A Real-Time Robot Arm Collision Avoidance System

Clifford A. Shaffer, *Member, IEEE,* and Gregory M. Herb

***Abstract*— A data structure and update algorithm are presented for a prototype real-time collision avoidance safety system simulating a multirobot workspace. 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. We use the *N*-objects octree, which indexes a collection of 3-D primitive solids. These primitives make up the two seven-degrees-of-freedom robot arms and workspace 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 world model for our application can be implemented 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 arm joint angles, updating the octree to reflect new positions, and detecting/reporting imminent collisions averages 30 ms on an Intel 80386 processor running at 20 MHz.**

***Index Terms*— Collision avoidance, hierarchical data structures, octrees, tele-operated robots.**

I. Introduction

This paper describes the use of a hierarchical data structure, the *N*-objects octree, in a collision avoidance safety system for simulating a multirobot workspace. 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 another object. In particular, our algorithm repeatedly tests a set of 3-D geometric primitives (which collectively represent the robot arms and their workspace, with the primitives for moving objects expanded to include a safety buffer) to determine if any primitives intersect. The algorithms described can be used to provide a collision-avoidance capability for a variety of robotics applications.

Our initial motivation for this work was to provide a real-time safety system to support NASA's tele-operated robot arms on the proposed space station. A tele-operated robot is one whose motions are dictated by an operator through a controlling device. In this case a minimaster, 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 that will prevent unintentional collisions. Ideally, a collision avoidance system will insure the safety of the robot arms without hindering the operator during normal operation.

Each robot arm in our testbed has seven degrees of freedom, with each link of the arm being nearly cylindrical in shape. The workspace is not static—the system must accommodate movement of both the arms and other objects. An important aspect of our problem is that motions are not predetermined. Many proposed collision avoidance and path planning algorithms are based on access to information about future motion. In contrast, our operating paradigm is one of receiving a current position for moving objects, updating the representation of the workspace to reflect that movement, and reporting any imminent collisions. As a result, we cannot solve a problem of continuous motion but rather must solve a series of static collision detection problems based on current positions of objects, possibly augmented with velocity information derived from the recent history of object movements. A microprocessor from a larger distributed robot control system may be dedicated to the safety system, but we expect this to have modest computational capability due to the requirements of reliability testing for space flight.

From the above characterization of the problem, we see that the safty system requires the following of its representation. First, the representation must allow determination in real time of imminent collisions. Second, the representation must constantly be updated, adjusting to the movements of robot arms and objects. Both updating and collision avoidance checking 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 octrees [31], [32] have been applied to spatial problems 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 Fig. 1. Changing the decomposition rule gives rise to many varieties of octrees. These different types of octrees have vastly different capabilities, and as with all computer science applica-

Manuscript received June 18, 1990; revised November 6, 1991. This work was supported in part by the NASA Goddard Space Flight Center under Grant NAG 5-1183.

The authors are with the Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061.

IEEE Log Number 9106453.

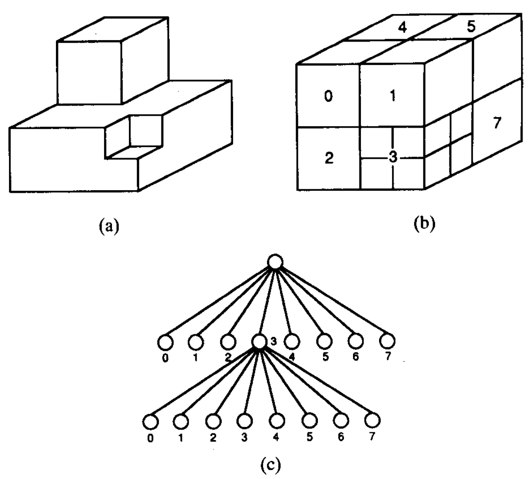

Fig. 1. An example region octree. (a) The object. (b) Its region octree block decomposition. (c) The resulting tree structure.

tions, selecting the correct data structure is crucial to success.

The most well-known form of octree is the *region octree*. It is most appropriate for defining the shapes of homogeneous objects that 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 Fig. 1).

Another type of octree, which we refer to here as the *N*-objects octree, has been used to speed ray tracing of images to simulate realistic lighting effects [13], [14], [24]. 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.

We use the *N*-objects octree to serve as the indexing method for the robots and their workspace for a number of reasons. The *N*-objects octree provides a compact spatial index of the world model. Its decomposition rule keeps the size of the *N*-objects octree small, which, as described below, allows for fast updating in response to movement. The *N*-objects octree easily adjusts to changes in the workspace through the splitting and merging of octants. Only objects sharing an octree node may intersect, thus localizing imminent collision detection and greatly reducing the number of intersection tests required. Finally, small object motions rarely require that the structure of the *N*-objects octree be changed. As a result, our system operates with an average cycle time of 30 ms (and under 60 ms in the worst case) for our test simulations.

The remainder of this paper is organized as follows. Section II present previous work related to our approach to collision avoidance and the *N*-objects octree. We describe our safety system prototype in Section III. Section IV discusses constraints in selecting the safety buffer size and explains why our algorithm does not run afoul of the completeness problem. Section V presents a series of experiments performed to evaluate the efficiency of our method. Section VI presents our conclusions and a number of open problems.

II. Previous Work

Many researchers have studied path planning problems in both static and dynamic environments (for overviews, see [33], [35]). With the expanded use of tele-operated robots in space, manufacturing, and the nuclear industry, the problem of collision avoidance has become a significant problem in its own right.

