University of Nevada
Reno

# A Parallel Linear Octree Collision Detection Algorithm

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science with a major in Computer Science.

by

Benjamin J. Lucchesi

Dr. Frederick C. Harris, Jr., Thesis advisor

May 2002

## Abstract

In this thesis a new collision detection algorithm is presented that solves the all-pairs collision detection problem using parallel processing. The design of the algorithm is based on a linear octree and runs in parallel with a theoretical performance of $\mathcal{O}((n\ log\ n)/k)$ run time. The algorithm has been implemented as a collision detection system using object-oriented design techniques and a client-server architecture. The architecture of the collision detection system is designed to use the parallel capabilities of both shared-memory, multi-processor computers and clusters of networked computers. Additionally, the modularity of the collision detection system gives application developers the flexibility to choose the level at which the collision detection system is integrated into an application. Using the collision detection system, experimental results have been generated that demonstrate how the algorithm performs according to the calculated theoretical performance.

## Acknowledgements

I would like to thank Professor Harris, my advisor, for his generous help and guidance throughout my education, and for supporting my research interests in graphics and virtual reality.

I am very grateful to Professor Wishart and Professor Edberg for their valuable time, suggestions, and for serving on my committee.

I would like to thank my parents for supporting me throughout my education and for their unique form of encouragement during difficult times. I really would like to thank them for not selling my car.

I would like to thank Maggie Jameson for her hard work and guidance during the editing stage of this thesis.

I would also like to thank Professor Harris' wife for her willingness to proofread this thesis, and her humorous insights and comments.

Finally, I would like to thank Rong Fan for her moral support and endless patience throughout this research project.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Interactive 3D graphical applications are a class of application which continuously query a user for input and provide output by means of three-dimensional graphics. Techniques used to implement these types of applications are enabling technologies which make other types application areas possible, for example Virtual Reality. Virtual reality applications that simulate visual environments are examples of interactive 3D graphical applications. In [5] and [25], two applications are described that apply virtual reality to workplace training in the surface mining industry. In the first application, virtual reality is used to train off-highway vehicle operators on how to inspect a vehicle before operation in an open pit mine. In this application, a user is presented with a simulated vehicle to inspect. The purpose of the application is to train vehicle operators on how to identify indications of problems that may compromise the safety of a vehicle. The second application uses virtual reality to simulate operating a vehicle. In this application, the user drives a simulated truck in an open pit mine to practice safe driving practices unique to open pit mine operations. The applications described in these two papers illustrate the necessary qualities of a successful interactive 3D graphical application.

In order for an interactive 3D graphical application to be successful, it must present the user with a visually believable environment. The believability of a simulated environment can be broken down into three main factors. First, the models used in the simulation must be recognizable. Second, the user must be able to interact with the simulated environment at a rate comparable to a real-world experience. Third, objects in the simulation must behave in a manner similar to their real-world counterparts. In interactive 3D graphical application, these three factors are mutually dependent. In general, the more detailed a model is, the slower the model is to render to a display. When models in a simulation are slow to render, the slower the simulation is at responding to user input. When the application is slow at responding to user input, the period of time with which realistic object behavior can be implemented is decreased.

In summary, in order for an interactive 3D graphical application to be successful, all calculations performed by the application must be completed in a timely manner before the graphical display can be updated. When an application fails to update the display rapidly, a time lag will exist between when the user inputs a command or action and when the graphical display is updated. When users experience lag, they often re-input a command expecting the application to respond faster. When the application eventually does respond, the graphical display is updated according to input from a prior point in time or using the cumulative sum of all input collected from the last display update. The display generated is usually much different than what the user expects and causes a great deal of confusion. For example, if turning the steering wheel of a driving simulation is not reflected by the graphical display for several seconds after the wheel it turned, the user will have difficultly judging what effect steering actually has in the simulation.

One of the most costly operations performed in interactive 3D graphics applications is collision detection. Collision detection is a key technology used for implementing realistic object behavior in interactive 3D graphics applications. Specifically, collision detection is used to determine when objects in a simulated environment are intersecting with one another. This information allows the simulation to prevent solid objects from passing through one another. Collision detection is a nontrivial problem and is actively researched in many fields of study besides interactive 3D graphics including robotics, physically based simulation, and computational geometry.

In this thesis, the collision detection problem is explored in Chapter 2 by presenting the challenges imposed by interactive 3D graphics on collision detection algorithms and discussing several previously developed methods for solving collision detection problems. A new collision detection algorithm is then presented in Chapter 3 which solves one aspect of the general collision detection problem using parallel computing. A description of software that implements the proposed algorithm is described in Chapter 4 followed by performance results in Chapter 5. This thesis concludes in Chapter 6 with final remarks concerning what the project has accomplished and a discussion of future work.

# Chapter 2

# Introduction to the Collision Detection Problem

For the purpose of this discussion, the collision detection problem will be limited to its applicability in 3D interactive graphics applications. However, the concepts, ideas, and algorithms presented have been drawn from many areas of research concerned with collision detection including robotics [10, 23, 24], computational geometry, and physical simulation [6, 26, 27].

This chapter is organized as follows: the collision detection problem is presented in context by describing how it applies to three-dimensional interactive graphics applications; a discussion of common types of collision detection algorithms is presented along with an explanation of a collision detection pipeline; and examples of popular collision detection algorithms and data structures are explained.

## 2.1 Collision Detection and Interactive Graphics

The purpose of collision detection is to determine if, and in what manner, objects are colliding at a moment in time. To illustrate this purpose, one might simulate dropping a ball onto a flat level floor. In this simulation, collision detection is used to determine when the ball hits the floor. The obvious solution is to calculate the time of impact using the initial height of the ball and a gravitational constant. Unfortunately, this solution doesn't constitute collision detection, but rather collision prediction. By calculating the time of impact, the assumption is made that the path of the ball remains unimpeded for the duration of the simulation. However, if unpredictable changes in the simulation occur, the prediction may be incorrect. This is the case imposed by interactive 3D graphics on collision detection algorithms.

In an interactive 3D graphics application, the flow of control runs in a continuous loop, sampling time at discrete intervals. This flow control is illustrated in Figure 2.1. At the end of each interval of time, the state of the simulated objects is updated to reflect changes that occur during the interval. For example, an object moving along a path is translated to a new position according to its velocity vector. Interactive 3D graphics applications create the illusion of smooth, animated motion by redrawing the screen at the end of each interval of time, as shown in Figure 2.2.

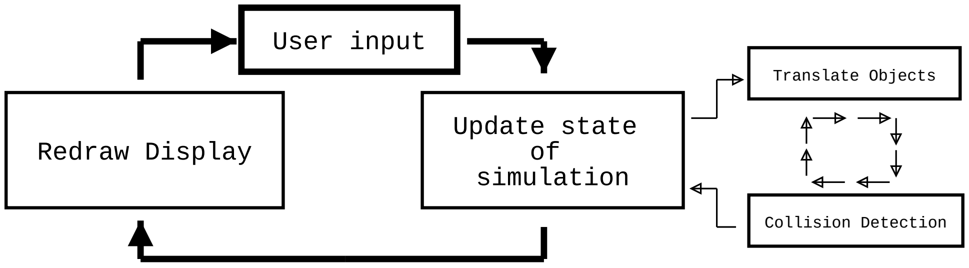
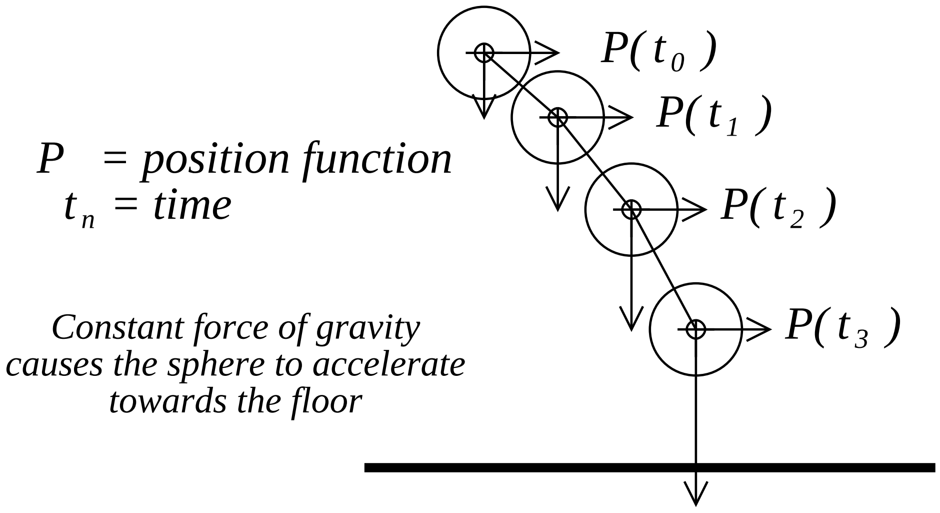


Figure 2.1: *Control flow of an interactive 3D graphics application.*

Along with updating the state of simulated objects, collision detection is also performed to determine whether the new position of an object causes it to interfere with the positions of other objects. It is important to note that collision response is an application-dependent issue and is not addressed by collision detection algorithms.

The mechanics of the simulation impose restrictions on collision detection algorithms. By sampling time at discrete intervals, only a fraction of the total elapsed



P = position function
t_n = time
Constant force of gravity causes the sphere to accelerate towards the floor
P( t_0 )
P( t_1 )
P( t_2 )
P( t_3 )

Figure 2.2: *Position of object is updated as a function of time to create smooth animation.*

time is accounted for in the simulation. As a result, it is possible for important events to be overlooked.

For example, consider the simulation under the conditions shown in Figure 2.3. At time interval $t_n$, the ball is positioned just above the floor. Then, at $t_{n+1}$, the ball's position is calculated to be just below the floor. If the floor is represented as a plane and collision detection is done using an intersection test, then the simulation would have incorrectly determined that no collision occurred.

A common solution to correct the problem of objects "jumping through" other objects is to increase the sample rate of the simulation timeline which will increase the probability of all collision being detected. Using this solution, one must determine the sample rate to be employed. If the rate is too high, application performance suffers. If the rate is too low, collisions might be missed.

One might consider expanding the simulation's capabilities to allow multiple balls to collide with the floor and with each other. Under the previous conditions, intersection between the ball and the floor could be tested rapidly at the end of each interval

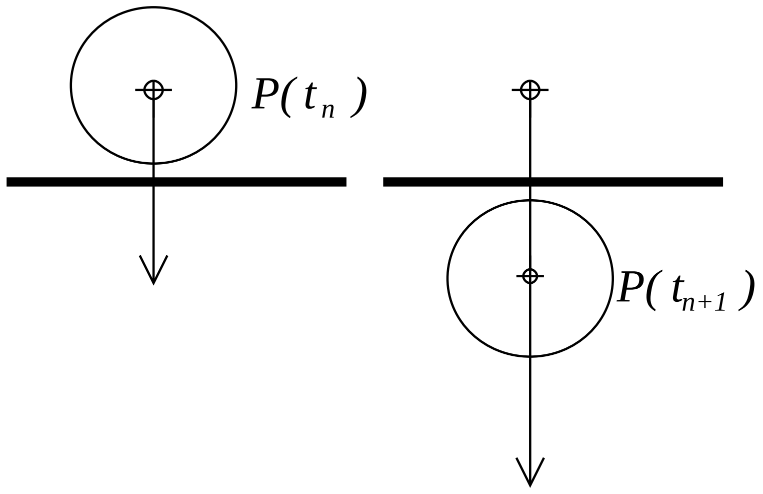


Figure 2.3: *Ball "jumping through" floor as a result of too long of a time interval.*

of time. However, with the new conditions, collision must be detected between all pairs of balls, $\binom{N}{2}$, and between each ball and the floor. The naive approach to solving this problem tests all pairs of balls and each ball with the floor for collision. This approach runs in $\mathcal{O}(n^2)$ time.

The situation is further complicated when objects that are more complex than spheres are introduced into the simulation. Testing for collision between two spheres or between a sphere and plane can be performed in a few operations. Due to the symmetry of a sphere, both tests reduce to a distance calculation using the center of the sphere. Detecting a collision between spheres runs in constant time.

This is not the case for general polyhedra. As show in [23], testing the intersection between two convex polyhedra can be done in $\mathcal{O}(n)$ time in the worst case where $n$ is the combined number of vertices of two polyhedra. Convex polyhedra are a special case of general polyhedra and are easier to test for intersection. For this reason, general polyhedra are often decomposed into convex entities. As a result of the running time of testing polyhedra for intersection, the number of polyhedra that can be tested for collision in a given period of time is much smaller than the number of spheres that could be tested in that period of time. As shown in Figure 2.4, complex objects may collide in many different ways making testing for collision a more complex and time consuming process than testing spheres for collision.

Testing for collision between multiple moving objects and testing for collision between complex objects are some of the cases effective collision detection algorithms must address. The manner in which an algorithm handles these situations determines the problem domain for which an algorithm is suitable. Issues of object representation, resource utilization, and acceptable performance characteristics also constrain applicability of an algorithm from one problem domain to another. Consequently, the breadth of highly specialized collision detection algorithms is great.

Although collision detection algorithms are specialized, many processes, concepts, and structures they employ are common throughout collision detection research. This is particularly true for algorithms that use similar data structures and geometric principles as a basis for their design and for hybrid algorithms that use multiple existing algorithms in conjunction with one another.

Many surveys and comparative overviews of collision detection algorithms are available in [13, 17, 34]. These sources discuss application domains, useful geometric principals, and solving strategies for collision detection problems are available. Rather than exploring all aspects of collision detection algorithms, the remainder of this chapter will discuss the components of general collision detection solutions followed by examples of collision detection algorithms.

