A real-time robot arm collision detection system

Article · July 1990

Source: NTRS

CITATIONS
4

READS
134

2 authors, including:

Clifford A. Shaffer
Virginia Polytechnic Institute and State University
205 PUBLICATIONS **2,468** CITATIONS

SEE PROFILE

A Real-Time Robot Arm Collision Detection System

By Clifford A. Shaffer and Gregory M. Herb

TR 90-28

A REAL TIME ROBOT ARM COLLISION DETECTION SYSTEM

Clifford A. Shaffer
Gregory M. Herb

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

ABSTRACT

A data structure and update algorithm are presented for a prototype real time collision detection safety system for a multi-robot environment. The data structure is a variant of the octree, which serves as a spatial index. An octree recursively decomposes three dimensional space into eight equal cubic octants until each octant meets some decomposition criteria. Our octree stores cylspheres (cylinders with spheres on each end) and rectangular solids as primitives (other primitives can easily be added as required). These primitives make up the two seven-degrees-of-freedom robot arms and environment modeled by the system. Octree nodes containing more than a predetermined number N of primitives are decomposed. This rule keeps the octree small, as the entire environment for our application can be modeled using a few dozen primitives. As robot arms move, the octree is updated to reflect their changed positions. During most update cycles, any given primitive does not change which octree nodes it is in. Thus, modification to the octree is rarely required. Incidents in which one robot arm comes too close to another arm or an object are reported. Cycle time for interpreting current joint angles, updating the octree, and detecting/reporting imminent collisions averages 30 milliseconds on an Intel 80386 processor running at 20 MHz.

Keywords and Phrases: hierarchical data structures, octrees, collision detection, collision avoidance, tele-operated robots.

June 11, 1990

This work was partially supported by NASA/Goddard SFC under Grant NAG 5-1183.

1. INTRODUCTION

This paper describes the use of a hierarchical data structure, the *N-Objects* octree, in a collision detection safety system for a multi-robot work environment. The goal is not to perform robot arm path planning, but rather to support a real-time safety system to warn of imminent collisions between two robot arms or between a robot arm and objects in the environment. The algorithms described can be used to provide a collision detection capability for a variety of robotics applications.

Our test environment is modeled on NASA's proposed space station operating bay containing two tele-operated robot arms and various objects to be manipulated and avoided by the arms. A tele-operated robot is one whose motions are dictated by an operator through a controlling device. In this case a mini-master, which is a scaled-down model of the arm, is manipulated by the operator to move the robot arm. Given the working conditions of the operator and the tasks being performed, the probability of an accident occurring is unacceptably high. Hence, a safety system is needed which will prevent unintentional collisions. Ideally, a collision avoidance system will provide safe operation of the robot arms without hindering the operator.

Each robot arm in our application has seven degrees of freedom, with each section of the arm being nearly cylindrical in shape. The operating environment is not static – the system must accommodate movement of both the arms and other objects. Nor are motions predetermined. The operating paradigm is one of receiving an indication of movement by certain objects, updating the representation of the environment to reflect that movement, and reporting any imminent collisions.

From the above characterization of the problem, we see that the collision detection system requires the following of its representation. First, the representation must determine in real time if robot arms/objects are about to collide. Second, the representation must constantly be updated, adjusting to the movements of the robot arms and objects. Both updating and collision detection must consistently be performed within the permitted time period to be acceptable as a real time

safety system. Third, the representation must be a reliable, but not necessarily exact model. That is, since we are trying to warn of and avoid imminent collisions, exact representation of the objects is not required — as long as the approximation does not lead to missing imminent collisions, nor leads to reporting too many false warnings.

Quad- and octree [Same89, Same90] have become popular data structures for applications in computer vision, robotics, computer graphics, and Geographic Information Systems as well as other related disciplines. The octree is a hierarchical data structure that recursively subdivides a cubic volume into eight smaller cubes (called octants) until a certain criterion, known as the decomposition rule, is met. This decomposition process is often represented as a tree of out-degree eight as shown in Figure 1. Changing the decomposition rule gives rise to many varieties of octrees. The most well known form is the *region octree*. It is most appropriate for defining the shapes of homogeneous objects which are difficult to model with higher-level primitives. Beginning with a cube that encloses the set of objects to be modeled, splitting occurs until each octant lies completely within an object or is completely empty (see Figure 1).

Another type of octree, which we here refer to as the *N-Objects* octree, is widely used for ray tracing of images to simulate realistic lighting effects [Glas84, Glas89, MacD89]. The N-objects octree subdivides space into octants, as does the region octree. However, the N-objects octree stores a list of the objects that inhabit each node. Beginning with a cube that encloses all of the objects, splitting occurs until no more than N objects lie in any leaf node. The value we choose for N may depend on the characteristics of the environment being modeled and the task being performed.

We use the N-Objects octree to serve as the underlying representation of our collision detection system for a number of reasons. The N-Objects octree provides a compact representation of the environment. An appropriate decomposition rule keeps the size of the tree small which, as we shall see, allows for fast updating. The N-Objects octree is a dynamic structure which easily adjusts to changes in the environment through the splitting and merging of octants. Equally important to

the efficiency of the system, small object motions rarely require that the structure of the tree be changed. As a result, our system operates with an average cycle time of 30 msec (and under 60 msec in the worst case) for our test simulations.

2. PREVIOUS WORK

Many researchers have studied path planning problems in moving environments (for overviews, see [Shar89, Whit85]). With the expanded use of tele-operated robots in space, manufacturing, and the nuclear industry, the problem of collision detection and prevention has become a significant problem in its own right. Here we describe previous work relating to various aspects of our representation.

One popular approach to path planning and collision avoidance is to model the robot's workspace in terms of the robot arm's configuration space [Lozo81, Lozo83, Fave84]. Unfortunately, the computational complexity of this approach grows rapidly with the number of robot arms and the number of joints making up the robot arms. In addition, this approach is not suited to a dynamic real time environment. Fujimura and Samet [Fuji86, Fuji88] have studied using four dimensional region octrees (three spatial dimensions and time) to do robot planning. Hong and Shneier [Hong85] have also considered the region octree for planning.

The use of region octrees to support collision detection is presented in [Boaz84, Roac87]. First, a plan to perform a given task is generated by the planning system. To ensure that no collisions occur, the planned motions are sampled at discrete times and inter-object interference is checked for. A region octree is constructed a priori for static objects in the environment. At each sample, an octree is constructed for each moving object and static intersections tests are performed by parallel traversal of the octrees.

Hayward [Hayw86] describes a collision detection tool based on region octrees for an off-line robot programming system. It takes as input a geometric description of a workspace and

a robot trajectory and reports where and when a collision would occur should the trajectory be executed. Yu and Khalil [Yu86] present a system for collision detection of a robot working in a fixed environment. The robot and the environment are modeled by means of simple primitives (i.e. spheres, cylinders, parallelepipeds, cones, and planes). The authors observe that, in spite of the simple methods used for modeling, an "on-line" application based on testing the intersection of the robot links with all obstacles at each control point is not practical. In order to accelerate the calculation of the collision detection algorithm, a table look-up procedure is used. Free space is represented by discretizing joint space and is stored in a table structure. This table is used to map the position of a robot link to the obstacles that lie close to that link. Thus, the number of intersection tests performed at each sample is reduced.

Many different approaches have been reported for the general problem of collision detection. Some of the systems are targeted for off-line applications where system performance is not a major factor and operator interaction is possible. Others are used in conjunction with planning systems to decide if pre-planned robot motions are collision free. Still others propose the use of custom hardware to efficiently detect collisions [Smit85]. We desire a system which will provide real time response with little or no interaction with the operator. Furthermore, no prior knowledge of robot arm movements will be available. Finally, a system using specialized hardware is not acceptable in our particular application due to the high cost of testing and accepting a new computer for space flight.

3. GENERAL DESCRIPTION OF THE ALGORITHM

Our approach to collision detection is to maintain a model of the working environment and through that model, detect when objects in the real world are about to collide. In effect, a real time simulation of the environment is performed. Information regarding the position of objects is input to the system, the representation is updated to reflect the current state of the world, and any

imminent collisions are reported.

