence of non-predetermined, jerky motions); object motion can be both translation and rotation. Objects can be deformed during motion. Given this kind of environment, the goal is to be able to detect the colliding objects in the world and, in particular, the face pairs, between objects that are interfering. Collision will be checked for all objects at discrete time instants (i.e., at each time instant the new positions of objects will be determined, and collision will be checked for at that time instant *before* the computer graphic images of the objects are drawn to the screen). It is assumed that the speeds of objects are sufficiently slow compared with the sampling interval so that collisions are not missed. Finally, it is assumed that there is a large cube which completely bounds the world (i.e., that all objects will stay inside of this cube); let the side length of this cube be L.

3.3.2 Outline of the Method

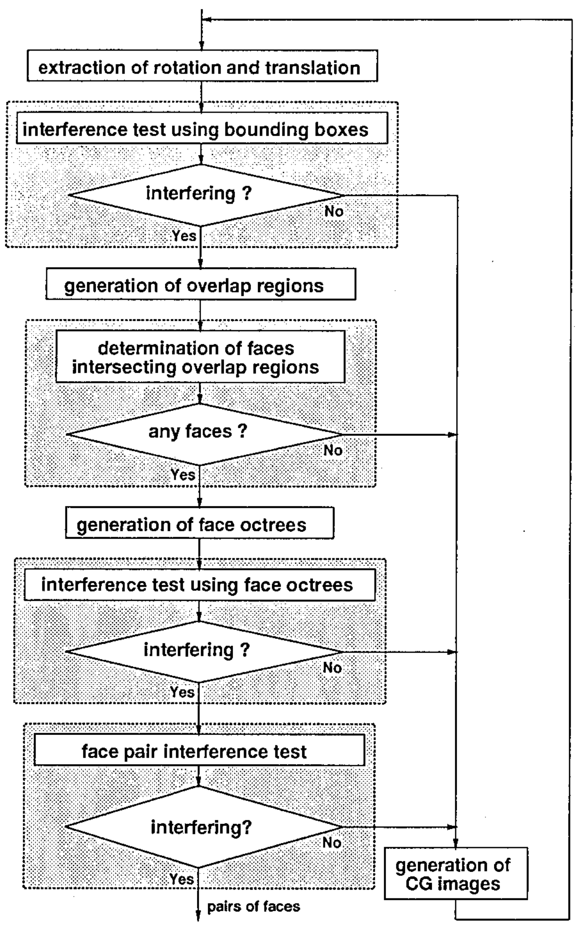
Figure 3.1 shows the control flow of our method. Suppose there are n objects in the workspace. The bounding boxes for each object are updated periodically (at discrete time instants $\cdots$, t_{i-1}, t_i, t_{i+1}, $\cdots$) using observed object motion parameters. Updated bounding boxes are checked for interference. For each object with an interfering bounding box all overlap regions of the object's bounding box with other objects' bounding boxes are determined. Next, for each such object all faces of the object are checked for intersection with the overlap regions of the object; a list of the object's faces which intersect one or more of the overlap regions is stored. Then, if there is a list of faces for more than one object, a face octree (i.e., an octree where a node is black if and only if it intersects faces) is built for the remaining faces (for all objects' face lists, together), where the root node is the world cube of side length L and the face octree is built to some user specified resolution. Finally, for each pair of faces which are from separate objects and which intersect the same face octree voxel (i.e., smallest resolution cube) it is determined whether the faces intersect each other in three-dimensional space. In this way, all interfering face pairs are found. Note that the intersection of faces with overlap regions and face octree stages repeatedly test for intersection of a face with an axis-aligned box; thus, we describe an efficient algorithm for testing this intersection.

3.3.3 Approximate Interference Detection Using Bounding Boxes

At every time instant, axis-aligned bounding boxes are computed for all objects and all pairs of objects are compared for overlap of their bounding boxes. For each pair of objects whose bounding boxes overlap, the intersection between the two bounding boxes is determined (called an overlap region as shown in Figure 3.2) and put into a list of overlap regions for each of the two objects. The overlap regions are passed to the next step.

3.3.4 Determination of Faces Intersecting Overlap Regions

For every object which has a list of overlap regions, all faces of the object are compared for intersection with the overlap regions. Once a face of an object is determined to be intersecting with at least one overlap region



extraction of rotation and translation
interference test using bounding boxes
interfering ?
No
Yes
generation of overlap regions
determination of faces intersecting overlap regions
any faces ?
No
Yes
generation of face octrees
interference test using face octrees
interfering ?
No
Yes
face pair interference test
interfering?
No
Yes
pairs of faces
generation of CG images

Figure 3.1: Control flow of collision detection.

it is placed in a face check list for the object. If there are face check lists for two or more objects then these are passed on to the next stage. Figure 3.3 shows an example of faces intersecting an overlap region.

3.3.5 Face Octree Spatial Subdivision Stage

A face octree is built down to a user specified resolution for the remaining faces starting from the world cube of side length L as the root. To minimize computation, only as much of the face octree as is necessary for collision detection is built; in particular, a parent node is subdivided into its 8 children only if it contains faces from two or more objects, and only the faces which were found to intersect the parent node are tested for intersection with the children nodes. Also, there is no condensation of the face octree (i.e., 8 black child nodes are not erased and replaced by their single, black parent node). If there are voxels in the face octree, then in each voxel there are faces from two or more objects. For each voxel, all possible pairs of faces, where the faces are from different objects, are determined and put into a face pair checklist. However, a face pair is only put into this face pair checklist if it was not previously put there by examination of another voxel. The face pair checklist is then passed to the next stage. Note that it is not necessary to allocate memory and actually build a face octree; faces can simply be checked for intersection with the standard cubes of an octree and checked recursively for lower-level cubes (thus no memory, beyond the small amount used by the stack during recursion, for storing octrees is necessary). Also note that a face octree is built for only a very small portion of all the faces; the previous stage eliminates most faces as not interfering.

3.3.6 Face Pair Interference Check

A pair of faces is checked for intersection at a time instant as follows. First, the bounding boxes of the faces are computed and checked for overlap; if there is no overlap in the bounding boxes then the faces do not intersect. Otherwise, the plane equation of the face plane of the first face is computed and the vertices of the second face are evaluated in this equation; if all vertices lead to the same sign (+ or -) then the second face is completely on one side of the face plane of the first face and thus there is no intersection. The plane equation of the face

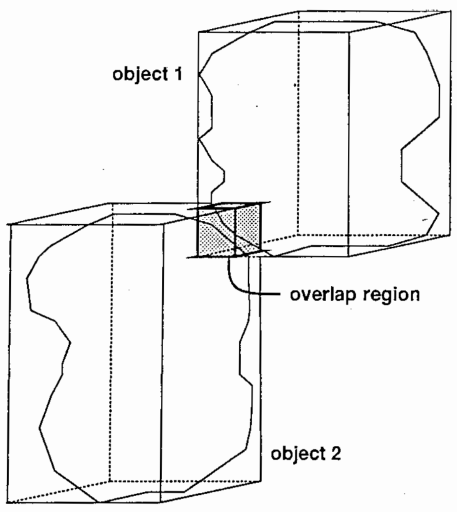


Figure 3.2: An overlap region

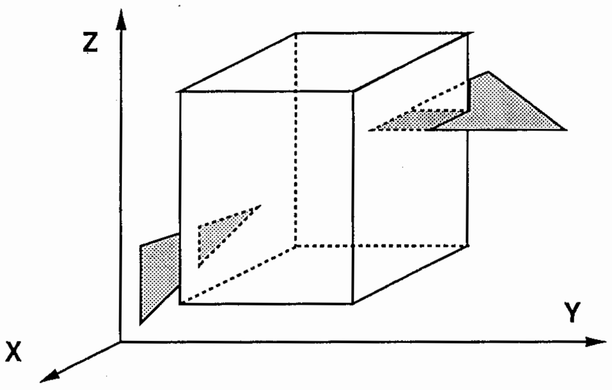


Figure 3.3: Faces intersecting the overlap region

plane of the second face is then determined and the vertices of the first face are evaluated in it in the same way. If neither face is found to be completely on one side of the face plane of the other face, then more detailed checks are done as follows. For each edge, in turn, of face 1 the intersection point of it with the face plane of face 2 is found and checked to see if it is inside face 2 (i.e., three-dimensional point-in-polygon check—the method used is described in [Arv91]); if the point is inside the face then the two faces intersect. The case when an edge and face plane are coplaner is handled by projecting the edge and face onto the two-dimensional coordinate axis most parallel to the face plane and performing a two-dimensional intersection check between the projected face and edge. In the same way, the edges of face 2 are checked for intersection with face 1. If no edges of either face are found to intersect the other face, then the two faces do not intersect.