## 2.2 The Collision Detection Pipeline

General solutions to collision detection problems in interactive 3D graphics applications are multifaceted. For example, detecting collisions between many objects and

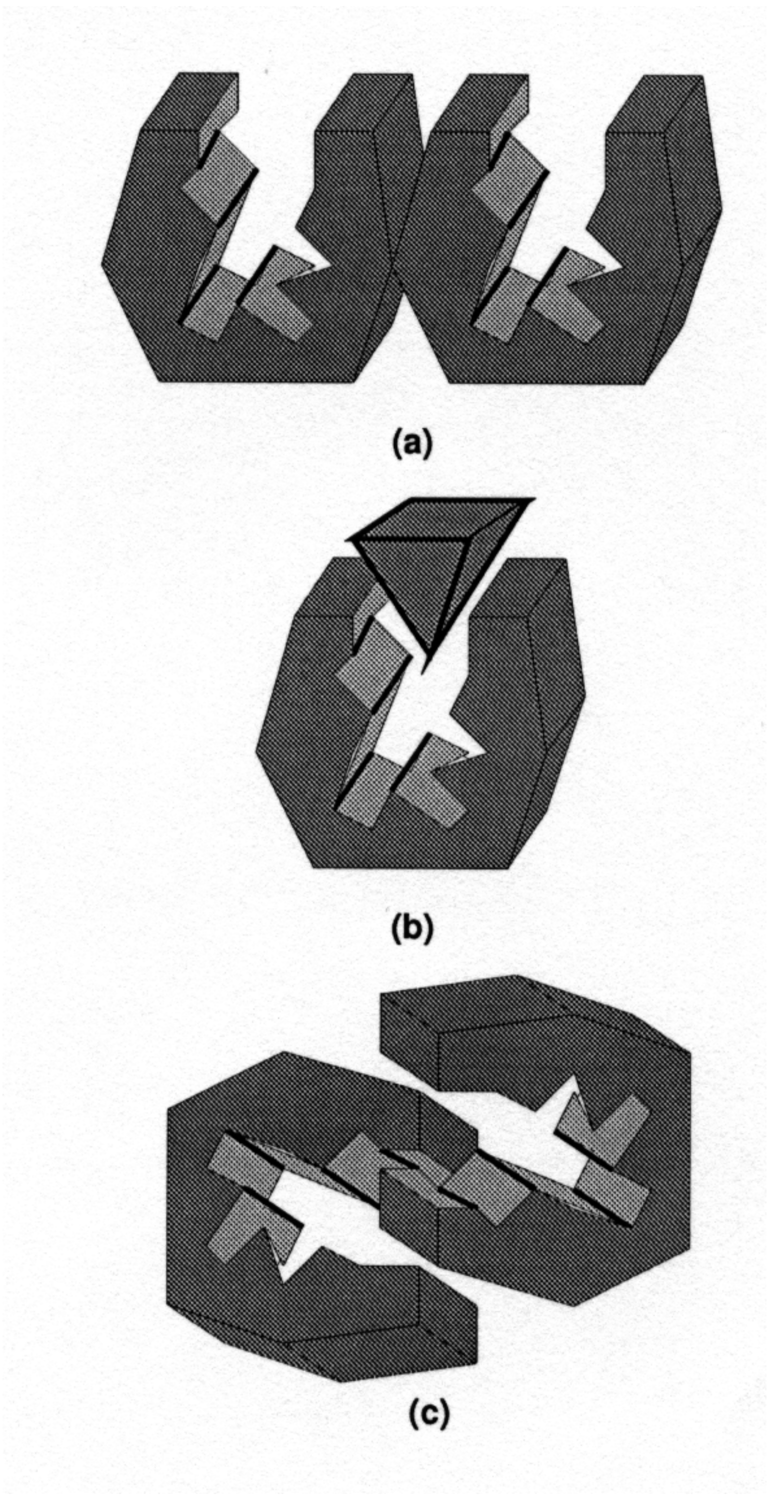


Figure 2.4: *Examples of complex objects colliding [22].*

detecting collision between two complex objects are separate problems with separate solutions. These two problems are commonly referred to as the all-pairs problem and the exact-object problem. Another facet of the collision detection problem is concerned with exactly how the geometric features of two objects intersect. Geometric features, or simply features, refer to a region or part of an object comprised of one or more polygons. This problem is commonly called the exact-feature problem. Software that implements a general solution to collision detection typically uses a collection of algorithms to address each facet.

Despite the disparate nature of each sub-problem, the algorithms for solving facets of the collision detection problem have well-defined relationships. These relationships have been studied in [34]. In this article, the relationship among aspects of collision detection algorithms is described as a pipeline of successive filters. Filters correspond to algorithms that address aspects of the collision detection problem, and the pipeline describes the order in which filters are applied to data. Input to the pipeline is a set of objects, and output is a pairwise description of collisions between objects. Internally, data flows from one filter to the next and is successively refined at each stage.

The relationship among various aspects of the collision detection problem has also been described as a multi-phase [8, 14, 30] and a multi-stage [15] process. In this description, the broad phase and the narrow phase are respectively analogous to the first stages and the last stages of pipeline filters.

Although these descriptions of the relationships among facets of the collision detection problem express the same idea, the pipeline paradigm is preferable because it emphasizes the composition of solutions to the general collision detection problem. Because each pipeline filter has a well-defined role, filters can be implemented as reusable components. Using the pipeline as a framework, specialized application-specific collision detection systems can then be assembled from filter components.

Another quality of the pipeline paradigm is that it remains conceptually flexible, allowing for modification without breaking existing conventions. As new collision detection algorithms are developed, pipeline filters can be inserted or replaced as a natural and expected process.

## 2.3 Algorithms for Solving the All-Pairs Problem

Stated formally, the all-pairs problem is to determine all pairs of $n$ objects that are colliding. Algorithms that solve the all-pairs problem are positioned at the beginning of the collision detection pipeline and are designed to reduce the number of exact-object tests performed in the latter stages of the pipeline.

The naive solution to the all-pairs problem is to test "all pairs" of objects for collision. This solution is considered naive because it is a brute force approach and runs in $\mathcal{O}(n^2)$ time. To solve the all-pairs problem efficiently, a strategy must be devised that can quickly differentiate between those objects that might collide from those that definitely will not.

Many algorithms have been developed to address the all-pairs problem. A common feature among them is their use of bounding volumes. The bounding volume of an object is another object that completely encloses the original object. Bounding volumes are designed to have a simple representation compared to the object they bind. Common choices for bounding volumes include cubes, boxes, and spheres. Algorithms that solve the all-pairs problem using bounding volumes instead of the original objects for collision detection are more efficient because testing for intersection between bounding volumes is, by design, much faster than testing for intersection using the actual objects.

### 2.3.1 Sweep and Prune Algorithm

One of the earliest algorithms developed to address the all-pairs problem is the sweep and prune algorithm [28]. The basic idea behind sweep and prune is to sweep a plane across a volume of space, testing pairs of objects for collisions that are simultaneously intersecting the plane. Pairs of objects that are not simultaneously intersecting the plane cannot collide and are eliminated from further collision tests.

In practice, implementations of sweep and prune operate by projecting all objects onto a coordinate axis, which results in intervals along the axis line. Overlapping intervals indicate possible collisions between objects. This is demonstrated in Figure 2.5. Determining overlapping intervals is a two-step process:

1. Sort the list of intervals in ascending order using the minima of the intervals.

2. Traverse the list in ascending order, testing intervals against the successive intervals to determine overlap. If the end-point of one interval is greater than or equal to the beginning-point of a subsequent interval, then the intervals overlap.

Sweep and prune is also called dimension reduction because it reduces the number of dimensions in which objects are compared.

A variation on sweep and prune is implemented in the I-Collide collision detection library [4]. In I-Collide, the sweep and prune algorithm projects objects onto three coordinate axes. Pairs of objects are considered for further collision tests only when their intervals overlap in all three dimensions. Costs associated with this algorithm include sorting three lists of intervals and testing for interval overlap on all three intervals. In the general case, sorting the lists runs in $\mathcal{O}(n\ log\ n)$ and testing for overlap

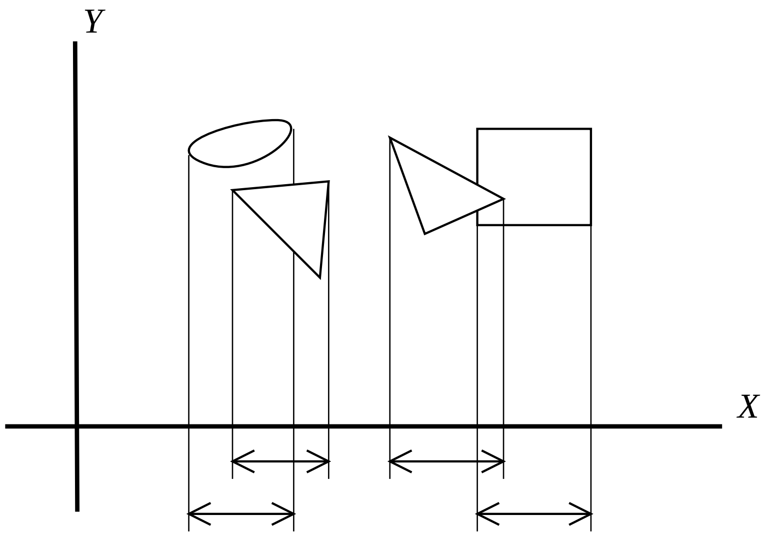


Figure 2.5: *Example of Dimension Reduction: Objects are projected onto the x-axis. Overlapping intervals along the axis indicate possible colliding objects.*

runs in $\mathcal{O}(n^2)$ in the worse case. However, as cited in [4], when objects maintain temporal and geometric coherence, performance of this algorithm is improved.

> Temporal coherence is the property that application state does not change significantly between time steps or frames. The objects move only slightly from frame to frame. The slight movements of the objects translates into geometric coherence because their geometry, defined by the vertex coordinates, changes minimally between frames [4].

Moreover, if the interval of time between object updates is small, objects can be expected to move relatively little between time steps. This makes the next position of an object predictably close to its previous position.

The sweep and prune algorithm described in I-Collide takes advantage of coherence by keeping the three lists of intervals between time steps. As a result of coherence, if the sample rate is high, the lists of intervals will remain in an approximately sorted order. This reduces the problem of sorting the lists of intervals to that of resorting partially sorted lists from the previous time step. In the I-Collide collision detection library, the lists are resorted by first updating the endpoints of the intervals and then using an insertion sort. When coherence is maintained, resorting the list using an insertion sort runs on average in $\mathcal{O}(n)$ time.

An additional cost incurred by this algorithm is in maintaining a data structure that stores the overlap status of objects. Again, if coherence is maintained, updating the overlap status of an object runs in $\mathcal{O}(n)$ time. Total running time of this algorithm is $\mathcal{O}(n + s)$ [4] where $n$ is the number of objects and $s$ is the number of overlapping intervals.

### 2.3.2 Octree Data Structures and Algorithms

Octree data structures belong to a class of structures that represent a volume space as a hierarchy of discrete units. Octree algorithms use a divide and conquer approach to navigating and searching a volume of space and have many uses in the fields of collision detection and interactive 3D graphics.

As the name implies, an octree data structure is a tree structure in which each node has eight child nodes. To represent a volume of space, the root node of an octree is associated with a cubic region within which the octree structure is contained. Below the root, each of the eight node children are associated with an even subdivision of the space enclosed by the root, as shown in Figure 2.6. The relationship between the root and its eight children can be recursively expanded to any number of levels, creating further subdivisions of the original cube associated with the root.

When discussing octrees, several terms are used for describing their components. The term octant is used to describe a cube that is one of eight even subdivisions of a larger cube. In the octree data structure, as with other tree data structures, nodes

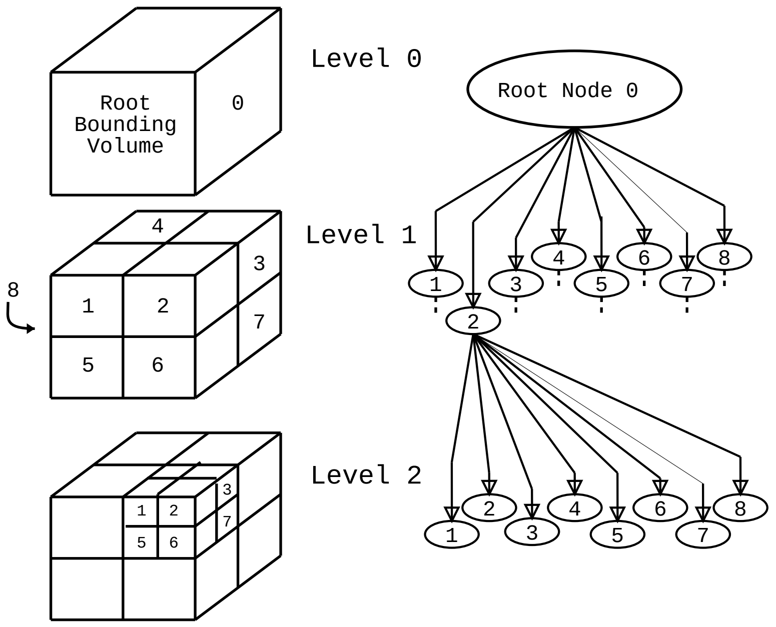


Figure 2.6: *A three-level octree and the corresponding tree data structure that represents it.*

that terminate a branch are called leaves. Octants that correspond to leaves in an octree are called voxels. A voxel describes the smallest unit of space that can be addressed in a discretized three-dimensional volume.

Octree data structures are commonly implemented as pointer-based tree data structures because pointer-based trees allow for a high degree of flexibility and simplicity in tree construction and traversal. As a feature of pointer-based trees, octants can be added and removed from the octree as needed. Internal to the octant data structure, parent octants store pointers to child octants which are used for navigating the octree data structure. In addition to pointers for navigation, octants store a record of data or a pointer to data that is associated with the octant. A good description of pointer-based octree can be found in [33].

Because of the manner in which octrees subdivide and index volumes of space, they are well suited to representing three-dimensional data volumetrically. To represent data as a volume of space in an octree, the data is associated with an octant. The position and size of the octant indicate the volume and location of the data. When data is stored in an octant that is not a leaf, it is implied that the data occupies all octants below the octant within which it is contained. These relationships between octants and data are shown in Figures 2.7 and 2.8.