Each object in the working environment that can collide with another object is included in the world model. A constructive geometry approach is used to represent objects. That is, complex objects are described as the union of simpler primitive objects. For our representation, we have chosen *cylspheres*, cylinders with spheres on each end, to represent each link in the robot arms. The cylsphere both provides an acceptable approximation for the links and allows for efficient intersection tests. The operating environment is not well modeled by cylspheres, so our representation also supports *rectangular solids* as additional primitives. Each primitive object in the environment is considered a separate entity and is assigned a unique identification number. A geometric description of each primitive object and its current position in the world is stored in a table and is accessed through this ID. The use of cylspheres and rectangular solids allows for a satisfactory representation of our working environment (shown in Figure 2) with only 35 primitives. We expect that the workspace for a wide range of applications can be modeled with at most a couple of hundred primitive objects.

When primitive objects are used to construct more complex objects, such primitives may appear to overlap, but no collision should be reported. To handle this problem, the notion of *compatible* primitives is introduced. Two adjacent primitives that make up an object (such as consecutive links in a robot arm) will never collide and thus are defined to be compatible. Compatibility between objects is represented by a two-dimensional array in which the entry at row i and column j is TRUE if objects i and j are compatible and FALSE otherwise.

We now turn to the problem of collision avoidance. We don't actually want to detect collisions. We wish to detect *imminent* collisions, with the intention of avoiding them. Thus, the collision detection system shall issue a warning when two objects come "too close" to one another. When a certain distance between non-compatible objects is to be maintained, the standard technique for a static environment is to extend each primitive by 1/2 the minimum tolerance

distance in all directions. Whenever extended (incompatible) primitives overlap, a collision warning can be issued.

In a dynamic robotics environment, several factors go into determining the minimum distance allowed between an arm and other objects. First, the current position of the arm (i.e., the current joint angles) must be passed from the controller to the collision detection system. The detection system (which is the part of the overall system described in this paper) must recognize that a collision is about to occur and issue a shutdown command. The controller must than engage the breaks. Finally, the arm must actually stop, which may in turn cause oscillations or bending in the arm.

Cycle times for the controller to propagate joint positions to the detection system can vary widely between systems, ranging from only a couple milliseconds (the expected time for the arms NASA intends to fly on the space station) to typically 50 msec (as required for NASA's current test robots). In our tests, our collision detection algorithm requires 30-60 msec to issue the shutdown command. Time to actually stop is in the 10-20 msec range at maximum speed. Thus, the tolerance value should be based on the distance that the robot arm can move toward an object in approximately 50 to 100 msec at maximum speed. With maximum speed of the end effectors limited to 24 inches/second, that yields between a one and three inch tolerance zone around each moving object, depending on the values selected.

In a dynamic environment we may also wish to make provision for the relative speed of moving objects. In other words, if an object is moving at less than maximum speed, the minimum tolerance for that object can be reduced. Our cylsphere representation provides only an approximate model for the links of the arms, and in some cases is in error by more than one inch. This is close enough to the tolerance required at maximum speed that the overhead incurred by changing the model for the arms to account for varying speeds is unjustified. We use a fixed tolerance for each arm such that at least 1.2 inches of buffer area beyond the approximated boundary of the robot arm

is provided by the cylsphere. This is an acceptable approximation for our target environment since we do not expect that the operator will ever intend to have two arms (or objects) within less than two inches of each other (the exception being when the operator wishes to grasp an object). Given two objects in the working environment and their associated tolerances, determining an imminent collision between them is now reduced to simply detecting an intersection between the primitive objects (including their tolerances) which represent them. Changes in the tolerance limit should have little or no effect on the algorithm's performance.

To reduce the number of unnecessary intersection tests between primitive objects during the collision detection phase, an indexing scheme over the working environment is needed to determine which objects, and the primitives representing them, are close to each other and which are not. The octree provides such an index.

The octree consists of two types of nodes: internal nodes and leaf nodes. Whether a particular volume in space is represented by an internal node or a leaf node is time dependent. That is, a leaf node may be split because of movements in the environment and thus becomes an internal node. The following record structure in Pascal-like notation is used to represent both types of nodes. Since our octree will be small and memory is not a bottleneck, a node representation that is space inefficient but minimizes computing time has been chosen.

```
OctNode = record
    vertices : array[0..7] of Point;    { Coordinates for corners of octant }
    isSplit : Boolean;    { Internal or leaf node }
    children : array[0..7] of ↑OctNode;    { Pointers to internal node's children }
    parent : ↑OctNode;    { Node's parent }
    childNo : 0..7;    { Which child node is of parent }
    numObjects : integer;    { Number of objects contained }
    assocObjects : ObjectList    { Head of contained objects list }
end;
```

The first step in modeling the world is to build the octree. We begin with an empty cube enclosing the working environment represented by the root of the octree. Objects are added to the tree one at a time, and splitting is performed as directed by the decomposition rule. The

decomposition rule for our N-objects octree is to split a node if more than N objects lie within it. The value chosen for N is determined by the complexity of the objects supported and the denseness of the environment, although we shall see that our application is not sensitive to the value of N (also see [Nels86]). Figure 3 shows a 2D workspace stored in an N-objects quadtree with $N = 5$.

Given an object and a node in which to insert it (initially the root), insertion proceeds as follows. If no part of the object lies inside the node, then do nothing and return. If the node is an internal node, then recursively insert the object into each of the node's children. Otherwise, apply the decomposition rule to the node to determine if the node should be split before inserting the object into it. If the number of objects already in the node is less than N, then add the new object's ID to the node's object list. Otherwise, split the node, inserting all of its objects into the node's newly created children. Next, the new object is recursively inserted into each of the children. This split-insert process is repeated until all leaf nodes contain no more than N objects. The following Pascal-like psuedo code formalizes the insertion process.

```
{ Insert a primitive object into a node. }
procedure InsertObject(objectId : integer; node : ↑OctNode);
  { ObjectNodeIntersect returns TRUE iff the object lies within the node. AddObjectToNode
    and RemoveObjectFromNode inserts and removes the object ID from the node's object list,
    respectively. }
var child : Octant;
begin
   if ObjectNodeIntersect(objectId, node) then
      if node↑isSplit then
         for child := oct0 to oct7 do InsertObject(objectId, node↑children[child])
      else
         if node↑numObjects < N
            then AddObjectToNode(objectId, node)
            else SplitInsert(objectId, node)
end;

{ Recursively split and insert a primitive object into a node. }
procedure SplitInsert(objectId : integer; node : ↑OctNode);
var
   objId : integer;
   child : Octant;
```

```
begin
   SplitNode(node); { SplitNode creates children and links them to node }
   for each objId in node↑assocObjects do begin
      for child := Oct0 to Oct7 do
         if ObjectNodeIntersect(objId, node↑children[child])
            then AddObjectToNode(objId,node↑children[child]);
      RemoveObjectFromNode(objId,node)
   end;
   for child := Oct0 to Oct7 do
      if ObjectNodeIntersect(objectId, node↑children[child]) then begin
         if node↑children[child]↑numObjects < N
            then AddObjectToNode(objectId, node↑children[child])
            else SplitInsert(objectId, node↑children[child])
      end
end;
```

Each cycle, the system receives two sets of joint angle values corresponding to the current configuration of the two robot arms. These joint angles are measured relative to a "home" position where every cylsphere is parallel to a coordinate axis. Using these values, simple kinematic transformations are applied to determine the current position of each line in Cartesian space. For each arm, beginning with the end effector and working backwards towards the joint attached to the base, we rotate each joint (and all joints dependent on it) to the position specified by its corresponding joint angle. Rotating a joint is performed by simply rotating the two end points of the cylsphere which represents it. As a by product of these transformations, the locations of objects currently attached to the end effectors of the robot arms are also updated.

The second step in the update process modifies the octree representation to reflect any changes in position of the robot arms. Three possible approaches to updating have been considered. The naive approach is to completely rebuild the octree for each cycle, checking for possible collisions as each object is inserted into the tree. This might be a good idea if a large portion of the environment changed during each cycle. However, due to the short cycle time (30-60 msec) combined with restrictions on robot arm speed, we expect only small changes in the environment during each cycle. Such changes rarely require modification to the octree. A second approach is

to delete and re-insert moving objects. This approach requires modification of a relatively small portion of the tree, but will cause expensive and unnecessary merges and splits. Objects move very small distances during a short cycle time, and thus a moving object is frequently re-inserted into the same nodes from which it was deleted. However, when such a node and its siblings contain N+1 objects, the nodes will be merged when the moving object is deleted from the tree, only to be split again when the object is re-inserted.