3.3.7 Efficient Triangular Patch and Axis-Aligned Box Intersection Determination

To determine whether or not a triangular patch intersects with an axis-aligned box, we perform clipping against 4 of the face planes of the faces that comprise the box; the 4 face planes are the maximum and minimum extents of two of the three x,y,z dimensions (e.g., in our implementation we arbitrarily chose to clip against the maximum and minimum x extents and the maximum and minimum y extents). For the final dimension, it is only necessary to check whether or not the remaining vertices of the clipped triangular patch are either all greater than the maximum extent or all less than the minimum extent; if either case is true then there is no intersection, otherwise there is intersection. In addition, it is often not even necessary to clip against four planes. During clipping, whenever the intersection point of a segment with the current face plane is calculated this point can be quickly checked to see if it is inside of the face of the face plane; if it is inside, then the triangular patch and box intersect and no more computation needs to be done. Finally, before performing any clipping at all, two quick tests are done. As a first step, a quick overlap check between the bounding box

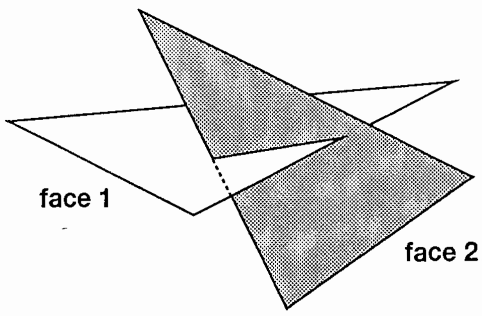


Figure 3.4: Face pair intersection test.

of the triangular patch and the axis-aligned box can be done to quickly determine non-intersection in many cases. Second, the three vertices of the triangular patch can be checked to see if one of them is inside of the axis-aligned box; if so, then the triangular patch and axis-aligned box intersect.

3.4 Experiments

The algorithm and an experimental environment were implemented and run on a Silicon Graphics Indigo^2 (this has an R4400/150 MHZ processor); experiments were done to determine the efficiency of the proposed algorithm. In all experiments described in this section, face octrees were built to resolution level 6.

3.4.1 Standardized Objects

For performance evaluation, sphere-like objects approximated by differing numbers of triangular patches were used; spheres were selected for testing because of their orientation invariance. Figure 3.5 shows some of the spheres which were used in the experiments. The basic experiment done was to have two identical sphere objects start at different (non-penetrating) positions and have them move towards each other (with both translation and rotation motion) until they interfere. This basic experiment was done with sphere objects having respectively 8, 10, 24, 48, 54, 80, 120, 168, 224, 360, 528, 728, 960, and 3968 triangular patches. Figure 3.6 shows the computation time required at each processing cycle from $t = 1(cycle)$, when there is no interference, until $t = 72(cycle)$, when faces from the two sphere objects are found to be intersecting, for four of the experimental sphere objects; at the last cycle, 70, 24, 16, and 11 milliseconds of computation are required to determine the colliding faces for the spheres with 3968, 960, 528, and 168 faces. Finally, figure 3.7 shows the computation required at the last stage (i.e., when faces from the two objects are found to be interfering—this requires maximum computation and is the true measure of the efficiency of a collision detection algorithm) of the proposed collision detection algorithm between two sphere objects against the number of triangular patches of the sphere objects.

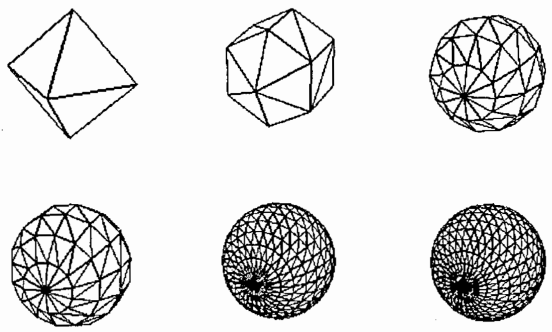


Figure 3.5: Examples of experimental objects (standardized spheres with different numbers of faces)

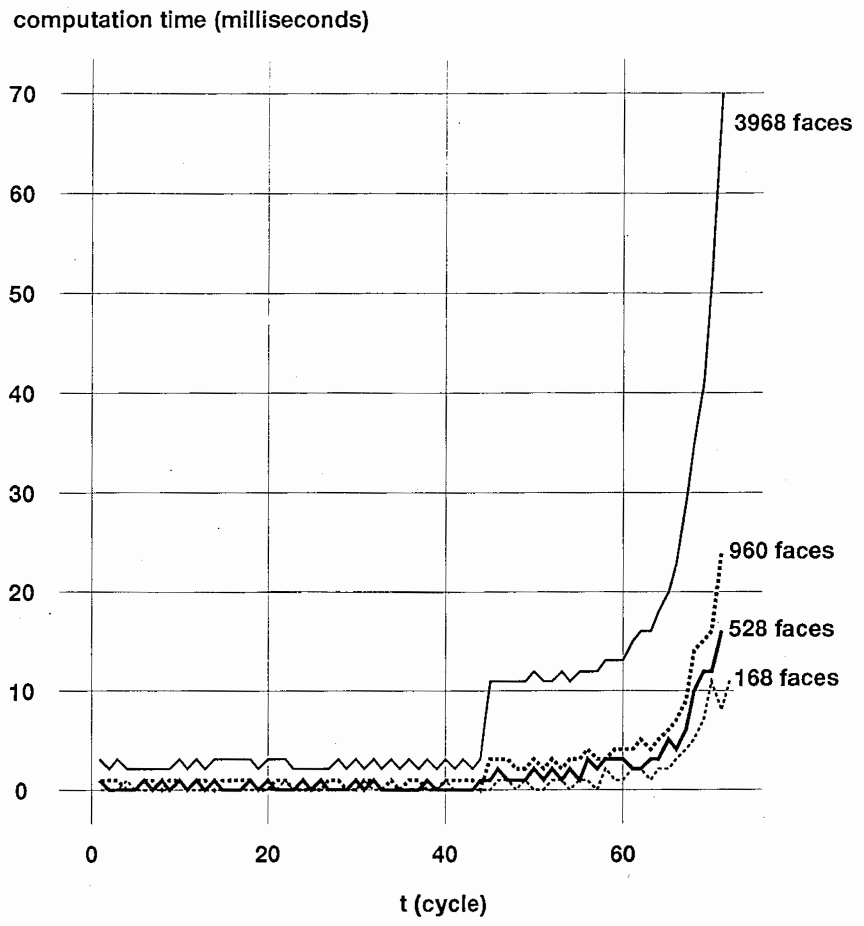


Figure 3.6: Computation time for each processing cycle for two identical sphere objects with 168, 528, 960 and 3968 faces.