Cameron [4] offers three different approaches to determining collisions based on geometric modeling of the workspace. In the first, the motion is sampled at a finite number of times and interference detection between objects is performed at each time step. In the second approach, models are created of the objects and their motions in space time, and intersections between these four-dimensional entities are detected. The third approach models volumes swept out by the moving objects and checks for intersections. Most researchers combine collision avoidance with path planning, using either the second or third of Cameron's approaches. The method reported in this paper is based on Cameron's first approach.

One popular paradigm for path planning and collision avoidance is to model the robot's workspace in terms of the robot arm's configuration space [22], [23], [8]. 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. One advantage of our representation is that, unlike configuration space approaches, adding degrees of freedom to the robot arm either through modifying the joints or adding new links will not significantly affect the running time.

The use of region octrees in three or four dimensions to support path planning has been widely reported. Fujimura and Samet [12] have studied using four-dimensional region octrees (three spatial dimensions and time) to do robot planning. Samet and Tamminen [30] consider conversion of CSG trees to region bintrees (a close variant of the region octree) for use in solving object intersections to support both path planning and static intersection detection. Haywood [15], Hong and Shneier [18], Herman [17], and Noborio *et al.* [27], [21] have also considered the region octree for planning. As a typical example of these, Hayward [15] 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. Unfortunately, all of these region octree-based proposals require availability of complete information on future motion and thus are more appropriate to path planning than to a collision avoidance safety system.

A few researchers have explored solving a series of static collision detection problems at discrete times. Cameron [4] describes potential problems with this approach that result from inappropriate sampling rates (we discuss this issue further in Section IV). Roach and Boaz [2], [29] present the use of region octrees to detect intersection at a series of discrete times, although they do so to support path planning. 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 workspace. At each sample, a region octree is constructed for each moving object and static intersections tests are performed by parallel traversal of the octrees. Ng and Sakata [26] represent the static workspace environment with a region octree, and the moving robot as a series of *cylspheres*. At each update cycle, each cylsphere is tested against the octree for possible intersection.

Velocity and distance bounds are introduced in [7] as a means of detecting imminent collisions. Given two objects in space, we can determine the maximum velocity at which they are moving toward each other and the minimum distance between any two points on their surfaces. These bounds are used to determine the minimum amount of time *dt* that can elapse before these objects could possibly collide. The collision detection algorithm works by tracking this relation between each pair of solids in the world model. As the value for *dt* approaches zero for a pair of objects, an imminent collision is reported. Simple enclosing solids are used as fast intersection checks to improve efficiency. Foisy *et al.* [9] improve on this approach by organizing the N^2 possible collisions between the N primitives that make up the robot and its environment in order of earliest possible collision. At each time step, the primitive pair most likely to collide next is checked and reinserted into the list.

Minimum distance algorithms are discussed in [19] as a means of detecting imminent collisions in an off-line collision detection system with graphical simulation facilities. As objects move, a minimum distance measure is used to determine if any components of the pair of objects are about to collide. To reduce the number of pairwise comparisons between the components, each component was enclosed in a bounding box. The minimum distance algorithm (which is computationally expensive) was applied only to pairs of components whose bounding boxes were "close."

Yu and Khalil [36] present a system for collision avoidance of a robot working in a fixed workspace that uses a world model based on 3-D solid primitives. The robot and workspace 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.

Two very different approaches to real-time collision detection should be noted. Borenstein and Koren [3] apply the technique of *potential fields* in the form of a repulsive force-field around obstacles to keep collisions from occurring. Cheung and Lumelsky [7] use a skin of proximity sensors to determine when the robot arm has come too close to an obstacle—an approach radically different from the others described here since it is based on direct sensing rather than simulation of the workspace.

Many approaches have been reported for the general problem of collision detection and avoidance, a few of which have been described above. Some of the systems are targeted for off-line applications where system performance is not a major factor and operator interaction is possible [15]. Others are used in conjunction with planning systems to decide if preplanned robot motions are collision free [5], [12], [21], [27], [30]. Still others propose the use of custom hardware to support a real-time system [20], [34]. We desire a system that will provide real-time response with little or no interaction with the operator. Furthermore, no advance 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.

III. General Description of the Algorithm

Our approach to collision avoidance is to maintain a model of the robots' workspace and, as often as possible, update and evaluate the model to detect imminent collisions. In effect, a real-time simulation of the robots' workspace is performed. Imminent collisions are defined to be situations in which moving objects are closer than a specified distance to another object.

The major contribution of our work is the combination of a world modeled by solid primitives with an octree indexing scheme that allows static collision detection to be performed quickly enough to allow real-time sampling on a standard CPU. Ours is the first reported attempt to apply a hierarchical spatial index other than the region octree to collision detection. The region octree is not well suited to this task since it requires a relatively large number of nodes to represent the robot and its workspace, resulting in increased processing time. By selecting a more appropriate representation, we have made a significant efficiency advance on the solution to our formulation of the collision avoidance problem.

We represent complex objects as the union of simpler *primitives*. The choice of primitives does not affect our method of representation; however, the primitives used should reflect a good balance between efficiency of primitive–primitive intersection operations and the number of primitives required to adequately represent the world model (which affects the total number of primitive–primitive intersection operations performed). Actual intersection tests for a given pair of primitives are done analytically. While algorithms do exist to compute the intersection of arbitrarily complex solids [28], thus allowing the use of arbitrary shapes as primitives, we use only primitives for which intersection tests can be done quickly.