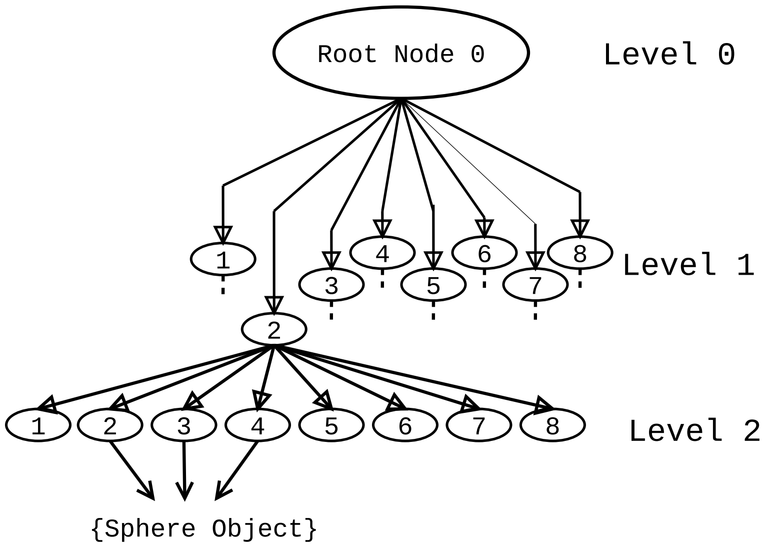


Figure 2.7: *Octree data structure storing the location of the sphere in the nodes of the tree. Note: not all octants that intersect the sphere are represented in the diagram.*

A typical strategy for inserting data into an octree is to perform a recursive traversal beginning with the root of the octree and searching for octants within which to store data. Beginning at the root, data is checked for intersection with each of the root's eight children. The traversal is continued within each child the data intersected. Depending on the needs of the algorithm, traversal of the octree can be terminated at a suitable time. Generally, most octree traversal algorithms run in $\mathcal{O}(log\ n)$ time with $n$ being the depth of the tree as a result of a divide and conquer approach to navigating the octree.

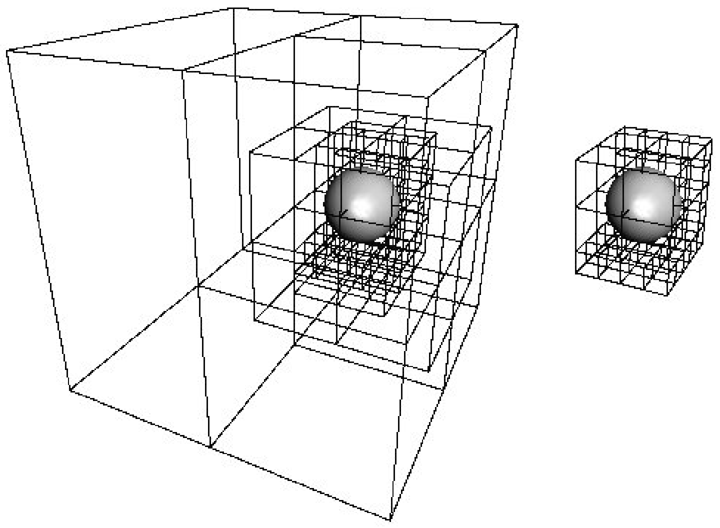


Figure 2.8: *On the left is an octree drawn with all octants that intersect a sphere. On the right only the octants binding the sphere are drawn.*

Octree data structures have been used in several collision detection algorithms [8, 12, 19, 30], the most successful of which operate in a similar manner. After a pointer-based octree is constructed and populated with objects, objects are moved in and out of octants as they move through space. To test for collision, the octree structure is searched to determine which objects share octants. Only objects that share octants are considered for further collision tests.

The differences among algorithms that use octrees are the way in which objects are moved between octants and how the octree data structure is searched. Two examples of collision detection algorithms that use octrees are [19] and [30]. In both examples, an octree is constructed so that it contains paths to only non-empty octants. This guarantees that every search path in the octree yields objects that need to be tested for collision. Also, both of these algorithms use an indexing scheme which allows the octants an object occupies to be calculated as a function of the center of the object. The algorithm in [30] uses coherence when moving an object from one octant to another and is able to maintain the octree in $\mathcal{O}(n)$ time. To move objects between octants, the algorithm in [19] builds additional search trees to manage collision events. This operation takes $\mathcal{O}(n\ log\ n)$ time.

In [8], a collision detection algorithm based on an octree uses coherence to achieve $\mathcal{O}(n)$ when moving objects between octants. However, unlike [30], [8] uses an auxiliary data structure to determine when an object is moving between octants. In [8], three lists are used to store the $x$, $y$, and $z$ endpoints of intervals defined by the axis aligned bounding boxes of the objects. After sorting the lists, the indices of the intervals that straddle the leaf octant boundaries are recorded. The recorded indices define the boundaries of buckets within the lists. By using an insertion sort, the order of the intervals in the buckets can be maintained in linear time as long as the positions or lengths of the intervals do not change significantly between sorts. This algorithm reduces the number of times that objects are checked for movement between octants by limiting its search to the objects that move between buckets.

Besides being suitable for solving the all-pairs problem in collision detection algorithms, octrees have been successfully applied in other fields relating to computer graphics such as Constructive Solid Geometry (CSG) [3]. Constructive Solid Geometry is concerned with performing high-level, logical operations between objects to construct new objects. This is illustrated in Figure 2.9. The motivation for this type of representation is to facilitate an interactive mode for solid modeling [32]. An example of a logical operation between two objects is to Boolean OR their volumes together to produce a new object.

Octree data structures are well suited for CSG applications because of the way in which volumetric regions are represented in octrees. In CSG applications, an octree

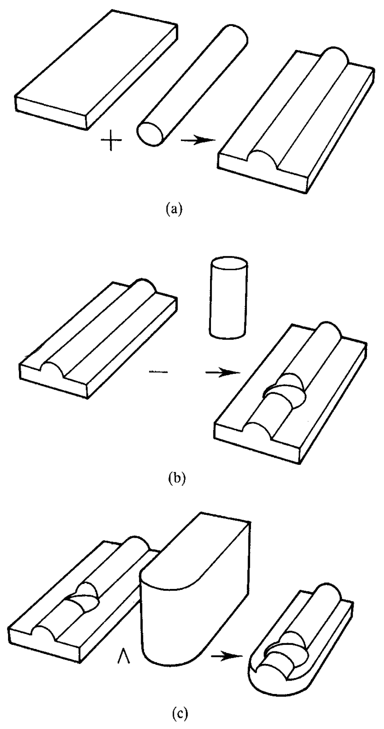


Figure 2.9: *Boolean operations between solids in CSG modeling: (a) union, (b) subtraction, and (c) intersection [32].*

is used to encode the volume of an object to a high degree of detail. This is done by finding the intersections between the surface of the object and the octants in a high resolution octree. A high resolution octree has exceedingly high number of leaf octants. The resulting octree is a hierarchical voxelization of the object's volume that captures the details of the shape of the object.

Logical operations between objects are performed using their octree representations. For example, subtracting one object from another is done by combining the octree data structures of two objects and deleting the common leaf octants.

In addition to octrees, other hierarchical and space-indexing data structures exist. These include BSP/k-d tree [13] and k-dop trees [21], which have been used extensively in collision detection algorithms.

## 2.4 Exact-Object Algorithms

In interactive 3D graphic applications, it is important to present users with a visual display of believable objects. One way to accomplish this is by representing objects with detailed models. Typically, these models are constructed using polygons positioned in 3D space. Polygons positioned in 3D space are preferable because they are convenient for constructing continuous surfaces that simulate the features and contours of real-world objects. Unfortunately, detecting collisions between these types of models is substantially more difficult than between simple objects. In interactive 3D graphic applications, algorithms that perform exact-object collision detection are designed to address the complexities of testing for collision between objects constructed from polygonal surfaces.

In much the same way that algorithms for solving the all-pairs problem prune the number of objects passed to exact-object collision detection algorithms, exact-object algorithms prune the number of features between two objects passed to exact-feature collision detection algorithms.

Many methods have been developed for analyzing the geometry of two objects in order to decide which features to test for collision. Of those methods, the most widely used are Bounding Volume Hierarchy (BVH) algorithms.

A BVH is a set of bounding volumes that recursively encloses the geometric features of an object. The resulting volumes form a hierarchy of enclosed spaces where levels of the hierarchy represent the resolution of enclosure around a geometric feature. Moreover, bounding volumes at lower levels of the hierarchy have tighter fits around their corresponding geometric features. This is shown in Figure 2.10.

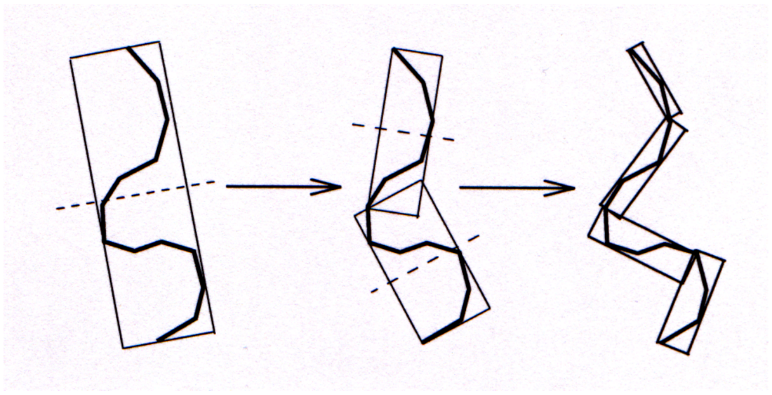


Figure 2.10: *Example of a Bounding Volume Hierarchy: bounding boxes recursively bound polygons [11].*

BVHs are designed to isolate quickly geometric features that are participating in collisions. This is done using an object's BVH as a guide to search for geometric features using a divide and conquer strategy.

BVHs are typically implemented as tree data structures. Tree nodes are used to store the volume bounding a geometric feature, and the relationship between parent and child nodes is such that the bounding volume of the parent encloses the bounding volume of the child. An octree data structure can be used as a BVH.

Algorithms that use BVHs to the find colliding features between two objects follow the same fundamental steps. Beginning at the root of each BVH tree, bounding volumes are tested for intersection with each other. When intersections are detected, the corresponding branches in each tree are descended. The differences among BVH implementations lie in the type of volumes used, how intersections between volumes are tested, and the algorithms used to build volume hierarchies for objects.

An example of an algorithm that uses a BVH in a parallel collision detection algorithm is presented in [20]. In [20], an octree BVH is built around the faces of two objects that may be colliding. When the octrees of two objects are compared with one another and voxels from the objects are found to overlap, the faces within the voxels are distributed to another processor where exact-feature tests are performed.

Another example of a algorithm that uses a BVH in a parallel collision detection algorithm is presented in [31]. In [31] a bounding volume hierarchy is constructed from bounding boxes and bounding spheres to obtain tight fits around the features of an object. Internal to the bounding volumes, balanced binary trees are used to store polygons enclosed by the bounding volume. This algorithm implements parallelism by spawning threads to perform searches between the branches of the BVH.

Other examples of BVH implementations include Oriented Bounding Box trees (OBB) [11], Axis-Aligned Bounding Box trees [1], Brep-Index trees [18], Binary Space Partitioning trees [2, 29], Sphere Trees [14], and octrees [28].

## 2.5 Exact-Feature Testing

The final stage of the collision detection pipeline concludes with an exact-feature test. Exact-feature tests are used to determine if the geometric features from two objects are, in fact, intersecting. Algorithms for these tests are based on mathematical solutions to geometry problems. When objects are constructed from polyhedral surfaces, this problem reduces to detecting if the edge of one surface pierces the face of another. For a discussion of efficient techniques used to solve this problem, refer to [17] and [32].

# Chapter 3

# Linear Octree-based Parallel Collision Detection Algorithm

## 3.1 Introduction

As presented in Chapter 2, collision detection is a multi-faceted problem requiring individual solutions to subsets of the collision detection problem. Current collision detection algorithms use a collection of algorithms to solve subset problems. The relationships among subsets of collision detection algorithms and how they can be used in conjunction with one another is best described as a pipeline. By optimizing each algorithm used in a collision detection pipeline, fast solutions to the general collision detection problem can be devised.

This chapter describes the data structures and algorithms explored in the development of a parallel collision detection algorithm designed to solve the all-pairs problem. This chapter is organized into four sections. In Sections 3.2 and 3.3, two octree-based data structures considered for use in a parallel collision detection algorithm are described. In Section 3.4, a new parallel collision detection algorithm for solving the all-pairs problem is presented and an algorithmic analysis of the new algorithm is discussed.

## 3.2 Distributed Octree Data Structure

In the process of designing a parallel collision detection algorithm, two strategies were explored, the first of which involved pursuing a distributed octree implementation that would facilitate parallel processing for collision detection.

A distributed octree is an octree data structure in which the octants and the objects they contain, are assigned to separate processors. In this configuration, objects can be operated on concurrently by the processors that manage the octants within which the data are contained. See Figure 3.1 for an example of a distributed octree data structure.

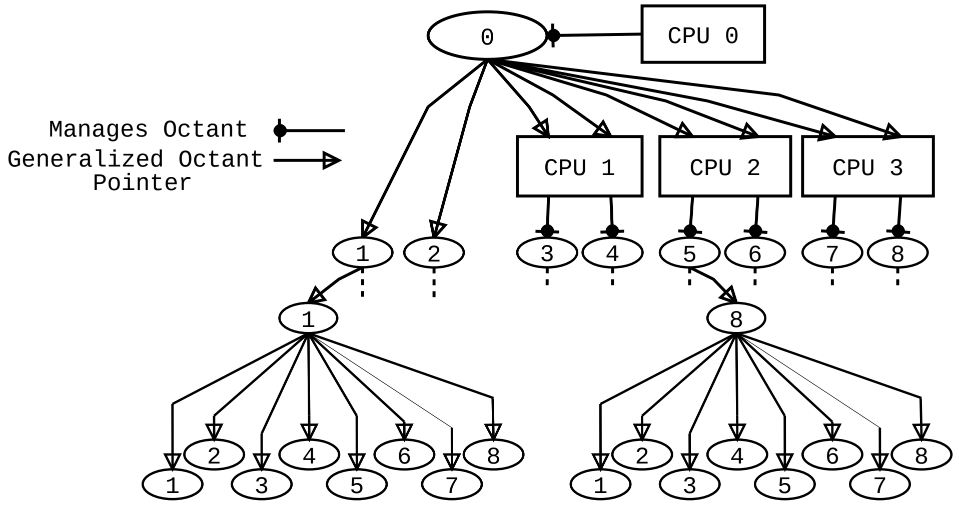


Figure 3.1: *Distributed Octree: 4-level octree data structure distributed over 4 processors.*

A distributed octree data structure has several qualities that make it appropriate for implementing a parallel collision detection algorithm. The five major qualities are:

1. Decomposition of an octree hierarchy for distribution is easy to understand and implement as a modified pointer-based tree. As stated in the description of

pointer-based octrees in Chapter 2, an octree node stores pointers to its child nodes. To facilitate the distribution of octant nodes to multiple processors, pointers used for referencing child octants are generalized, enabling them to reference a processor that manages octants. From a distributed octree implementor's perspective, the generalized pointer can be viewed as addresses used for communicating with processors that manage octants.