3.4.2 Multiple General Objects

An experiment was also done with multiple general (i.e., concave—a real-world type of object) objects. Specifically, 15 identical objects (space shuttles with 528 triangular patches—see figure 3.12) were moved (both translation and rotation) in the test environment for many processing cycles and the computation time required at each cycle to perform collision detection was measured. At every cycle, many objects' bounding boxes were overlapping; thus, many triangular patches had to be tested for intersection with overlap regions at every cycle. At the last cycle of the test, faces from two objects were found to be interfering, taking 31 milliseconds of computation. Figure 3.8 shows the results of this experiment. Also in this figure, in order to provide a basis for comparison, are the results for when only the two interfering space shuttle objects are in the test environment; here, the last step, where faces are determined to be colliding, required 16 milliseconds of computation.

3.4.3 Comparison Against Competing Algorithms

In order to show that our algorithm is truly efficient, we directly compared the performance of our algorithm against two other competing algorithms. In general, it is difficult to make such direct comparisons because authors of collision detection papers do not normally give out the code that they used to get experimental results. Fortunately, however, we found the C language code for the first competing collision detection algorithm in [Hec94], and the second competing algorithm is a slight modification of the algorithm proposed in this chapter.

Separating Plane Algorithm

The first competing algorithm is based on ideas from [GJK88] and [Bar90]. This algorithm can only be used for convex, rigid objects and it does not return the list of face pairs that are interfering, as ours does. Thus, it is not completely fair to compare our algorithm against this algorithm because our algorithm is more general and

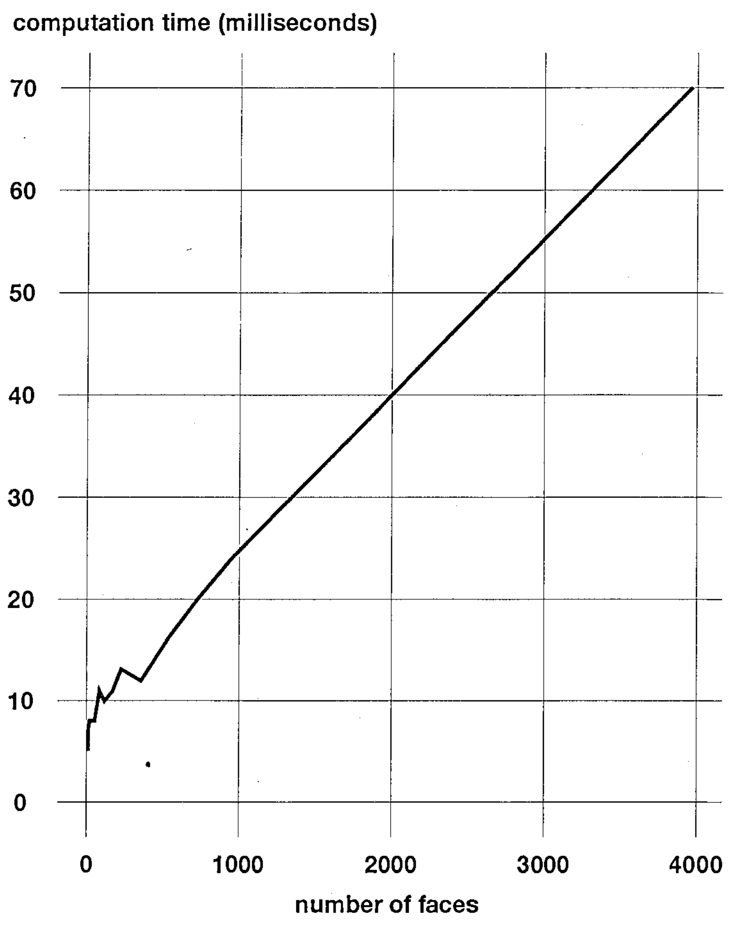


Figure 3.7: Computation time at the last stage of the proposed collision detection between two identical sphere objects against the number of planar patches of the objects.

gives more complete collision analysis. Even so, however, our algorithm gives better performance for non-trivial virtual environments.

The details of this competing algorithm are given in [Hec94]. However, briefly, the algorithm works by initially finding a separating plane between each pair of objects. A separating plane is found for two objects by finding the two closest vertices on the two objects (using the method in [GJK88]); the vector between these two points is the normal vector of the plane and the plane passes through one of the two points. Separating planes are cached between time instants and the previous time instant's separating plane is checked at the current time instant to see if it still separates the two objects; if it no longer separates then an attempt is made to find a new separating plane, which is then cached. If no new separating plane can be found then there is collision. Note that the complexity for this test (for n objects) is $O(n^2)$.

We compared our algorithm against this competing algorithm for environments containing differing numbers of same sphere objects (528 triangular patches). In particular, we tested both algorithms in environments with 10, 20, 30, and 40 moving sphere objects; at the last cycle of the tests two of the sphere objects were interfering. For 10 sphere objects, our algorithm performed roughly the same as the competing algorithm; in particular, our algorithm required 16 milliseconds of computation at the last cycle, while the competing algorithm required approximately 10 milliseconds per cycle. However, for 20, 30, and 40 objects our algorithm performed better. In particular, for 20, 30, and 40 objects, our algorithm required 21, 22, and 41 milliseconds at the last cycle; against this, the competing algorithm required approximately 35, 76, and 140 milliseconds per cycle. The results of these experiments can be seen in figure 3.9 (the competing algorithm's times are drawn with dotted lines, while the proposed algorithm's are drawn with solid lines).

Octree Update Algorithm

The second competing algorithm is a slight modification of the algorithm proposed in this chapter [KSTK94], and is representative of the bounding region hierarchy, octree and voxel approaches described in the section on related work (in fact, it is the algorithm described in more detail in the first chapter). Essentially, the

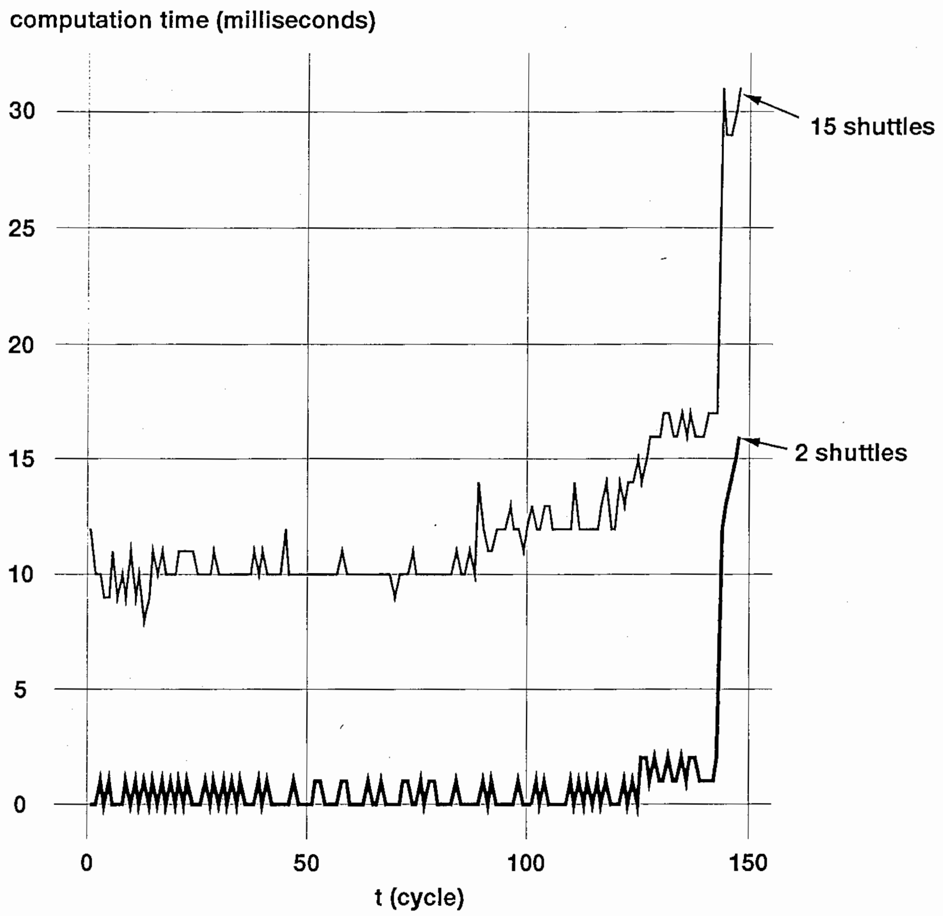


Figure 3.8: Computation time at each processing cycle for 15 space shuttle objects-collision between two objects is detected at the last cycle.