For our prototype, 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. Other objects in the workspace such as the experiment modules

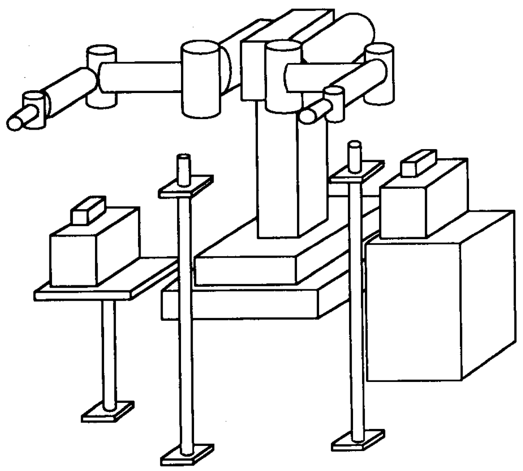
Fig. 2. Working environment used to test collision detection algorithms.

to be manipulated by the arms are not well modeled by cylspheres, so our prototype also supports *rectangular solids* as additional primitives. Each primitive in the model is considered a separate entity and is assigned a unique identification number. A geometric description of each primitive 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 world model (shown in Fig. 2) with only 35 primitives. We expect that the world model for a wide range of applications can be represented with at most a couple of hundred primitives.

When primitives are used to construct more complex objects, such primitives may 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. Due to the small number of primitives required by our world model, compatibility between primitives is represented by a two-dimensional array in which the entry at row *i* and column *j* is true if primitives *i* and *j* are compatible and false otherwise. Larger world models should use a more sophisticated approach to storing compatibility information.

We now turn to the problem of collision avoidance. We wish to detect *imminent* collisions, with the intention of avoiding them. Thus, the safety system should issue a warning when two primitives come "too close" to one another. During each time step, we must check the current position of all primitives to see if this has happened. When a certain distance between noncompatible objects is to be maintained, the standard technique for a static environment is to extend each primitive by half the width of the *safety buffer* in all directions. Since we are working with a dynamic environment in which a few primitives move, but most primitives are static, we instead extend moving primitives by the full width of their safety buffer and do not extend static primitives at all. An *extended primitive* is one that has been enlarged to include its safety buffer. Whenever extended (incompatible) primitives overlap, a collision warning can be issued (the proper response to a collision warning is beyond the scope of this paper). Criteria for determining the appropriate safety buffer size will be discussed in Section IV.

To minimize the number of unnecessary intersection tests between extended primitives during the collision detection phase, an indexing scheme over the workspace is needed to determine which extended primitives are close to each other and which are not. The *N*-objects octree provides such an index. Two extended primitives may intersect only if they share a node in the *N*-objects octree. We have selected the *N*-objects octree over other octree variants for efficiency reasons. The standard region octree will require far too many nodes to allow efficient update (as experienced by Roach and Boaz [29]). The *nonminimal division* octree of Ayala *et al.* [1] and the closely related *polytree* of Carlbom *et al.* [6], while an improvement over the region octree, will likewise require too many nodes and redundant intersections. We prefer the approach of intersecting simple primitives analytically, with the *N*-objects octree serving as an index to minimize the number of intersection tests.

The *N*-objects octree (referred to hereafter simply as "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 workspace 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
   { Coordinates for corners of octant }
   vertices : array[0..7] of Point;
   { Internal or leaf node }
   isSplit : Boolean;
   { Pointers to node's children }
   children : array[0..7] of ↑OctNode;
   parent : ↑OctNode; { Node's parent }
   { Which child this node is of parent }
   childNo : 0..7;
   { Number of primitives contained }
   numObjs : integer;
   { Head of contained primitives list }
   assocObjs : ObjList
end;
```

The first step in modeling the workspace is to build the octree. We begin with the root of the octree as an empty cube enclosing the workspace. Primitives 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 primitives supported and the denseness of the workspace, although we found that our application is not sensitive to the value of *N* (also see [25]). Fig. 3 shows a 2-D workspace stored in an *N*-objects quadtree with $N = 5$.

Given a primitive and a node in which to insert it (initially

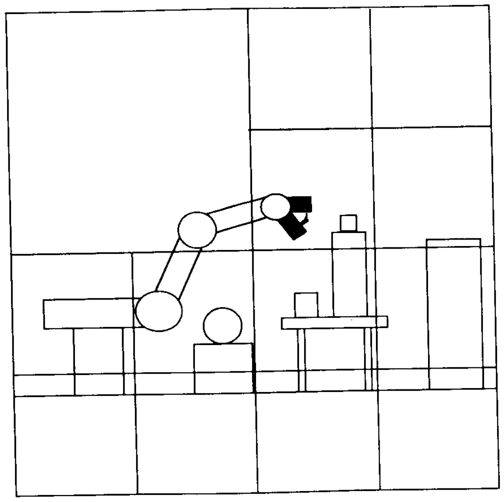

Fig. 3. Decomposition for sample environment using a five-object quadtree.

the root), insertion proceeds as follows. If no part of the primitive lies inside the node, then do nothing. If the node is an internal node, then recursively insert the primitive into each of the node's children. If the node is a leaf and the number of primitives already in the node is less than N, then add the new primitive's ID to the node's object list. Otherwise, split the node, inserting all of its primitives into the node's newly created children, and recursively repeat the insertion process for the new primitive at each child. This split-insert process is repeated until all leaf nodes contain no more than N primitives. The following Pascal-like pseudocode formalizes the insertion process.