2. Distributing an octree across a cluster of networked computers is conceptually the same as distributing an octree across the processors of a shared-memory, multi-processor computer. As a function of the structure of an octree, clearly defined boundaries between regions of space are created. As a result of the logical structure of an octree, octree data structures have clearly defined regions by which data are segregated. In an octree, these boundaries are defined by the volume of an octant, but in an octree data structure, the boundaries are defined by the tree nodes in which data are stored. This feature allows tree nodes to be separated physically without compromising the logical structure of the octree. To distribute an octree data structure over a cluster of networked computers, pointers to octants are allowed to be computer network addresses where child octants are located.

3. When an octree is distributed across a cluster of computers, the aggregate size of the octree data structure can exceed the largest octree that can be stored on a single computer. This allows for a larger octrees to be constructed. On a single computer, the maximum size of an octree data structure that can be stored is limited to the amount of available random access memory. By distributing octants to several computers, the memory from each computer contributes to

the total space available for storing the data structure. It is important to note that although an octree data structure can be stored on disk, accessing the data structure is prohibitively slow as a result disk access speeds.

4. Distribution of data processing is a function of storing data in octants. As objects are placed in octants that bind their position and volume, they are accordingly distributed among the processors that manage octants.

5. The complexity of distributing an octree across multiple processors does not grow as a function of the number of processors over which it is distributed. As a result of the structure of an octree, once a program has been written to handle a single distributed octant, the same program should scale to any number of distributed octants.

In our first attempt to develop a parallel collision detection algorithm, a distributed octree was used to partition and manage objects among multiple processors and memories. Each processor would then perform collision detection between objects stored locally. However, after studying distributed octree implementations, we determined that distributed octrees are unsuitable for use in a parallel collision detection algorithm because they require load balancing to remain efficient, and, in some cases, must duplicate an object on multiple processors in order to build a tight-fitting bounding volume for the object.

### 3.2.1 The Load Balancing Problem

When objects become concentrated in small regions of space, their representations become concentrated in the octants of the octree representing that space. In the case of distributed octrees, processors managing octants where concentrations of objects exist have a disproportionately high workload. As a result of an unevenly distributed workload, performance of the distributed octree gained through parallelism is decreased. To solve this problem, load balancing can be used to redistribute work among the processors. Redistribution of workload is accomplished using three operations: dynamic octant splitting, octant migration, and octant consolidation.

Dynamic octant splitting is the process of subdividing octants at runtime to create additional levels in an octree. Dynamic octant splitting is performed by subdividing the volume of an octant into eight new child octants and then inserting objects contained in the original octant into the newly created children. The goal of dynamic octant splitting is to create additional subdivisions of a volume to further segregate objects. If dynamically splitting an octant fails to adequately partition a concentration of objects, the procedure is repeated until the objects are sufficiently segregated. Octants containing high concentrations of objects are candidates for dynamic octant splitting.

Octant migration is the process of moving octants from one processor to another. To move an octant between processors, the definition of an octant and the data the octant contains are reassigned to another processor. After an octant has been reassigned, all pointers that reference the octant are updated to reflect the change of management. Depending on the architecture of the parallel computer for which a distributed octree is implemented, octant migration can involve as little as reassigning octant pointers, as is the case for a multi-processor, shared-memory computer, or as much as transporting octant definitions and data across a network connection when an octree is implemented for a networked cluster of computers.

Combined, dynamic octant splitting and octant migration can be used to distribute data and processing evenly throughout a distributed octree. However, as a result of these operations, the structure of a distributed octree becomes fragmented over processors and memory. Fragmentation reduces the performance of octree operations and consumes excessive amounts of memory as branches of the octree become unused and the paths through the octree span an excessive number of processors. To repair a fragmented octree, octant consolidation is used to remove unnecessary octants and decrease the number of processors spanned by paths through the octree. Octant consolidation is the opposite operation of dynamic octant splitting. Octants are consolidated based on the number of objects contained in a branch of an octree. If the children of an octant, combined, contain a small number of objects, the objects are moved into the parent and the children are deleted. This process reduces the amount of memory consumed by the data structure, decreases the number of levels in the octree, and decreases the separation distance between the root and leaves of the octree on a parallel computer.

An example of a distributed octree is described in [7]. Although not designed specifically for collision detection, the distributed octree implementation discussed in this article can also be used for parallel collision detection. A distributed octree is used for storing and manipulating scientific data sets for visualization. By distributing a data set across multiple processors, operations on data can be performed in parallel. Due to the large amounts of data, a distributed octree is well suited for this type of application.

### 3.2.2 The Object-Octant Membership Problem

Object-octant membership is a term that will be used to describe which octants an object intersects. The other factor which makes distributed octrees unsuitable for use in a parallel collision detection algorithm is the manner in which object-octant membership must be managed. The fundamental purpose of using an octree is to segregate objects based on their position. Octrees do this by creating a spatial relationship between objects based on which octants objects occupy. A spatial relationship exists between two objects that occupy a common octant. Several methods have been developed to determine the object-octant membership of an object. In this discussion, a search method will be used to illustrate problems with distributed octrees.

As discussed in Chapter 2, a typical strategy for inserting data into an octree is to recursively search for octants that intersect the object. Depending on how an algorithm terminates recursive descents into the tree, upon completion of an insertion operation, a reference to the object will have been stored with octree data structure nodes that correspond to the location and volume of the object. Octree nodes that store references to an object define the object-octant membership of the object.

As seen in Figure 3.2 and Figure 3.3, when the volume of an object straddles octants managed by different processors, the object must be stored by each processor that manage octants that intersect the object.

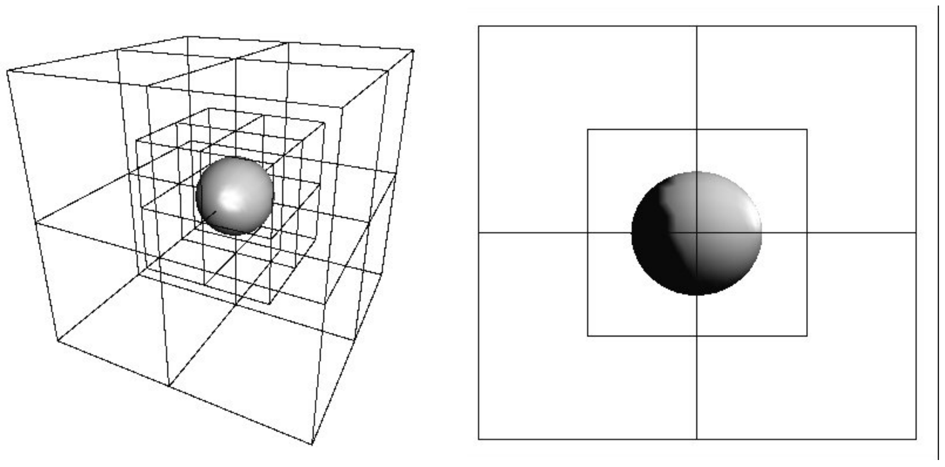


Figure 3.2: *A perspective and orthogonal view of a sphere located in the center of an 3-level octree.*

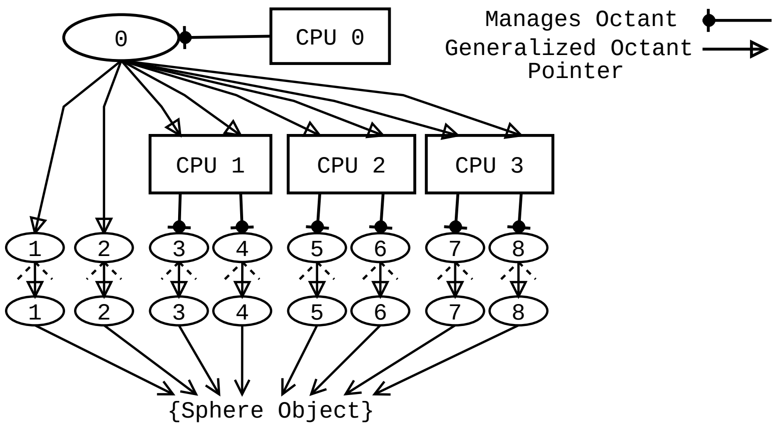


Figure 3.3: *A distributed octree data structure storing the sphere from Figure 3.2.*

Storing objects on multiple processors is problematic for distributed octrees because maintaining the state of an object requires coordinating the processors which store the object.

Although the difficulties surrounding distributed octrees can be overcome, the cost of load balancing in terms of performance outweigh the benefits of using it for parallel collision detection. For this reason, distributed octrees are not suitable for use in a parallel collision detection algorithm.

## 3.3 Linear Octree

The second data structure we considered for use in a parallel collision detection algorithm was a linear octree. A linear octree is not a tree-like data structure. It is an encoding scheme derived from the structure of an octree. Codes generated using a linear octree encoding scheme uniquely identify voxels positioned within an octree. These codes are used for describing three-dimensional objects volumetrically based on the voxels an object intersects. The set of codes that describe the volume of an object are called a linear octree. A linear octree is stored in a list data structure.

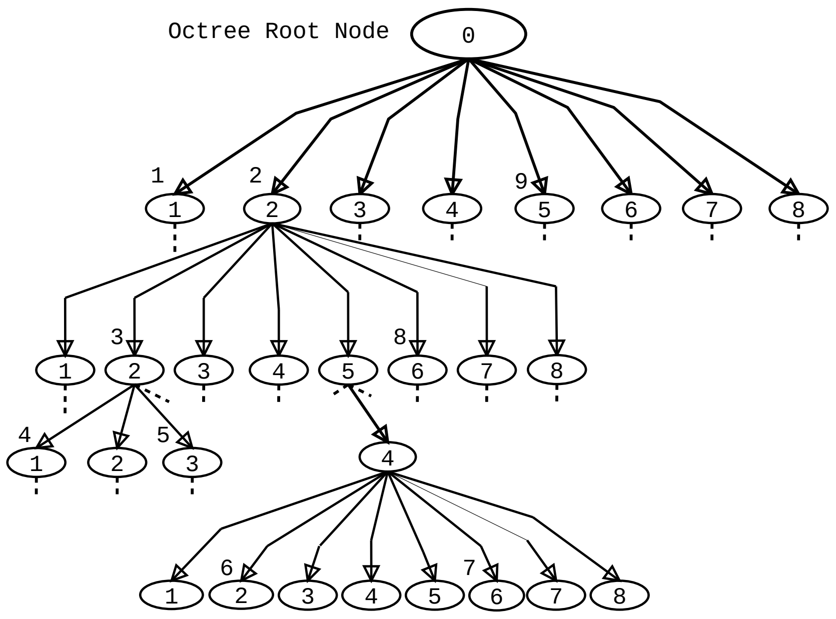
The basic unit of a linear octree is an octal code, also called octnode code. An octal code is a sequence of numbers that, when read from left to right, describe a path from the root of an octree to an octant. The numbers used in the sequence of an octal code correspond to the indices of octants. The position of a number in the sequence of an octal code corresponds to a level in the octree. For example, the octal code `{3, 1, 4}` refers to an octant located on the third level of an octree and can be found by recursively descending into octant 3, followed by octant 1, and finally octant 4.

The sequence of numbers in an octal code can be stored conveniently in an unsigned integer by using the digit positions to store the octal code sequence. For example, the sequence `{3, 1, 4}` can be represented by the integer `{314}`.

Several variations of the linear encoding scheme have been developed. [28] provides a comprehensive overview of these variations as well as a description of their origins. [16] also provides a thorough description of linear encoding schemes as well as discusses set operations between linear octrees. [9] briefly describes a linear encoding scheme and then describes how to use linear octrees for ray tracing.

Linear octrees have several qualities that make them useful for parallel computing environments. The most useful quality is that a linear octree preserves the hierarchical nature of the data it represents, while avoiding the need to retain a pointer-based tree. Furthermore, the data representation of a linear octree is convenient for transportation by means of interprocess communications.

Another useful quality of linear octrees is a unique property that octal codes exhibit. By sorting in ascending order a list of octal codes represented as integers, the resulting sequence is the pre-order traversal of an octree [28]. As show in Figure 3.4, by visiting octants from a list of octal codes that has been sorted in ascending order,



```
Octal Codes: 1000, 2000, 2200, 2210, 2230, 2542, 2546, 2600, 5000
Visit Order: 1,    2,    3,    4,    5,    6,    7,    8,    9
```

Figure 3.4: *Octal codes sorted in ascending order and the order octree nodes are visited.*

nodes in the tree are visited in order on lower to higher numbered branches and from top to bottom. This property of octal codes is useful because it provides a method for merging disparate linear octrees into a single octree. Given a set of octal codes, a pre-order traversal of an octree that only contains octants that intersect objects can be found by simply sorting the list of octants. This property is the cornerstone of the new algorithm presented in the next section.

## 3.4 A Parallel Linear Octree Collision Detection Algorithm

In this section, a new collision detection algorithm is proposed that uses a linear octree for solving the all-pairs collision detection problem. The algorithm has $\mathcal{O}((n\ log\ n)/k)$ performance, where $n$ is the number of objects and $k$ is the number of processors.

The strategy behind the proposed algorithm is to construct the linear octrees of $n$ objects in parallel using $k$ processors and then merge the resulting octrees into a single linear octree on a single processor. In the description of this algorithm, the following assumptions are made:

1. All objects are cubes.
2. The parallel program consists of one master process and $k$ slave processes.
3. Each slave process either shares a single octree, or all processes have an exact copy of the same octree.
4. The number of objects is an even multiple of the number of processes.