modification is to precompute complete face octrees for all of the polyhedral objects, and to store a list for each black node of the faces which intersect that black node. Then, the proposed collision detection algorithm is modified as follows. Instead of determining the polyhedral faces which intersect with overlap regions, the octree update algorithm determines the black nodes from the precomputed face octrees which intersect with the overlap regions; these intersecting black nodes are then put into a "node check list" (as opposed to a "face check list"). Then, in the next stage (face octree spatial subdivision stage), instead of creating a face octree by testing for intersections between the polyhedral faces in the face check list and the standard octree nodes, the octree update algorithm builds an octree by testing for intersections between the transformed (i.e., using the same transformation matrix as for the polyhedral objects) black nodes of the node check list and the standard octree nodes. Finally, for each standard octree voxel which was found to contain transformed black nodes from more than one object, all unique pairs of faces, where the faces are inside a precomputed face list of one of the transformed black nodes and the faces are from different objects, are enumerated and checked for intersection (using the method described in the face pair interference check section). Basically, the octree update algorithm substitutes precomputed face octree black nodes for faces in checking for intersection with overlap regions and standard octree nodes. Note that this algorithm can be used for concave objects, but that objects must be rigid; thus, it is not as general as the proposed algorithm.

We tested the proposed algorithm against this competing algorithm for the environment of figure 3.11; this environment contains a sphere (120 faces), a space shuttle (528 faces), a chair (146 faces), and a Venus head (1816 faces). The experiment that was performed was to move the venus head and the space shuttle towards each other (with translation and rotation) until they collided at the last cycle; the other two objects also translated and rotated slightly (without any collision). The proposed algorithm performed much better than the competing algorithm for all cycles after cycle 17 (when bounding boxes first overlapped); in particular, at the last cycle the competing algorithm required 161 milliseconds of computation, while the proposed algorithm required only 11 milliseconds (roughly 16 times better performance). Figure 3.10 shows the results of this experiment.

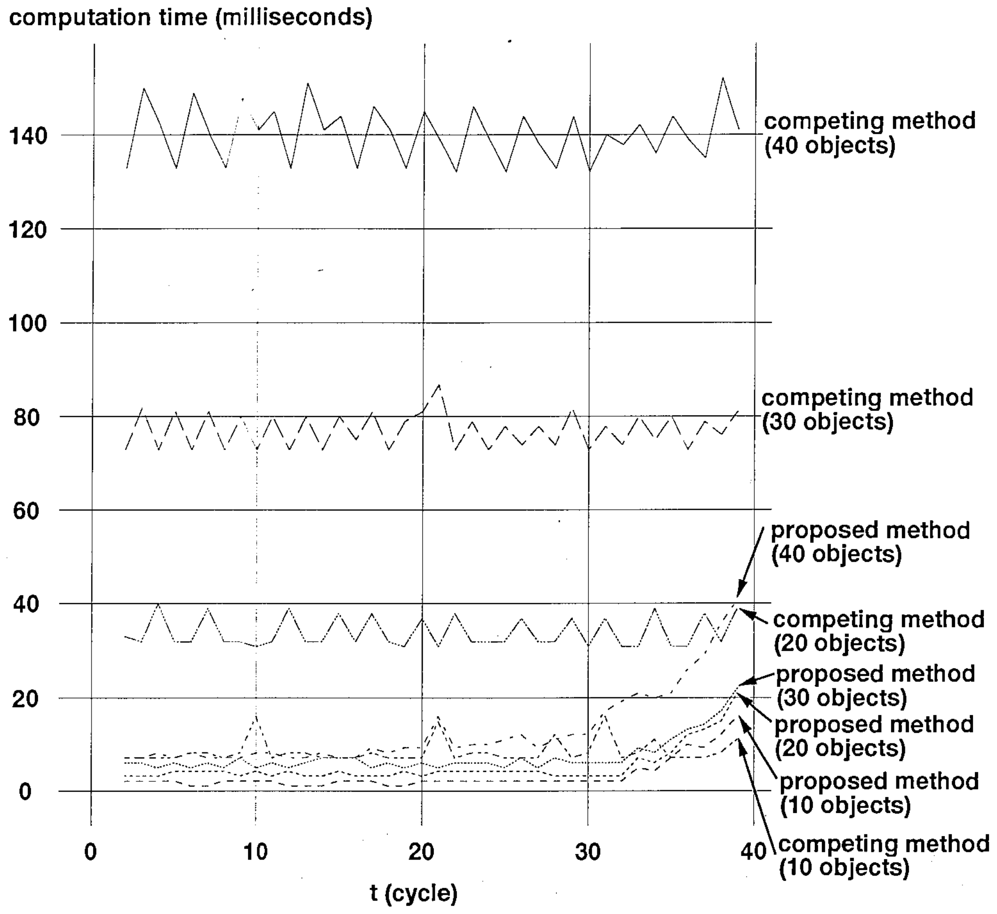


Figure 3.9: Computation time for each processing cycle for the proposed algorithm and the separating plane competing algorithm for 10, 20, 30, and 40 identical sphere objects (528 triangular patches each).

3.5 Discussion

As can be seen from the various graphs given, our collision detection algorithm is quite efficient. A common definition for "real-time" performance of a computer graphics application is being able to render 10 frames per second [CMN83]. Using this definition, our algorithm is able to perform real-time collision detection for objects having up to approximately 5936 faces (extrapolated from figure 3.7). Also important is the fact that the algorithm takes negligible compute time (rarely more than 10 milliseconds) when no objects in the environment are interfering. In addition, adding many objects to the environment increases computation time only slightly (i.e., for the case that only two objects at a time interfere—if more objects interfere at the same time then computation time will increase, but not greatly).

We did not implement the basic, naive collision detection algorithm in order to compare it to our algorithm (because our algorithm is clearly better—see [KTK94] and [SF91] to see how ludicrously long the naive algorithm can take for even very simple environments). The important basis of comparison should be with other authors accurate collision detection algorithms for general, deformable polyhedra; as shown in the section on related work, there are very few collision detection algorithms which providerevised this generality. We were not able to compare directly our algorithm against another competing algorithm which is as general as ours; however, even against the more restrictive algorithms of the previous section our algorithm gives better performance.