A more efficient update process changes the structure of the tree only when changes in the environment dictate. We observe that the tree structure changes only when an object exits a node (causing the object to be deleted from that node) or moves into a new node (causing the object to be inserted). These events, in conjunction with the decomposition rule, may cause nodes in the tree to merge or split. Further, when an object enters a new node, that node is a *neighbor* of a node in which it currently resides. Two nodes are considered neighbors if they share a face, edge, or corner. With this approach, tree updates work as follows. First, locate all nodes that the moving object resided in before it moved. For each of these nodes locate all of its neighbors using neighbor finding techniques described by Samet [Same89]. If the object has moved into a neighbor node, then insert it at that node and split if necessary. Upon completion, check if the object has exited any of the nodes it resided in before the move. If so, then delete the object from such nodes and try to merge them with their siblings.

This algorithm allows us to perform efficient updates of the octree by ignoring parts of the tree where no movement has occurred. However, collision checking must be incorporated into the update. After an object moves, all objects in all nodes that contain it are checked for possible collisions. In the octree, multiple moving objects may reside in the same node or a moving object and a static object may share more than one node. To eliminate any redundant intersection tests, we keep track of which pairs of objects have been checked for collisions during the current cycle. Thus, a complete intersection test between a pair of objects will be performed at most once (although

our update algorithm may check several times to see if a given pair has been tested).

Optimization of the neighbor finding process results from locating neighbors only in the direction of object motion. In fact, often the number of neighbors processed can be reduced to zero. If a moving object's bounding box remains completely within a single node, then there is no need to check for entry into any of the neighboring nodes. Determining if a bounding box lies completely within a node requires at most six comparisons. This quick check can save a significant amount of processing time, particularly if many small objects are moving (e.g. the fingers of a gripper).

The amount of computation needed for an update when an object moves into a neighboring node can be further reduced. When an object moves into a neighbor, the algorithm described above will insert the object into that node and perform any required splitting. If the neighbor is split, the algorithm will attempt to recursively insert the object into *all* of the neighbors' children (and possibly their offspring). Due to restrictions in robot motion, we only need to update the part of the neighbor which lies closest to the original node. So, the object is inserted into only the leaf descendents of the neighbor which lie on the common face, edge, or corner between the two nodes.

When a single object lies in many nodes, there will be some overlap in the neighbors of these nodes. This presents a problem for our algorithm because it will visit the same neighboring node multiple times. For example, if an object moves into a node X which is a neighbor to three of the nodes in which it currently resides, then our algorithm will process X three times. In fact, it will insert the object's ID into X's associated object list three times when only once is necessary. Furthermore, two of the nodes which an object lies in can be neighbors, causing the algorithm to add an object to a node's associated objects list in which it is already stored.

To prevent such anomalies, we augment our algorithm to mark all nodes which have been visited while moving an object. Each node in the tree includes the field *lastCheckNo*, which stores an integer denoting the last update during which this node was visited. The variable *ThisCheckNo*

is used to denote the current object update. When an object moves, each node that is processed has its *lastCheckNo* assigned the value of *ThisCheckNo*. Before processing a node, we first check its *lastCheckNo* to determine if it has been processed. If *ThisCheckNo* is equal to *lastCheckNo*, then the node is ignored. After processing an object, *ThisCheckNo* is incremented by one. Using an unsigned 32 bit integer, *ThisCheckNo* will reset to zero after 4,294,967,296 updates. At this point, the tree should be traversed and every node's *lastCheckNo* reset. Assuming a 50 msec cycle time with 14 updates of the octree required during each cycle (all seven links of both arms moved), this occurs approximately every 6 months during continuous operation. The following Pascal-like psuedo code provides a more formal description of the updating process.

```
{Update the octree when an object has moved.}
procedure UpdateObject(objectId : integer);
  { ObjectId is the label for a primitive object. Location of objectId is a member of the set of
    nodes containing the primitives. ObjectNodeIntersect determines if a primitive object lies in
    a node. CollisionInNode determines if an object collides (intersects) with any of the objects
    in a node. GetUnCheckedNeighbors returns a list of all neighbors of a node which have not
    been visited during the current update. BoundingBoxInsideNode determines if an object's
    bounding box lies completely inside a node. NeighborInDirection determines if a neighbor lies
    in the direction of the moving object. DeleteObjectFromNode removes an object's association
    with a node and performs any possible node merging. }
var
  location : ↑OctNode; {A node in which the object resides (see Section 4)}
  neighbor : ↑OctNode;
  neighbors : NeighborList;
begin
  ThisCheckNo := ThisCheckNo + 1;
  for each location of objectId do begin
    if CollisionInNode(objectId, location) then HandleCollision; {Warn system of collision}
    location↑lastCheckNo := ThisCheckNo
  end;
  for each location of objectId do begin
    if not BoundingBoxInsideNode(objectId, location) then begin
      GetUnCheckedNeighbors(location, neighbors);
      for each neighbor in neighbors do
        if NeighborInDirection(neighbor) then
          if ObjectNodeIntersect(objectId, neighbor)
            then UpdateNeighbor(objectId, neighbor)
            else neighbor↑lastCheckNo := ThisCheckNo
    end;
```

```
        if not ObjectNodeIntersect(objectId, location) { Object moved our }
            then DeleteObjectFromNode(objectId, location)
    end;


{Update a node that an object has just moved into.}
procedure UpdateNeighbor(objectId : integer; neighbor : ↑OctNode);
var child : Octant;
begin
    if neighbor↑lastCheckNo <> ThisCheckNo then begin
        if neighbor↑isSplit then
            for each child of neighbor on common face, edge, or corner do
                UpdateNeighbor(objectId, node↑children[child])
        else begin
            neighbor↑lastCheckNo := ThisCheckNo
            if ObjectNodeIntersect(objectId, neighbor) then begin
                if CollisionInNode(objectId, neighbor) then
                    HandleCollision {Warn system of collision}
                else if neighbor↑numObjects < N then
                    AddObjectToNode(objectId, neighbor)
                else UpdateSplitInsert(objectId, neighbor)
            end
        end
    end;


{Recursively split and insert an object into a node during update.}
procedure UpdateSplitInsert(objectId : integer; node : ↑OctNode);
var
    objId : integer;
    child : Octant;
begin
    SplitNode(node);
    for each objId in node↑assocObjects do begin
        for child := oct0 to oct7 do
            if ObjectNodeIntersect(objId, node↑children[child])
                then AddObjectToNode(objId,node↑children[child]);
        RemoveObjectFromNode(objId,node)
    end;
    for child := oct0 to oct7 do
        node↑children[child]↑lastCheckNo := ThisCheckNo;
        if ObjectNodeIntersect(objectId, node↑children[child]) then
            if node↑children[child]↑numObjects < N
                then AddObjectToNode(objectId, node↑children[child])
                else UpdateSplitInsert(objectId, node↑children[child])
end;
```

A fundamental assumption used by our update algorithm is that an object cannot move *through* a node between consecutive updates. If this were not true, then our premise that between updates an object can move only into neighbor nodes would no longer hold. Consider, for example, an object that has moved out of a node, completely through one of its neighboring nodes, and into a non-neighboring node. The update algorithm described above would recognize that the object has left its original node, but has not moved into any of the neighboring nodes. The object would "disappear" from this part of the octree and possible collisions would be ignored. Alternatively, an algorithm that deleted a moving object from its current node and then inserted the object at its destination could miss collisions at intervening nodes. Recall that the tolerance zone was set to be greater than the distance that an object can move in one cycle.

