A REAL-TIME ROBOT COLLISION AVOIDANCE SAFETY SYSTEM

by

Gregory M. Herb

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

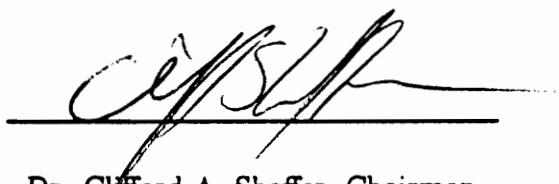
for partial fulfillment of the requirements for the degree of

Master of Science

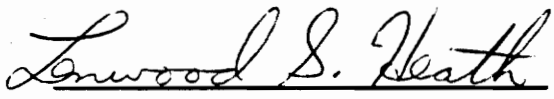
in

Computer Science and Applications

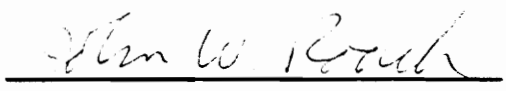
APPROVED:



Dr. Clifford A. Shaffer, Chairman



Dr. Lenwood S. Heath



Dr. John W. Roach

May, 1990

Blacksburg, Virginia

A REAL-TIME ROBOT COLLISION AVOIDANCE SAFETY SYSTEM

by

Gregory M. Herb

Dr. Clifford A. Shaffer, Chairman

Computer Science

(ABSTRACT)

A data structure and update algorithm are presented for a prototype real-time collision avoidance safety system supporting tele-operated robot arms. 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 (nodes) until each octant meets some decomposition criteria. Our octree stores cylspheres (cylinders with spheres on each end) and rectangular solids as primitives. 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 the other arm or an object in the environment are reported. Cycle time for receiving current joint angles, updating the octree, and detecting/reporting collisions is about 30 milliseconds on an Intel 80386 processor running at 20 MHz.

Acknowledgements

I would like to express my thanks to Dr. Clifford A. Shaffer for his guidance during the research and writing of this thesis. His advice and suggestions have been invaluable. I would also like to thank Dr. John W. Roach and Dr. Lenwood S. Heath for serving as members of my committee. Thanks are also due to NASA/Goddard for providing the funding that made this research possible. Finally, I would like to thank Isabel, Sam, and Barney for their unending patience and support.

CONTENTS

List of Illustrations

1. INTRODUCTION

1.1. Overview

This thesis describes the use of a hierarchical data structure, the *N-Objects* octree, in a collision avoidance 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 algorithm described can be used to provide a collision avoidance capability for a variety of robotics applications.

In particular, our target environment is NASA's space station operating bay containing two tele-operated robot arms and various objects to be manipulated and avoided by these two arms. A tele-operated robot is a robot 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. The operator's view of the work area is limited to two small windows and two video monitors receiving input from cameras mounted in the operating bay. The tasks performed by the operator include positioning and initiating experiment modules, and capturing and repairing damaged satellites. Given the working conditions of the operator and the tasks being performed, we can conclude that the probability of an accident occurring is unacceptably high. Furthermore, recovery from such an accident may be quite expensive, if not impossible. Hence, a safety system is needed which would prevent unintentional collisions. The purpose of a collision avoidance system is to provide safe operation of the robot arms without hindering the efforts of the operator.

Each robot arm in our model has seven degrees of freedom, with each section of the arm being nearly cylindrical in shape. The operating environment is not static so we must model the movements of the arms, and possible movements of other objects. The operating paradigm is one of receiving an indication of movement by certain objects, updating our 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 avoidance system requires the following of its representation. First, the representation must provide a spatial index that can determine if robot arms/objects are about to collide. Second, the representation must be capable of adjusting to the movements of the robot arms or objects. 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. Finally, since we are designing a real-time system, the updating and collision detection must consistently be performed within the permitted time period.

The octree [Chen85, Same89, Same90] has become a popular data structure for applications in computer vision, robotics, computer graphics, and CAD/CAM 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. 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 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 completely outside of one (see Figure 1).

Another type of octree, which we 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, recording the objects that inhabit each disjoint region of space. Beginning with a cube that encloses all of the objects, splitting occurs until no more than N objects lie in any unsplit octant. The value we choose for N depends on the characteristics of the environment being modeled and the task being performed. We observe that with the N-Objects octree we are not so much interested in the precise shapes of the objects as we are in their relative positions to each other.

We have chosen the N-Objects octree to serve as the underlying representation of our collision avoidance 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 will see, allows for fast updating. The N-Objects octree is also 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 overall system, small object motions rarely require that the structure of the tree be changed.

1.2. Background and Definitions

Before describing our representation we define several terms pertaining to octrees. An octree is split into eight octants labeled 0 through 7. Corresponding to this splitting is a tree representation of the decomposed space (as shown in Figure 2). Nodes within the octree are associated with octants in space. An octree contains two types of nodes: *internal*

nodes and *leaf* nodes. For our implementation, the leaf nodes hold the actual data to be stored in the tree; internal nodes serve as pathways to the leaves of the tree. The *root* node of the octree represents the entire space. When an octant splits, eight sub-octants are formed. The newly formed nodes are the *children*; the original node is called their *parent*. *Siblings* are nodes that have the same parent node. *Ancestors* of a node are all of the nodes on the path from the current node to the root node of the tree.

Two storage methods can be used to represent an octree; *linear* octrees and *pointer-based* octrees. Linear octrees [Garg82] maintain the octree as a list of cubes indexed by their coordinates. A pointer-based octree uses the traditional tree structure to represent the block decomposition hierarchy, complete with pointers linking a parent node to its children. Pointer-based octrees are typically used when storing the octree completely in main memory. Conversely, linear representations are traditionally used for disk-based applications [Same89]. Since our memory requirements are relatively small while time requirements are rigid, we have adopted the pointer-based approach to provide quick access to the octree.

The rest of this paper describes our implementation as follows. Chapter 2 presents a brief overview of previous work done in this area. Chapter 3 provides a general overview and implementation independent aspects of the algorithm while Chapter 4 describes our implementation in more detail. Chapter 5 discusses experimental results, while Chapter 6 is devoted to conclusions and future work.

2. PREVIOUS WORK

2.1. Collision Detection and Avoidance Systems

Many researchers have studied path planning problems in moving environments (for an overview, see [Whit85]). We restrict ourselves to the simpler problem of collision detection and do not consider the more general planning problem further. 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. Thus, researchers have increasingly dedicated time and money to producing effective solutions.

One application of collision detection systems is to allow safe operation in a flexible assembly cell. Gouzenes [Gouz84] proposes collision detection at two different levels: gross motion security and inter-system security. Gross motion security prevents catastrophic collisions which lead to deformation or partial destruction of the robot and other objects in its immediate environment. It is provided by maintaining a set of all possible robot configurations which keep the robot at least a minimum distance from the obstacles. This set is structured into a graph. Safe gross motion from one configuration to another is obtained by a graph search. Inter-system security is needed to prevent collisions between multiple robot arms operating asynchronously in a shared workspace. The use of Petri nets to protect a multi-robot system from collisions is explained by the author. Space and work areas are considered as resources which will be allocated and deallocated for the different robots. As robots perform their tasks, these resources are requested and allocated such that no collisions are possible.

The use of region octrees to support collision detection is presented in [Boaz84, Roac87]. The problem of collision detection arises when the motions of multiple robot arms

in a common workspace need to be coordinated. 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.

Cameron [Came85] also addresses the problem of collision detection with respect to robot motion planning. That is, given a pre-ordained motion, determine whether a collision will occur. The author offers three different approaches to determining collisions based on geometric modeling of the environment. In the first, the motion is sampled at a finite number of times and interference detection between objects is performed at each time. 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. In the third approach, models of the volumes swept out by the moving objects are created and checked for intersections.

A method using clipping algorithms is presented in [Smit85] to efficiently determine interference between two modeled objects. This method is used to perform real-time collision detection in a robot simulation program implemented on a graphics workstation. When a robot arm moves through the environment, the convex polyhedra which represents it is clipped against other objects in the geometric model of the robot's world. To facilitate real-time response, standard VLSI graphics hardware is used to perform the clipping tests between the robot and other objects. Smith describes how the graphics hardware could be integrated into a robot control system to provide on-line collision prediction and avoidance.

Information about the robot's trajectory is provided to the collision detection system in advance. Information about the robots' surroundings are either pre-programmed or acquired through the use of sensors. The detection algorithm is applied to points lying ahead of the robot on its current path and imminent collisions are reported.

Hayward [Hayw86] describes a collision detection tool for an off-line robot programming system. Such a system 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. The robot workspace is initially represented by a region octree. The author offers two methods for determining collisions at a given sample of the trajectory. In the first method, the robot is approximated by bounding volumes made of cylinders ended by hemispheres and a different octree of the workspace is associated with each bounding volume in the robot arm. The freespace represented by the octree for a given volume is expanded with respect to the radius of the volume, and the volume is reduced to a line segment. Detecting collisions is then reduced to the simple task of determining the interference of segments with the volume described by the octree at each trajectory sample. The second method proposed by the author is to allocate control points on the surface of the robot and at each trajectory sample, determine if these control points lie outside of the freespace represented by the octree.

A collision detection system directly incorporated into the control system of a robot manipulator is introduced in [Koka86]. Kokaji presents a collision avoidance/control system which maintains an internal geometric model of the robot and its workspace. The system is given the desired position of the gripper as input, and generates the appropriate sequence of joint angles to move the gripper to that position. Two types of imaginary force vectors — an

attractive and a repulsive force vector — are generated. The attractive force vector is defined between the manipulator hand and the target position. This "pulls" the hand towards the target position. The repulsive force vector is defined between a part of the manipulator and an obstacle. This inhibits the manipulator from approaching the obstacle too closely. The joint angles at the next sampling time are determined using these vectors and the world model, and are sent to the manipulator driver to perform the actual movements.

Velocity and distance bounds are introduced in [Cull86] 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, a pending collision is reported.

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. The representation of the free space is carried out by the discretization of the 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.

Minimum distance algorithms are discussed in [Hurt88] as a means of detecting imminent collisions in an off-line robot programming system with graphical simulation facilities. It is desirable that such systems include automatic collision detection facilities, since graphical depiction of a manipulator task can be confusing when scenes of even mild complexity are depicted, whether wireframe or hidden surface displays are used. A simulation system is described in which robots and objects are constructed of finite unions of convex polyhedra and finite (ellipsoidal) cylinders. To provide collision detection support for the system, an algorithm for determining the minimum distance between two objects is presented. As objects move, this 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 hyper-rectangle with sides parallel to the coordinate axis (i.e. a bounding box). The minimum distance algorithm (which is computationally expensive) was applied only to pairs of components whose bounding boxes were "close".

As we have seen, 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. None of these solutions are applicable to our specific problem. 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 due to the high cost of testing and accepting a

computer for space flight.

2.2. Octrees and Fast Ray Tracing

Ray tracing is a popular algorithm for computer rendering of synthetic images. Primary reasons that the use of ray tracing is so widespread are the simplicity of coding and the comparative ease with which ray tracing renders many realistic effects, including shadows, specular and diffuse lighting, and motion blur. The principle drawback of ray tracing is its comparatively high computation cost, which is due primarily to the high occurrence of one basic operation, the ray-scene intersection test. The simplest, brute force method of determining the ray-scene intersection is to test the ray against each object, remembering which object (if any) has the nearest point of intersection. An efficient method of reducing the number of ray-object tests is to implement a spatial index on the objects from which we can derive the relative positions of the objects in the scene. Using such an index, we only test objects which lie close to the path of the ray for possible intersection with the ray. Octrees have been introduced as a method of improving performance of ray tracing algorithms by providing such a spatial index on a group of objects in a three dimensional model [Glas84, MacD89]. Each node in the octree is used to store a list. This list describes all the objects in the scene that have a piece of their surface in that node. Our N-Objects octree is closely related in structure and behavior to the octrees proposed for fast ray tracing.

3. GENERAL DESCRIPTION OF THE ALGORITHM

This chapter provides a general description of our collision detection algorithm. We discuss how the world is modeled using primitive objects, how the system determines when two objects are about to collide, and how the octree representation is used to determine when objects are close to one another. The algorithms presented in this chapter are independent of the particular primitives supported, and of the specific implementation for the N-Objects octree. Chapter 4 presents detailed algorithms for calculating primitive intersections and for octree manipulations.

3.1. World Modeling

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.

The foundation of our representation is a three dimensional coordinate system. The position of each object in the environment is described relative to this coordinate system. Lying in the coordinate system is a bounding cube which contains the entire working environment. It has one of its vertices at the origin and each of its edges is parallel to an axis (see Figure 3). This cube encloses the space decomposed by the octree.

Each object in the working environment that can collide with another object is included in the world model. A constructive geometry approach to representing each object has been adopted. That is, complex objects are described as the union of simpler primitive

objects. As previously described, each of the robot arms has seven degrees of freedom with each section of the arm being nearly cylindrical in shape. Thus, the arms can be modeled as a series of seven adjoining cylinders. For our representation, we have chosen *cylspheres*, cylinders with spheres on each end, to represent each link in the arm. The cylsphere both provides an acceptable approximation for the links of the arms and allows for efficient intersection tests. The operating environment of the arms 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 through which it is referenced. 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.