Based on these experiments, it seems reasonable to conclude that our algorithm would perform quite well in many applications. Unfortunately, however, we cannot assert, based solely on these experiments, that our algorithm is the fastest for all possible applications. There has already been much research into efficient collision detection, and many different efficient approaches have been proposed. We feel that, in addition to exploring new collision detection approaches, "comparative collision detection" would be a worthy new research topic. We feel that our proposed algorithm would fare well in such a comparative study, and we have made a start towards such research with our comparisons against two competing algorithms. However, more comprehensive research, which does more complete comparisons and which tests variations and combinations of the various

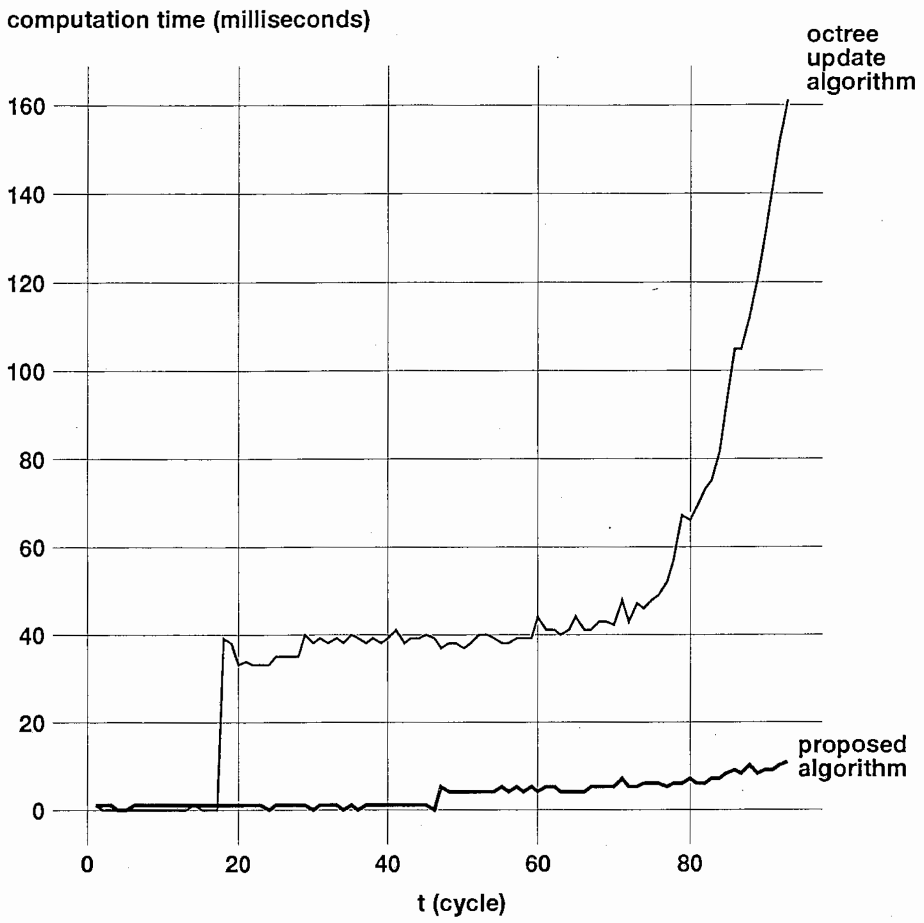


Figure 3.10: Computation time for each processing cycle for the proposed algorithm and the octree update competing algorithm for the environment of figure 3.11.

algorithms in situations that mimic real applications, is necessary. For the time being, however, we feel that, considering the generality of our algorithm, its ease of implementation, its small memory requirements, and its proven efficiency, we have provided a practical solution to the problem of real-time collision detection.

3.6 Conclusion

In this chapter, we have presented an efficient algorithm for accurate collision detection among polyhedral objects. The algorithm can be used for both convex and concave objects; both types of objects are dealt with in the same way and there is no performance penalty for concave objects. The algorithm can be used for objects whose motion is not prespecified, and both translation and rotation motion are allowed. The algorithm can also be used for objects that deform during motion. Thus, the algorithm is very general. The algorithm is fairly straightforward and should be easy to implement. The algorithm does not require the precomputation and update of memory intensive auxiliary data structures, which some collision detection algorithms require and which can sap the memory resources of an application, making it impossible to perform collision detection for a large number of objects. And finally and most importantly, even though the algorithm is very general it is extremely fast; Adding many objects to the environment does not require much more computation and the algorithm can run in real-time on a graphics workstation for polyhedra containing several thousands of faces.

We are currently exploring various optimizations to this algorithm, such as using face bintrees instead of face octrees, using a more efficient bounding box check (to reduce the $O(n^2)$ complexity for n objects), and determining the optimal level for face octree subdivision (the PM-octree [Sam90] might be useful for this). In addition, we are implementing a parallel version of the algorithm, which should be quite effective because of the many independent intersection calculations done by the algorithm. The algorithm is already sufficiently fast for most applications. However, with anticipated speedups from optimization and parallelization, our algorithm should be suitable for very large, practical virtual environments.

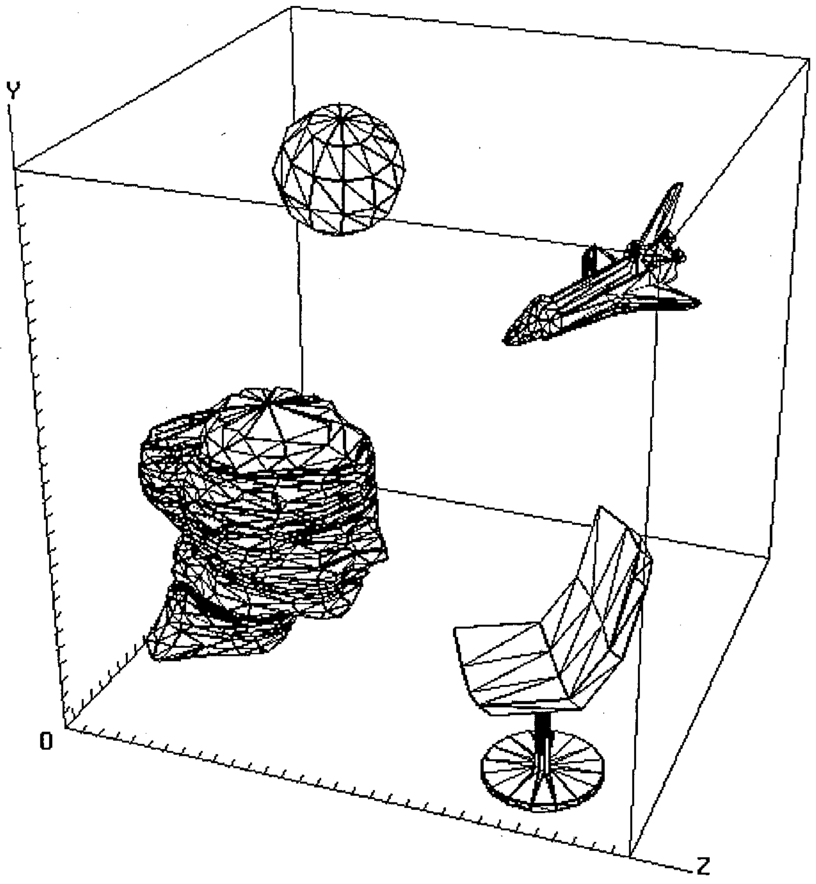


Figure 3.11: The experimental environment used to obtain the data for figure 3.10.

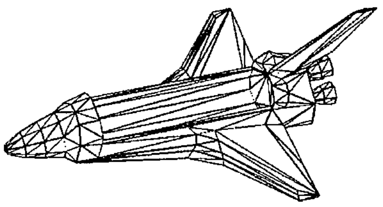


Figure 3.12: The space shuttle experimental object (528 triangular patches).

Chapter 4

Parallelization of Collision Detection

4.1 Introduction

This chapter describes parallelization strategies for a slightly modified, more accurate version of the collision detection algorithm described in the previous chapter. The basic algorithm is described in detail in the previous chapter, but as a brief review the four main steps are as follows:

1. Compute and store the intersections (called overlap regions) between objects' bounding boxes.
2. Determine the faces of objects which intersect with overlap regions.
3. Perform octree-like spatial subdivision on the faces from step 2 (found to be intersecting overlap regions) to localize possibly colliding faces into small sized (i.e., voxels) volumes of the world.
4. Check all unique pairs of faces, where the faces are from different objects and where the faces intersected with the same voxel from step 3, for interference.