The maximum speed at which an object can move, combined with the cycle time between updates, allows us to calculate the maximum distance that an object can move between updates. This distance is used to determine the minimum size for a leaf node. For example, if the robot arm tip can move 1 inch in an update cycle, then the smallest voxel allowed in the octree would have an edge length of 1 inch. If during the splitting process a node with this size is created, then we prevent any more splitting and allow this node to exist without regard to the decomposition rule. Since the minimum size of any primitive in the smallest dimension is at least twice the tolerance value, it is also not possible for a primitive to move into, through, and out of the corner of a node in a single cycle.

The bounding cube used to enclose the entire working environment of our test scenarios is 250 inches wide in each dimension. The robot arms themselves have a maximum reach of 75 inches when fully extended. While the minimum resolution for our octree was calculated to be 1.95 inches, during testing this level of decomposition was never approached, in part due to the value of N (10).

When an object moves, we must quickly locate what nodes contain the object. This is done using *location links*. The set of location links for an object is a linked list of leaf nodes in the tree. Each object's description contains a pointer to one of the leaf nodes containing the object. This leaf node in turn contains an object list entry with a pointer to another leaf node in which the object lies (see Figure 4). A sequence of links is formed that includes every leaf node containing that object. When an object enters a leaf node, the node is added to the object's location list. Similarly, when an object exits a leaf node, the node is removed from the object's location list. During an update for a moving object, each node on the object's location list is processed.

4. INTERSECTION ALGORITHMS

Intersection tests are an essential part of the collision avoidance system. Tests between primitive objects (rectangular solids and cylspheres) are performed to determine possible collisions. Tests between primitive objects and nodes are performed to build and update the octree. We briefly describe intersection tests involving cylspheres. Intersections between rectangular solids are common in graphics applications, and not further discussed. Additional primitives can be included by simply adding the appropriate intersection tests. For more complete details on our intersection operations, see [Herb90].

Determining if two cylspheres intersect is a straightforward process. Cylspheres are described using two points (*endP1* and *endP2*) and a radius. In addition, the length of the axis between the two end points is stored since it is needed throughout the intersection calculation. To determine if a point P lies within a cylsphere, first the distance from P to the line segment formed by *endP1* and *endP2* is calculated. If this distance is less than the cylsphere's radius, then P lies within the cylsphere.

To determine if two cylspheres *cs1* and *cs2* intersect, we generalize this approach. Given two lines in space, there is a unique point on each line where the lines are closest to each other (unless

the lines are parallel). This point is quickly found using simple line-plane intersections. For the lines containing the axes of *cs1* and *cs2*, assume that we have found these two points, *P1* and *P2*, respectively. If *P1* lies outside the two end points on the axis of *cs1*, then the closest end point is taken as *P1*. The same is done for *P2* and *cs2*. Next the distance between *P1* and *P2* is calculated. If this distance is less than the sum of the radii of the two cylspheres, then they intersect.

For all intersection tests, a bounding box test is performed first in hopes of quickly ruling out an intersection. To calculate the bounding box for a cylsphere the maximum and minimum x, y, and z values for points on its surface are found. To find the minimum x value, the smaller of the x coordinates of the two end points is determined and the cylsphere's radius is subtracted from it. The remaining values for the bounding box are found similarly. Every time a cylsphere's position changes, its bounding box is recalculated.

Due to constraints on object motion, for a cylsphere to intersect a rectangular solid (box), either an edge of the box passes through some part of the cylsphere's surface or part of the cylsphere passes through a face of the box. If both of these cases are checked and no intersection is found, then it is assumed that the box and the cylsphere do not overlap. Intersections tests involving nodes are similar to intersection tests involving rectangular solids. A node is just a special case of a rectangular solid in that its edges are all equal in length and parallel to a coordinate axis.

5. EXPERIMENTAL RESULTS AND ANALYSIS

Our collision detection algorithm was implemented using the C language, running under UNIX. All timing results are for an Intel 80386 CPU with math co-processor running at 20 MHz. This configuration matches NASA's planned computing environment on the space station. Our algorithm was tested by first generating joint angles using a robot simulation program. This program allowed us to direct the two robot arms through a task within a 3 dimensional graphical model, sampling the robot arms' joint angles at discrete intervals, and storing them into a file. For

each task, the corresponding joint angle file was used as input to our algorithm. The algorithm proceeds by first reading in a block of joint angles. For each set of joint angles in the block, kinematics are applied to produce the arm's new position, and the octree is updated. Whenever a collision is detected, the algorithm terminates, indicating which objects have collided. Upon completion, the algorithm reports timing results.

Our testing was comprised of running three separate tasks using a fixed working environment (see Figure 2). Our environment was modeled after the test bed constructed at NASA's robotics laboratory at the Goddard Space Flight Center. The tasks consisted of the two robot arms being navigated through the environment to simulate realistic operation. The first task was comprised of the left arm moving towards the box located on the table in front of the robots while the right arm simultaneously positioned itself above the box located on the table to the right. The second task was similar to the first except that the left arm collided with the table in front. The third task consisted of the right arm colliding with the table located in front while attempting to grasp the box lying on top of it. The tasks required 1000 cycles, 150 cycles and 2000 cycles, respectively.

The average time per cycle (processing one complete set of joint angles and checking for collisions) over the three tasks was about 30 msec. However, the actual time for each cycle varied depending on how many primitives moved during that cycle. For example, all seven links of both arms moving required more computation time then if a single link of one arm was moving. This is because the first case requires updating the tree 14 times (14 objects have moved) whereas the second case requires updating the tree only once. The time required for each cycle of the algorithm was measured using the system clock, which had a resolution of 10 msec. The observed upper bound for the range of update cycle times was under 60 msec.

Further data was collected on how much computation is done by each part of the algorithm. About 28% of the computation time was devoted to performing the kinematics for the robot arms. The kinematics algorithm that we implemented was selected for its simplicity rather than it ef-

ficiency. A more efficient algorithm could offer significant improvement. The remainder of the computation time was used to update the octree and check for collisions. About 27% of the time was spent performing object-object intersection tests and about 7% calculating rotation angles and bounding boxes for moving objects. Object-node intersection tests required 13% of the time while retrieving neighbors took about 7%. The remaining computation time was dedicated to overhead incurred by other parts of the algorithm (i.e. splitting, merging, etc.). In summary, kinematics, collision detection, and octree maintenance each required roughly one third of the computation time.

Two characteristics of the N-Objects octree which make it a desirable representation for a collision avoidance system is that it is compact and changes in its structure rarely occur. The initial configuration of the octree representing our test scenario (35 objects) was split only two levels below the root and contained 33 nodes, of which 29 were leaf nodes. The average number of objects in each non-empty leaf node was about 5 (with $N = 10$), while 15 of the nodes were empty. Each object resided in about 2 nodes on the average. So the occupancy of each leaf node as well as the number of nodes occupied by each object were both low. For the three tasks used to test our algorithm, the average number of cycles between a split or merge was around 600. Given a 30 msec cycle time, this translates into once every 18 seconds.

In Section 3, techniques for improving system performance were discussed. In each case a positive effect on performance was observed. To illustrate how fine tuning of the algorithm can effect system performance, we compared computational requirements with and without each technique incorporated into the algorithm. Eliminating redundant intersection tests between objects decreased computation time by 4%. Calculating the direction(s) of a moving object and updating only neighbors in this direction reduced computation time by 8%. Checking if an object's bounding box is completely contained within a node (to preclude checking for movement into the nodes' neighbors) reduced computation time by 13%. The use of bounding boxes to eliminate object-

object and object-node intersection tests had the most signifi ant effect by reducing computation time by 80%.

The decomposition rule for the N-Objects octree is simply "split a node if more than N objects lie within it." The value chosen for N, though, affects system performance. A large portion of the computation required by an octree update consists of object-node and object-object intersection tests. Object-node tests are needed to determine if an object has moved into a new node. Object-object tests are used to detect collisions between objects. The chosen value of N directly effects the number of each type of intersection test performed during an update. For example, the choice of a small N causes the tree to decompose to a much lower level than a large N. This deeper splitting, in general, increases the number of nodes that an object lies in. This in turn increases the number of neighbors that need to be checked for possible entry.