### 3.4.1 Algorithm Outline

The following is an outline of the steps performed by the algorithm to generate a linear octree of a set of objects and test the objects for collision.

1. The master process divides a list of $n$ objects by $k$ processes and distributes a list of $n/k$ objects to each process. The master process then waits for all slaves to complete processing before execution is resumed.
2. Each slave waits to receive a list of objects from the master. When the list of objects is received, each slave builds a linear octree for each object. The resulting octal codes are paired with an index of corresponding objects from the main list of objects. All resulting octal-code/object-index pairs are stored in a single list.

3. Each slave process sorts in ascending order its list of octal-code/object-index pairs by the integer representation of the octal codes.

4. Each slave sends its list of octal-code/object-index pairs back to the master process. The slave process is complete.

5. The master process receives $k$ list of octal-code/object-index pairs from the slave processes.

6. The master process merges $k$ lists of octal-code/object-index pairs into a single list.

7. The master process iterates in ascending order through the list of octal-code/object-index and finds all pairs of octal codes that share octants. The indices of objects that correspond to octant-codes that share octants are stored in a list.

### 3.4.2 Algorithm Pseudo Code

The algorithm can be summarized in two pseudo code programs, one for the master process and one for the slave processes.

```
Master Process()
Begin
  integer k := number of slaves
  address slave[ 0 to k ] := locations of slave processes
  cube object_list[ 0 to n ] := list of n objects
  Pair<octalcode,integer> slave_lists[0 to k][] := NIL
  Pair<octalcode,integer> master_list[] := NIL
  Pair<integer,integer> colliding_pair_list[] := NIL
  octalcode smallest_octalcode := NIL

// Step 1
  sublist_size := sizeof(object_list[]) / k
  index := 0
  For each x := from 0 to k
  {
    Send( slave[x], object_list[ index to (index + sublist_size) ] )
    index := index + sublist_size
  }
```

```
// Step 5
  num_octal_codes := NIL
  For each x := from 0 to k
  {
    Wait(slave[x])
    Receive(slave[x], slave_list[x])
  }


// Step 6
  master_list[] := MergeSortedLists(slave_lists[])

// Step 7
  index := 0
  For( object1 := 0 ... sizeof(master_list[]) - 1 )
  {
    For( object2 := object1 ... sizeof(master_list) )
       if( CommonPath( master_list[object1], master_list[object2] ) == true )
       {
         colliding_pair[index] := object1,object2
         index := index + 1
       }
       else
         break from inner for-loop
  }
End

Slave Process()
Begin
  cube object_list[] := NIL
  octalcode linear_octree[] := NIL
  Pair<octalcode,integer> slave_list[] := NIL
  address master := address of master process

// Step 2
  Wait( master )
  Receive( master, object_list )

  For each x := object_list[]
  {
    linear_octree[] := SearchOctree(object_list[x])
    For each y := linear_octree[]
    {
      slave_list[] append pair(linear_octree[y],x)
    }
  }

// Step 3
  Sort( slave_list[] )
// Step 4
  Send( master, slave_list )
End
```

```
Subroutine Send( address, data )
{
  Send "data" to a process at "address"
}

Subroutine Wait( address )
{
  Wait for communications from the process at "address"
}

Subroutine Receive( address, data )
{
  Receive "data" from the process located at "address"
}

array Subroutine MergeSortedLists( "list of sorted lists" )
{
  Merges a "list of sorted lists" into a single "list"
  return "list"
}

Boolean CommonPath( octalcode1, octalcode2 )
{
  Determines if the octal codes share an octant.
  Two octal codes share an octant if both codes define
  octants on the same path through the octree.

  Return true if two octants are shared, otherwise return
  false.
}

array SearchOctree( cube )
{
  Determine the object-octant membership of an object return
  the octal codes of the octants this object intersects.
}

Sort( octant[] )
{
 Sorts a list of octants in ascending order
}
```

## Bounding Cube Representations

Like other algorithms for solving the all-pairs collision detection problem, this algorithm uses bounding representations of objects to gain performance. In this algorithm, bounding cubes are used. Bounding cubes are used for two reasons. First, testing for intersection between two cubes is efficient. This is important because the octree search algorithm makes extensive use of cube intersection tests. Second, cubes can be stored efficiently in memory. Storage efficiency is important because it reduces the overall memory requirements of the collision detection algorithm and can be quickly transported over a network connection.

### Octree Search

To build a linear octree for an object, the slave processes search an octree for a set of octants that. When combined, these octants form a cube that binds the bounding cube representation of the object.

### Octal Code Comparisons

The `CommonPath` subroutine is used to determine if two objects are colliding by determining if they share octants. The `CommonPath` subroutine is so named because octal codes that define similar paths through an octree have overlapping octants. The general algorithm for comparing two octal codes is a piecewise equivalence test between the sequences of numbers stored as octal codes. If any of the numbers between the sequences differ, then the octal codes identify disparate regions of space. Otherwise, the octal codes identify overlapping regions of space, and the corresponding objects are considered for further collision testing.

It is possible for two octal codes to define paths of different lengths. When this occurs, the octant indices are compared up to the maximum length of the shorter octal code. For example, consider the two octal codes `{1, 2, 3, 4, 5}` and `{1, 2, 6, 0, 0}`. In this case, because the second octal code defines a shorter path than the first octal code, only the first three numbers in the code would be compared.

In practice, the implementation of the general algorithm exploits the integer representation of octal codes. When two octal codes differ in length, the longer of the two is truncated using a modulus and a subtraction operation. The octal codes can then be tested for equivalence using an integer comparison. Using the previous example of octal codes in integer form, octal code 12345 would be transformed to 12300 so that it can be compared to octal code 12600 using an integer comparison.

**Pre-order Traversal of a Linear Octree**

The feature of linear octrees that makes the proposed algorithm efficient is the property of octal codes that, when sorted in ascending order, result in a list that defines a pre-order traversal of an octree [28]. This feature is used to merge the linear octree representations of multiple objects into a single space by concatenating the linear octree representations of multiple objects and sorting the resulting list of octal codes.

The sorted list of octal codes is used to determine all colliding pairs of objects using the `CommonPath` function. This is done by testing each octal code for a common path with subsequent octal codes from the list until an octal code representing a non-overlapping region of space is tested. At this point, no additional octal codes need to be tested for a common path because the region of space represented by the octal code does not overlap any additional regions of space represented in the list of octal codes.

To understand why this works, recall that within an octree each octant is a fully enclosed subdivision of a higher level octant. Therefore, only parent and child octants that lie on the same path along a branch of the octree overlap. Because octal codes represent paths along octree branches, octal codes that represent different paths indicate non-overlapping regions of space. As a result of sorting octal codes in ascending order, octal codes representing overlapping regions of space appear contiguously and in order within the list such that higher level octants intersecting lower level octants appear first. For example, in the sequence of octal codes `{1200, 1210, 1214, 1330, ...}`, `1200` identifies the parent of both `1210` and `1214` and overlaps both of their spaces. Octal code `1210` is the parent of `1214` and overlaps its space. Octal code `1330` defines a region of space that doesn't overlap any of the previous spaces and indicates that the previous spaces do not overlap any other regions of space represented in the list of octal codes. This property allows the search for overlapping octants to be performed in the fewest number of steps. As a result of the ordering, once two octal codes identifying non-overlapping regions of space have been tested, no further comparisons with subsequent octal codes will reveal any additional overlapping regions of space.

### 3.4.3 Analysis of the Algorithm

On the master processor, steps 1 and 5 of the algorithm run in linear time as a function of the data distributed to, and collected from, the slave processes. Because every object is passed to the search subroutine in step 2 to build the linear octree for each object, each search time becomes a constant factor. Therefore, searching for $n$ objects runs in linear time as a function of the number of objects. In Step 3, each slave sorts $n$ resultant octal codes which takes $\mathcal{O}((n\ log\ n))$ time in the worst case. In step 6, merging $k$ lists of sorted octal codes is done in linear time. Finally, step 7 runs in $\mathcal{O}(n^2)$ time in the worst case, but, if objects are not allowed to remain in an intersecting state, then the running time is $\mathcal{O}(n)$ on average as a result of the pre-order traversal ordering of the list.

Based on the individual run times of each step, the runtime performance of this algorithm will be bound by the sort performed in Step 3 by each slave processor. Therefore, the expected runtime of this algorithm is $\mathcal{O}((n\ log\ n)/k)$.

# Chapter 4

# Parallel Collision Detection System Implementation

This chapter discusses the implementation of a collision detection system that uses the new algorithm proposed in the previous chapter. In this chapter, the architectural approach and design used in the implementation will be described along with the components that constitute the collision detection system.

## 4.1 Client-Server Architecture, Object-Oriented Design, Modular Implementation

The structure of the collision detection system is a client-server architecture. The application serves as the client, and the collision detection system acts as the server. A client-server approach is used to separate logically and physically the application from the collision detection.

Logical separation of the collision detection system and the application is achieved by using object-oriented design techniques to encapsulate the data and functionality of the collision detection system in a collection of reusable components. A feature of this design is that it allows the collision detection system to be used by client applications with varying degrees of integration.

As part of the design of the collision detection system, parallel aspects of the underlying algorithm are implemented as separate processes. This feature requires the collision detection system to be separated physically from the client application and was made possible by the underlying object-oriented design. Additionally, the physical separation of the collision detection system from the application allows the collision detection system to take advantage of both shared-memory, multi-processor computers and a cluster of networked computers.

## 4.2 Operating Environment

The collision detection system has been implemented for use on Unix operating systems using the C++ programming language. Interprocess communication between components of the collision detection system on a single computer is implemented with System V semaphores and shared memory. Berkeley sockets are used for communicating over a network connection.

## 4.3 Integrating Collision Detection into an Application

As part of the object-oriented design of the collision detection system, application developers may choose from three levels of integration. This choice is made by selecting certain components of the collision detection system to build into an application.

When the highest level of integration is used, the entire collision detection system is run in a process owned by the client application. In this configuration, the client application has the highest degree of control over initializing data structures internal to the collision detection system. The drawback to fully integrating the collision detection system is that the client application cannot take advantage of the parallel features of the collision detection system.

A lower level of integration removes the task of building linear octrees from a client-application-owned process and performs this task in processes owned by the collision detection system. The task of merging linear octrees is still performed by a client- application-owned process using components from the collision detection system. Components from the collision detection system that run in a client-application-owned process communicate with processes owned by the collision detection system using shared memory and semaphores. When this configuration is used, an application can begin to take advantage of the parallel features of the collision detection system.

By using the lowest level of integration, all tasks performed by the collision detection system are removed from client-application-owned processes. In this configuration, an application can take full advantage of the parallel features of the collision detection system as well as communicate asynchronously with the collision detection system. Asynchronous communication with the collision detection system allows a client application to submit a job to the collision detection system and continue processing. At a later time, the client application can query the collision detection system to see if a previously submitted job is complete.

There are two reasons for running the collision detection system as part of a client-application-owned process:

1. The target platform is a single-processor computer.

   In the collision detection system, multiple processes are used to facilitate concurrent execution in parallel computing environments. If the client application is not run on a parallel computer, then no performance advantage will be gained by using the parallel features of the collision detection system.

2. The number of objects the client application manages is small.

   The parallel features of the collision detection system are designed to enable client applications to exceed the number of objects other collision detection systems can handle. When the number of objects is small, there is no benefit to using the parallel features of this collision detection system.

## 4.4 Class Objects

The following subsections describe the core components of the collision detection system and their role in performing collision detection.

### 4.4.1 The Octree class

The **Octree** class is used to encapsulate an octree data structure and its associated algorithms. Important issues this class addresses are the storage of an octree data structure in shared memory and octree search algorithms.

#### Shared-Memory Octree Data Structure

On multi-processor, shared-memory computers, the parallel implementation of the collision detection system uses several processes to concurrently build linear octrees. Because the octree data structure is used as a static reference frame for positioning objects within a volume of space, it is not necessary for each processor to share the same octree data structure in memory. Searching identical octrees for the same object always produces identical results. However, because the size of an octree grows exponentially with its depth, it is advantageous for all processes on the same machine to share a single octree data structure in memory. The **Octree** class addresses this issue by storing its internal octree data structure in shared memory.

The main technical issue of sharing an octree between multiple processes is how to represent the octree data structure in shared memory. As described in Chapter 2, octree data structures are typically implemented as pointer-based trees. However, a pointer-based tree does not lend itself to being stored in shared memory because dynamically allocated nodes are used to construct the tree and are referenced using pointers. Typically, the memory sharing facilities that come with Unix operating systems are designed to map a small number of memory regions into address spaces of multiple processes, not to share many small blocks of dynamically allocated memory.

To overcome this limitation, the octree data structure is constructed from an array of octants that is stored in a single region of shared memory. An array-based octree can be conveniently shared among many processes.

In an array-based octree, the hierarchical relationship between octants is established by storing array indices with each octant. Array indices are used by octants to locate the position of their parent and child octants in the array. In array-based octrees, array indices are analogous to pointers in pointer-based trees. An example of an array-based octree is depicted in Figure 4.1. The **NamedVector** class, which is described in section 4.5.4, is used to store an array of octants in shared memory.

#### Octant class

The **Octant** class implements the node data structure used in an octree data structure. An instance of this class corresponds to an octant in an octree. Internally, this class stores a **Cube** class to represent the volume of an octant and array indices that identify the location of parent and child octants in an array-based octree.