When using primitive objects to construct more complex objects, there may be cases where multiple primitives describing the same object actually overlap. Even though these primitives do intersect, this should not be considered a collision. To handle this problem, the notion of *compatible* primitives is introduce. Two primitives that intersect in making 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 stored using 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.

3.2. Determining Imminent Collisions Between Objects

We now examine the problem of collision avoidance. Rather than detecting collisions as they occur, we would like to detect imminent collisions, with the intention of avoiding them. Thus, the collision avoidance system shall issue a warning when two objects come

"too close" to one another. If a certain distance between two objects is to be maintained, then the standard technique for a static environment is to increase the dimension of each primitive representing the objects by 1/2 the standard tolerance distance in all directions. Then, whenever the expanded primitive objects overlap, a collision warning can be issued.

In a dynamic environment, though, we may also wish to make provision for the relative speed of moving objects. In other words, if an object is moving particularly fast, the minimum tolerance for that object is increased to account for the added distance the object may move during shutdown time. Furthermore, when the robot arm is shut down during high speed operation, the effector may develop a swinging motion before settling into its braked position. Whether one chooses to incorporate this capability into the representation is dependent upon the maximum speed of the arms, the time required to report a collision and stop the arms, and the accuracy of the model. For example, in our application the maximum speed at which any point on the robot arm is allowed to move is 24 inches per second. The amount of time required to detect a collision and stop the arms is on the order of about 50 milliseconds. The greatest distance that the two arms could move toward each other during a 50 millisecond span is 2.4 inches. Consequently, the tolerances placed around the moving arms would range from zero (the link is still) to 1.2 inches (the link is moving at maximum speed). However, the cylsphere provides only an approximate model for the links of the arms, and in some cases is in error by more than one inch (i.e. roughly equivalent to the tolerance required at maximum speed). Thus, the overhead incurred by changing the model for the arms to account for varying speeds is unjustified, given the coarseness of the primitives chosen to represent the arms. For our application, a fixed tolerance for each arm was chosen such that at least 1.2 inches of buffer area (beyond the approximated boundary

of the robot arm) is provided by the cylsphere for each point on the surface of the link it represents. This is an acceptable approximation for our target environment where it is not expected 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, which is a problem we have chosen not to address at this time).

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.

3.3. Spatial Indexing Using an Octree

When an object moves, we wish to determine if it is about to collide with other objects in the environment. The naive approach would be to determine if the primitive objects that represent it collide with any other primitive objects in our world model. This is an efficient approach given a sufficiently simple environment. However as the environment becomes more complex, the computation time to detect collisions involving a single moving object grows linearly, with respect to the number of primitive objects in the world model. If all objects are to be tested, the order of complexity for the naive algorithm is $O(n^2)$ when there are n primitives. The problem with this approach is that we are checking for collisions with objects in the environment which may be nowhere near the object that has just moved.

To reduce the number of unnecessary intersection tests between primitive objects, 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 us with such an index. Using neighbor finding techniques described below, it can

easily be determined which nodes in the tree that are close to one another and hence, which primitives are close to each other.

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.

```
OctNode = record
    vertices : array[0..7] of Point;
    isSplit : Boolean;
    children : array[0..7] of ↑OctNode;
    parent : ↑OctNode;
    childNo : 0..7;
    numObjects : integer;
    assocObjects : ObjectList
end;
```

Since our octree will be small and memory is not limited, a node representation that is space inefficient but minimizes computing time has been chosen. The array *vertices* contains the coordinates for the eight corners of the octant which defines the region of space represented by this node. If memory were limited, then only two vertices of this region would be stored, and the remaining vertices would be calculated as they are needed. However, the extra computation time needed to repeatedly derive these values justifies the added storage space used to store all vertices. The flag *isSplit* stores whether this is an internal node (*isSplit* is true) or a leaf node. The array *children* stores pointers to this node's eight children, provided that it has been split. Figure 2 illustrates the numbering

system that has been adopted. *parent* is a pointer to this node's parent, which is needed when performing neighbor finding. *childNo* is the number of this octant with respect to its parent. If the node is a leaf node, then *numObjects* is a count of the number of primitive objects which lie in this node and *assocObjects* is a list of the ID's of the primitive objects which lie in this node. For the remainder of this paper primitive objects will be referred to simply as objects.

3.3.1. Building the Octree

The first step in modeling the world is to build the octree. We begin with an empty cube enclosing the working environment. 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 in 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]).

Given an object and a node in which to insert it (initially the root node), 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 inserted into each of the children. This split-insert process is repeated until

all offspring nodes have no more than N objects in them. The following Pascal-like psuedo code provides a detailed illustration of how the insertion proceeds.

```
procedure InsertObject(objectId : integer; node : ↑OctNode);
  {Insert a primtive object into a node.}
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;

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

```
{The following primitive routines are needed to perform the update.}

function ObjectNodeIntersect(objectId : integer; node : ↑OctNode) : boolean;
  {Determine if a primitive object lies in a node (discussed in Chapter 4).}

{The following three primitive routines are implementation dependent.}

procedure SplitNode(node : ↑OctNode);
  {Allocate storage for the children, initialize them and connect them to the parent.}

procedure AddObjectToNode(objectId : integer; node : ↑OctNode);
  {Add a primitive object's ID to a node's associated object list.}

procedure RemoveObjectFromNode(objectId : integer; node : ↑OctNode);
  {Remove a primitive object's ID from a node's associated object list.}
```

3.3.2. Updating the Octree

After construction, the octree describes the starting state of the working environment. When objects in the environment move, the octree is updated to maintain an accurate model of the world. The update process is a two-step cycle, first receiving position information about objects in the environment, and then modifying our representation to reflect any changes, during which any imminent collisions are reported.

The position information received by the system is 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 of the robot arms where every cylsphere is parallel to a coordinate axis. Using these values, simple kinematic transformations are applied to our model of the arms in their home position. 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. The result of the kinematics is the current position of the cylspheres representing the individual links in the robot arms. As a by product, 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 first and most obvious approach is to completely rebuild the octree from scratch, checking for possible collisions as each object is inserted into the tree. This would be a desirable approach if a large portion of the environment changed during each cycle. However, due to the short cycle time (30 msec) combined with restrictions on robot arm speed, we expect only small changes in the environment during each cycle. In particular, the nature of the N-Objects octree is such that minor changes in the environment only rarely require modifications to the octree. A second approach is to delete and re-insert objects into the tree whenever they move while checking for collisions. This approach allows us to modify a relatively small portion of the tree at a time, but at the cost of causing 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 exact 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.

Our algorithm uses an approach that minimizes the number of modifications to the structure of the tree. Because only small changes take place in the environment between cycles, an update process is desired that changes the structure of the tree only when changes

in the environment dictate it. We observe that changes in the tree structure only occur 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, cause nodes in the tree to merge or split. We further observe that 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. The process of updating the tree in light of a movement by an object can now be explained 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, performing any required splitting. 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 the node and try to merge that node with its 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 still be incorporated into the update. During the update process, the resident nodes of each object that has moved are located. It is within these nodes that we check for possible collisions by performing intersection tests with all objects sharing a node with the moving object. In addition, it must be considered that the moving object can enter a new node and collide with another object there. To handle this case, intersection tests are performed with any incompatible objects already in the node upon insertion. 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 (relatively expensive) 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).

During the update process we find all of the neighbors to a node in which the moving object resides and then check if the object has moved into any of them. Checking all of these neighbors is not always necessary. The direction(s) in which the object is moving can be determined using the object's current and previous position. Using these directions, the number of neighbor nodes processed can be reduce by only checking those nodes which lie in a direction in which the object is moving.

In some cases 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 (see section 3.3.3), 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 siblings 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 which is a neighbor to three of the nodes in which it currently resides, then our algorithm will process this node three times. In fact, it will insert the object's ID into the node'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 include a mechanism to mark all nodes which have been visited while moving an object. For each node in the tree, the field *lastCheckNo*, which stores an integer denoting the last update during which this node was visited, is included. 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 previously visited. If *ThisCheckNo* is equal to *lastCheckNo*, then the node is ignored. After the update of the octree has been completed for a moved 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 millisecond cycle time with 14 updates of the octree required during each cycle (all seven links of both arms moved), this only occurs approximately every 6 months during continuous operation. The following Pascal-like psuedo code provides a more formal description of how the updating proceeds.

```
procedure UpdateObject(objectId : integer);
  {Update the octree when an object has moved.}
var
    location : ↑OctNode; {A node in which the object resides (see Chapter 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 BoundingIBoxInsideNode(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)
            then DeleteObjectFromNode(objectId, location)
    end;

procedure UpdateNeighbor(objectId : integer; neighbor : ↑OctNode);
  {Update a node that an object has just moved into.}
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;

procedure UpdateSplitInsert(objectId : integer; node : ↑OctNode);
  {Recursively split and insert an object into a node during update.}
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;

{The following primitive routines are needed to perform the update.}

function ObjectNodeIntersect(objectId : integer; node : ↑OctNode) : boolean;
  {Determine if a primitive object lies in a node (discussed in Chapter 4).}

function CollisionInNode(objectId : integer; node : ↑OctNode) : boolean;
  {Determine if an object collides (intersects) with any of the objects in a node (discussed
in Chapter 4).}

procedure GetUnCheckedNeighbors(node : ↑OctNode; var neighbors : NeighborList);
  {Find and return all neighbors of a node which haven't been visited during the current
update (discussed in Chapter 4).}
```

```
{The following three primitive routines are implementation dependent.}

function BoundingBoxInsideNode(objectId : integer; node : ↑OctNode) : boolean;
  {Determine if an object's bounding box lies completely inside a node.}

function NeighborInDirection(neighbor : ↑OctNode) : boolean;
  {Determine if a neighbor lies in the direction of the moving object.}

procedure DeleteObjectFromNode(objectId : integer; node :↑OctNode);
  {Remove an object's association with a node and perform any possible merging.}
```

3.3.3. Minimum Octree Resolution

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.

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 dimension that a cube in the tree can have. 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.

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.

4. INTERSECTION AND NEIGHBOR-FINDING ALGORITHMS

This chapter provides descriptions for intersection and neighbor-finding routines used in the collision detection and update algorithms presented in the previous chapter. Specifically, we describe how intersection tests are performed, how objects are located in the tree during the update process, and how neighboring nodes are found.

4.1. Intersection Tests

Intersection tests are an essential part of the collision avoidance system. Intersection tests between primitive objects (rectangular solids and cylspheres) are performed to determine possible collisions. Intersections between primitive objects and nodes are performed to build and update the octree. Below, we provide a detailed description of how these intersection tests were implemented.

4.1.1. Intersections Between Rectangular Solids

Given two rectangular solids (or boxes) in arbitrary positions and orientations, we wish to determine if they intersect. To reduce the amount of computation required to detect intersections, *bounding boxes* are employed. For each rectangular solid, its bounding box is calculated and stored. The bounding box is the smallest box whose edges are parallel to the axis and which completely contains the rectangular solid. If two bounding boxes do not intersect, then it can be concluded that the rectangular solids enclosed by them also do not intersect. Otherwise, an intersection detection algorithm is applied to the two rectangular solids. Whenever a rectangular solid's position changes, its bounding box is recalculated.

The current position of a rectangular solid in the world model is represented by the eight points corresponding to its eight vertices. Figure 4a illustrates how these vertices

are numbered. Figure 4b indicates the numbering system adopted for the six faces of the rectangular solid. To perform the intersection test between two rectangular solids, a technique related to three dimensional clipping in computer graphics [Fole82] is used. With this technique, a line segment is clipped against a box which is *normalized*. By normalized, we mean that the box is positioned so that each of its edges is parallel to an axis and one of its vertices lies at the origin. A rectangular solid is defined to be normalized when *edge30*, the edge connecting vertices 3 and 0, lies on the positive y-axis, *edge23* lies on the positive x-axis, *edge37* lies on the negative z-axis, and vertex 3 is positioned at the origin (see Figure 6).

Given a normalized box, its eight vertices are used to segregate the world into 27 different regions called *quadrants*. Figure 8 illustrates how the quadrants are labeled. Similar to the Cohen-Sutherland line clipping algorithm [Fole82], these quadrants reduce the number of line-plane intersection tests needed to determine if an edge of one rectangular solid intersects a face of another. For example, to determine if an edge intersects a box, the naive approach would be to perform line-plane intersections using the edge and the six sides of the box. However, if it is known that one end point of the edge lies in quadrant 1 and the other lies in quadrant 7, then no intersection tests are required since the entire edge must lie outside the box. If one end point lies in quadrant 11 and the other end point lies in quadrant 15, then only two faces need to be checked for a possible intersection (faces 2 and 3). Finally, if either end point lies in quadrant 14 (inside the box), then an intersection has occurred and no tests are necessary.

As previously stated, either rectangular solid can have an arbitrary position or orientation in the coordinate system. However, the clipping algorithm requires that the box

being clipped against be normalized. To facilitate this, some transformations are performed on the two boxes. For each rectangular solid, a set of *rotation angles* that will normalize it is calculated and stored.