On the other hand, if we choose a larger N, the tree is not as deep and we have fewer nodes to process during the update. However, since more objects are allowed to share a node, when an object moves, more object-object intersection tests are required within the nodes to detect for possible collisions. Thus, the value of N controls the relative amount of each type of intersection test performed during an update. Depending on the relative cost of performing object-node and object-object intersections, the value chosen for N directly effects system performance. If the cost of performing an object-object intersection test is much more expensive than the cost of an object-node intersection test then a large value for N would optimize the update process.

Given the primitives supported by our representation and the working environment of the robot arms, we have chosen a value of 10 for N. This value resulted in optimal performance given the relative costs of object-node intersection tests (70 microseconds) and object-object intersection tests (150 microseconds). Figure 5 shows how system performance varied for different values of N. Although 10 provided the lowest average cycle time, this number falls within a wide range of values providing similar performance. Thus, we can be confident that a different task or environment

would not require that a different a value for N be used.

A natural question to ask is how does the octree compare in performance to the naive approach to collision detection? A naive algorithm is one which, when an object moves, would check for possible collisions with *all* other objects in the world. The computation time for such an algorithm grows in proportion to the number of objects in the world (assuming a constant number of objects have moved). This may be acceptable behavior if the computation cost for the intersection tests is very low. The naive algorithm also does not require nearly as much overhead as the octree. For a sufficiently simple environment, the naive approach is more efficient than the octree. Conversely, the octree is more efficient for a sufficiently complicated environment. The question is at what point does the octree perform better?

The naive algorithm was implemented and tested with the same three tasks as described above. The number of objects in the robot's environment was varied and timing results were recorded. The same tests were repeated using the octree version of the algorithm. Figure 6 illustrates the behavior of the two algorithms. The cycle times for both algorithms increased as objects were added to the environment. In both cases though, the objects were added into the immediate area surrounding the two robot arms. Other objects could have been strategically placed in the working environment which would have no effect on the cycle time for the octree algorithm but would still increase the cycle time for the naive algorithm. For example, objects could be placed in parts of the tree where no updating takes place, in which case no increase in cycle time would be observed. Thus, our testing was biased against the octree method, yet the octree showed greater performance gains over the naive method as the environment became more complex.

A grid representation is similar to the octree in that it provides a spatial index by partitioning the space into disjoint regions. Grid structures have been suggested for use in performing geometric operations on large data bases [Fran83, Fran89]. A grid is overlayed onto the data and

for each grid cell, a set containing each object that lies in that cell is formed. Such a representation was implemented to perform collision detection and compared to the octree approach. To simplify implementation, the grid was represented as a three-dimensional $GxGxG$ array of octree nodes, where G was the number of cells along each dimension of the fixed-size world. Updating was similar to octree updating with the exception of finding neighbors. Finding the neighbors of a node was simplified to acquiring the 26 surrounding grid cells. To provide for a fair comparison, all of the performance enhancing techniques that were incorporated in the octree implementation were also included in the grid implementation.

To test the grid implementation, we used our standard three tasks and varied the value of G. Figure 7 provides an illustration of how the algorithm performed for different values of G. In our tests, the best grid was slightly worse than the octree. Although the grid representation is a simple one, it is a static structure whose performance suffers when the distribution of the geometric data it represents is not uniform. Furthermore, choosing a good value for G may be difficult since the optimal value can vary from task to task or even during a task.

An alternative method for modeling three dimensional objects is the region octree. The region octree represents an object as a set of cubes of varying size. Each cube is colored black or white, depending on whether it is inside the object (black) or outside the object (white). A region octree representation of our world model can be constructed in which the black nodes represent robot arms and objects, and the white nodes represent the free space. Using methods described in [Boaz84], nodes in the tree can be transformed to reflect the movement of robot arms. Using a fine resolution octree, we can develop an accurate model of the robot arms and their working environment. However, the storage space required by the region octree and the amount of computation needed to perform the required transformations during robot movement may be unacceptable for real time robot collision detection.

A region octree was implemented using our world model of cylspheres and rectangular solids. Using a resolution of 1.2 inches (the same as that used for the N-Objects octree), the number of leaf nodes needed to represent the scenario model was around 30,000. The upper six links of both robot arms required a total of about 3,000 leaf nodes. Given that these 12 links moved during a typical cycle, it is inconceivable that updating 3000 nodes on today's microprocessors could be done in real time (i.e within a 30-60 msec time span).

6. CONCLUSIONS AND FUTURE WORK

The performance of the N-Objects octree is directly affected by the number of stationary and moving objects in the environment. When a stationary object is added to the environment, it may increase the algorithm's cycle time in a number of ways. The new object may cause the tree to split, in which case the algorithm may be required to process more nodes during an update. The object may also share a node with a moving robot arm, increasing the number of intersection tests performed during an update. On the other hand, the object may be added in an area not close to a robot arm, in which case no effect on cycle time will be observed. In general, stationary objects only increase cycle time if they are included in nodes that also contain moving objects. Thus, the number of stationary objects affecting cycle time is closely related to the percentage of nodes containing moving objects.

Adding a moving object, for example another robot arm, will have a more predictable effect on cycle time. In a given cycle, every moving object in the environment causes the octree to be updated with respect to that object. When the number of moving objects in the world is doubled, we would expect the time needed to update the octree to also double (assuming that the moving objects make up a reasonably small percentage of the total number of objects). If for example we wish to increase the number of primitives used to represent a robot arm, a corresponding increase in cycle time should result. This behavior was observed during our testing, when cycle time was

proportional to the number of links which moved during that cycle.

With the expanded use of tele-operated robots in space, manufacturing, and the nuclear industry, the problem of collision detection and prevention has become more important. Providing a mechanism to ensure safe operation of robots is of high priority, given the consequences of accidentally damaging expensive robotic equipment or nuclear waste containers. The goal of our research was to provide such a mechanism, with the specific application studied for testing purposes being the operating bay of NASA's manned space station. A collision avoidance system must be *efficient* enough to provide timely information about possible collisions, *reliable* enough to not miss any imminent collisions, and *usable* enough so as not to hinder normal operations. The N-Objects octree representation presented here meets all of these requirements.

The algorithm we have presented is suited to a variety of different applications in robotics where a collision avoidance capability is needed. However, there are some restrictions on the kinds of problems that our system can be applied to. Information about the robots and their working environment must be available to the system in the form of a geometric model using supported primitives. Thus, the shape of the environment must be suitable to such a representation. If information about the environment is known a priori, then it can be manually entered by the user. Otherwise, some form of sensing is required so that the system may acquire this knowledge. Position information about moving objects is needed by the system in order to maintain the model. Joint angles were used in our application of the system to tele-operated robots on the space station. However, if an astronaut walks into the robot's work area and moves a box, its new position must be made available to the system.

To obtain real time performance, some sacrifices were necessary in terms of the accuracy of our model. Other primitives may have better represented the robots and working environment but at the cost of more expensive intersection tests. Hence, the system is well suited for an application where an approximate model is acceptable. We also made the assumption that no object will move

into, through, and out of a node in a single cycle. Since our cycle time is so short, this should not limit most applications. Finally, if a large number of primitives are moving, cycle times will likely increase beyond acceptable limits. On the other hand, a small number of moving primitives can likely operate in real time even with a relatively large number of stationary objects.

The octree provides a flexible means for indexing three dimensional space in that it easily supports dynamic modeling of robot arms. If we wish to change the model of the arm based on the type of task it is performing, we simply delete the model of the old arm from the tree and insert the new model. For example, when an arm is performing gross motions the entire gripper could be represented, for efficiency reasons, as a single primitive which completely encloses it. However, when the gripper is being used to grasp an object, a more detailed model is desired. This capability is supported by deleting the coarse model and inserting the detailed model at the appropriate time. In the same manner, the octree also provides for changing the model of the arm based on how fast it is moving (although our test application did not require this capability). Similarly, tolerances for the arm may be related to the mass of a grasped object.

The research presented here provides a foundation on which to develop a collision avoidance system. However, an issue that has been ignored in this paper is that of object grasping by the robot. That is, the system must be able to distinguish between intentional and unintentional collisions. When a robot gripper is about to collide with an object, the system should not report an imminent collision if the operator is actually attempting to pick up that object. A simple solution to this problem would be to interrogate the operator to determine his intentions. However, as little interaction with the operator as possible is desired. An alternative approach might be to declare ahead of time all objects that will be manipulated by the robot to be compatible with the gripper. Unfortunately, this still leaves open the possibility that the gripper may unintentionally collide with one object while trying to grasp another. A third approach would be to adopt a simple heuristic such as when the gripper approaches an object while moving at a low speed, assume that

the operator is trying to manipulate that object. In any case, this is not a trivial problem.