```
{ Insert a primitive into a node. }
procedure InsertObj(objId: integer;
                    node: ↑OctNode);
{ObjNodeIntersect returns TRUE iff the
 primitive lies within the node.
 AddObjToNode and RemoveObjFromNode
 inserts and removes the primitive ID
 from node's object list, respectively. }
var child: Octant;
begin
  if ObjNodeIntersect(objId,node)
  then
    if node↑isSplit then
      for child := 0 to 7 do
        InsertObj(objId,
                  node↑children[child])
    else if node↑numObjs< N
      then AddObjToNode(objId,node)
      else SplitInsert(objId,node)
end;

{ Recursively split and insert a primitive
  into a node. }
procedure SplitInsert(objId : integer;
                      node  :↑OctNode);
var
  objId: integer;
  child: Octant;
begin
  { SplitNode creates children and links
    them to node }
  SplitNode(node);
  for each objId in node↑assocObjs do
  begin
    for child:=0 to 7 do
      if ObjNodeIntersect(objId,
                          node↑children[child])
      then
        AddObjToNode(objId,
                     node↑children[child]);
        RemoveObjFromNode(objId,node)
  end; { for each objId in node }
  for child:= 0 to 7 do
    if ObjNodeIntersect(objId,
        node↑children[child])then
    begin
      if node↑children[child]↑numObjs< N
      then AddObjToNode(objId,
                        node↑children[child])
      else  SplitInsert(objId,
                        node↑children[child])
    end
end;
```

Once the world model has been built, the safety system operates as a continuous series of update/collision detection cycles. At the start of 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 robot arm link is parallel to a coordinate axis. Using these values, simple kinematic transformations are applied to determine the current position of each link in Cartesian space. For each arm, beginning with the end effector and working backward toward 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 that represents it. As a byproduct 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 primitive is inserted into the octree. This might be a good idea if a large portion of the workspace changed during each cycle. However, due to the short cycle time (30–60 ms) combined with restrictions on robot arm speed, we expect only small changes in position during each cycle. Such changes rarely require modification to the octree since the updated primitives rarely move to new octree nodes. A second approach is to delete and reinsert moving primitives. This approach requires modification of a relatively small portion of the octree, but will cause expensive and unnecessary merges and splits. Objects move very small distances during a short cycle time, and thus a moving primitive is frequently reinserted into the same nodes from which it was deleted. However, when such a node and its siblings contain N+1 primitives, the nodes will be merged when the moving primitive is deleted from the octree, only to be split again when the primitive is reinserted.

A more efficient update process changes the structure of the octree only when changes in the workspace dictate. The tree structure changes only when a primitive exits a node (causing the primitive to be deleted from that node) or moves into a new node (causing the primitive to be inserted). These events, in conjunction with the decomposition rule, may cause nodes in the octree to merge or split. Further, when an primitive enters a new node, that node must be a *neighbor* of a node in which it currently resides (this will be discussed further in Section IV). Two nodes are considered neighbors if they share a face, edge, or corner. With this approach, octree updates work as follows. First, locate all nodes that the moving primitive resided in before it moved. For each of these nodes locate all of its neighbors using neighbor finding techniques described by Samet [31]. If the primitive has moved into a neighbor node, then insert it at that node and split if necessary. Upon completion, check if the primitive has exited any of the nodes it resided in before the move. If so, then delete the primitive from such nodes and try to merge them with their siblings.

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

The amount of computation needed for an update when a primitive moves into a neighboring node can be further reduced. When a primitive moves into a neighbor, the algorithm described above will insert the primitive into that node and perform any required splitting. If the neighbor is split, the algorithm will attempt to recursively insert the primitive 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 that lies closest to the original node. So, the primitive is inserted into only the leaf descendents of the neighbor that lie on the common face, edge, or corner between the two nodes.

When a single primitive 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 a primitive moves into a node X that 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 primitive's ID into X's associated object list three times when only once is necessary. Furthermore, two of the nodes in which a primitive lies can be neighbors, causing the algorithm to add the primitive 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 that have been visited while moving a primitive. Each node in the octree 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 time step. When a primitive 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 a primitive, *ThisCheckNo* is incremented by one. Using an unsigned 32-b integer, *ThisCheckNo* will reset to zero after 4 294 967 296 updates. At this point, the octree should be traversed and every node's *lastCheckNo* reset. Assuming a 50-ms cycle time with 14 updates of the octree required during each cycle (all seven links of both arms moved), this occurs approximately every six months during continuous operation.

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

The third step of the safety system cycle is collision checking. After a primitive moves, all primitives in all nodes that contain it are checked for possible collisions. In the octree, multiple moving primitives may reside in the same node or a moving primitive and a static primitive may share more than one node. To eliminate any redundant intersection tests, we keep track of which pairs of primitives have been checked for collisions during the current cycle. Thus, a complete intersection test between a pair of primitives will be performed at most once (although our update algorithm may check several times to see if a given pair has been tested).

Intersection tests are an essential part of the collision avoidance system. Tests between primitives are performed to determine possible collisions. Tests between primitives and nodes are performed to build and update the octree. Additional primitives can be included by simply adding the appropriate intersection tests. To improve the efficiency of all intersection tests, a standard bounding box test is performed first in hopes of quickly ruling out an intersection. For more complete details on our intersection operations, see [16]. The following Pascal-like psuedocode provides a more formal description of the updating and collision detection process.