```
RotationAngles = record
    yRot : real;
    zRot : real;
    alignRot : real
end;
```

To normalize a box, a simple translation that will position vertex 3 at the origin is performed. Next, *edge03* is projected onto the x-z plane. The value *yRot* represents the angle between the projected edge and the positive x-axis. The box is rotated around the y-axis using *yRot* so that *edge03* now lies on the x-y plane. The value *zRot* represents the angle between *edge03* and the positive y-axis. Using zRot the box is rotated around the z-axis so that *edge03* now lies on the positive y-axis. The value *alignRot* represents the angle between *edge37* and the negative z-axis. Using *alignRot*, the box is rotated around the y-axis so that *edge37* lies on the negative axis. The box is now normalized. Whenever the position of the box in the world model changes, these rotation angles are recalculated.

In addition to the rotation angles, the dimensions for each box is also stored. These dimensions represent the height, width, and depth of the box when it has been normalized.

```
Dimensions = record
    height03 : real;
    width23 : real;
    depth73 : real
end;
```

The intersection test for two boxes, *box1* and *box2*, now proceeds as follows. First, all of the edges of *box1* are clipped against the faces of *box2*. Before doing so, *box2* must be normalized. The transformations that would normalize *box2* are applied to the vertices of *box1*. Note that it is not necessary to apply these transformations to *box2* itself because the position of its vertices after it has been normalized are already known (from its dimensions). For each edge of *box1*, the quadrants of its endpoints with respect to the normalized *box2* are determined. Using a table lookup, the faces (if any) of *box2* that need to be checked for a possible intersection are found. If an intersection is detected, then a collision is declared. Otherwise the process continues with the next edge of *box1*. If no intersections are found, then we repeat the same procedure, clipping *box2* against *box1* this time, and report any collisions. The following Pascal-like psuedo code provides a more precise description of the intersection algorithm.

```
function RectSolidIntersect(r1, r2 : RectSolid) : boolean;
  {Determine if two rectangular solids intersect.}
var
    rot1, rot2 : RectSolid;
begin
    NormalizeRectSolid(r1, r2, rot2);
    if ClipRect(rot2, r1.dimensions) then
        RectSolidIntersect := true
    else begin
        NormalizeRectSolid(r2, r1, rot1);
        if ClipRect(rot1, r2.dimensions)
            then RectSolidIntersect := true
            else RectSolidIntersect := false
    end
end;

function ClipRect(rotR : RectSolid; normDim : Dimensions) : boolean;
  {Clip the edges of rotR against the normalized rectangular solid specified by normDim.}
var
    edge : EdgeDefinition;
    q1, q2 : 1..27;
    chkFaces : string;
    i : integer;
begin
    for each edge of rotR do begin
        GetEdgeQuadrants(edge, normDim, q1, q2);
        chkFaces := FacesToCheck(q1, q2);
        if chkFaces = "Intersection" then begin
            ClipRect := true;
            exit
        end
        else if chkFaces = "NoIntersection" then begin
            ClipRect := false;
            exit
        end
        else
            for i := 1 to length(chkFaces) do
                if EdgeFaceIntersection(edge, checkFaces[i], normDim) then begin
                    ClipRect := true;
                    exit
                end
end;
```

{The following primitive routines are needed to perform the rectangular solid intersection test.}

procedure NormalizeRectSolid(*r1*, *r2* : RectSolid; **var** *rot2* : RectSolid);
{Apply to *r2* the transformations which would normalize *r1* and return result in *rot2*.}

procedure GetEdgeQuadrants(*edge* : EdgeDefinition; *normDim* : Dimensions;
var *q1*, *q2* : 1..27);
{Find the quadrants that the end points of an edge lie in with respect to the normalized rectangular solid defined by *normDim*.}

function FacesToCheck(*q1*, *q2* : 1..27) : string;
{Determine which faces an edge can intersect, using the quadrants of its end points.}

function EdgeFaceIntersection(*edge* : EdgeDefinition; *face* : char;
normDim : Dimensions) : boolean;
{Determine if an edge intersects a face of the normalized rectangular solid defined by *normDim*.}

4.1.2. Intersections Between Cylspheres

Determining if two cylspheres intersect is a straightforward process. As previously stated, a cylsphere is a cylinder with spheres on each end whose centers lie on the cylinder's axis. The cylinder and the spheres all have the same radius. Cylspheres in our representation are described using two points and a radius. In addition, the length of the axis between the two end points is also stored since it is needed throughout the intersection calculations.

```
CylSphere = record
    endP1 : Point;
    endP2 : Point
    radius : real
    length : real
end;
```

The points, *endP1* and *endP2*, lie on the axis of the cylinder, one at each end, and denote the centers of the two ending spheres. The field *radius* denotes the radius for both the cylinder and the ending spheres. To determine if a point P lies within a cylsphere is simple. 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 greater than the sum of the radii of the two cylspheres, then the cylspheres do not overlap. Otherwise, an intersection is assumed.

As with rectangular solids, 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 similarly found. Every time a cylsphere's position changes, its bounding box is recalculated. The following Pascal-like psuedo code provides a more precise description of the cylsphere intersection algorithm.

```
function CylsphereIntersect(c1, c2 : Cylsphere) : boolean;
  {Determine if two cylspheres intersect.}
var
    p1, p2 : Point;
begin
    FindClosestPoints(c1, c2, p1, p2);
    if Distance(p1, p2) > (c1.radius + c2.radius)
        then CylsphereIntersect := false
        else CylsphereIntersect := true
end;

procedure FindClosestPoints(c1, c2 : Cylsphere; var p1, p2 : Point);
  {Find the unique point on each axis which is closest to the other axis.}
var
    v1, v2, vn, vn1, vn2 : Vector;
    pln : Plane;
begin
    {Find the vector which is perpindicular to both axis.}
    v1 := TwoPointsToVector(c1.endP1, c1.endP2);
    v2 := TwoPointsToVector(c2.endP1, c2.endP2);
    vn := CrossProduct(v1, v2);
    {Find the plane which is parallel to vn and contains axis of c2.}
    vn2 := CrossProduct(vn, v2);
    pln := PointAndVectorToPlane(c2.endP1, vn2);
    {Find closest point to that plane which is on the axis of c1.}
    p1 := SegmentPlaneIntersection(c1.endP1, c1.endP2, pln);
    {Find the plane which is parallel to vn and contains axis of c1.}
    vn1 := CrossProduct(vn, v1);
    pln := PointAndVectorToPlane(c1.endP1, vn1);
    {Find closest point to that plane which is on the axis of c2.}
    p2 := SegmentPlaneIntersection(c2.endP1, c2.endP2, pln);
end;

{The following primitive routines are needed to perform the cylsphere intersection test.}

function TwoPointsToVector(p1, p2 : Point) : Vector;
  {Calculate a vector using two points.}

function CrossProduct(v1, v2 : Vector) : Vector;
  {Calculate the cross product of two vectors.}
```

function PointAndVectorToPlane(p : Point; v : Vector) : Plane;
{Calculate the plane descibed by a point and a vector.}

function SegmentPlaneIntersection(*p1*, *p2* : Point; *pln* : Plane) : Point;
{Find closest point on a segment to a plane by intersecting the line containing the segment with the plane.}

function Distance(*p1*, *p2* : Point) : real;
{Calculate the distance between two points.}

4.1.3. Intersections Between Cylspheres and Rectangular Solids

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. We will now address these cases individually.

To test if an edge of a box passes through the surface of the cylsphere a clipping technique similar to the one used to detect rectangular solid intersections is employed. We re-introduce the notion of normalization and apply it to cylspheres. Given a cylsphere with a radius of r and a length of l between its two end points, it is normalized when its *endP1* is at $(0, 0, -r)$ and its *endP2* is at $(0, 0, -(l + r))$ (see Figure 7). As with rectangular solids, the rotation angles needed to normalize a cylsphere are calculated and stored ahead of time . To save storage space, these angles could be re-calculated every time an object is involved in an intersection test. This, however, becomes too expensive computationally. Therefore, these values are re-calculated only when an object has moved.

```
RotationAngles = record
    yRot : real;
    xRot : real
end;
```

To normalize a cylsphere, a simple translation is performed that will position *endP1* at the origin. Next, the segment defined by *endP1* and *endP2*, which we will call *seg12*, is projected onto the x-z plane. The value *yRot* represents the angle between the projected segment and the negative z-axis. The cylsphere is rotated around the y-axis using *yRot* so that *seg12* now lies on the y-z plane. The value *xRot* represents the angle formed by *seg12* and the negative z-axis. Using *xRot*, the cylsphere (i.e *seg12*) is rotated around the x-axis to place *endP2* on the negative z-axis. Finally, a translation is performed on the cylsphere in the negative z direction so that *endP1* is position at $(0, 0, -r)$. The cylsphere is now normalized.

Now, to clip the edges of an arbitrary rectangular solid, *box*, against a cylsphere, *cs*, we proceed as follows. First, the transformations which would normalize *cs* are applied to the vertices of *box*. Note that it is not necessary to apply these transformations to *cs* itself because the position of its end points after it has been normalized is already known (from its radius and length). Next, a bounding box is placed around the normalized *cs*. This bounding box is used to assign quadrants to the space, similar to the intersection algorithm used for detecting intersections between rectangular solids. For each edge of *box*, the quadrants of its end points with respect to this bounding box are found. Using a table lookup on the quadrants, we determine if it is necessary to check for a possible intersection with the cylsphere. For example, if the vertices of an edge lie in quadrants 1 and 7, then the edge does not intersect with the bounding box of the normalized cylsphere, so it cannot

possibly intersect the cylsphere itself. If an edge can intersect with the bounding box of the normalized *cs*, then some closer examination is necessary to determine if it does, in fact, intersect with *cs*. This requires checking for a possible intersection with either of the ending spheres or the cylinder which, together, comprise the cylsphere.

To determine if an edge intersects a sphere, we first find the point on the line containing the edge which is closest to the center of the sphere. If this point lies outside of the two end points of the edge, then the closest end point is taken. If the distance between this point and the center of the sphere is greater than the radius of the sphere, then the edge does not intersect the sphere. Otherwise, an intersection is declared.

To determine if an edge intersects the cylinder, the cylinder and the edge are projected onto the x-y plane. The problem now becomes a simple edge/circle intersection test. If the line containing the projected edge does not intersect the circle, then there is no edge cylinder intersection. Otherwise, the line intersects the circle at either one or two points. For each point of intersection on the line, if the point lies outside of the projected edge, then an intersection is not possible. Otherwise, we calculate its corresponding z value and check if it is in the range of the z coordinates for the two end points of *cs*. If it is, then the edge intersects the cylinder between the ending spheres and an intersection between *box* and *cs* is asserted.

If none of the box's edges intersect the cylsphere, then the second part of the intersection test is performed. That is, we check if part of the cylsphere, *cs*, passes through a face of the box, *box*. This procedure depends on *box* being normalized and begins by applying the transformations that would normalize *box* to the end points that define *cs*. Note, again, that it is not necessary to apply these transformations to *box* itself because

its position after it has been normalized is already known (based on its dimensions). We observe that for *cs* to penetrate a face of *box* and not intersect with any of the edges of *box*, either the segment defined by *endP1* and *endP2* of *cs*, *seg12*, must intersect with *box*, or one of the ending spheres of *cs* must intersect with *box* (or both). To check the first case, *seg12* is clipped against *box* using the techniques described above to clip an edge against a rectangular solid.

Determining an intersection between one of the ending spheres and *box* is more involved. The basic idea is to find the point on the surface of *box* which is closest to the center of the sphere. If its distance from the center is less than or equal to the radius of the sphere, then an intersection is assumed. Given an ending sphere *sp* and *box*, our first step is to determine which quadrant the center *c* of *sp* lies in with respect to *box*. Assuming *c* does not lie inside of *box* (in which case we have an intersection), this quadrant falls into one of three categories. Either it shares a face with *box* (quadrants 5, 11, 13, etc.), an edge with *box* (quadrants 2, 4, 6, etc.), or a corner with *box* (quadrants 1, 3, 7, etc). For a face quadrant, the distance from the center of the ending sphere, *c*, to the closest face is calculated. For an edge quadrant, the distance from *c* to the closest edge is calculated. From a corner quadrant, the distance from *c* to the closest corner is calculated. In any case, if the distance is greater than the radius of the ending sphere *sp*, then no overlap occurs. Otherwise, an intersection is reported. The following Pascal-like psuedo code provides a more formal description of the intersection algorithm.