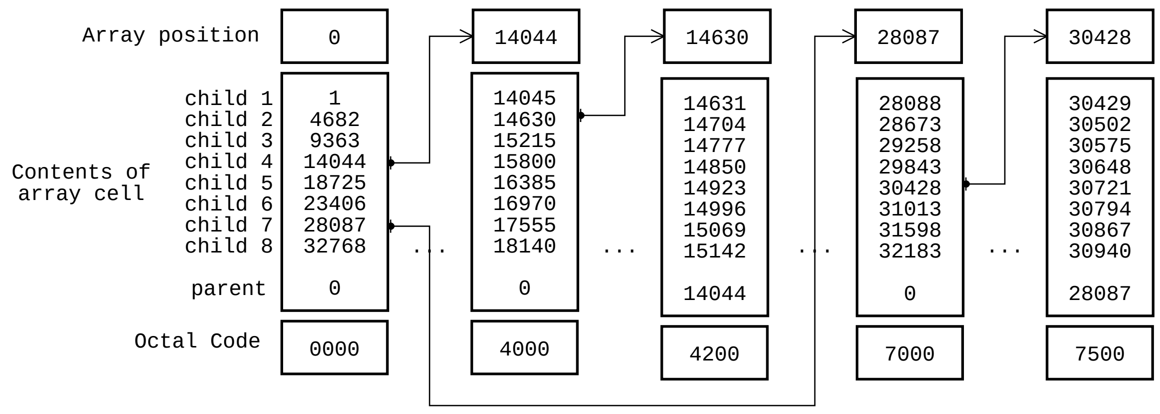


Figure 4.1: *Portions from an array-based octree demonstrating how to traverse from parent to child octant using the array indices stored in each array cell.*

**Octree::iterator class**

To traverse an array-based octree, indices stored with octants are used to navigate the array to parent and child octants according to the structure of the octree. The **Octree::iterator** class simplifies the process of traversing an array-based octree by encapsulating the mechanics of inspecting and storing array indices. This class is based on a container-iterator design pattern and is designed to work like the iterator class in the C++ Standard Template Library. Readers unfamiliar with iterators can think of them as intelligent pointers used for navigating data containers. In this case, the container is an array, and the data are **Octant** class instances. Because the natural structure of an octree is not a linear array, the iterator class is used to abstract client code from the array storage container and make it appear more like an octree. The **Octree::iterator** class simplifies writing octree algorithms.

**Octree::ResolveObject()**

The **Octree::ResolveObject()** function implements a recursive search of the octree that finds the octants which bind an object. This function is used to build the linear octree representation of an object. The parameters of the **Octree::ResolveObject()**

function are a **Cube** class, a maximum search depth, and a list of **OctalCode** classes. The **Octree::ResolveObject()** function searches the octree data structure stored in the **Octree** class instance for octants that intersect a cube. The resulting octal codes are stored in the **OctalCode** list parameter. These octal codes are the linear octree representation of the **Cube**. The **OctalCode** list parameter is a value-result argument and is used to pass the linear octree out of this function.

### 4.4.2 OctreeMaster Class

The **OctreeMaster** class is the intermediate interface between client applications and processes that perform octree searches. The primary responsibilities of the **OctreeMaster** class include assigning objects to slaves, signaling slaves to begin octree searches, and merging search results. Moreover, this class acts as the interface to processes that build linear octree representations of objects.

#### Asynchronous Communication with Client Application

One of the features implemented by the **OctreeMaster** class is an asynchronous communication method for client applications. Using this feature, client applications are able to submit jobs to the collision detection system and continue running while the job is being processed. Clients can then verify that the job has been completed through the **OctreeMaster** class. Alternatively, the client application can wait for the job to complete. While waiting, execution of the client application is suspended.

#### Object transfer and storage

Object data are transfered to and from an **OctreeMaster** class instance using shared memory. In most cases, the **NamedVector** class is used for this purpose. To transfer data to an instance of an **OctreeMaster** class, a process constructs a **NamedVector** class that attaches to the shared memory of a **NamedVector** class owned by an **OctreeMaster** class. The process and the **OctreeMaster** can read and write the same region of shared memory using their respective instances of a **NamedVector** class. Reading and writing operations are synchronized using a **NamedSemaphore** class.

### 4.4.3 OctreeClient Class

The **OctreeClient** class is the interface used by client applications to communicate with the collision detection system. The **OctreeClient** class provides an interface for clients to send and receive signals and data with the collision detection system.

This class provides three functions to communicate with the collision detection system.

- **OctreeClient::ResolveObjects():** The **ResolveObjects()** function is used to transmit objects to the collision detection system and signal the **OctreeMaster** to begin octree searches. This function returns immediately.

- **OctreeClient::Finished():** The **Finished()** function is used to determine if a job submitted using the **ResolveObjects()** function has been completed.

- **OctreeClient::Wait():** The **Wait()** function is used to wait for a job that was submitted using the **ResolveObjects()** function. This function blocks the process that invokes it until the requested job has been completed and data is ready to be delivered to the client application.

Internally, this class uses **NamedVector** and **NamedSemaphore** classes to transport data and communicate with the collision detection system.

### 4.4.4 NamedVector Class

The **NamedVector** class is a template class data structure designed to facilitate sharing memory between processes. This class is implemented as part of the class library for the collision detection system. The interface of the **NamedVector** class is designed to simulate the C++ Standard Template Library vector class. Data is stored internally in a shared-memory region. When a **NamedVector** class is constructed, the class creates or attaches itself to a shared memory region. Upon construction, if the shared memory region the class instance should use exists, then the class instance attaches to the memory, otherwise the memory is allocated. The **NamedVector** class uses a mnemonic to identify the shared-memory region with which to associate.

### 4.4.5 NamedSemaphore Class

Throughout the collision detection system, processes and shared memory must be synchronized. To facilitate synchronization operations, the **NamedSemaphore** class is implemented as part of the class library of the collision detection system. The **NamedSemaphore** class encapsulates operations on System V semaphores. Like the **NamedVector** class, upon initialization, the **NamedSemaphore** class uses a mnemonic to determine with which semaphore to identify.

## 4.5 Service Components

To implement the previously described classes as components running in separate processes, harness programs are used to instantiate classes and initialize communication facilities. The resulting programs function as service components in the collision detection system.

The collision detection system consists of four services excluding the the client application: **octree_master**, **octree_slave**, **octree_netclient**, and **octree_netserver**.

### 4.5.1 Octree_master Service

The **octree_master** service is the harness for the **OctreeMaster** class. This service interfaces with an instance of the **OctreeClient** class to forward data and signals to an instance of the **OctreeMaster** class. The **octree_master** service then drives the **OctreeMaster** class by invoking its functions to communicate with the **octree_slave** service.

### 4.5.2 Octree_slave Service

The **octree_slave** service is the harness for the **Octree** class. The **octree_slave** service receives signals and data from an instance of the **OctreeMaster** class and, in turn, drives the **Octree** class to perform octree searches. The **OctreeMaster** class is capable of interfacing with multiple **octree_slave** service processes. Multiple **octree_slave** service processes running on the same computer are used to achieve parallelism on a multi-processor computer.

### 4.5.3 Class-service Relationships

The relationship between the **OctreeClient** class, **octree_master** service, **OctreeMaster** class, **octree_slave**, and **Octree** class is depicted in Figure 4.2. Because the roles of the service components and the classes with which they communicate are isolated from one another, application developers have the option of choosing the level in which the collision detection system is integrated into the client application. As show in Figure 4.2, the **OctreeClient** class, **OctreeMaster** class, and **Octree** class all expose an interface called **ResolveObject()**. Functionally, the **Re-**

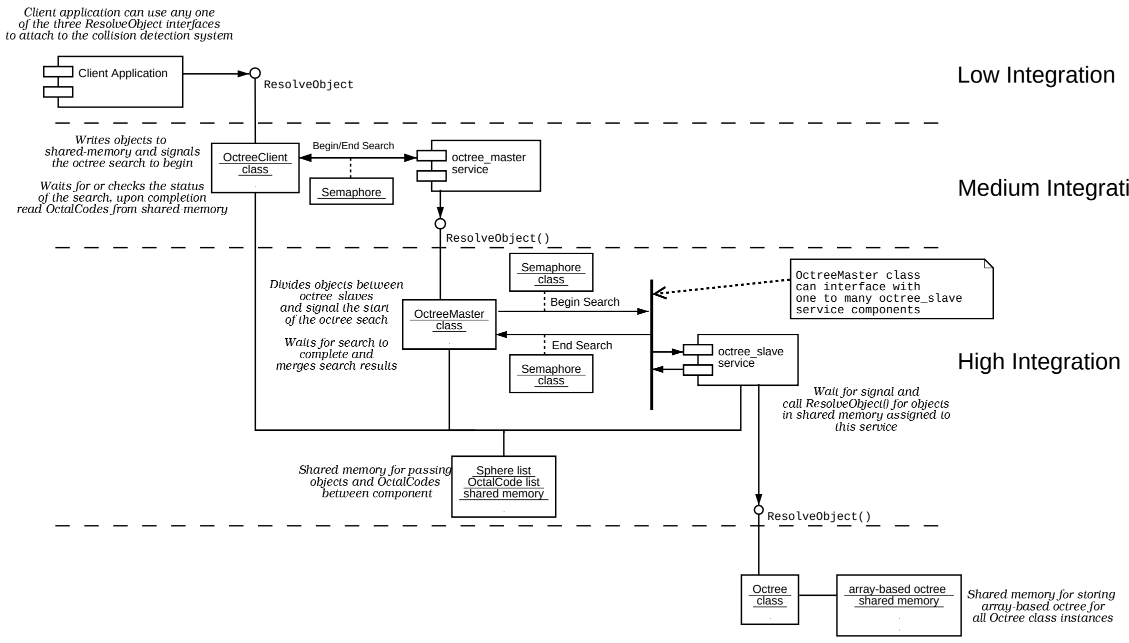

Client application can use any one of the three ResolveObject interfaces to attach to the collision detection system
Client Application
ResolveObject
Low Integration
Writes objects to shared-memory and signals the octree search to begin
Waits for or checks the status of the search, upon completion read OctalCodes from shared-memory
OctreeClient class
Begin/End Search
Semaphore
octree_master service
Medium Integration
ResolveObject()
Divides objects between octree_slaves and signal the start of the octree seach
Waits for search to complete and merges search results
OctreeMaster class
Semaphore class
Begin Search
End Search
Semaphore class
OctreeMaster class can interface with one to many octree_slave service components
octree_slave service
High Integration
Wait for signal and call ResolveObject() for objects in shared memory assigned to this service
Shared memory for passing objects and OctalCodes between component
Sphere list OctalCode list shared memory
ResolveObject()
Octree class
array-based octree shared memory
Shared memory for storing array-based octree for all Octree class instances

Figure 4.2: *Instance diagram of the collision detection system.*

**solveObject()** interface works approximately the same between all three classes that implement it. This allows the client application to be attached to the collision detection system using any one of the three **ResolveObject()** interfaces. By choosing the **ResolveObject()** interface to attach to, the application developer chooses the level in which the collision detection system is integrated into the client application.

### 4.5.4 Octree_netclient and Octree_netserver Services

The **octree_netclient** and **octree_netserver** services are the glue that enables the collision detection system to distribute the task of building linear octrees over a network of computers. These services mimic the behavior of **octree_slave** and client applications. They operate by intercepting data and signals from client applications and **octree_slave** services and forward information across network connections.

The **octree_netclient** service implements the communication interface that the **octree_slave** service component uses to communicate with an instance of the **Oc-**

**treeMaster** class. This allows the **octree_netclient** to receive signals and data from an **OctreeMaster** class. When an **OctreeMaster** initiates a search, an **octree_netclient** service receives the request and sends it across a network connection to a corresponding **octree_netserver**.

When an **octree_netserver** service receives a request from an **octree_netclient** service, the **octree_netserver** simulates the behavior of a client application and submits the search request to an **octree_master** service running on the same computer. The process is reversed to send results back to the **octree_master** service that originally requested the octree searches to be performed. The configuration of a collision detection system for parallel processing is shown in Figure 4.3.

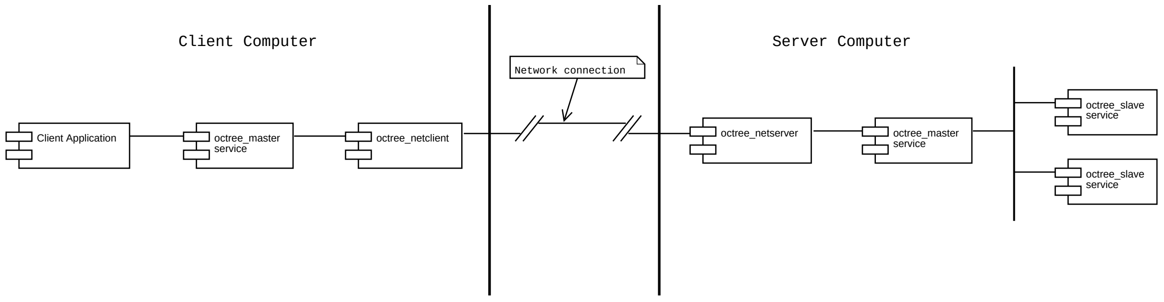


Figure 4.3: *Collision detection system utilizing network connections for distribution.*

## 4.5.5 Bandwidth Utilization

Both the **octree_netclient** and **octree_netserver** services have the ability to bind themselves to a specific network connection. This feature gives the collision detection system the ability to take full advantage of all available network bandwidth for moving data to and from networked computers. This feature is particularly useful when computers are multi-homed on a network. For each network interface, a corresponding **octree_netclient/octree_netserver** can be installed, allowing data transportation to be distributed over the interfaces.

Using **octree_netclient** and **octree_netserver** services in combination with the collision detection system services allows for a variety of configurations to be constructed that take full advantage of all available resources.

Examples of how the collision detection can be configured for maximum utilization of resources are shown in Figure 4.4.