```
{ Update the octree when a primitive has
  moved. }
procedure UpdateObj(objId : integer);
{ ObjId is the label for a primitive.
```

```
  Location of objId is a member of the
  set of nodes containing that primitive.
  ObjNodeIntersect determines if a
  primitive lies in a node.
  CollisionInNode determines if a
  primitive collides with (intersects) any
  of the primitives in a node, by simply
  performing a primitive-primitive
  intersection check against each such
  primitive. GetUnCheckedNeighbors returns
  a list of all neighbors of a node which
  have not been visited during the current
  update. BoundingBoxInsideNode determines
  if a primitive's bounding box lies
  completely inside a node.
  NeighborInDirection determines if a
  neighbor lies in the direction of the
  moving primitive. DeleteObjFromNode
  removes a primitive's association with a
  node and performs any possible node
  merging. }
var
  location : ↑OctNode; { A node in which
                         primitive resides }
  neighbor : ↑OctNode;
  nbrs : NeighborList;
begin
  ThisCheckNo := ThisCheckNo + 1;
  { Check intersections in node already
    containing objId }
  for each location of objId do begin
    if CollisionInNode(objId, location)
    then HandleCollision; { Send Warning }
    location↑lastCheckNo := ThisCheckNo
  end;
  for each location of objId do begin
    { Check nodes objId moves into }
    if not BoundingBoxInNode(objId,
                             location)
    then begin
      GetUnCheckedNbrs(location, nbrs);
      for each neighbor in nbrs do
        if NbrInDirection(neighbor) then
          if ObjNodeIntersect(objId,
                              neighbor)
          then UpdateNbr(objId, neighbor)
          else neighbor↑lastCheckNo :=
                             ThisCheckNo
    end;
    if not ObjNodeIntersect(objId,
                            location)
    then  { Primitive moved out }
      DeleteObjFromNode(objId, location)
  end
end;

{ Update a node that a primitive has just
  moved into. }
procedure UpdateNbr(objId : 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
        UpdateNbr(objId,
                  node↑children[child])
    else begin
      neighbor↑lastCheckNo := ThisCheckNo;
      if ObjNodeIntersect(objId, neighbor)
      then begin
        if CollisionInNode(objId,
                           neighbor) then
          HandleCollision { Send warning }
        else if neighbor↑numObjs < N
        then  AddObjToNode(objId, neighbor)
        else UpdateSplitInsert(objId,
                               neighbor)
      end { if ObjNodeIntersect }
    end { else }
  end { for }
end;

{ Recursively split and insert a primitive
  into a node during update. }
procedure UpdateSplitInsert(
    objId : integer; node : ↑OctNode);
var
  objId : integer;
  child : Octant;
begin
  SplitNode(node);
  for each objId in node↑assocObjs do
  begin
    for child := 0 to 7 do
      if ObjNodeIntersect(objId,
                          node↑children [child])
      then  AddObjToNode(objId,
                         node↑children [child]);
      RemoveObjFromNode(objId, node)
  end;