```
function CylsphereRectSolidIntersect(c : Cylsphere; r : RectSolid) : boolean;
  {Determine if a cylsphere and a rectangular solid intersect.}
var
    rotC : Cylsphere;
    rotR : RectSolid;
begin
    NormalizeRectSolid(r, c, rotC);
    if CylspherePierceRectSolid(rotC, r.dimensions) then
        CylsphereRectSolidIntersect := true
    else begin
        NormalizeCylsphere(c, r, rotR);
        if ClipCylprism(rotR, c.radius, c.length)
            then CylsphereRectSolidIntersect := true
            else CylsphereRectSolidIntersect := false
    end
end;

function CylspherePierceRectSolid(rotC : Cylsphere; normDim : Dimensions) : boolean;
  {Check if an ending sphere or the axis of a cylsphere passes through a face of a normalized
rectangular solid defined by normDim.}
begin
    if SphereRectSolidIntersect(rotC.endSphere1, normDim) then
        CylspherePierceRectSolid := true
    else if SphereRectSolidIntersect(rotC.endSphere2, normDim) then
        CylspherePierceRectSolid := true
    else
        CylspherePierceRectSolid := ClipCylsphereAxis(rotC.endPoint1,
                                        rotC.endPoint2, normDim)
end;

function ClipCylsphere(rotR : RectSolid; normRadius, normLength : real) : boolean;
  {Clip edges of rectangular solid against normalized cylsphere defined by normRadius and
normLength.}
var
    edge : EdgeDefinition;
    chkFaces : string;
    q1, q2 : 1..27;
begin
    for each edge of rotR do begin
        GetEdgeQuadrants(edge, normRadius, normLength, q1, q2);
        chkFaces := ClipContainingBox(q1, q2, normRadius, normLength);
        if chkFaces = "Intersection" then begin
```

```
            ClipCylsphere := true;
            exit
        end
        else if EdgeCylsphereIntersect(edge, normRadius, normLength) then begin
            ClipCylsphere := true;
            exit
        end
end;
```

{The following primitive routines are needed to perform the cylsphere-rectangular solid intersection test.}

procedure NormalizeRectSolid(*r* : Cylsphere; *c* : RectSolid; **var** *rotC* : RectSolid);

{Apply to *c* the transformations which would normalize *r* and return result in *rotC*.}

procedure NormalizeCylsphere(*c* : Cylsphere; *r* : RectSolid; **var** *rotR* : RectSolid);

{Apply to *r* the transformations which would normalize *c* and return result in *rotR*.}

function ClipCylsphereAxis(*p1*, *p2* : Point; *normDim* : Dimensions) : boolean

{Determine if the axis of a cylsphere defined by two points intersects a face of the normalized rectangular solid defined by *normDim*.}

procedure GetEdgeQuadrants(*edge* : EdgeDefinition; *normRadius*, *normLength* : real;
var *q1*, *q2* : 1..27);

{Find the quadrants that the end points of an edge lie in with respect to the containing box of the normalized cylsphere defined by *normRadius* and *normLength*.}

function ClipContainingBox(*q1*, *q2* : 1..27; *normRadius*, *normLength* : real) : boolean;

{Using the quadrants of an edge's end points, determine if the edge can intersect the containing box of a normalized cylsphere defined by *normRadius* and *normLength*.}

function EdgeCylsphereIntersect(*edge* : EdgeDefinition; *normRadius*,
normLength : real) : boolean;

{Determine if an edge intersects a normalized cylsphere defined by *normRadius* and *normLength*.}

4.1.4. Intersection Tests Involving Nodes

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. Algorithms used to detect intersections with a node treat the node as if it were a rectangular solid, with a single exception. Normalizing a node requires only a single translation and no rotations. Otherwise, the algorithms proceeds the same. Bounding boxes are used to speed up intersection tests involving nodes (the bounding box for a node is the node itself).

4.2. Locating Objects in the Octree

When an object moves, the initial step in updating the octree is to locate where in the tree the object is. As described above, an object which lies in a node has its corresponding ID stored in the node's associated object list. The problem is to locate these nodes quickly. The brute force approach would be to perform a depth-first search on the tree and process the nodes as they are encountered. This may be acceptable for a very small tree but in most cases is too slow. To speed up the location process we introduce the notion of *location links*. A set of location links for an object can be thought of as a sequence of links that connect different leaf nodes in the tree. A pointer pointing to a leaf node in the tree in which the object resides is added to an object's description. The object's corresponding associated object list entry in that leaf node includes a pointer to another leaf node in the tree in which the object lies (see Figure 8). This sequence of links is maintained to include every leaf node that an object lies in. 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 the update process, this list is traversed, processing each node one at a time.

4.3. Neighbor Finding Algorithms

When updating the octree in light of a movement, it must be determined if the moving object has entered any new nodes. Given a node that the object lies in, we proceed by checking if the object has moved into any of its neighbors. But first, these neighboring nodes must be located. Samet [Same89] provides an in-depth discussion of algorithms which can be used to locate neighbors in an octree. We have implemented one of these algorithms, *GetFaceNeighbor*, which, given node *n* and face *f*, locates the smallest node in the tree which has a face which lies on the same plane as face *f* of *n* and completely contains it. This node is termed a *face neighbor*. Below, we describe how *GetFaceNeighbor* is implemented and how it is used to locate all neighbors of a node.

4.3.1. Finding a Face Neighbor

To find a neighboring node of node *n* at a given face *f*, the first step is to locate a *common ancestor* of the two nodes. Starting at *n*, the tree is traversed upward towards the root until a node which is not adjacent to face *f* of its parent is encountered. This node's parent is the common ancestor. For example, to find a neighbor over face 3, the common ancestor is the first node reached through its 0, 3, 4 or 7th child. Once the common ancestor has been located, the traversal is retraced downward while making mirror image moves about a plane containing the common face of the two nodes. If locating a neighbor at face 3, for example, then the mirror image of an upward move through child 0 would be a downward move through child 1.

4.3.2. Locating All Neighbors of a Node

When searching for the neighbors of a node, we have two goals in mind. First, we wish to locate the neighboring nodes which completely surround the node in question. This includes any node which shares a face, edge or corner with the node. Our second objective is to locate all neighboring nodes as quickly as possible. In the worst case, if a node n is nested in a grid of equal-sized nodes, then n has 26 neighboring nodes (see Figure 9). To locate all of these nodes can be time consuming, so we would like to optimize this process.

We observe that for a given node n, its siblings are neighbors which completely cover three faces of the node. In addition, they cover some edges and corners as well (see Figure 10). Hence, by maintaining a pointer to a node's parent, seven of a node's neighbors can be located without any tree traversal at all. There are, however, more neighbor nodes to be found. These remaining neighbors of n can be located using *GetFaceNeighbor*.

Assuming that n is nested in a grid of equal sized nodes, there are 19 non-sibling nodes that are neighbors of n. Now we examine the case where the parent node p of n is nested in a grid of equal size nodes. Only 7 neighbor nodes of p along with the siblings of n are needed to completely surround n (see Figure 11). Three of the neighbors of p are face neighbors, three are edge neighbors and one is a corner neighbor. So the method adopted to retrieve all neighbors of a node is to first get the siblings of the node. Then, the next step taken is to get the three face neighbors of the node's parent using *GetFaceNeighbor*. Using these three face neighbors and *GetFaceNeighbor*, the remaining three edge neighbors of p are found. Finally, using an edge neighbor and *GetFaceNeighbor*, the lone corner neighbor of p is retrieved.

The algorithm described above works for the case when p is surrounded by equal

sized nodes. This however is not always the case. If a face neighbor of p is larger than p, then it could possibly cover an open edge (or corner) of p (see Figure 12). When such is the case, we need not find the corresponding edge neighbor. Our algorithm is easily modified to detect when these cases arise by simply checking the size and position of the neighbor nodes as they are found. For example, suppose the neighbor $f4$ of face 4 of p has been retrieved. Using its vertices, we would check if $f4$ "covers" an edge or corner of p. If vertex 1 of $f4$ is "higher" then vertex 0 of p, then it is not necessary to locate the edge neighbor which is to the left and above p (see Figure 12).

Presented below is a psuedo-code description of the algorithm to find all neighbors of a node. A numbering system similar to the one used to label quadrants has been adopted for the neighbor nodes of p. Assuming p is nested in the center of a grid of 27 equal-sized nodes, then the nodes are numbered 1 thru 27 with p being node 14.

```
procedure GetChild0Neighbors(parent : ↑OctNode; var neighbors : NeighborList);

  {Get all neighbors of a node which is child 0 of its parent. Similar procedures are written
for children 1 thru 7.}
var
    child : Octant
    n2, n4, n5, n10, n11, n13 : ↑OctNode;    {ni is ith neighbor of parent (i=1..27).}
    ful : Point;    {Front upper left vertex of parent.}
    cornerCovered : boolean;    {True if a neighbor encloses corner of p.}
begin
    {Add siblings to list of neighbors.}
    for child := oct1 to oct7 do
        AddNeighborToList(parent↑children[child], neighbors);

    ful := parent↑vertices[0];
    cornerCovered := false;

    {Get face neighbor in front of parent.}
    n5 := GetFaceNeighbor(parent, 1);
    if n5 <> nil then begin
        AddNeighborToList(n5, neighbors);
        if (n5↑vertices[4].x < ful.x) and (n5↑vertices[4].y > ful.y)
            then cornerCovered := true
    end
    else {No neighbor in front, parent adjacent to border face of world cube.}
        cornerCovered := true;

    {Get face neighbor on top of parent.}
    n11 := GetFaceNeighbor(parent, 2);
    if n11 <> nil then begin
        AddNeighborToList(n11, neighbors);
        if (n11↑vertices[3].x < ful.x) and (n11↑vertices[3].z > ful.z)
            then cornerCovered := true
    end
    else
        cornerCovered := true;

    {Get face neighbor to left of parent.}
    n13 := GetFaceNeighbor(parent, 4);
    if n13 <> nil then begin
        AddNeighborToList(n13, neighbors);
        if (n13↑vertices[1].z > ful.z) and (n13↑vertices[1].y > ful.y)
            then cornerCovered := true
```

```
end
else
    cornerCovered := true;

{Get edge neighbor in front and above parent.}
if n5 = nil and n11 = nil then    {No neighbor there.}
    n2 := nil
else if (n5↑vertices[4].y = ful.y) and (n11↑vertices[3].z = ful.z) then begin
    n2 := GetFaceNeighbor(n11, 1);
    AddNeighborToList(n2, neighbors);
    if (n2↑vertices[7].x < ful.x)
        then cornerCovered := true
end
else {Front or above neighbor covers the edge.}
    n2 := nil;

{Get edge neighbor in front and to the left of parent.}
if n5 = nil and n13 = nil then
    n4 := nil
else if (n5↑vertices[4].x = ful.x) and (n13↑vertices[1].z = ful.z) then begin
    n4 := GetFaceNeighbor(n13, 1);
    AddNeighborToList(n4, neighbors);
    if (n4↑vertices[5].y > ful.y)
        then cornerCovered := true
end
else {Front or left neighbor covers the edge.}
    n4 := nil;

{Get edge neighbor to left and above parent.}
if n11 = nil and n13 = nil then
    n10 := nil
else if (n11↑vertices[3].x = ful.x) and (n13↑vertices[1].y = ful.y) then begin
    n10 := GetFaceNeighbor(n13, 2);
    AddNeighborToList(n10, neighbors);
    if (n10↑vertices[2].z > ful.z)
        then cornerCovered := true
end
else {Front or above neighbor covers the edge.}
    n10 := nil;

{Get corner neighbor if not covered by another neighbor.}
if not cornerCovered then begin
    {Use an edge neighbor to get the corner neighbor.}
    if n2 <> nil then
        n1 := GetFaceNeighbor(n2, 4)
    else if n4 <> nil then
```

```
            n1 := GetFaceNeighbor(n4, 2)
        else if n10 <> nil then
            n1 := GetFaceNeighbor(n10, 1);
        AddNeighborToList(n1, neighbors)
    end
end
```

{The following primitive routine is needed to locate all neighbors.}