The slightly modified version of this algorithm differs in the way that it computes the interference between faces in step 4. The method described in the previous chapter (called simple intersection method) was simply to determine whether or not the two faces intersect. The method used for the slightly modified version (called swept-space method) is to determine whether the spaces swept out by the two faces between time instants intersect. The swept-space method achieves better accuracy over the simple intersection test, because no collisions are missed between time instants (the simple intersection test only checks for collisions at the end of the time instant, not during a time instant). Unfortunately, however, this greater accuracy is also accompanied by a much greater computation time. Thus, in this chapter, we describe two parallelization strategies useful for enhancing the performance of the collision detection algorithm using the swept-space method; we also give experimental results showing the efficiency of the parallelization strategies.

4.2 Face Pair Swept-Space Interference Method

Face pairs are checked for interference between time instants as follows. At any time instant t_i, in order not to miss the collisions between time intervals, the possibility of collision between t_i and t_{i+1} is tested by considering the volume expected to be swept by each face during the interval $[t_i,\ t_{i+1}]$ (see Figure 4.1). To be conservative, collision is assumed if these volumes intersect even though such intersections are a necessary, but not sufficient, condition for the occurrence of collisions.

For each moving face A, we compute the convex hulls $V_A^{t_i}$ of a set of vertex points of A^{t_i} (i.e. $a_0^{t_i}$, $a_1^{t_i}$, $a_2^{t_i}$,...) and $A^{t_{i+1}}$ (i.e. $a_0^{t_{i+1}}$, $a_1^{t_{i+1}}$, $a_2^{t_{i+1}}$,...) (chapter 3 in [PS88]) which are expected to be swept by face A during the interval $[t_i,\ t_{i+1}]$. For each face B^{t_i} with which intersection of A^{t_i} is to be tested during the interval $[t_i,\ t_{i+1}]$, the convex hulls $V_B^{t_i}$ of a set of vertex points of B^{t_i} and $B^{t_{i+1}}$ are computed. Here, face A and face B at time $t = t_i$ are specified by A^{t_i} and B^{t_i}, respectively.

Then the intersection between $V_A^{t_i}$ and $V_B^{t_i}$ is tested. The intersection is detected by testing whether one of the following positional relationships of all combinations of faces and edges exists: both endpoints of an edge lie on the same side of the plane containing the face (Edge 1), an edge intersects the outside of the face plane (Edge 2), or an edge intersects the inside of the face plane (Edge 3). We detect an intersection in the case of Edge 3.

This identifies all pairs of faces that are expected to collide in the time interval $[t_i,\ t_{i+1}]$ by testing for collisions between faces in the face pair checklist. Figure 4.1 illustrates this method.

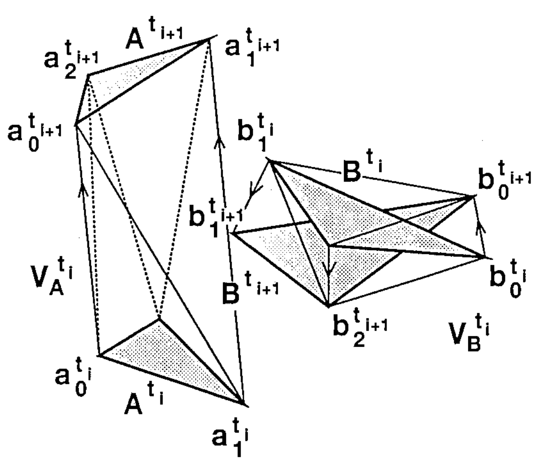


Figure 4.1: Face pair swept-space interference method.

4.3 Parallelization Strategy Consideration

To determine where parallelization effort should be effected, we measured the computation times of the four steps of the collision detection algorithm using the swept-space method. First, figure 4.2 shows the total computation time required at each cycle for two spheres starting at non-interfering positions and moving towards each other (with translation and rotation) until colliding at t=45 (cycle); this was done for two spheres with 960 faces and two spheres with 3968 faces. The important measurement is the last stage, when faces are found to be colliding; at this last stage the spheres with 960 faces required 126 milliseconds to determine that 21 out of 186 checked pairs were colliding, and the spheres with 3968 faces required 434 milliseconds to determine that 121 out of 1160 face pairs were colliding. Breaking up this computation time required at the last stage for the case of the spheres with 3968 faces into the time for each step of the algorithm gives the following times:

1. 2 milliseconds (0.5%)
2. 10 milliseconds (2.4%)
3. 58 milliseconds (13.2%)
4. 364 milliseconds (83.9%)

Similar percentages were obtained for the spheres with 960 faces each, and, in general, for other experiments. Thus, it is clear from these numbers that steps 3 and 4 dominate the total computation and should be the main focus of parallelization.

4.4 Parallelization of Swept-Space Method

For the swept-space method, steps 1 and 2 take such a small percentage of the computation time that it is best to do them serially (i.e., the parallelization overhead will probably be too much). Thus, the parallelization of the swept-space method concentrates on steps 3 and 4.

4.4.1 Single Program, Multiple Data Method

The simplest parallelization method follows the Single Program, Multiple Data (SPMD) [Ala87] abstract model of parallel computation in parallelizing only step 4; in SPMD, the processors are all running the exact same program, but on different data. The method is as follows. Perform steps 1, 2 and 3 serially (i.e., using just one processor). The output of step 3 will be a list of possibly interfering face pairs (i.e., face pairs for which both faces intersected the same voxel). Step 4 is then easily parallelized by dividing up the possibly interfering face pairs equally among the available processors; each processor then runs the same code to determine whether the face pairs assigned to it interfere or not.

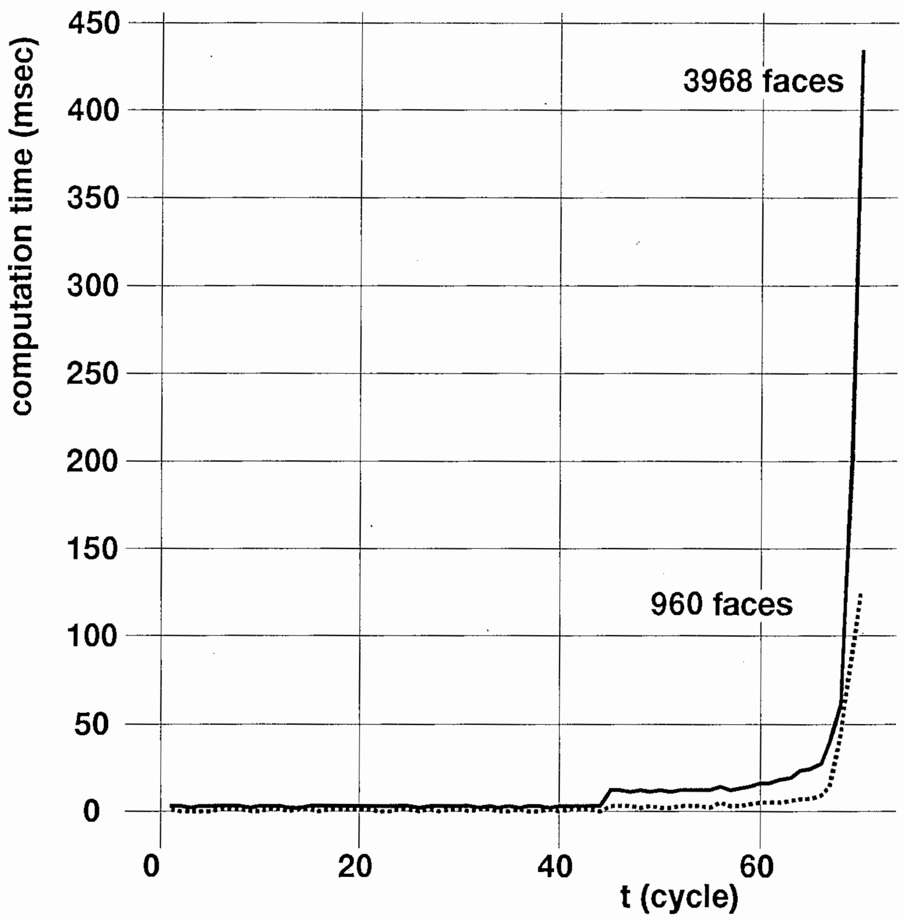


Figure 4.2: Computation required at every cycle for two spheres moving from non-colliding to colliding positions.