  for child := 0 to 7 do begin
    node↑children[child]↑lastCheckNo :=
          ThisCheckNo;
    if ObjNodeIntersect(objId,
                 node↑children[child]) then
      if node↑children[child]↑numObjs < N
      then AddObjToNode(objId,
                        node↑children[child])
      else UpdateSplitInsert(objId,
                        node↑children[child])
end;
```

IV. The Completeness Problem

Since our system checks for violations of primitives' safety buffers only at discrete time intervals, we must insure that a collision cannot take place between consecutive checks. This is known as the *completeness problem.* This problem is avoided by making the safety buffer wide enough so that an imminent collision will be detected and appropriate action taken regardless of when in the update/detection cycle of the safety system that safety buffer violation occurs. The converse problem is to determine, for a given safety buffer size, how often imminent collision detection must take place. In other words, how long can the cycle time be to insure adequate coverage? For our application, we are concerned solely with minimizing the cycle time since we anticipate a microprocessor being dedicated to the safety system as part of a larger distributed robot control system.

In a dynamic robotics environment, several factors affect the minimum size of the safety buffer. First, the current position of the arms (which in our case is indicated by the current joint angles) must be passed from the controller to the safety system. The safety system must recognize that a collision is about to occur and issue a shutdown command. The controller must then 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 safety system can vary widely between different hardware, ranging from only a couple milliseconds (the expected time for the arms NASA intends to fly on the space station) to typically 50 ms. (as required for NASA's current test robots). In our tests, our safety system requires 30–60 ms to issue the shutdown command. The time to actually stop is in the 10–20 ms 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 40 to 120 ms at maximum speed. With maximum speed of the end effectors limited to 24 in/s, which yields between a 1- and 3-in tolerance zone around each moving primitive, depending on the values selected.

In a dynamic environment we may also wish to account for the relative speeds of moving primitives. If a primitive is moving at less than maximum speed, the minimum tolerance for that primitive 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 1 in. 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 in 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 2 in of each other (the exception being when the operator wishes to grasp an object). Changes in the width of the tolerance zone should have little or no effect on our algorithm's performance. Note that since we sample positions at specified times, it is still possible that the safety buffers of two moving primitives may briefly overlap between cycles. This does not present a problem since this does not represent an imminent collision.

A fundamental assumption used by our update algorithm is that a primitive cannot move *through* a node between consecutive updates. If this were not true, then our premise that between updates a primitive can move only into neighbor nodes would no longer hold. Consider, for example, a primitive that has moved out of a node, through one of its neighboring nodes, and into the next nonneighboring node. The update algorithm described in the previous section would recognize that the primitive has moved into the neighboring node, but not into the node beyond.

The maximum distance that an object can move in one cycle is used to determine the minimum size for a leaf node. For example, if the robot arm tip can move 1 in in an update cycle, then the smallest node allowed in the octree would have an edge length of 1 in. 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. Thus, by placing a limit on the minimum node size that is proportional to object speed, we guarantee that all nodes containing a moving object are checked for collisions.

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

V. Experimental Results and Analysis

Our collision avoidance safety system was implemented using the C language, running under UNIX. All timing results are for an Intel 80386 CPU with math coprocessor running at 20 MHz. 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 three-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 an imminent collision is detected, the algorithm terminates, indicating which primitives are about to collide. Upon completion, the algorithm reports timing results.

Our tests consisted of three separate tasks using a fixed workspace (shown in Fig. 2). Our workspace 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 workspace to simulate realistic operation. The first task was comprised of the left arm moving toward 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, 150, 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 ms. 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 octree 14 times (14 primitives have moved) whereas the second case requires updating the octree only once. The time required for each cycle of the algorithm was measured using the system clock, which had a resolution of 10 ms. The observed upper bound for the range of update cycle times was under 60 ms.

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 its efficiency. A more efficient algorithm could offer significant improvement. The remainder of the computation time was used to update the octree and check for intersections. About 27% of the time was spent performing primitive–primitive intersection tests and about 7% calculating rotation angles and bounding boxes for moving primitives. Primitive-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, intersection tests, and octree maintenance each required roughly one third of the computation time.

Two characteristics of the N-objects octree that 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 primitives) was split only two levels below the root and contained 33 nodes, of which 29 were leaf nodes. The average number of primitives in each nonempty leaf node was about 5 (with $N = 10$), while 15 of the nodes were empty. Each primitive resided in about two nodes on the average. So the occupancy of each leaf node as well as the number of nodes occupied by each primitive 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-ms cycle time, this translates into once every 18 s.