```
procedure AddNeighborToList(neighbor : ↑OctNode; var neighbors : NeighborList);
  {Add a node to a list of neighbors, checking if it has already been visited.}
```

---

5. EXPERIMENTAL RESULTS AND ANALYSIS

Our collision detection algorithm was implemented using the C language, running under UNIX. The host hardware included an 80386 CPU running at 20 MHz, complete with math co-processor. 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 13). Our environment was modeled after the test bed constructed at NASA's robotics laboratory at the Goddard Space Flight Center. The tasks were each on the order of one minute in duration and 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 average time per cycle (processing

one complete set of joint angles and checking for collisions) over the three tasks was about 30 milliseconds. However, the actual time for each cycle varied depending on how much of the environment moved during that interval. 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 milliseconds. The observed upper bound for the range of update cycle times was 60 milliseconds. So, even in a worst case scenario, the algorithm performed within acceptable limits.

5.1. A Closer Look at Algorithm Performance

Further analysis was needed to understand how much computation was being 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 was implemented for this research was selected for its simplicity, rather than it efficiency. A more efficient algorithm would offer significant improvement in this area of performance. The remainder of the computation time was used to update the octree and check for collisions. About 37% of this time was spent performing object-object intersection tests. Another 10% was needed to calculate rotation angles and bounding boxes for moving objects. Object-node intersection tests required 18% of this time while retrieving neighbors took about 10%. 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.

5.2. Structure of the Octree

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 are rarely needed. 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. As stated, the structure of the N-Objects octree rarely changes. 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 millisecond cycle time, this translates into once every 18 seconds.

5.3. Tuning the Algorithm

In Chapters 3 and 4, some techniques for increasing 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 significant effect by reducing computation time by 80%.

5.4. System Performance and the Decomposition Rule

The decomposition rule for the *N-Objects* octree is simply stated as "split a node if more than N objects lie within it." The value chosen for N, though, has a significant effect on system performance. A large portion of the computation required by an octree update consists of performing 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 the choice of 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-object intersection tests (70 microseconds) and

object-node intersection tests (150 microseconds). Figure 14 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.

5.5. A Naive Approach Versus the Octree

The obvious question one might ask is how does the octree compare in performance to the brute force approach to collision detection. A brute force 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 would grow 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 of the intersection tests is very low. The naive algorithm also does not require nearly as much overhead as the octree. It is clear that 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 15 illustrates the behavior of the two algorithms over the different test data. 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 is taking place, and 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.

