MVA'94 IAPR Workshop on Machine Vision Applications Dec. 13-15, 1994, Kawasaki

EFFICIENT ALGORITHMS FOR OCTREE MOTION

Andrew SMITH, Yoshifumi KITAMURA , and Fumio KISHINO
ATR Communication Systems Research Laboratories

2-2 Hikaridai, Seika-cho, Soraku-gun, Kyoto, 619-02, Japan
<asmith, kitamura, kishino>@atr-sw.atr.co.jp

Abstract

This paper 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 paper 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.

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 [1]. 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 [2, 3].

This paper deals with the problem of efficiently updating moving octrees. The paper 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 [2]. The paper describes a new, more efficient octree arbitrary motion algorithm and provides data showing that the algorithm runs 3 to 4 times faster than [2]. 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 paper 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 paper 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 paper are also space efficient; this is because they can be used for octrees represented in the linear octree representation.

2 Octree Shape Representation

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 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 [4]. 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 1 has the DF-representation "(B(BWBBBWWWBWBWB(B(BWBBBWBWBBB WBW"

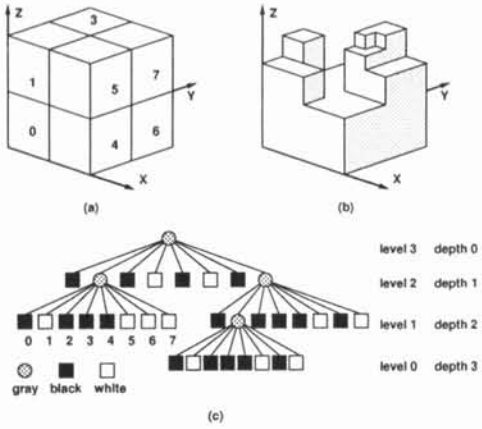

Figure 1: The octree shape representation. (a) The ordering of octants (b) An example octree (c) Pointer-based representation of the example octree.

2.2 Octree Motion Basic Algorithm

The basic algorithm for moving an octree [2] (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 (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.

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

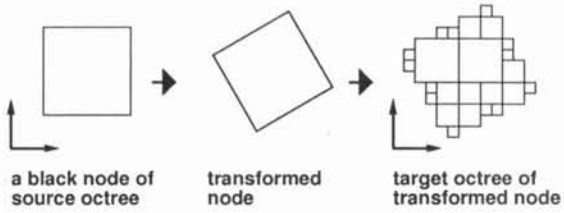

Figure 2: The octree motion basic algorithm illustrated for the 2D case (quadtree).

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. In this paper, 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 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).

4 Arbitrary Octree Motion

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. [2] 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, [2] 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, [2] claims that using an approximate intersection test is the efficient way to move octrees; however, an important result of this paper is that using the exact cube/cube intersection test described in this paper 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.

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

[1]The actual sphere/cube intersection test used is described on page 335 of [5]; 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.

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.

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 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.

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 $i, i+N, i+2N, \ldots$ 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.

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 6). To obtain the octree in the DF-representation, the algorithm for arbitrary motion (described in section 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.

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 [2][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 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 [2] against a version of [2] which uses the exact source cube/target cube intersection test described above (i.e., other than that the algorithm is the same as [2]—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 [2]. We also implemented the parallel version of the proposed efficient algorithm and performed the test using 2, 3, and 4 CPUs. Figure 5 shows the space shuttle test object. The results of the experiments for the arbitrary octree motion algorithms can all be seen in figure 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 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 4.