4.4.2 Producer-Consumer Method

The second method follows the Multiple Instruction, Multiple Data (MIMD) abstract model of parallel computation in parallelizing steps 3 and 4; this method should be faster because it parallelizes both steps 3 and 4. The method is based on the well-known parallel paradigm known as "producer-consumer" [Bar92]. In this general paradigm, one processor is the "producer" and produces items which the "consumers" grab and consume (i.e., do some computation on).

This second method works as follows. Perform steps 1 and 2 serially. Then, have one processor (producer) determine the possibly interfering face pairs (step 3); as soon as this processor finds a possibly interfering face pair it puts it on a list accessible by all the processors. The other processors (consumers) go directly to step 4, grab the possibly interfering face pairs from the list (as they are added to the list by the first processor), and check whether or not they interfere. After completing the list of possibly interfering face pairs, the producer becomes a consumer and helps to check whether any of the remaining face pairs interfere.

4.5 Implementations and Experimental Results

Both of these parallel algorithms were implemented using the shared-memory model of interprocessor communication; in this model, processors communicate by modifying variables that are accessible by all processors. The actual implementations were done in the C programming language, on a Silicon Graphics Onyx/Reality Engine with 24 150 MHZ R4400 RISC processors. The memory architecture of this machine is shared-memory, where each processor has a 16 Kbyte instruction cache, a 16 Kbyte data cache, and a 1 Mbyte secondary unified instruction/data cache; the main (shared) memory size is 512 Mbytes, and is 4-way interleaved. The parallelization was effected by using the Sequent compatible parallel programming primitive library [Bar92]. To effect parallelization using this library, the "m_fork" function is used to create multiple copies of a function and start them running on multiple processors; each processor then identifies itself by getting its ID using the "m_get_myid" function and performs unique computation based on this ID. In addition, the system function "sysmp" [Sil94] was used to schedule the processes to always run on the same processor; this was to take advantage of cache affinity (i.e., the fact that a process quickly fills up its cache with needed data—if the process is rescheduled to a new processor, it has to refill the cache of the new processor, which requires time consuming main memory accesses). In addition to being faster than rescheduling, this technique also caused the programs to run more smoothly (i.e., there were not wild variations in computation time at each step of the simulation).

The implementation of the first method was done by having the serial portion (i.e., steps 1,2, and 3) write the possibly interfering face pairs to an array accessible by all processors. Then, in the parallel portion, the face pairs are distributed evenly among the processors and each processor checks for interference of its face pairs. Note that the distribution of the face pairs can be done statically (i.e., processor N of P total processors checks interference for face pairs $N, N + P, N + 2P, N + 3P, ...$) or dynamically (i.e., the processors "grab" face pairs to check for interference from the array–note that this necessitates mutual exclusion overhead when grabbing, so that two processors don't grab the same face pair.) In general, dynamic distribution would be more effective if the time to test for intersection of individual face pairs varied greatly. However, since this is not the case we implemented the static distribution method. The implementation gave fairly good speedups and the computation time (at the last cycle, when faces were found to be colliding) versus the number of processors can be seen in figure 4.3.

The implementation of the second method gave better results, as expected. In this implementation, after steps 1 and 2 are finished serially, one processor finds the possibly interfering face pairs (step 3) and writes them to an array accessible by all processors. The other processors go directly to step 4 and wait for this array to fill up. These other processors grab face pairs as they are added by the first processor, and check them for interference (thus, the distribution of face pairs to processors is dynamic). The first processor, after creating the list of possibly intersecting face pairs, then goes on to step 4 and helps the other processors finish checking for intersection of the face pairs. This implementation gave very good speedups and the computation time (at the last cycle, when faces were found to be colliding) versus the number of processors can be seen in figure 4.3.

Notice that for all of these data sets, performance doesn't increase greatly for more than about 8 to 10 processors (and in fact, decreases for the producer-consumer method). This is not due to the algorithm, but is a general problem with shared-memory parallel architectures. Here, when more and more processors are used, they all compete for access to the shared bus (which only one processor can access at a time), and this creates considerable overhead. Silicon Graphics literature notes that, for most applications, the largest gains for parallelization are gotten from using between 4 and 8 processors [Sil94].

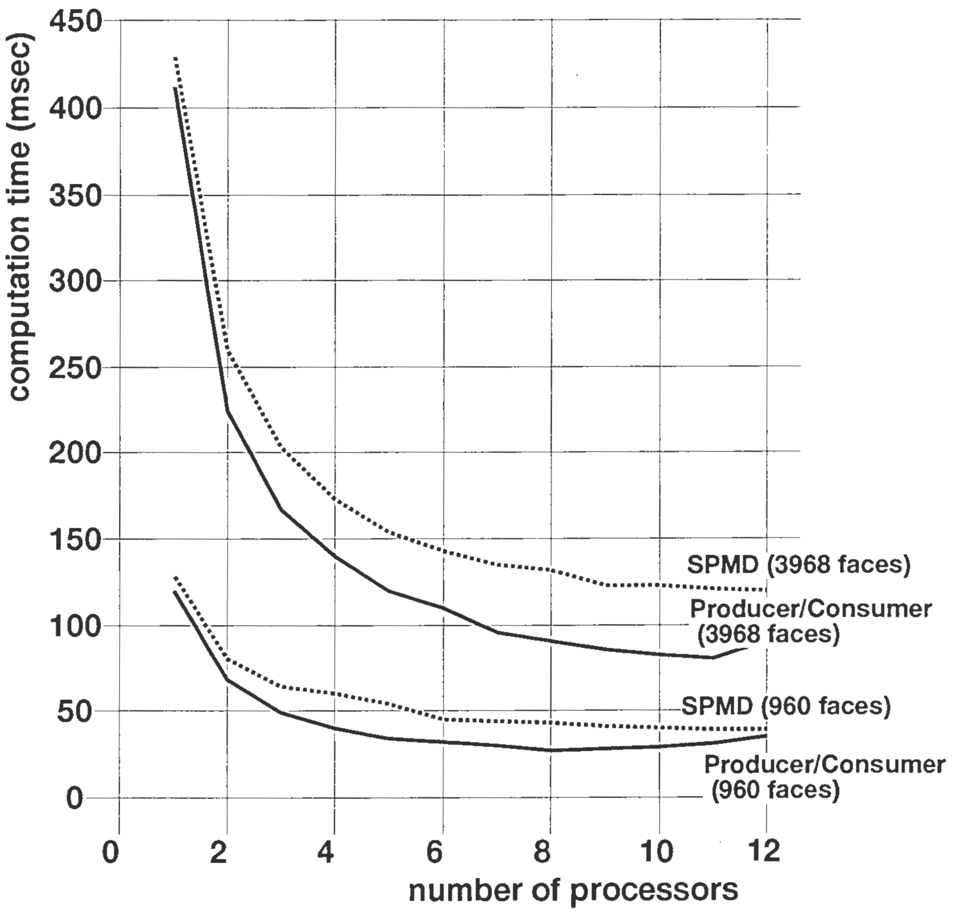


Figure 4.3: Computation time required at the last cycle by the SPMD and producer-consumer parallel methods, for two spheres moving from non-colliding to colliding positions.

Acknowledgements

I would like to thank Dr. K. Habara, Executive Vice President of ATR International and Chairman of the Board of ATR Communication Systems Research Laboratories, and Dr. N. Terashima, President of ATR Communication Systems Research Laboratories, for inviting me to ATR for 1 year. In addition, I would also like to thank them for allowing me to continue for an additional 2 and 1/2 months after this year so that I could continue my work, and so that I could present at the IAPR Machine Vision Applications Workshop in Kawasaki, Japan 1994.