5.6. A Grid Representation versus the Octree

A grid representation is similar to the octree approach 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]. For example, given a large set of line segments in two-dimensional space, a grid representation can be used to efficiently determine which pairs of line segments intersect. A grid is overlayed onto the data and for each grid cell, a set containing each segment which lies in that cell is formed. Only pairs of segments that lie in the same grid cell are checked for possible intersection. The grid can be extended to provide a spatial index in three dimensions. 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. Insertion required visiting each node in the grid. 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 representation were also included in the grid representation.

To test the grid representation, we used our standard three tasks and varied the value of G. Figure 16 provides an illustration of how the algorithm performed for different values of G. Even at the optimum G, the grid representation fell short in performance when compared to the octree representation. Although the grid representation is a simple one and lends itself to parallel execution [Kank88], 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.

5.7. The Region Octree versus the N-Objects Octree

An alternative method of modeling three dimensional objects is the region octree representation. 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 avoidance.

A region octree representation was constructed 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 our 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 all 12 links moved during a typical cycle, the amount of computation required to transform the corresponding leaf nodes would be directly proportional to the number of points used to represent each node. Assuming that only the center of each node was transformed, it is still inconceivable that applying the forward kinematics of a seven degree of freedom arm to 3000 points on a microprocessor could be done in real time (i.e within a 30-60 millisecond time span).

5.8. Expected Performance of the Algorithm

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 different 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 affect on cycle time will be observed. In general, stationary objects only affect 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 affect 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.

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.

6. CONCLUSIONS AND FUTURE WORK

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 accidently 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 in this thesis 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, a 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 percentage of the objects in the environment are moving, cycle times will likely increase beyond acceptable limits.

The octree provides a flexible means of 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 we have presented provides a solid 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 discern 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 as 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 primitives chosen to model the target working environment are simple and provide for efficient interference detection. However, the N-Objects octree can easily be modified to support *any* primitive as long as an efficient method for determining primitive-primitive and primitive-node intersections is available. Spheres, planes, and cylinders with different radii at each end are examples of primitives which could be incorporated into our representation.