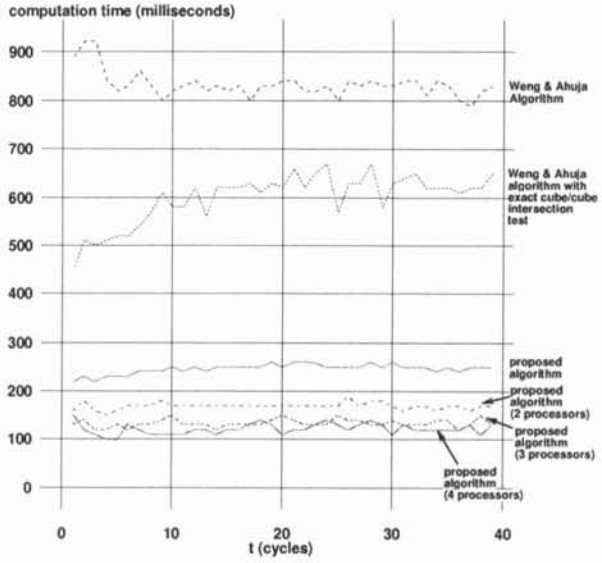

Figure 3: Results from the tests done for the arbitrary motion algorithms.

7 Discussion

As can be seen from the figures, the arbitrary motion algorithm is quite efficient in comparison to the algorithm of [2]. Figure 3 shows that, as we stated previously, the algorithm of [2] works approximately 40% faster when it uses the exact cube/cube intersection test. Even better, our new algorithm runs nearly 4 times faster than [2]. In addition, the parallel algorithm achieved reasonable speedups. The parallel algorithm (with four processors) achieves about an eight times speedup over the algorithm of [2]. 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.

[3] also describes an algorithm for updating octrees undergoing arbitrary motion; it works similar to [2], except that it avoids condensing octrees by comparing traversed target nodes for intersection with both the

[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 [2] 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 [2] cannot.

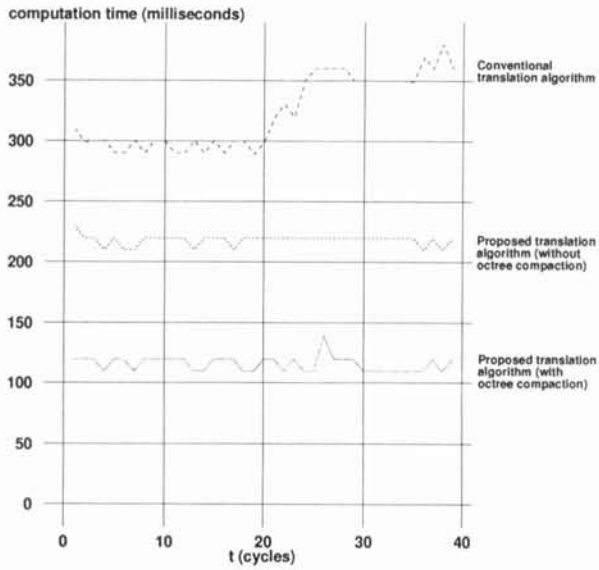

Figure 4: Results from the tests done for the translation motion only algorithms.

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 [3], our algorithm performs more than 115 times better, and just better hardware most likely cannot make up for this).

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 paper can easily handle this situation, whereas [2] cannot because it must traverse the target octree many times for each source black cube separately.

8 Conclusion

In this paper, 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.

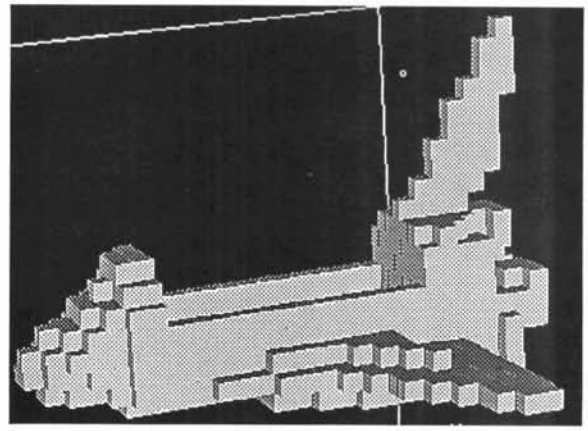

Figure 5: The space shuttle experimental object.

References

[1] 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.

[2] 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.

[3] 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.

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

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