In Section III, 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 primitives decreased computation time by 4%. Calculating the direction(s) of a moving primitive and updating only neighbors in this direction reduced computation time by 8%. Checking if a primitive'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 primitive–primitive and primitive–node intersection tests had the most significant 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." A large portion of the computation required by an octree update consists of primitive–node and primitive–primitive intersection tests. Primitive–node tests are needed to determine if a primitive has moved into a new node. Primitive–primitive tests are used to detect imminent collisions between primitives. The value chosen for N directly affects the number of each type of intersection test performed during an update. For example, the choice of a small N causes the octree to decompose to a much lower level than a large N. This deeper splitting, in general, increases the number of nodes that a primitive 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 octree is not as deep and we have fewer nodes to process during the update. However, since more primitives are allowed to share a node, when a primitive moves, more primitive–primitive 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 primitive–node and primitive–primitive intersections, the value chosen for N directly effects system performance. If the cost of performing a primitive–primitive intersection test is much more expensive than the cost of a primitive–node intersection test then a large value for N would optimize the update process.

Given the primitives supported by our representation and the workspace of the robot arms, we have chosen a value of 10 for N. This value resulted in optimal performance given the relative costs of primitive–node intersection tests (70 μs) and primitive–primitive intersection tests (150 μs). Fig. 4 shows how system performance varied for different values of N. Although $N = 10$ 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 workspace would not require that a different value for N be used.

A natural question to ask is how does the octree compare in performance to the naive approach to intersection testing? The naive algorithm is one that, when a primitive moves, would check for possible intersections with *all* other primitives in the world. The computation time for such an algorithm grows in proportion to the number of primitives in the world (assuming a constant number of primitives 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 as much overhead as the octree. For a sufficiently simple workspace, the naive approach is more efficient than

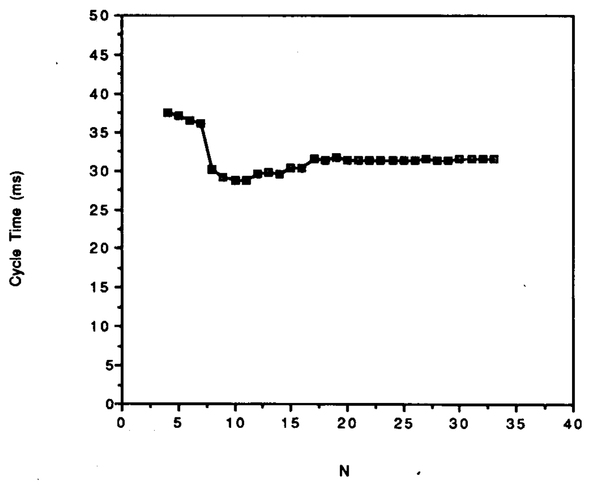

Fig. 4. System performance for different values of N in decomposition rule.

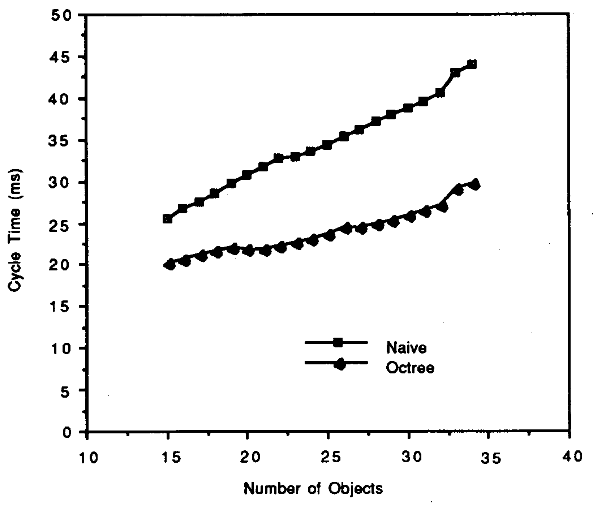

Fig. 5. System performance of naive and octree algorithms as environment grows in size.

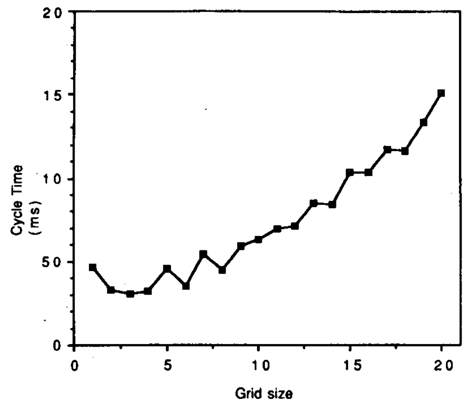

Fig. 6. System perofrmance of grid algorithm as grid size (G) is varied. Values are averaged over all 3 tasks.

the octree. Conversely, the octree is more efficient for a more complicated workspace. The question is, at what point does the octree perform better?

The naive algorithm was implemented and tested with the same three tasks described above. The number of primitives in the robot's workspace was varied, and timing results were recorded. The same tests were repeated using the octree version of the algorithm. Fig. 5 illustrates the behavior of the two algorithms. The cycle times for both algorithms increased as primitives were added to the workspace. In both cases though, the primitives were added into the immediate area surrounding the two robot arms. Other primitives could have been strategically placed in the workspace, 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, primitives could be placed in parts of the octree 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 ever greater performance gains over the naive method as the workspace became more complex. In all tests, the octree was superior to the naive method, even when only 15 primitives were used (the two robot arms and one box).

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 [10], [11]. A grid is overlayed onto the data, and for each grid cell, a set containing each primitive that lies in that cell is formed. Such a representation was implemented to minimize intersection tests and compared to the octree approach. To simplify implementation, the grid was represented as a three-dimensional $G \times G \times G$ 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. Fig. 6 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.

Finally, we compared the N-objects octree to the more traditional region octree. A region octree was implemented using our world model of cylspheres and rectangular solids. Using a resolution of 1.2 in (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 3000 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 ms time span).

The performance of the *N*-objects octree is directly affected by the number of stationary and moving primitives in the workspace. When a stationary primitive is added to the workspace, it may increase the algorithm's cycle time in a number of ways. The new primitive may cause the octree to split, in which case the algorithm may be required to process more nodes during an update. The primitive 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 primitive 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 primitives only increase cycle time if they are included in nodes that also contain moving primitives. Thus, the number of stationary primitives affecting cycle time is closely related to the percentage of nodes containing moving primitives.

Adding a moving primitive, for example, another robot arm, will have a more predictable effect on cycle time. In a given cycle, every moving primitive in the workspace causes the octree to be updated with respect to that primitive. When the number of moving primitives in the world is doubled, we would expect the time needed to update the octree to also double (assuming that the moving primitives make up a reasonably small percentage of the total number of primitives). 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 that moved during that cycle.

VI. Conclusions and Future Work

With the expanded use of tele-operated robots in space, manufacturing, and the nuclear industry, the problem of collision avoidance 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 proposed 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 to which our system can be applied. Information about the robots and their workspace must be available to the system in the form of a geometric model using supported primitives. Thus, the shape of the workspace must be suited to such a representation. If information about the workspace 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. However, if somebody 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. The system is well suited for an application where an approximate model is acceptable. If greater accuracy is required, other primitives may be selected that better represent the workspace (possibly at the cost of more expensive intersection tests), or more primitives may be used (at the cost of more intersection tests). We also require that no primitive will move into, through, and out of a node in a single cycle. This is guaranteed due to the use of a tolerance zone to extend the size of primitives and a minimum limit to octree node size. Since our cycle time is so short, both the resulting tolerance zone and the minimum node size should be reasonable in practice. 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 primitives.

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 octree 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 that 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 (since mass affects stopping time—a major factor in safety buffer size).