Much research has been done on determining intersections between two static convex polyhedra [Dobk83, Maru72, Mull78]. Researchers have also described algorithms for detecting imminent collisions between two convex polyhedra [Cull86, Gilb89, Hurt88]. These methods, along with the octree, can be used to provide efficient collision avoidance in a world of complex objects. We can use the octree to index the bounding boxes of the polyhedra, and only when the bounding boxes come close are the expensive polyhedra intersection/imminent collision tests applied. In this scenario, the octree provides a spatial index with very low computation time and storage space overhead. We experienced an

eight percent increase in system performance when our representation was modified to index objects using their bounding boxes. The number of object-object intersection tests increased because the bounding box for an object sometimes resided in more nodes then the object itself. However, this extra computation was offset by a much cheaper object-node intersection test (six comparisons).

The N-Objects octree could 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 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, V.P.I. and S.U., 1984.

2. [Bowy83] A. Bowyer and J. Woodwark, *A Programmer's Geometry*, Butterworths Publishing Company, Inc., London, England, 1983.

3. [Came85] S. Cameron, A Study of the Clash Detection Problem in Robotics, *Proceedings of the 1985 IEEE International Conference on Robotics and Automation*, St. Louis, MO, USA, 25-28 March 1985, 488-493.

4. [Chen85] H.H. Chen and T.S. Huang, Octrees: Construction, Representation, and Manipulation, *SPIE Vol. 579 Intelligent Robots and Computer Vision*, September1985, 448-458.

5. [Cull86] R.K. Culley and K.G. Kempf, A Collision Detection Algorithm Based on Velocity and Distance Bounds, *Proceedings 1986 IEEE International Conference on Robotics and Automation*, 7-10 April 1986, vol 2, 1064-1069.

6. [Dobk83] D. Dobkin, Fast Detection of Polyhedral Intersections, *Theoretical Computer Science 27*, 3(December 1983), 241-253.

7. [Fran83] W.R. Franklin, Adaptive Grids for Geometric Operations, *Proceedings of the Sixth International Symposium on Automated Cartography*, 16-21 October 1983, vol 2, 230-239.

8. [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.

9. [Fole82] J.D. Foley and A. Van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1982.

10. [Garg82] I. Gargantini, Linear Octtrees for Fast Processing of Three-Dimensional Objects, *Computer Graphics and Image Processing 20*, 4(December 1982), 365-374.

11. [Gilb89] E.G. Gilbert and S.M. Hong, A New Algorithm for Detecting the Collision of Moving Objects, *Proceedings 1989 International Conference on Robotics and Automation*, 1989, vol 1, 8-14.

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

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

14. [Gouz84] L. Gouzenes, Collision Avoidance for Robots in an Experimental Flexible Assembly Cell, *Proceedings International Conference on Robotics*, Atlanta, GA, USA, 13-15 March 1984, 474-476.

15. [Hayw86] V. Hayward, Fast Collision Detection Scheme by Recursive Decomposition of a Manipulator Workspace, *Proceedings 1986 IEEE International Conference on Robotics and Automation*, San Francisco, CA, USA, 7-10 April 1986, vol 2, 1056-1063.

16. [Hurt88] G. Hurteau and N.F. Stewart, Distance Calculation for Imminent Collision Indication in a Robot System Simulation, *Robotica*, January-March 1988, vol 6, pt 1, 47-51.