The N-Objects octree could also serve as the underlying representation for a robot planning system. Using a generate and test paradigm, the octree could serve as a means of determining if a given plan would be collision free. Alternatively, the octree could be used as a search space for a robot planning system. Using tree traversal techniques, the planner could search the octree itself for a collision-free path.

There is still much work to be done in the field of tele-operated robotics. The final goal is to allow for very high level human control. The limitations of current hardware and software technology prevents us from reaching this goal. However, the need still exists for collision avoidance and safety systems to meet the operational requirements of today. The octree has proven to be a useful data structure for both developing current systems and researching systems for the future.

7. REFERENCES

1. [Boaz84] M. Boaz, *Spatial Coordination of Transfer Movements in a Dual Robot Environment*, Master's Thesis, Virginia Tech, Blacksburg VA, 1984.

2. [Fave84] B. Faverjon, Obstacle avoidance using an octree in the configuration space of a manipulator, *IEEE Conference on Robotics*, Atlanta, 1984, 504-512.

3. [Fran83] W.R. Franklin, Adaptive Grids for Geometric Operations, *Proceedings of the Sixth International Symposium on Automated Cartography*, Vol 2, October 1983, 230-239.

4. [Fran89] W.R. Franklin, M. Kankanhalli, and C. Narayanaswami, Efficient Primitive Geometric Operations on Large Databases, *Proceedings GIS National Conference 1989*, Ottawa, Canada, March 1989, 59-67.

5. [Glas84] A.S. Glassner, Space Subdivision for Fast Ray Tracing, *IEEE Computer Graphics and Applications 4*, 10(October 1984), 15-22.

6. [Glas89] A.S. Glassner, *An Introduction to Ray Tracing*, Academic Press Inc., San Diego CA, 1989.

7. [Hayw86] V. Hayward, Fast Collision Detection Scheme by Recursive Decomposition of a Manipulator Workspace, *Proceedings 1986 IEEE International Conference on Robotics and Automation*, Vol 2, San Francisco, CA, April 1986, 1056-1063.

8. [Herb90] G.M. Herb, A real time robot collision avoidance safety system, Masters Thesis, Virginia Tech, Blacksburg VA, May 1990.

9. [Hong85] T.-H. Hong and M. Shneier, Describing a robot's workspace using a sequence of views from a moving camera, *IEEE Transactions on Pattern Analysis and Machine Intelligence 7*, 6(November 1985), 721-726.

10. [Lozo81] T. Lozano-Perez, Automatic planning of manipulator transfer movements, *IEEE Transactions on System, Man and Cybernetics 11*, 10(October 1981), 681-698

11. [Lozo83] T. Lozano-Perez, Spatial planning: a configuration space approach, *IEEE Transactions on Computers 32*, 2(February 1983), 108-120.

12. [MacD89] J.D. MacDonald and K.S. Booth, Heuristics for Ray Tracing Using Space Subdivision, *Graphics Interface 89*, 152-163.

13. [Nels86] R. Nelson and H. Samet, A Population Analysis of Quadtrees with Variable Node Size, Computer Science TR-1740, University of Maryland, College Park MD, December 1986.

14. [Roac87] J.W. Roach and M.N. Boaz, Coordinating the Motions of Robot Arms in a Common Workspace, *IEEE Journal of Robotics and Automation 3*, 5(October 1987), 437-444.

15. [Same89] H. Samet, *Applications of Spatial Data Structures; Computer Graphics, Image Processing, and GIS*, Addison-Wesley, Reading MA, 1989.

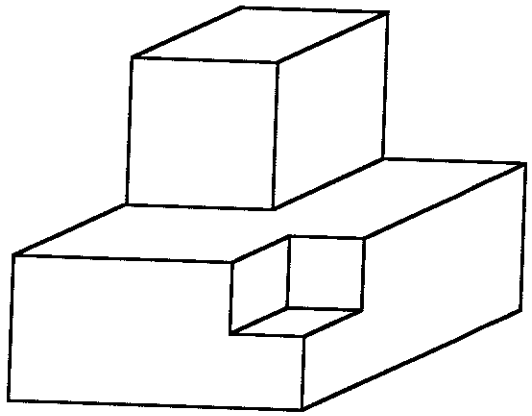
16. [Same90] H. Samet, *The Design and Analysis of Spatial Data Structures*, Addison-Wesley, Reading MA, 1990.

17. [Shar89] M. Sharir, Algorithmic Motion Planning in Robotics, *IEEE Computer 22*, 3(March 1989), 9-20.

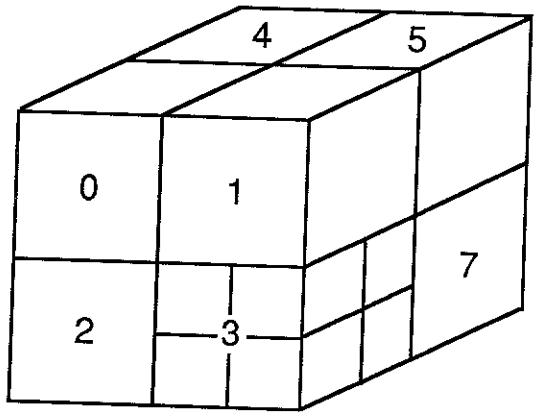
18. [Smit85] R.C. Smith, Fast Robot Collision Detection Using Graphics Hardware, *Proceedings of IFAC Symposium on Robotic Control*, Barcelona, Spain, 1985, 277-282.

19. [Whit85] S.H. Whitesides, Computational Geometry and Motion Planning, in *Computational Geometry* (Ed., G.T. Toussaint), North-Holland, Amsterdam, The Netherlands, 1985, 377-427.

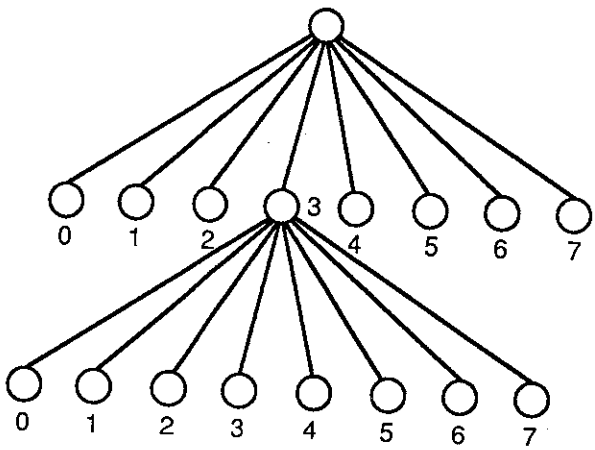
20. [Yu86] Z. Yu and W. Khalil, Table Look Up for Collision Detection and Safe Operation of Robots, *Theory of Robots*, selected papers from IFAC/IFIP/IMACS Symposium, Vienna, Austria, December 1986, 343-347.



(a)



(b)



(c)

Figure 1. An example region octree. (a) The object. (b) Its region octree block decomposition. (c) The resulting tree structure.