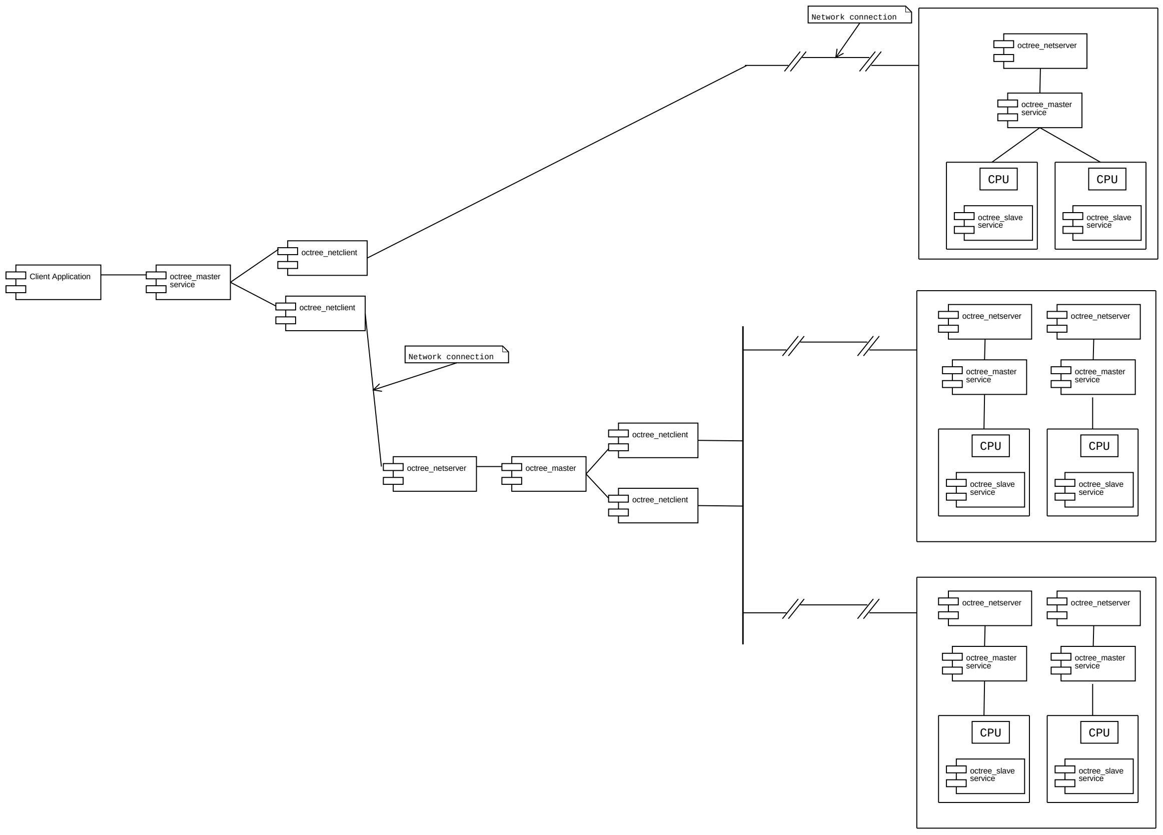


Figure 4.4: *Sample configurations of the collision detection system using both shared-memory and networked computers.*

# Chapter 5

# Results

This chapter presents results from our implementation of the collision detection algorithm. In the first section, empirical results obtained from running the collision detection system are described through a series of figures. In the second section, the actual performance of the algorithm is described using speedup and efficiency metrics. Finally, the last section presents an explanation of the performance of the algorithm by analyzing the figures presented in the first section of this chapter and describing the actual runtime of several components of the collision detection system.

## 5.1 Empirical Results

Results presented in this section were obtained by running the collision detection system on variable size data sets and recording the time taken to complete a collision detection operation. Two parallel computers where used to run the collision detection system. The first parallel computer consists of six dual processor AMD Athlon MP 1.2GHz computers networked together using a 100baseT network. Data in Figure 5.6 was obtained by running the collision detection system on the second parallel computer. This computer consists of twenty-one dual processor Intel 1GHz Pentium III computers interconnected with a high bandwidth, low latency network.

In each test, a five level octree measuring one thousand units across is used to perform collision detection on spherical objects, each with a radius of 0.5 units. The spheres are randomly positioned within the volume enclosed by the octree. The number of objects used in each test is indicated in the figure.

Figure 5.1 shows the results of running the collision detection system on a dual-processor computer using one and two processors. As shown by this figure, performing collision detection using two processors completes in approximately half the time of the same operation performed using a single processor.

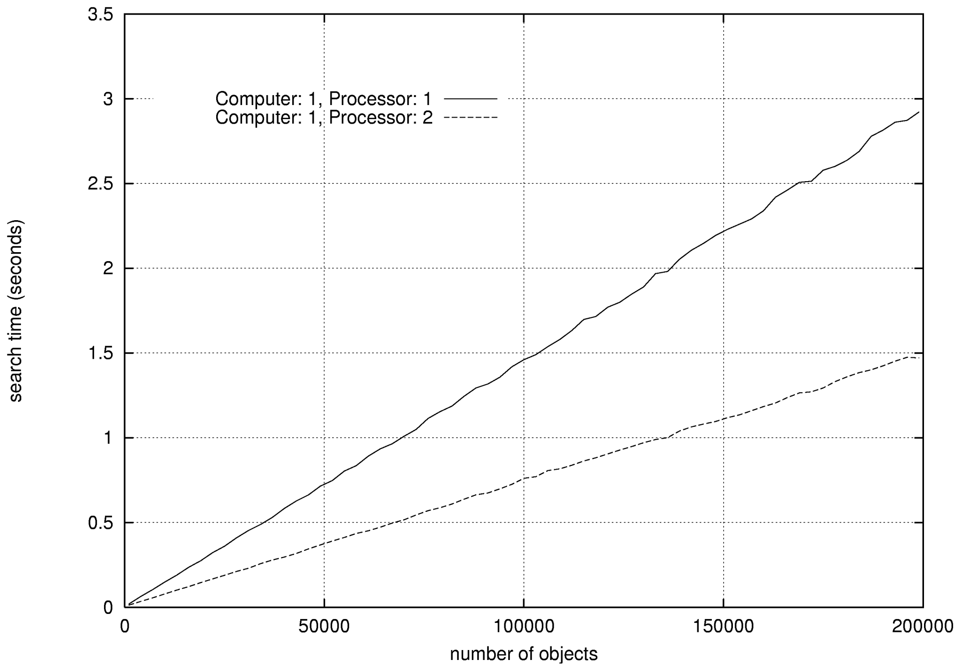


Figure 5.1: *Collision detection: local, 1 and 2 processors.*

In Figure 5.2, collision detection performed using a single processor located in the same computer and a remote computer are compared. In this figure, the distance between the line profiles represent the time needed to transport data to and from the remote computer. Figure 5.3 shows the same comparison but adds the use of a second processor.

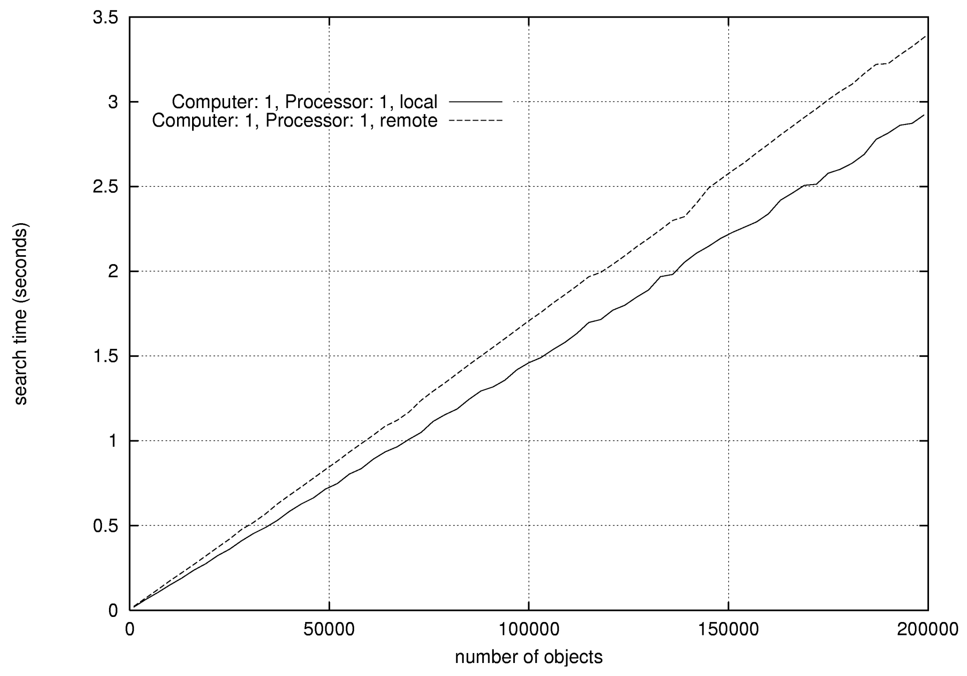


Figure 5.2: *Collision detection: local and remote, 1 processor.*

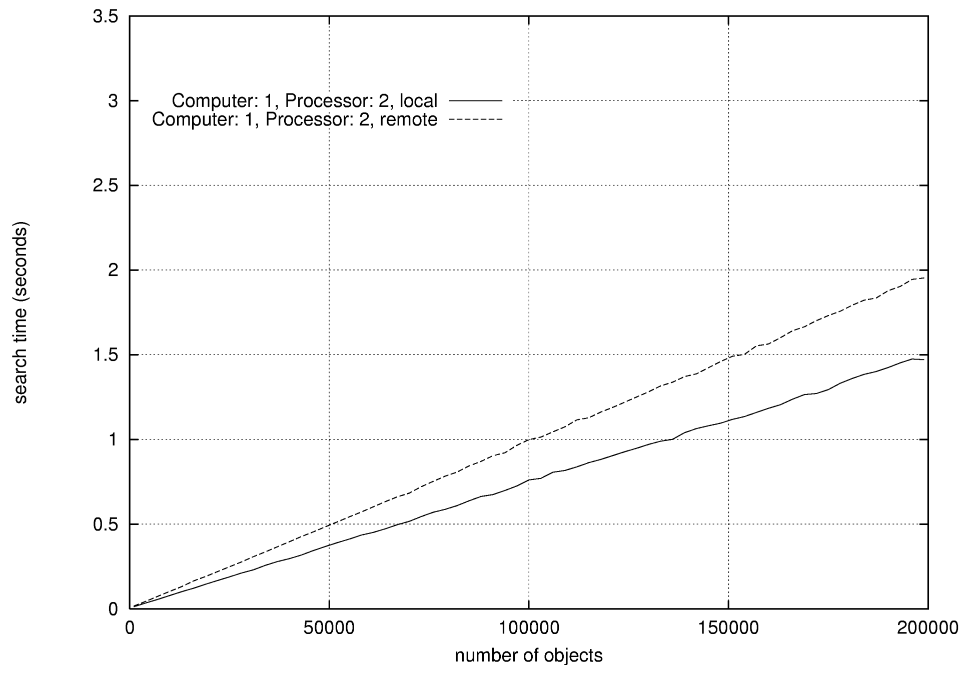


Figure 5.3: *Collision detection: local and remote, 2 processors.*

Figure 5.4 shows the scalability of the collision detection system by comparing the time needed to perform the same collision detection operation using one to five remote computers. Each computer uses only a single processor. Figure 5.5 shows the results of the same test performed in Figure 5.4 when each computer uses two processors.

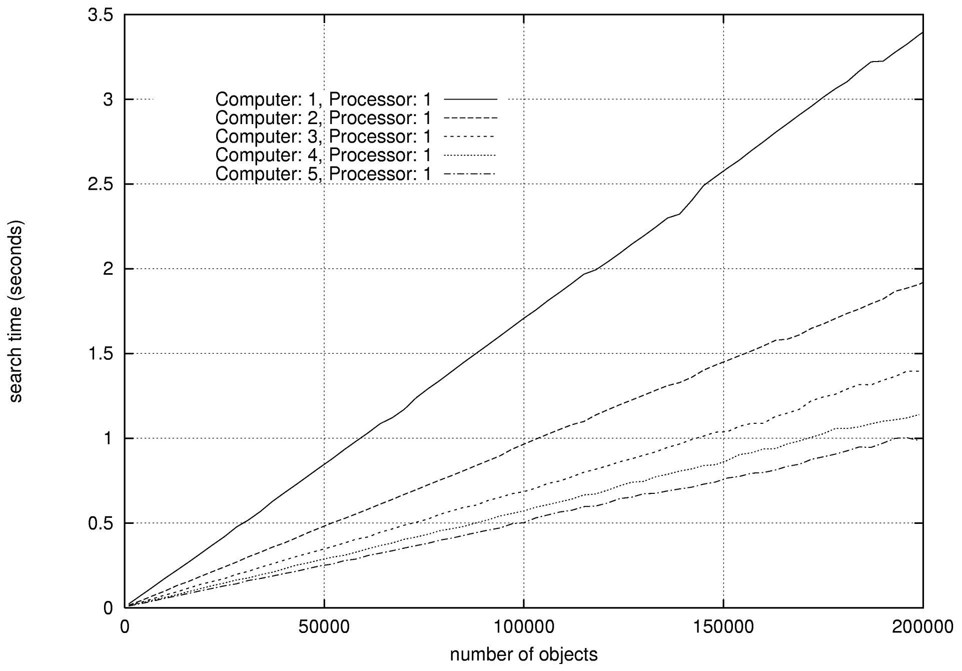


Figure 5.4: *Collision detection: remote, 1 processor per machine, 1 to 5 machines.*

Figure 5.6 shows the performance of the collision detection system by comparing execution times for 100,000 to 2,000,000 objects using one to forty processors. These computers are interconnected with a high bandwidth, low latency network. This network allows large amounts of data to be moved between computers at speeds in upward of eight times faster than a 100baseT Ethernet network. Besides helping show how scalable the collision detection algorithm is, the primary purpose of this graph is to provide supporting evidence to the claim that the collision detection algorithm runs in $\mathcal{O}((n\ log\ n)/k)$ time. Unfortunately, these graphs appear linear. An explanation of this phenomenon is presented in the last section of this chapter.