17. [Kank88] M. Kankanhalli, *The Uniform Grid Technique for Fast Line Intersection on Parallel Machines*, M.S. Thesis, Electrical, Computer, and Systems Engineering Dept., Rensselaer Polytechnic Institute, Troy, NY, April 1988.

18. [Koka] S. Kokaji, Collision-Free Control of a Manipulator with a Controller Composed of Sixty-Four Microprocessors, *IEEE Control Systems Magazine*, vol 6, no 5, (October1986), 9-14.

19. [Maru72] K. Maruyama, A Procedure to Determine Intersections between Polyhedral Objects, *International Journal of Computer and Information Sciences 1*, 1972, 255-261.

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

21. [Mull78] D.E. Muller and F.P. Preparata, Finding the Intersection of Two Convex Polyhedra, *Theoretical Computer Science 7*, 1978, 217-224.

22. [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.

23. [Roac87] J.W. Roach and M.N. Boaz, Coordinating the Motions of Robot Arms in a Common Workspace, *IEEE Journal of Robotics and Automation*, vol 3, no 5, (October1987), 437-444.

24. [Same89] H. Samet, *Applications of Spatial Data Structures; Computer Graphics, Image Processing, and GIS*, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1989.

25. [Same90] H. Samet, *The Design and Analysis of Spatial Data Structures*, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1990.

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

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

28. [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.

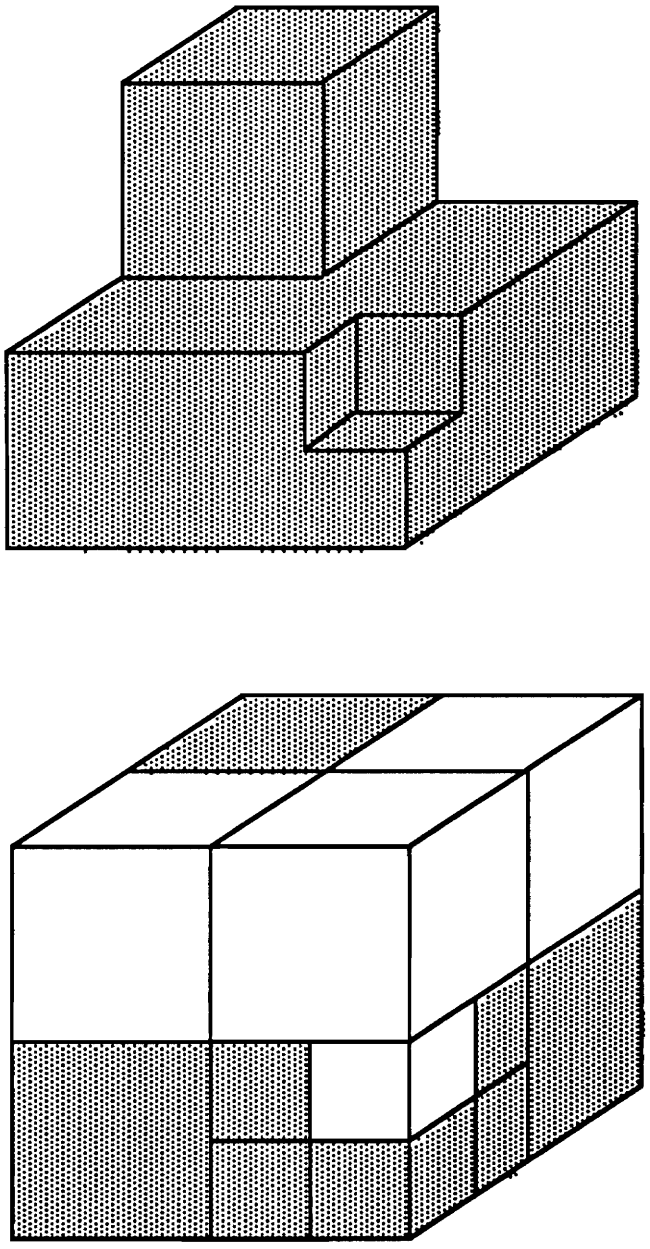


Figure 1. A three dimensional region and its corresponding region octree block decomposition.

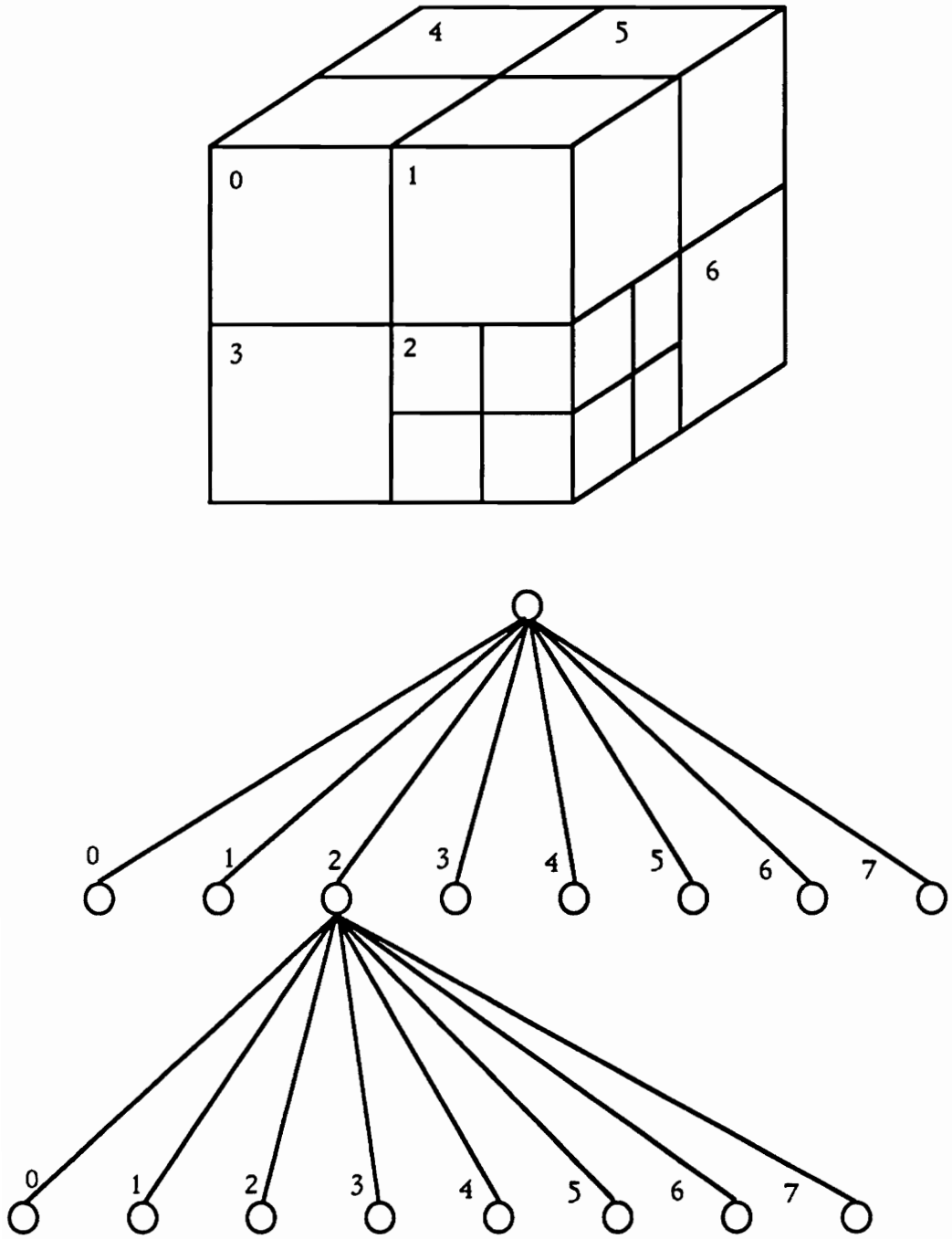


Figure 2. The tree representation of a block decomposition.

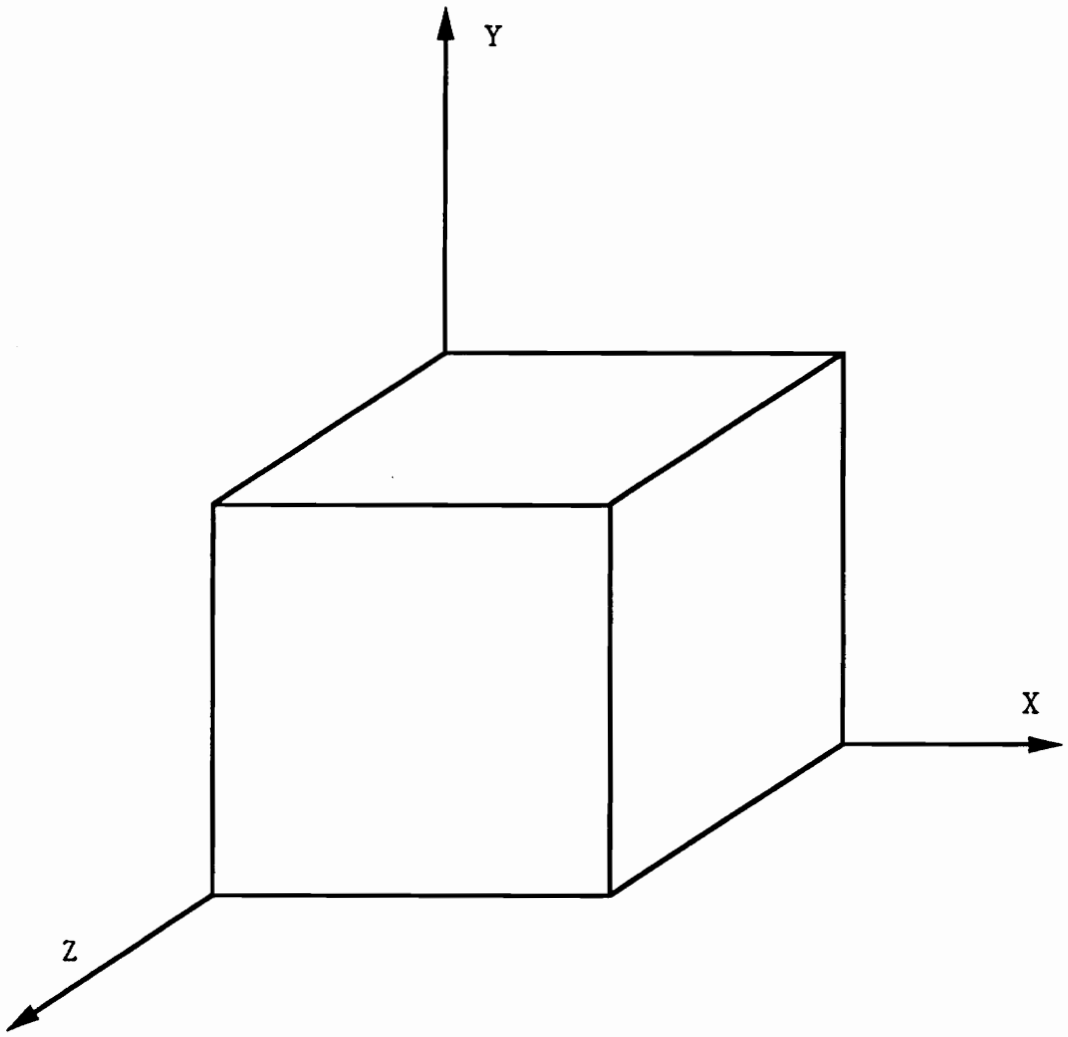
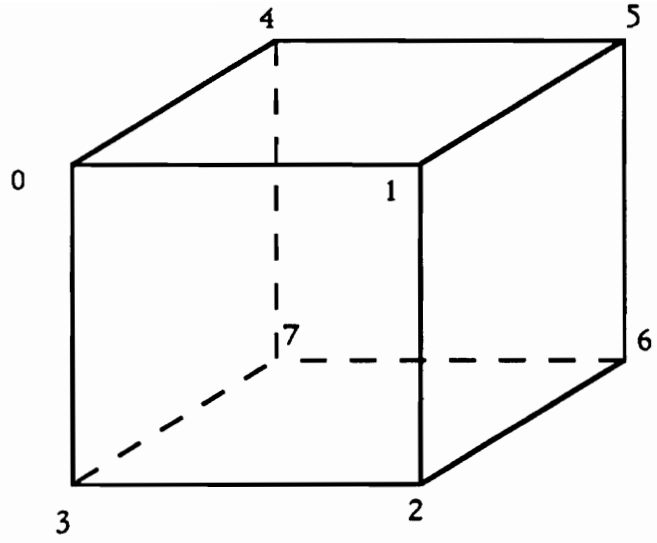
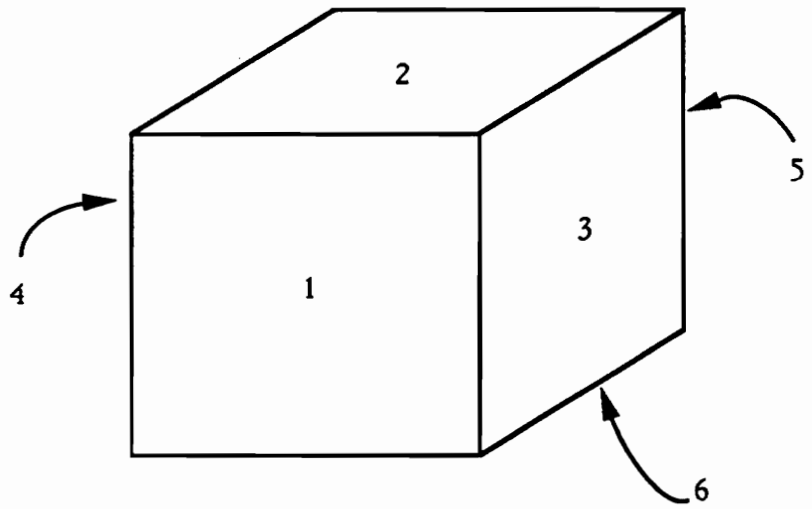


Figure 3. Bounding cube enclosing entire working environment.



(a)



(b)

Figure 4. Numbering system for faces and vertices of rectangular solids.

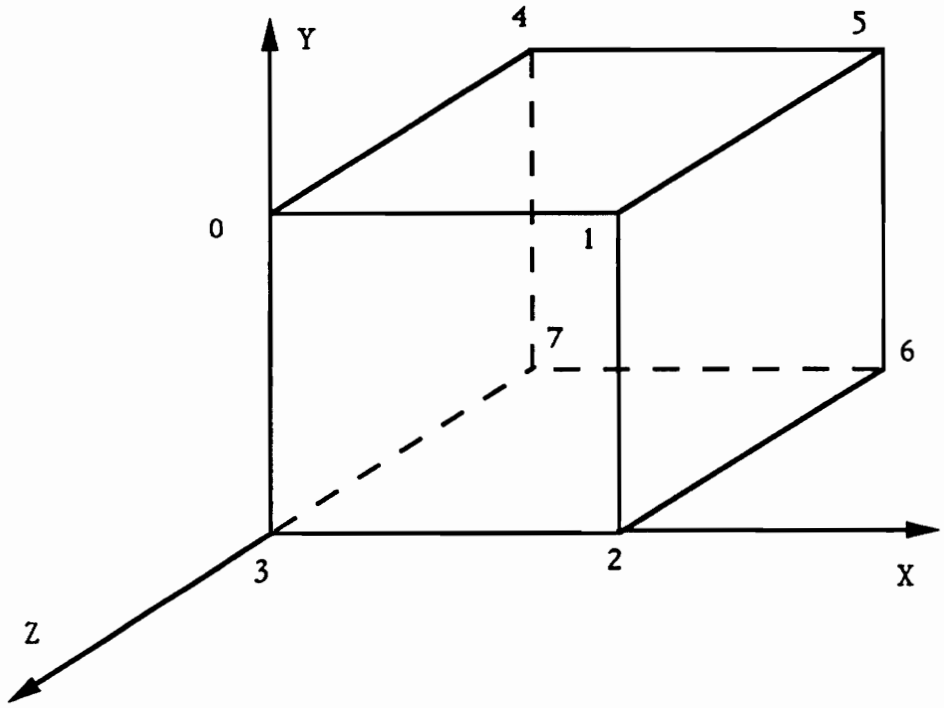


Figure 5. Position of a normalized rectangular solid.

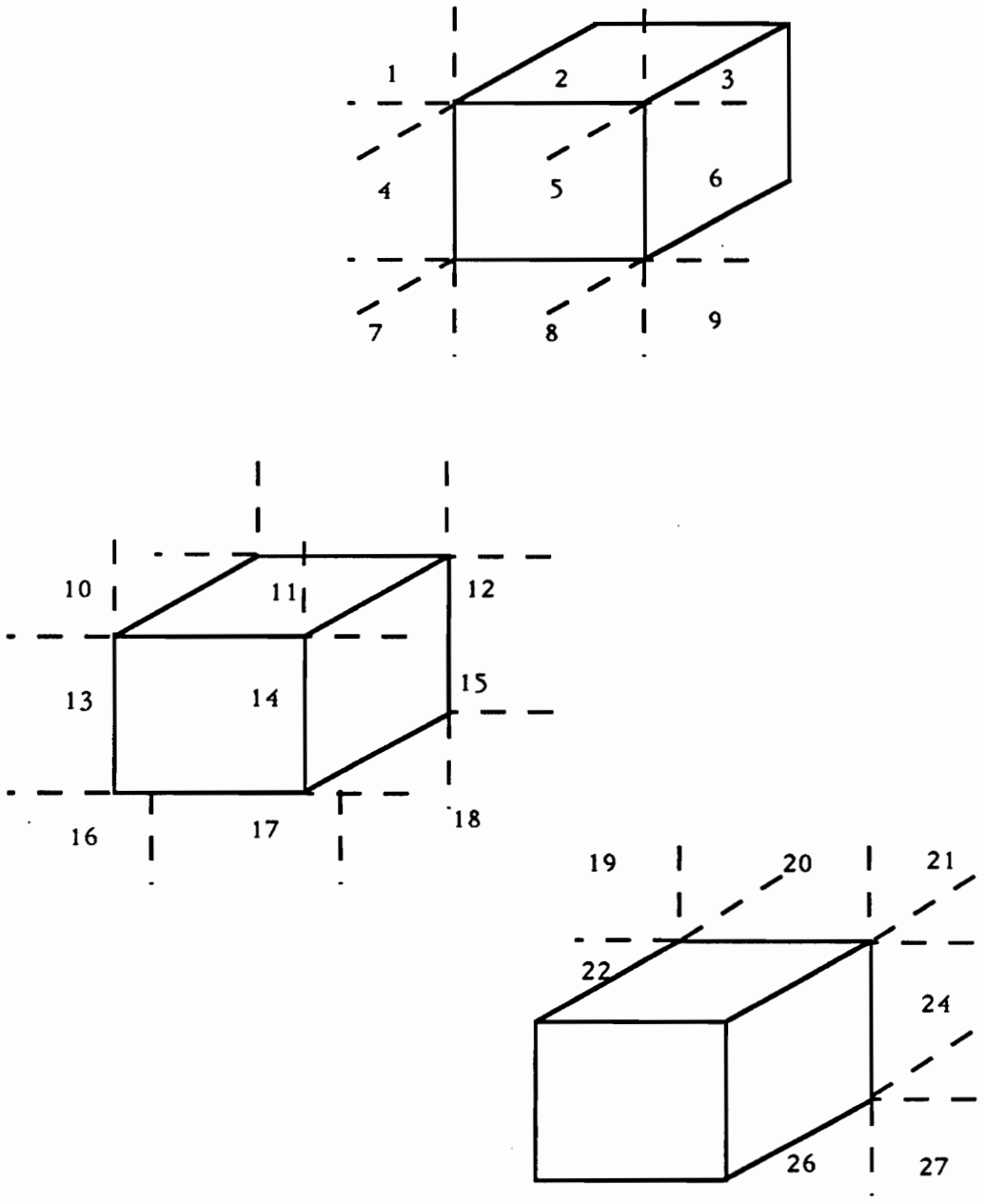


Figure 6. Numbering system adopted for quadrants.

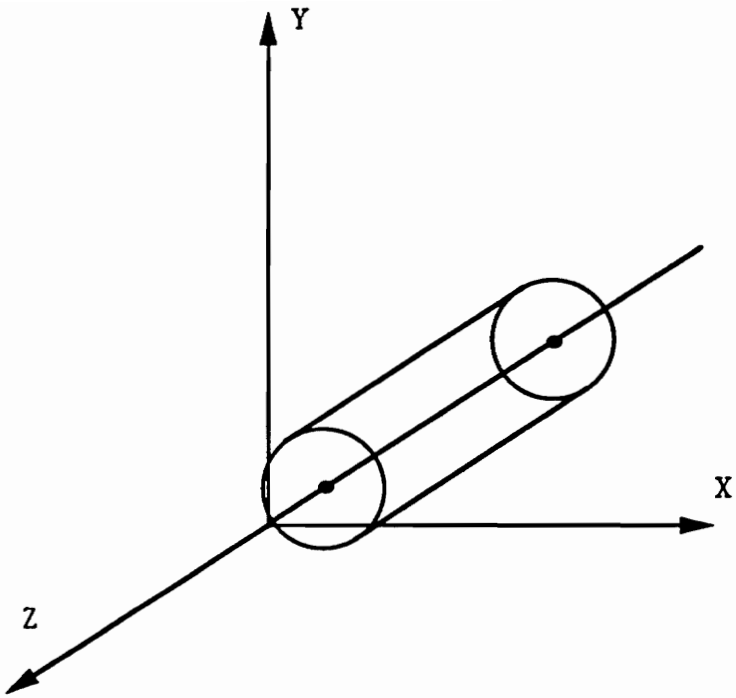


Figure 7. Position of a normalized cylsphere.

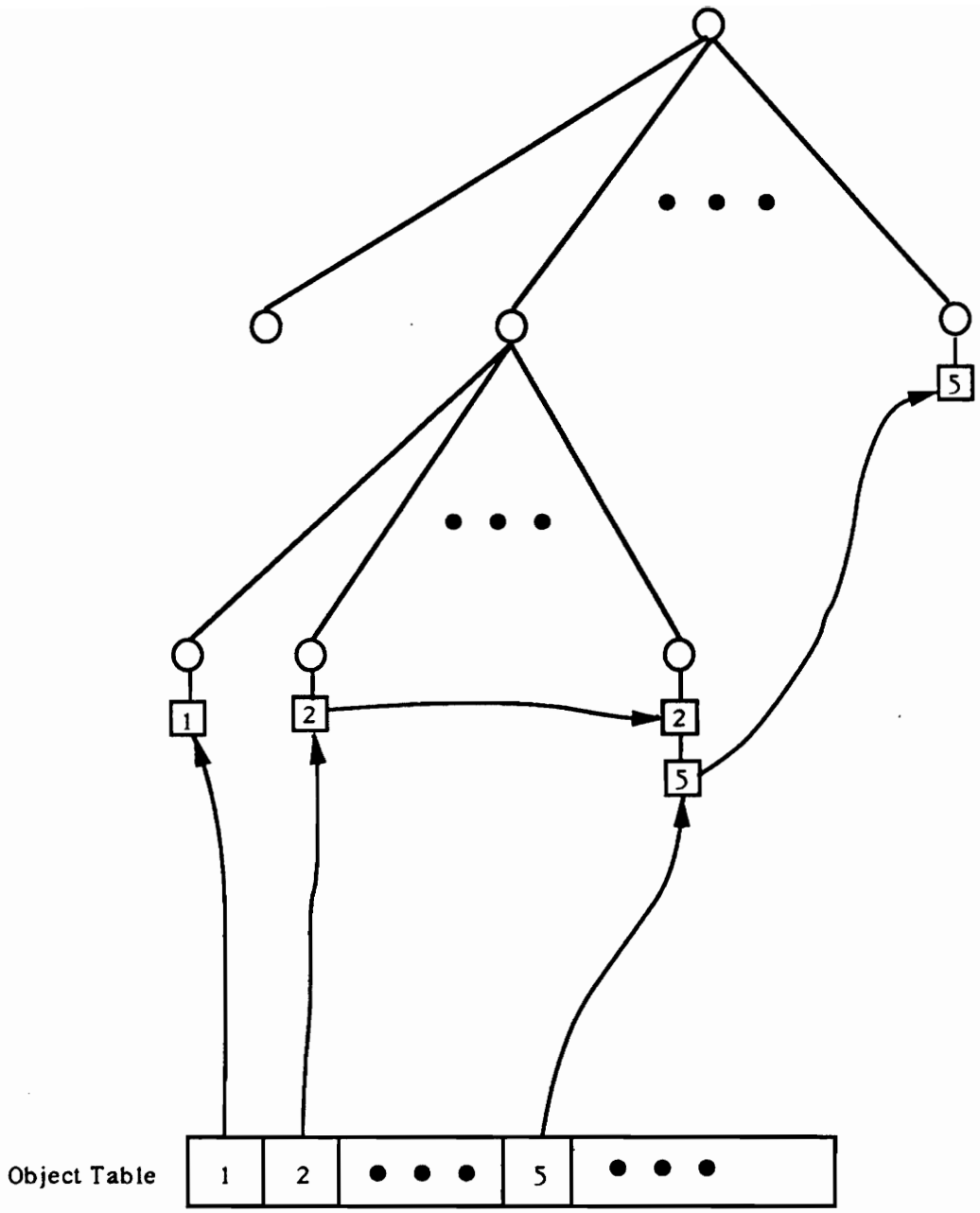


Figure 8. Location links for objects in the octree.

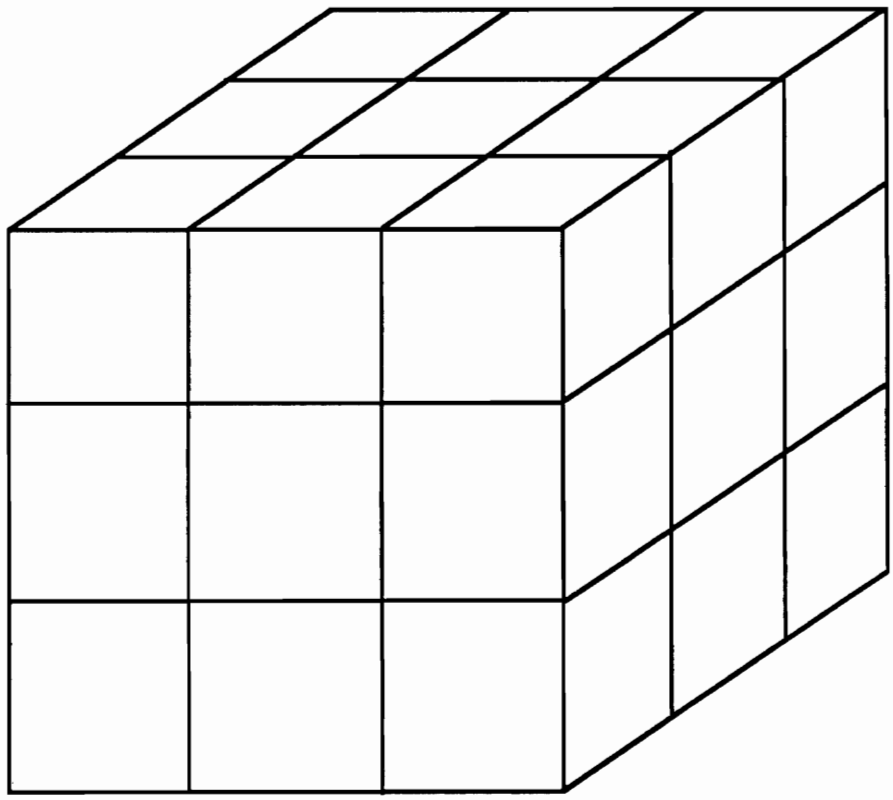


Figure 9. A node nested in 26 equal-sized neighbor nodes.

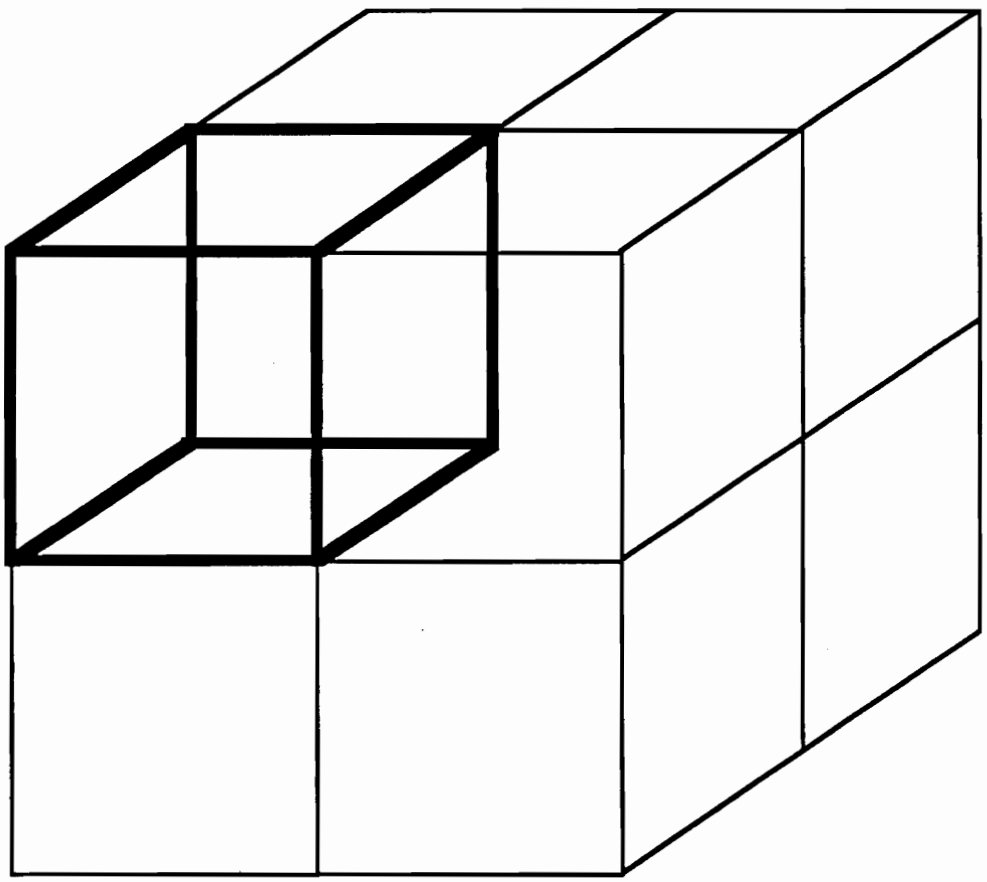


Figure 10. A node and its sibling neighbors.

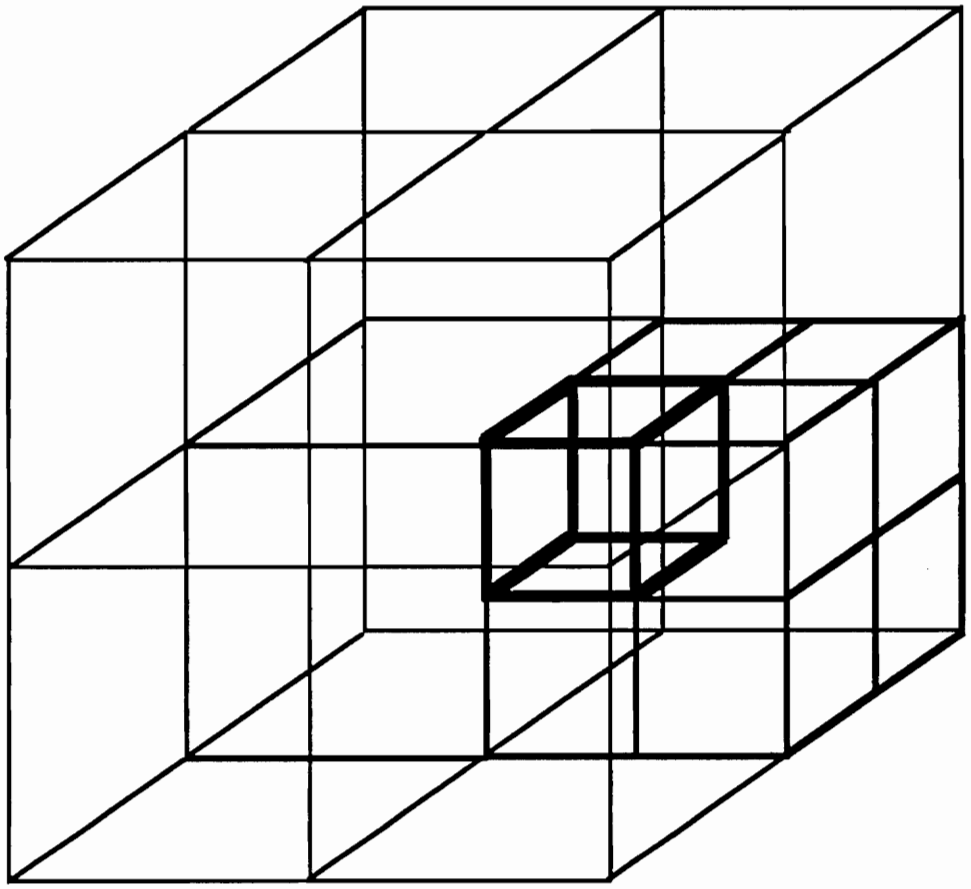


Figure 11. A node completely enclosed by its siblings and the neighbors of its parent.

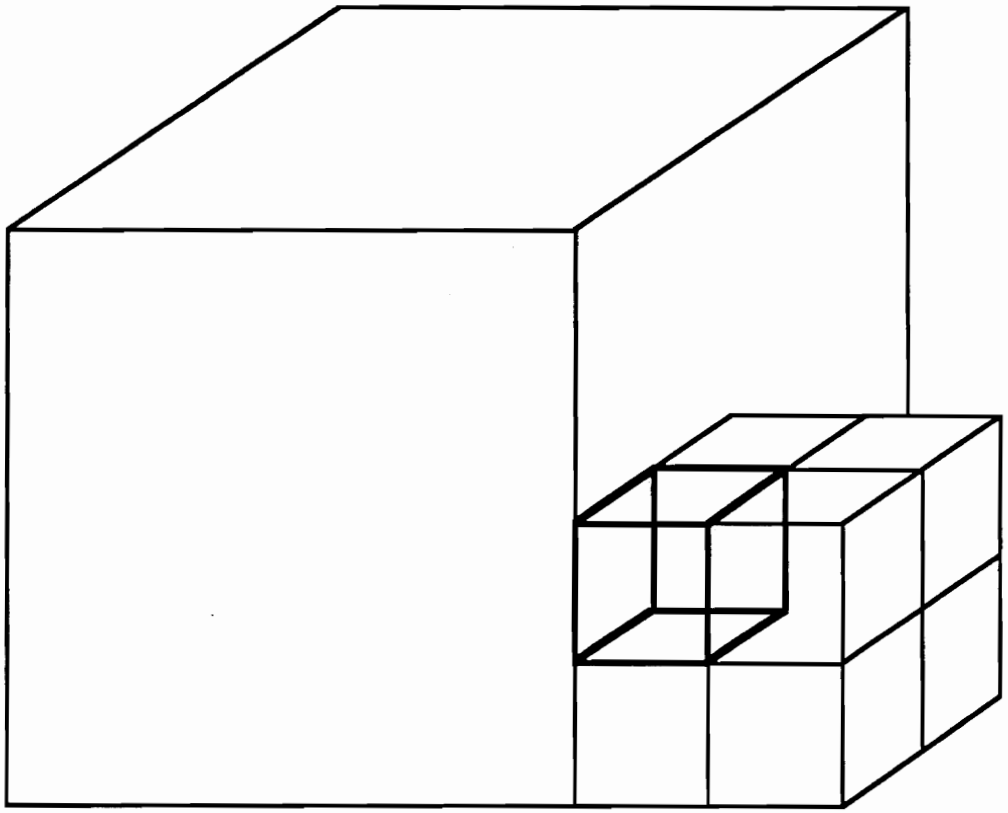


Figure 12. A face neighbor which also covers an edge.

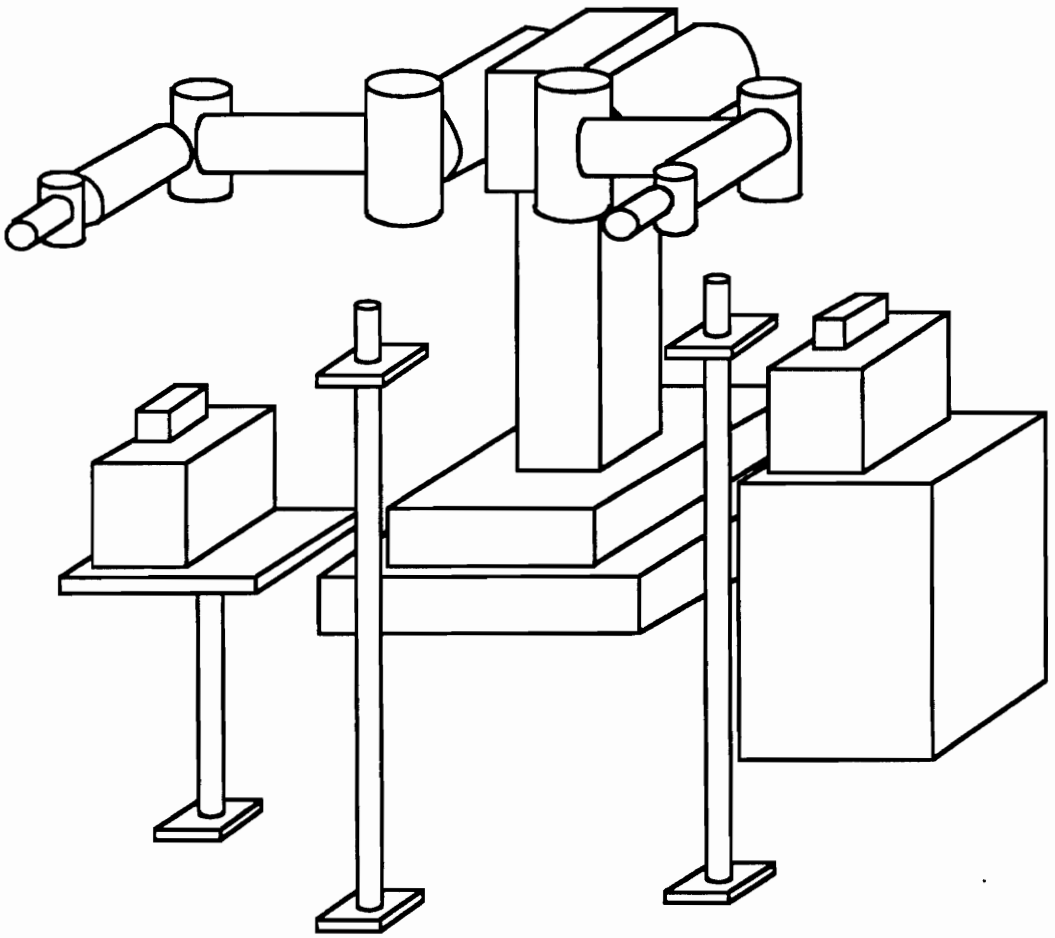


Figure 13. Sketch of working environment used to test collision detection algorithms.

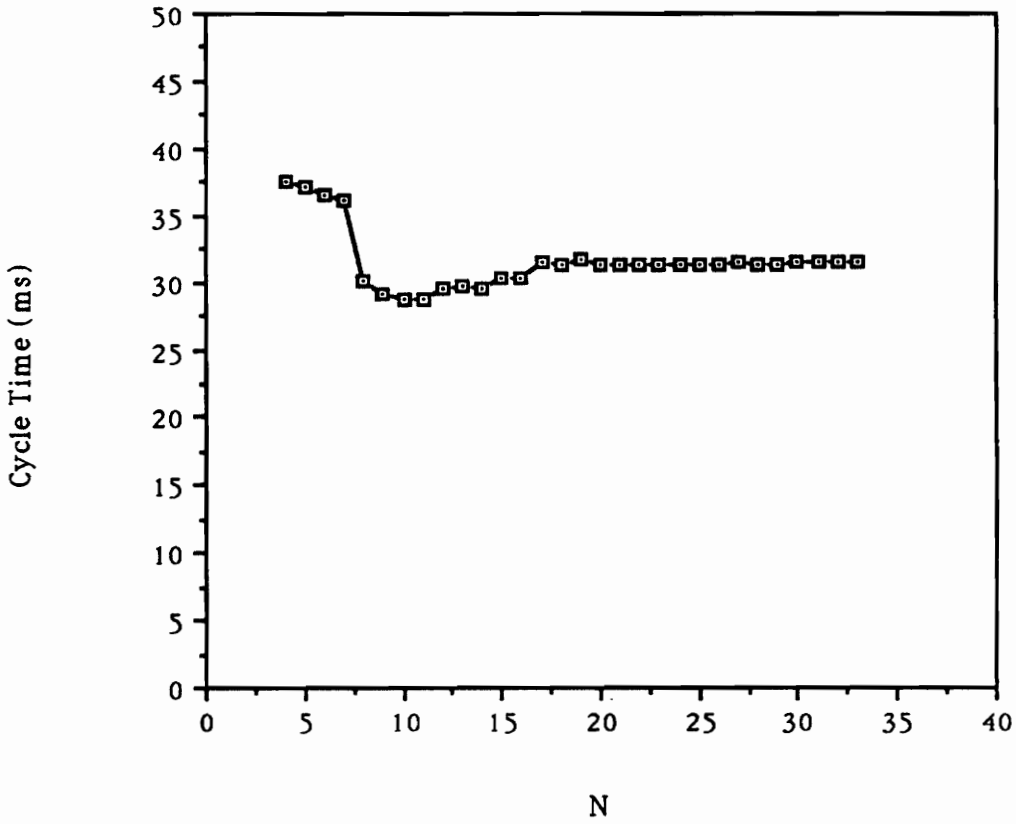


Figure 14. System performance for different values of N in decomposition rule.

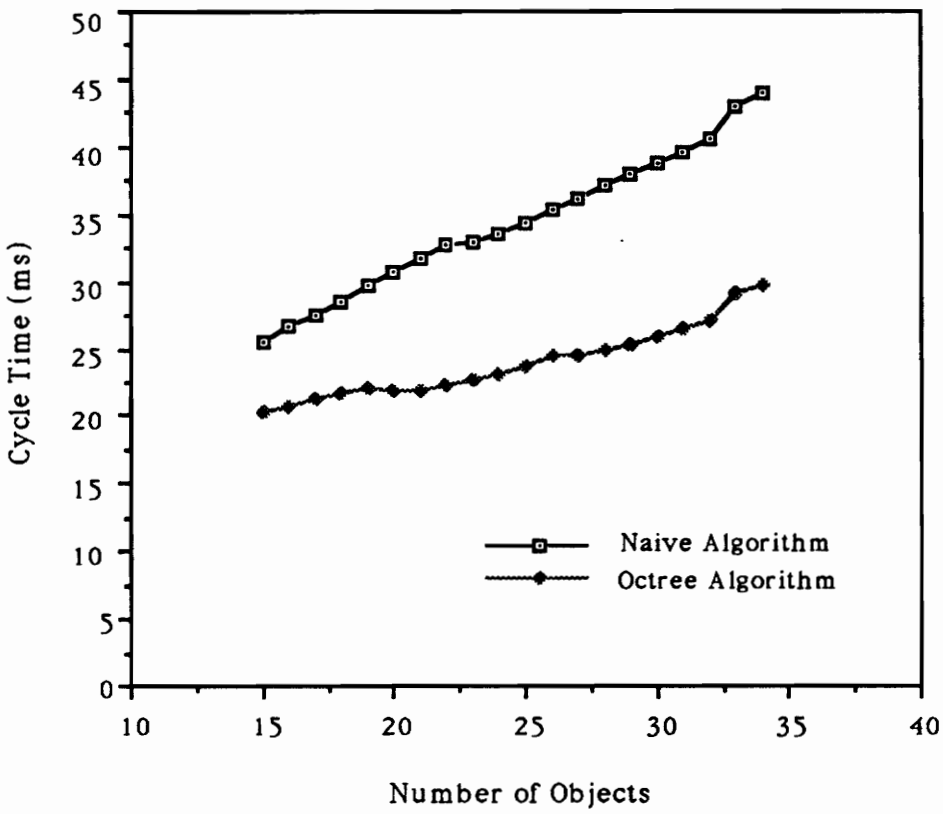


Figure 15. System performance of naive and octree algorithms as environment grows in size.

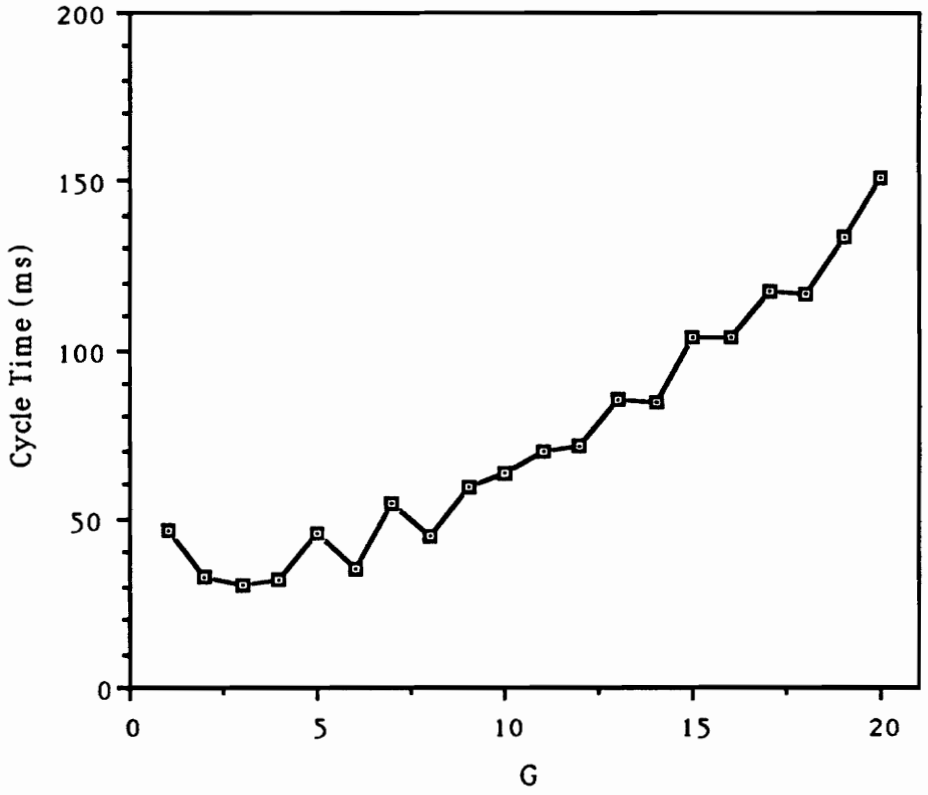


Figure 16. System performance of grid algorithm as grid size (G) is varied.

Vita

Gregory Mark Herb was born in Baltimore, Maryland on October 12, 1964. He graduated from Virginia Polytechnic Institute and State University in June, 1987 with a Bachelor of Science in Computer Science. He enrolled in the Computer Science Masters program at Virginia Polytechnic Institute and State University in the Fall of 1988 and graduated in the Spring of 1990.

Gregory M. Herb