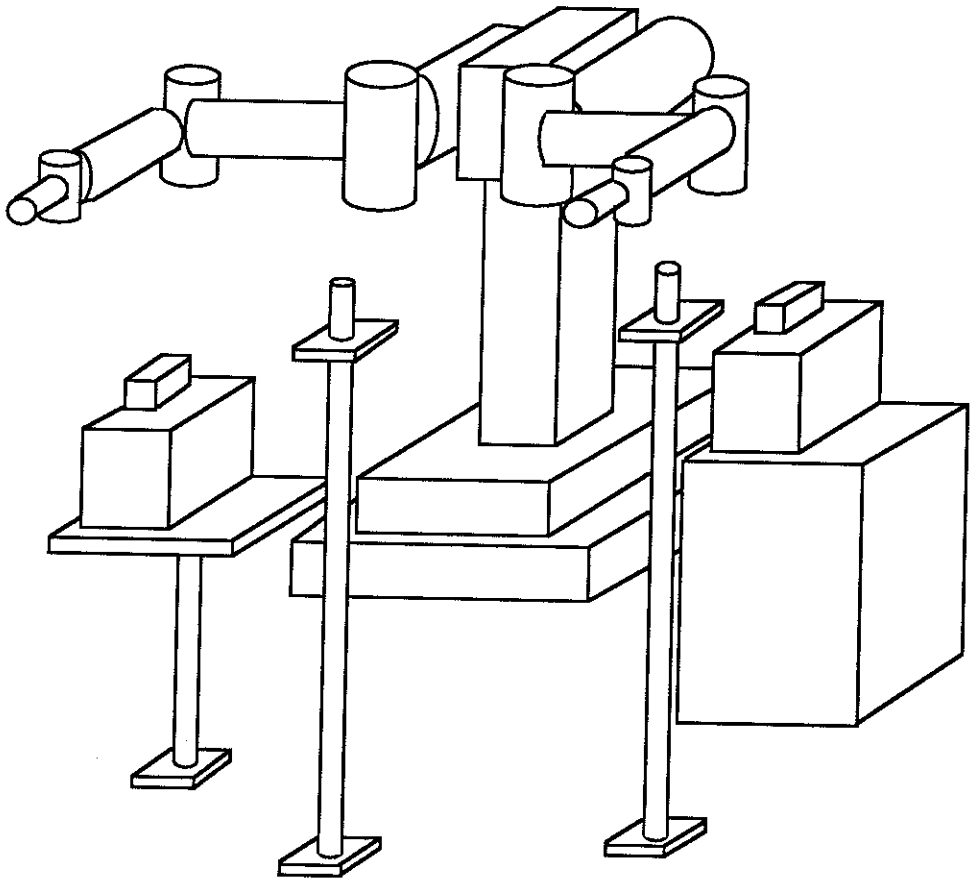


Figure 2. Working environment used to test collision detection algorithms.

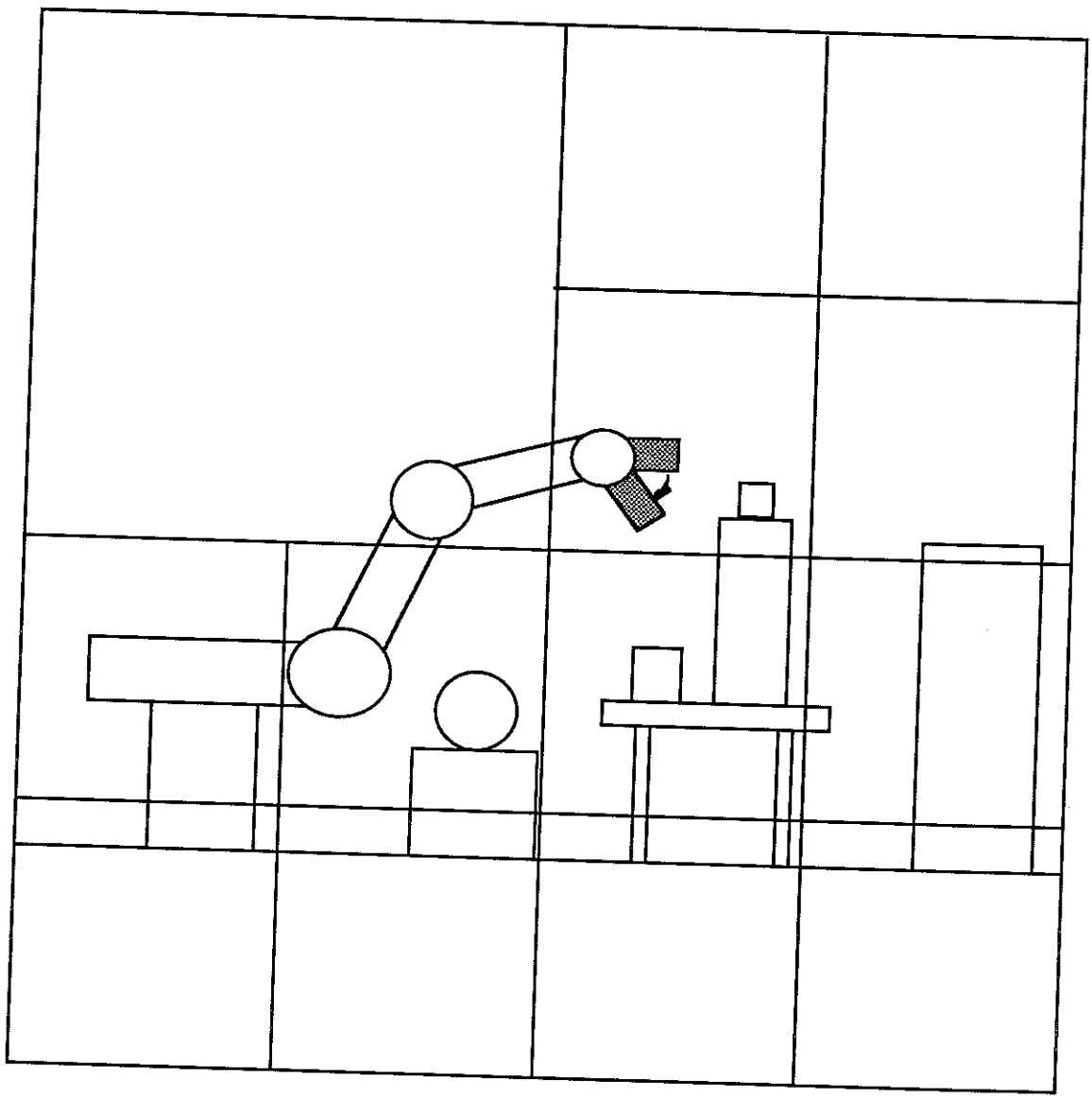


Figure 3. Decomposition for sample environment using a 5-objects quadtree.

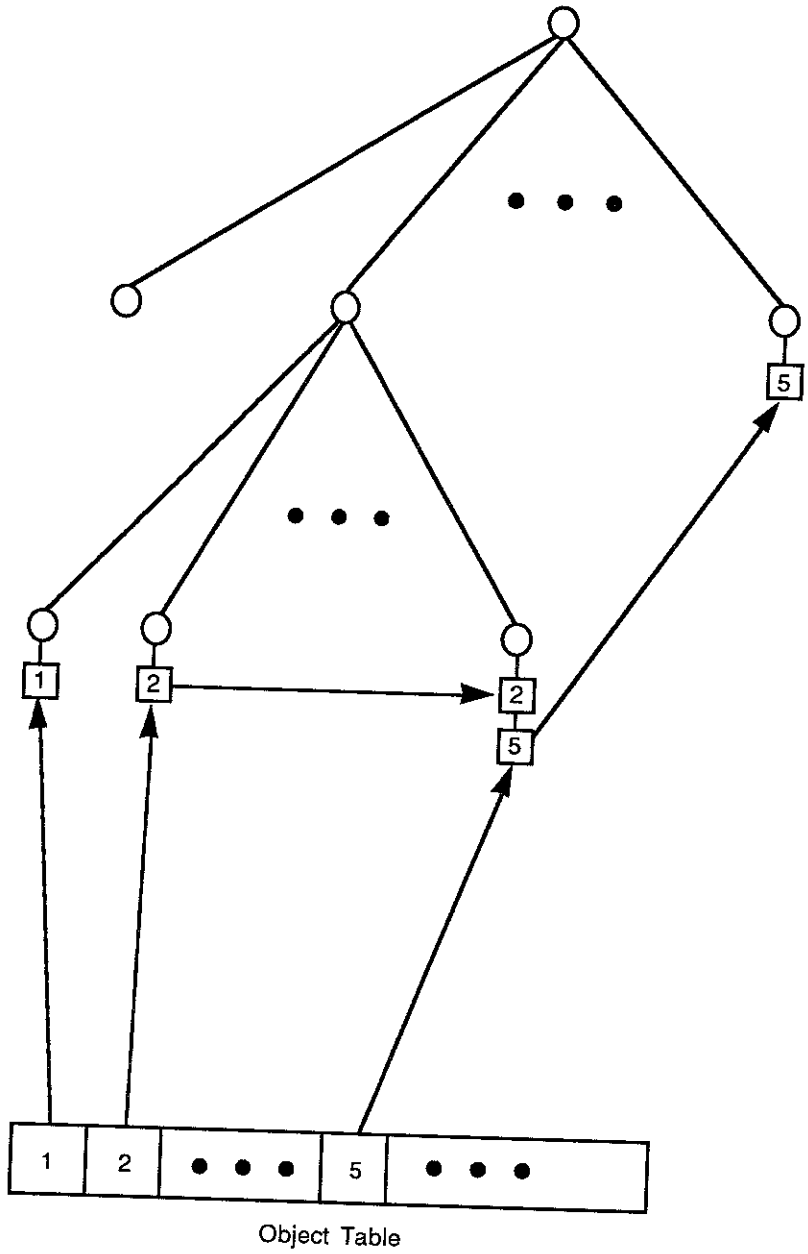


Figure 4. Location links for objects in the octree.