I would like to express my sincere gratitude to Mr. F. Kishino, Head of the Artificial Intelligence Department, who welcomed me in to his department, gave me the opportunity to attend various symposia and conferences (Info-Tech '93, Multimedia '94, Machine Vision Applications '94), and set up a very interesting laboratory visit with NTT laboratories in Tokyo.

I would like to thank my advisor Mr. Y. Kitamura, Researcher, for his guidance and support of my work, and for his very helpful tips and advice about making effective presentations.

Finally, I would like to thank the members of the Planning Division and the Planning Section of ATR Communication Systems Research Laboratories for having organized such a wonderful stay in Japan for me.

Bibliography

[Ala87] Alan H. Karp. Programming for parallelism. *IEEE Computer*, pp. 43–57, May 1987.

[Arv91] Arvo, James, editor. *Graphics Gems II.* Academic Press Professional, 1991.

[Bar90] Baraff, David. Curved surfaces and coherence for non-penetrating rigid body simulation. *Computer Graphics*, Vol. 24, No. 4, pp. 19–28, 1990.

[Bar92] Barr E. Bauer, editor. *Practical Parallel Programing.* Academic Press, Inc., 1992.

[Boy79] Boyse, John W. Interference decision among solids and surfaces. *Communications of the ACM*, Vol. 22, No. 1, pp. 3–9, 1979.

[BV91] Bouma, W. and Vanecek, G. Collision detection and analysis in a physical based simulation. In *Eurographics Workshop on Animation and Simulation*, pp. 191–203, September 1991.

[CMN83] Card, S. K., Moran, T. P., and Newell, A. *The Psychology of Human-Computer Interaction.* Hillsdale, NJ: Lawrence Erlbaum Associates, 1983.

[FHA90] Foisy, A., Hayward, V., and Aubry, S. The use of awareness in collision prediction. In *International Conference on Robotics and Automation*, pp. 338–343. IEEE, 1990.

[FK85] Fujimura, K. and Kunii, T. A hierarchical space indexing method. In *Visual Technology and Art (Computer Graphics Tokyo)*, pp. 21–33, 1985.

[FPB94] Fairchild, K. M., Poston, Timothy, and Bricken, William. Efficient virtual collision detection for multiple users in large virtual spaces. In *Virtual Reality Software and Technology*, 1994.

[GJK88] Gilbert, Elmer G., Johnson, Daniel W., and Keerth, S. Sathiya. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal of Robotics and Automation*, Vol. 4, No. 2, pp. 193–203, 1988.

[Gla90] Glassner, Andrew S., editor. *Graphics Gems.* Academic Press Professional, 1990.

[GSF94] Garcia-Alonso, A., Serrano, N., and Flaquer, J. Solving the collision detection problem. *Computer Graphics and Applications*, Vol. 14, No. 3, pp. 36–43, May 1994.

[Hah88] Hahn, James K. Realistic animation of rigid bodies. *Computer Graphics*, Vol. 22, No. 4, pp. 299–308, 1988.

[Hay86] Hayward, V. Fast collision detection scheme by recursive decomposition of a manipulator workspace. In *International Conference on Robotics and Automation*, pp. 1044–1049. IEEE, 1986.

[Hec94] Heckbert, Paul, editor. *Graphics Gems IV.* Academic Press Professional, 1994.

[HO87] Hong, T.H. and Oshmeier, M. Rotation and translation of objects represented by octree. In *International Conference on Robotics and Automation*, pp. 947–952. IEEE, 1987.

[Hub93] Hubbard, Philip M. Interactive collision decision. In *Symposium on Research Frontiers in Virtual Reality*, pp. 24–31. IEEE, 1993.

[Kir92] Kirk, David, editor. *Graphics Gems III.* Academic Press Professional, 1992.

[KSTK94] Kitamura, Y., Smith, A., Takemura, H., and Kishino, F. Optimization and parallelization of octree-based collision detection for real-time performance. In *IEICE Conference 1994 Autumn.* D-323, 1994. (in Japanese).

[KTAK94] Kitamura, Y., Takemura, H., Ahuja, N., and Kishino, F. Efficient collision detection among objects in arbitrary motion using multiple shape representations. In *12th International Conference on Pattern Recognition Jerusalem, Israel*, 1994.

[KTK94] Kitamura, Y., Takemura, H., and Kishino, F. Coarse-to-fine collision detection for real-time applications in virtual workspace. In *International Conference on Artificial Reality and Tele-Existence*, pp. 147–157, July 1994.

[LMC94] Lin, M. C., Manocha, D., and Canny J. F. Fast contact determination in dynamic environments. In *International Conference on Robotics and Automation*, pp. 602–608. IEEE, 1994.

[Man88] Mantyla, Martti. *An introduction to solid modeling.* Computer science express, 1988.

[Mar72] Maruyama, K. A procedure to determine intersections between polyhedral objects. *International Journal of Computer and Information Sciences*, Vol. 1, No. 3, pp. 255–266, 1972.

[MT83] Mantyla, M. and Tamminen, M. Localized set operations for solid modeling. *Computer Graphics*, Vol. 17, No. 3, pp. 279–288, July 1983.

[MW88] Moore, M. and Wilhelms, J. Collision detection and response for computer animation. *Computer Graphics*, Vol. 22, No. 4, pp. 289–298, 1988.

[Pen90] Pentland, Alex P. Computational complexity versus simulated environments. *Computer Graphics*, Vol. 24, No. 2, pp. 185–192, 1990.

[PS88] Preparata, F. P. and Shamos, M. I. *Computational geometry, an introduction.* Springer-Verlag, 1988.

[Qui94] Quinlan, Sean. Efficient distance computation between non-convex objects. In *International Conference on Robotics and Automation*, pp. 3324–3329. IEEE, 1994.

[Sam90] Hanan Samet. *The design and analysis of spatial data structures.* Addison-Wesley, 1990.

[SF91] Shinya, M. and Forgue, M. Interference detection through rasterization. *The Journal of Visualization and Computer Animation*, Vol. 2, pp. 132–134, 1991.

[SH92] Shaffer, C. A. and Herb, G. M. A real-time robot arm collision avoidance system. *IEEE Transactions on Robotics and Automation*, Vol. 8, No. 2, pp. 149–160, 1992.

[Sil94] Silicon Graphics, Inc., Mountain View, CA USA. *Silicon Graphics Online Documentation*, 1994.

[SKK94] Smith, A., Kitamura, Y., and Kishino, F. Efficient algorithms for octree motion. In *IAPR Workshop on Machine Vision Applications*, 1994.

[SKTK95] Smith, Andrew, Kitamura, Yoshifumi, Takemura, Haruo, and Kishino, Fumio. A simple and efficient method for accurate collision among deformable polyhedral objects in arbitrary motion. In *Virtual Reality Annual International Symposium, North Carolina, USA*. IEEE, March 1995.

[Tur89] Turk, Greg. Interactive collision detection for molecular graphics. M.sc. thesis, Department of Computer Science, University of North Carolina at Chapel Hill, 1989.

[Van94] Vanecek, George. Back-face culling applied to collision detection of polyhedra. Technical report, Purdue University Department of Computer Science, 1994.

[WA87] Weng, Juyang and Ahuja, Narendra. Octrees of objects in arbitrary motion: Representation and efficiency. *Computer Vision, Graphics, and Image Processing*, Vol. 39, No. 2, pp. 167–185, 1987.

[ZPOM93] Zyda, M. J., Pratt, D. R., Osborne, W. D., and Monahan, J. G. NPSNET: Real-time collision detection and response. *The Journal of Visualization and Computer Animation*, Vol. 4, No. 1, pp. 13–24, 1993.