The *N*-objects octree could also serve as the underlying representation for a robot planning system, as has been proposed for the region octree in [12], [15], [27], [29]. Using a generate-and-test paradigm, the *N*-objects octree could serve as a means of determining if a given plan would be collision free. Alternatively, the *N*-objects octree could be used as a search space for a robot planning system. Using octree 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 real-time collision avoidance and safety systems to meet the operational requirements of today. The octree has proven to be a useful data structure both for developing current systems and researching systems for the future.

References

[1] D. Ayala, P. Brunet, R. Juan, and I. Navazo, "Object representation by means of nonminimal division quadtrees and octrees," *ACM Trans. Graphics*, vol. 4, no. 1, pp. 41–59, 1985.

[2] M. Boaz, "Spatial coordination of transfer movements in a dual robot environment," Master's thesis, Virginia Polytech. Inst. and State Univ., Blacksburg, 1984.

[3] J. Borenstein and Y. Koren, "Real-time obstacle avoidance for mobile robots," *IEEE Trans. Syst. Man Cybern.*, vol. SMC-19, no. 5, pp. 1179–1187, 1989.

[4] S. Cameron, "A study of the clash detection problem in robotics," in *Proc. IEEE Int. Conf. Robotics Automat.* (St. Louis, MO), Mar. 1985, pp. 488–493.

[5] ——, "Collision detection by four-dimensional intersection testing," *IEEE Trans. Robotics Automat.*, vol. 6, no. 3, pp. 291–302, 1990.

[6] I. Carlbom, I. Chakravarty, and D. Vanderschel, "A hierarchical data structure for representing the spatial decomposition of 3-D objects," *IEEE Computer Graphics Applicat. Mag.*, vol. 5, no. 4, pp. 24–31, 1985.

[7] E. Cheung and V. Lumelsky, "Motion planning for a whole-sensitive robot arm manipulator," in *Proc. IEEE Int. Conf. Robotics Automat.* (Cincinnati, OH), 1990, pp. 344–349.

[8] B. Faverjon, "Obstacle avoidance using an octree in the configuration space of a manipulator," in *Proc. IEEE Int. Conf. Robotics Automat.* (Atlanta, GA), 1984, pp. 504–512.

[9] A. Foisy, V. Hayward, and S. Aubry, "The use of awareness in collision prediction," in *Proc. IEEE Int. Conf. Robotics Automat.* (Cincinnati, OH), 1990, pp. 338–343.

[10] W. R. Franklin, "Adaptive grids for geometric operations," in *Proc. 6th Int. Symp. Automated Cartography*, vol 2, Oct. 1983, pp. 230–239.

[11] W. R. Franklin, M. Kankanhalli, and C. Narayanaswami, "Efficient primitive geometric operations on large databases," in *Proc. GIS Nat. Conf.* (Ottawa, Canada), Mar. 1989, pp. 59–67.

[12] K. Fujimura and H. Samet, "Path planning among moving obstacles using spatial indexing," in *Proc. IEEE Int. Conf. Robotics Automat.* (Philadelphia, PA), Apr. 1988, pp. 1662–1667.

[13] A. S. Glassner, "Space subdivision for fast ray tracing," *IEEE Computer Graphics Applicat. Mag.*, vol. 4, no. 10, pp. 15–22, 1984.

[14] ——, *An Introduction to Ray Tracing*. San Diego: Academic, 1989.

[15] V. Hayward, "Fast collision detection scheme by recursive decomposition of a manipulator workspace," in *Proc. IEEE Int. Conf. Robotics Automat.* (San Francisco), Apr. 1986, vol. 2, pp. 1056–1063.

[16] G. M. Herb, "A real time robot collision avoidance safety system," Masters thesis, Virginia Polytech. Inst. and State Univ., Blacksburg, May 1990.

[17] M. Herman, "Fast, three-dimensional, collision-free motion planning," in *Proc. IEEE Int. Conf. Robotics Automat.* (San Francisco), Apr. 1986, pp. 1056–1063.

[18] T.-H. Hong and M. Shneier, "Describing a robot's workspace using a sequence of views from a moving camera," *IEEE Trans. Pattern Anal. Machine Intell.*, vol PAMI-7, no. 6, pp. 721–726, 1985.

[19] G. Hurteau and N. F. Stewart, "Distance calculation for imminent collision indication in a robot system simulation," *Robotica*, vol. 6, no. 1, pp. 47–51, 1988.

[20] S. Kokaji, "Collision-free control of a manipulator with a controller composed of sixty-four microprocessors," *IEEE Control Syst. Mag.*, vol. 6, no. 5, 9–14, 1986.

[21] Y.-H. Liu, S. Kuroda, T. Naniwa, and H. Noborio, "A practical algorithm for planning collision-free coordinated motion of multiple mobile robots," in*Proc. IEEE Int. Conf. Robotics Automat.* (Scottsdale, AZ), May 1989, pp. 1427–1432.

[22] T. Lozano-Perez, "Automatic planning of manipulator transfer movements," *IEEE Trans. Syst. Man Cybern.*, vol. SMC-11, no. 10, pp. 681–698, 1981.

[23] ——, "Spatial planning: A configuration space approach," *IEEE Trans. Comput.*, vol. C-32, no. 2, pp. 108–120, 1983.

[24] J. D. MacDonald and K. S. Booth, "Heuristics for ray tracing using space subdivision," in *Proc. Graphics Interface 89*, 1989, pp. 152–163.

[25] R. C. Nelson and H. Samet, "A population analysis for hierarchical data structures," in*Proc. SIGMOD Conf.* (San Francisco), May 1987, pp. 270–277.

[26] M. T. Ng and H. Sakata, "MSS implementation plan," SPAR Aerospace, Toronto, Canada, Internal Tech. Rep. SPAR-SS-PL-0501, Feb. 1989.

[27] H. Noborio, T. Naniwa, and S. Arimoto, "A feasible motion-planning algorithm for a mobile robot on a quadtree representation," in *Proc. IEEE Int. Conf. Robotics Automat.* (Scottsdale, AZ), May 1989, pp. 327–332.

[28] F. P. Preparata and M. I. Shamos, *Computational Geometry*. New York: Springer-Verlag, 1985.

[29] J. W. Roach and M. N. Boaz, "Coordinating the motions of robot arms in a common workspace," *IEEE J. Robotics Automat.*, vol. 3, no. 5, pp. 437–444, 1987.

[30] H. Samet and M. Tamminen, "Bintrees, CSG trees, and time," *Computer Graphics*, vol. 19, no. 3, pp. 121–130, 1985.

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

[32] ——, *The Design and Analysis of Spatial Data Structures*. Reading, MA: Addison-Wesley, 1990.

[33] M. Sharir, "Algorithmic motion planning in robotics," *Computer Mag.*, vol. 22, no. 3, pp. 9–20, 1989.

[34] R. C. Smith, "Fast robot collision detection using graphics hardware," in *Proc. IFAC Symp. Robotic Control* (Barcelona, Spain), 1985, pp. 277–282.

[35] S. H. Whitesides, "Computational geometry and motion planning," in *Computational Geometry*, G. T. Toussaint, Ed. Amsterdam: North-Holland, 1985, pp. 377–427.

[36] Z. Yu and W. Khalil, "Table look up for collision detection and safe operation of robots," in*Theory of Robots* (selected papers from IFAC/IFIP/IMACS Symp., Vienna, Austria), Dec. 1986, pp. 343–347.

Clifford A. Shaffer (S'83–M'86) received the Ph.D. degree in computer science from the University of Maryland, College Park, in 1986.

Since 1987, he has been an Assistant Professor in the Department of Computer Science at Virginia Polytechnic Institute and State University, Blacksburg. His primary research interests are in the area of spatial data structures. In addition to applying advanced data structures to support real-time collision avoidance systems, he has implemented several computer mapping systems based on hierarchical data structures.

Gregory M. Herb received the Master's degree from Virginia Polytechic Institute and State University, Blacksburg, in 1990.

Since then, he has been working for the Department of Defense. His current research interests include object-oriented design and programming and man–machine interface.