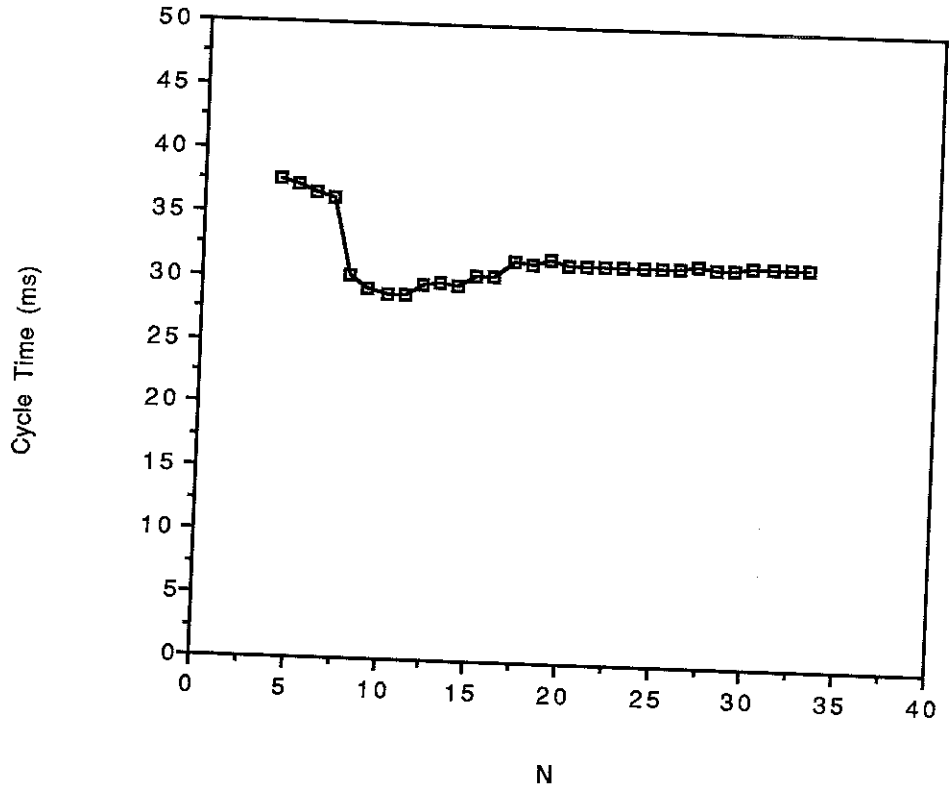


Figure 5. System performance for different values of N in decomposition rule.

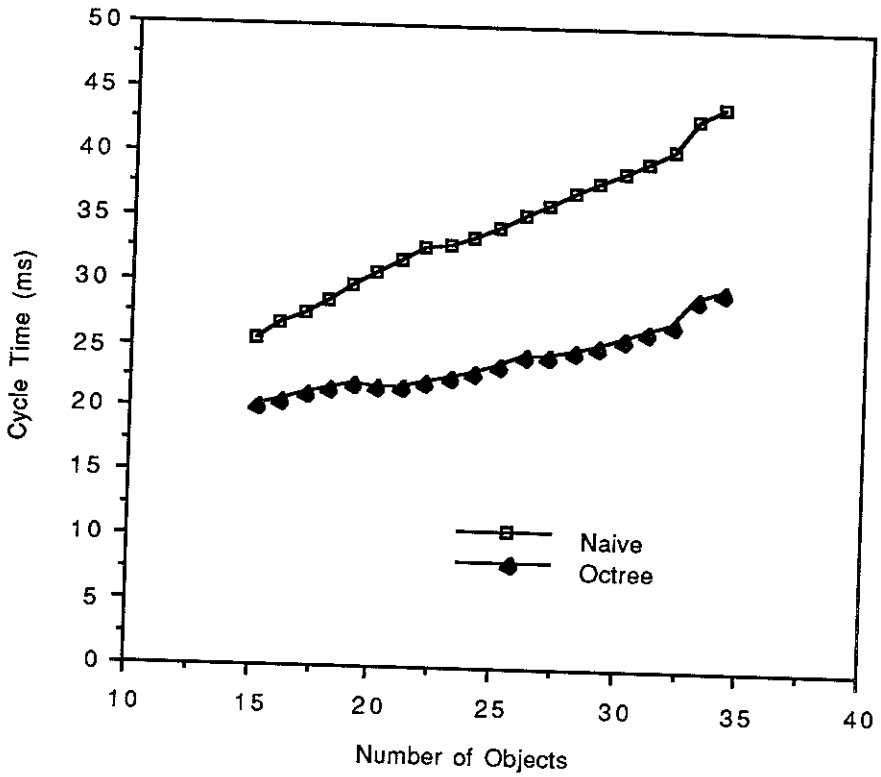


Figure 6. System performance of naive and octree algorithms as environment grows in size.

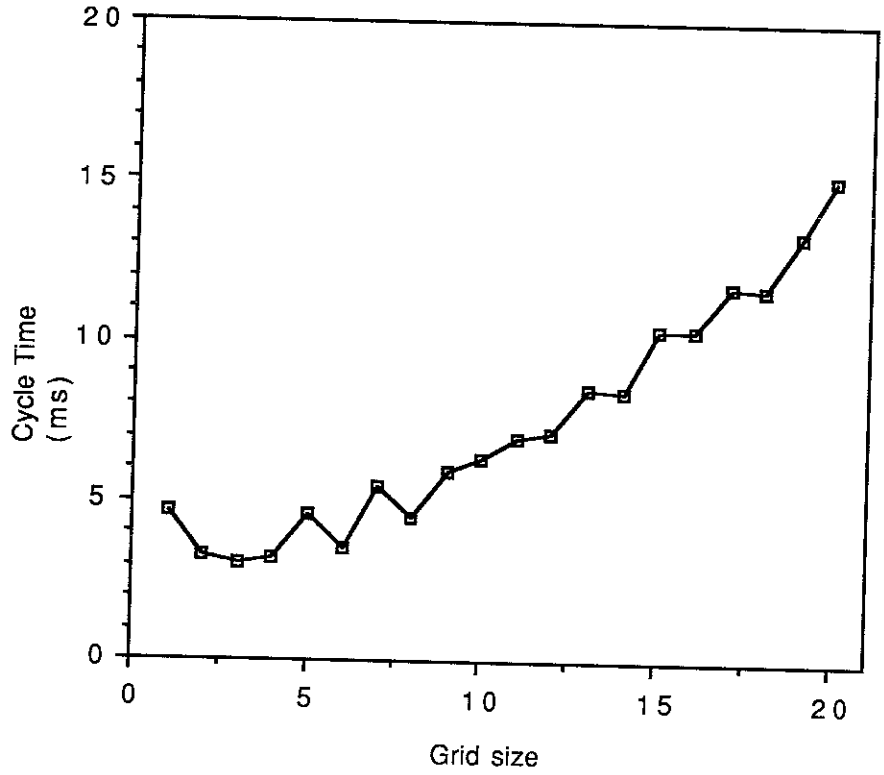


Figure 7. System performance of grid algorithm as grid size (G) is varied. Values are averaged over all 3 tasks.