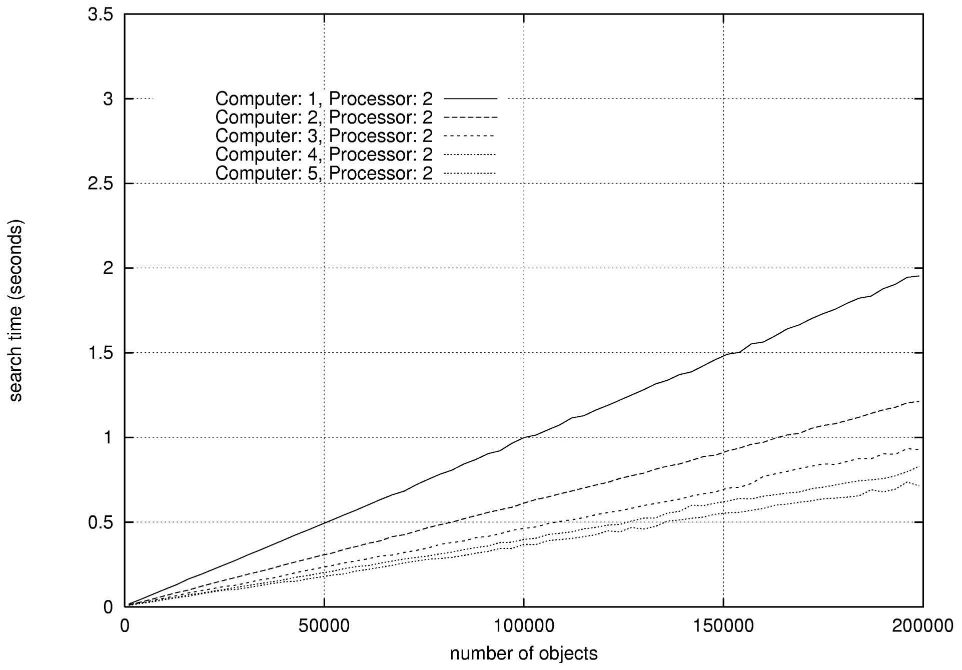


Figure 5.5: *Collision detection: remote, 2 processors per machine, 1 to 5 machines.*

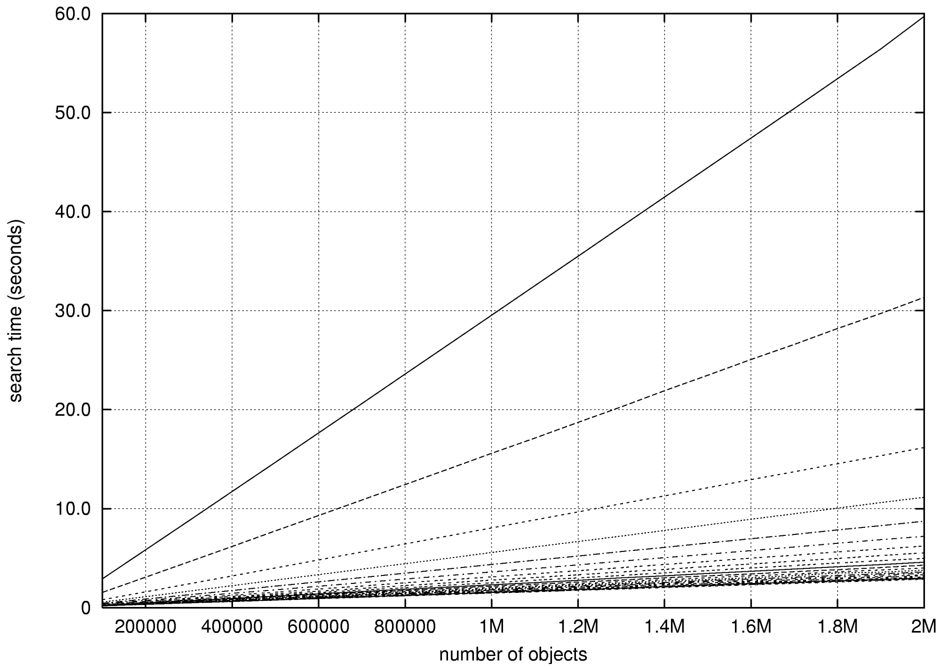


Figure 5.6: *Collision detection: remote, 2 processors per machine, 1 to 20 machines.*

## 5.2 Analysis of Empirical Results

The tables in this section describe the performance of the collision detection system using actual runtime data. Each table contains the speedup and efficiency of the collision detection system for a variable number of processors and data set sizes. In each case, speedup is calculated by dividing the runtime of the serial algorithm by the parallel runtime of the algorithm. Speedup indicates how many time faster the collision detection system performs using multiple processors compared to the serial implementation of the algorithm. Efficiency is calculated by dividing the speedup of the algorithm by the number of processors used to attain the speedup value. The efficiency of an algorithm is a percentage that indicates how much of the total processing capabilities of the parallel computer an algorithm can use to while executing.

In Table 5.1 and Table 5.2 speedup and efficiency are calculated for the collision detection system running on a dual processor computer with two different size data sets. As show in these tables, a larger data set causes the collision detection algorithm to perform more efficiently.

| Processor | Execution time ($\mu$sec.) | Speedup | Efficiency |
|---|---|---|---|
| 1 | 1,460,885 | NA | NA |
| 2 | 761,157 | 1.919 | 95.95% |

Table 5.1: *Collision detection: Local computer, 100,000 objects.*

| Processor | Execution time ($\mu$sec.) | Speedup | Efficiency |
|---|---|---|---|
| 1 | 2,921,999 | NA | NA |
| 2 | 1,471,116 | 1.986 | 99.31% |

Table 5.2: *Collision detection: Local computer, 199,000 objects.*

In Table 5.3, 5.4, 5.5, and 5.6, speedup and efficiency of the collision detection system are calculated using a variable number of processors. These tables demonstrate that the collision detection system is capable of scaling as the number of processors increases and still achieve speedup. However, as the number of processors increases, efficiency of the system decreases.

| Processor | Execution time ($\mu$sec.) | Speedup | Efficiency |
|---|---|---|---|
| 1 | 1,708,203 | NA | NA |
| 2 | 965,051 | 1.77 | 88.5% |
| 3 | 685,805 | 2.49 | 83.0% |
| 4 | 571,763 | 2.99 | 74.8% |
| 5 | 502,234 | 3.40 | 68.0% |

Table 5.3: *Collision detection: 1 processor per computer, 100,000 objects.*

| Processor | Execution time ($\mu$sec.) | Speedup | Efficiency |
|---|---|---|---|
| 1 | 3,379,940 | NA | NA |
| 2 | 1,908,001 | 1.77 | 88.5% |
| 3 | 1,395,906 | 2.42 | 80.7% |
| 4 | 1,140,158 | 2.96 | 74.1% |
| 5 | 986,878 | 3.42 | 68.5% |

Table 5.4: *Collision detection: 1 processor per computer, 199,000 objects.*

| Processors | Execution time ($\mu$sec.) | Speedup | Efficiency |
|---|---|---|---|
| 1 | 3,379,940 | NA | NA |
| 2 | 1,953,981 | 1.71 | 85.5% |
| 4 | 1,212,657 | 2.79 | 69.8% |
| 6 | 928,884 | 3.69 | 61.5% |
| 8 | 828,263 | 4.30 | 53.7% |
| 10 | 713,694 | 4.64 | 46.4% |

Table 5.5: *Collision detection: 2 processors per computer, 100,000 objects.*

| Processor | Execution time ($\mu$sec.) | Speedup | Efficiency |
|---|---|---|---|
| 1 | 3,379,940 | NA | NA |
| 2 | 1,953,981 | 1.73 | 86.5% |
| 4 | 1,212,657 | 2.79 | 69.7% |
| 6 | 928,884 | 3.64 | 60.6% |
| 8 | 828,263 | 4.08 | 51.0% |
| 10 | 713,694 | 4.74 | 47.4% |

Table 5.6: *Collision detection: 2 processors per computer, 199,000 objects.*

Tables 5.7 and 5.8 shows that as the number of processors increases the system can retain efficiency if the amount of work (the number of objects) is increased.

| Processor | Execution time ($\mu$sec.) | Speedup | Efficiency |
|---|---|---|---|
| 1 | 29,537,415 | NA | NA |
| 2 | 15,596,299 | 1.89 | 94.7% |
| 10 | 3,615,147 | 8.17 | 81.7% |
| 20 | 2,177,746 | 13.56 | 67.8% |
| 30 | 1,682,112 | 17.56 | 58.5% |
| 40 | 1,480,608 | 19.95 | 49.9% |

Table 5.7: *Collision detection: 2 processors per computer, 1,000,000 objects.*

| Processors | Execution time ($\mu$sec.) | Speedup | Efficiency |
|---|---|---|---|
| 1 | 59,706,876 | NA | NA |
| 2 | 31,332,310 | 1.91 | 95.3% |
| 10 | 7,215,860 | 8.27 | 82.7% |
| 20 | 4,298,207 | 13.89 | 69.5% |
| 30 | 3,331,193 | 17.92 | 59.7% |
| 40 | 2,889,864 | 20.66 | 51.7% |

Table 5.8: *Collision detection: 2 processors per computer, 2,000,000 objects.*

## 5.3 Analysis of Algorithm Runtime

One of the goals of obtaining empirical data of the runtime of this algorithm was to provide supporting evidence to the claim that it performs in $\mathcal{O}(n\ log\ n)/k$ time. Figure 5.6 was intended to do this by producing data that when graphed, resulted in a curve similar to an $n\ log\ n$ curve. Unfortunately, none of the plotted lines in Figure 5.6 are curved. In fact, these lines appear linear.

To understand why these lines do not fit a characteristic $n\ log\ n$ curve, individual components that contribute to the runtime of the algorithm where timed and compared. Based on our findings, all operations performed internal to the collision detection system are performing as expected, except for the **ResolveObject()** routine, which is responsible for generating the octal codes for each object.

The following list of functions and times are the primary contributing factors that make up the runtime of the collision detection system.

- Slave computer: **qsort()** 88,953 $\mu$ sec.
- Slave computer: **ResolveObject()** 1,307,631 $\mu$ sec.
- Networked slave computer: **MergeSlaveResults()** 38,622 $\mu$ sec.
- Master computer: **MergeSlaveResults()** 791,490 $\mu$ sec.

Obviously, the major contributing factor to the runtime of the collision detection algorithm is the **ResolveObject()** function. The runtime of the **ResolveObject()** function grows linearly with the number of objects it operates on and, because the runtime of the **ResolveObject()** function is an order of magnitude greater than the other runtimes, the effects other contributing components have on the overall runtime of the collision detection algorithm are diminished. This explains why the lines in Figure 5.6 appear so linear.

The second largest contributing factor in the list is the **MergeSlaveResults()** function performed by the master process. Although in this example, the runtime for the **MergeSlaveResults()** function appears large compared to the **qsort()** runtime, this is actually the effect of the number of processors being used in the collision detection system. In this example, forty slave processors are being used. If $k$ was smaller, or $n$ was sufficiently larger, then the runtime of the **qsort()** routine would dominate all other contributing factors and make the curves in Figure 5.6 appear more $n \, log \, n$.

# Chapter 6

# Conclusions and Future Work

In this thesis we presented a new collision detection algorithm that solves the all-pairs collision detection problem using parallel processing. The design of the algorithm is based on a linear octree and runs in parallel with a theoretical performance of $\mathcal{O}((n\ log\ n)/k)$ runtime. The algorithm has been implemented as a collision detection system using object-oriented design techniques and a client-server architecture. The architecture of the collision detection system is designed to use the parallel capabilities of both shared-memory, multi-processor computers and clusters of networked computers. Additionally, the modularity of the collision detection system gives application developers the flexibility to choose the level at which the collision detection system is integrated into an application. Using the collision detection system, experimental results have been generated that demonstrate how the algorithm performs according to the calculated theoretical performance.

## 6.1 Conclusions

Distributed octree data structures are well suited for collision detection when the data storage requirements of an application exceed the memory capacity of a single computer. In this case, a distributed octree data structure used in conjunction with a cluster of networked computers is an effective solution that will enable an application to meet the requirements of a memory-intensive application. However, distributed octree data structures are ill suited for implementing parallel collision detection algorithms that compete with the performance of existing collision detection algorithms. Issues of load balancing and object representation in the data structure prevent algorithms that use the distributed octree data structures from performing efficiently. Linear octrees are an efficient and robust method for representing objects in a collision detection algorithm.

## 6.2 Future Work

Many aspects of the algorithm and collision detection system developed in this thesis are the result of an evolutionary process in which experimental designs were explored. Successful ideas and techniques from these experiments became part of the final algorithm and collision detection system. A prime example of an evolutionary change the collision detection system underwent is the transition from a distributed octree to a linear octree used in the parallel collision detection algorithm. In the same way that the algorithm and collision detection system evolved into the current implementation, it can continue to evolve with further development. The following list outlines areas where the algorithm and collision detection system can be improved.

1. Faster octree searches

   In the collision detection system the algorithm used for finding the set of octants that enclose an object is a recursive search of an octree. The running time of the search is $\mathcal{O}(log\ n)$. An alternative algorithm could be used that directly calculates the octal code of the octant in which the center of an object resides.

The search for octants that enclose the object can then be restricted to the octants adjacent to the octant containing the center of the object. The formula for calculating the octal code of a three-dimensional coordinate is well known and runs in $\mathcal{O}(1)$ time. A description of this formula can be found in [30]. As a result of restricting the search to a constant number of octants, the algorithm for performing an octree search will run in $\mathcal{O}(1)$ time.

This search was not used because the original search algorithm developed for this project was designed for a distributed octree. At the stage of the project when the transition was made from using a distributed octree to a linear octree, the algorithm for performing an octree search was already well developed and debugged. This search works equally well for building linear octrees and inserting objects into a distributed octree. Another reason for not implementing the new algorithm when the transition was made was that it would not improve the algorithmic performance of the system as a whole. Because a linear octree is built for each object every time the collision detection system is invoked, the search routine that builds the linear octree becomes a constant factor in the collision detection algorithm. However, implementing this feature will improve the speed of the collision detection system.

2. Improved network performance

   When the collision detection system is run on a cluster of networked computers, performance is limited by the speed at which data can be transferred between computers. Two methods should be explored to reduce the amount of network traffic generated by the collision detection system in order to increase the throughput of network operations:

- **Data compression** By compressing data that is transferred across the network, the total number of bytes transferred is decreased. The choice of which compression scheme to use should be based on the total amount of time needed to compress, transfer, and decompress.
- **UDP transport protocol** Currently, the TCP protocol is used to move data across network connections reliably. In order to deliver data reliably, the TCP protocol incurs overhead which results in extra network traffic. Schemes have been developed to deliver data reliably using UDP, an unreliable protocol, and should be explored to determine if they are suitable for use in the collision detection system.

3. Parallel Merge Operation

   In Step 6 of the collision detection algorithm, the octal codes generated by the slave processes are sent to the master process to be merged. When four or more processors are being used and the number of objects is large, this step can possibly benefit from a parallel merge operation.

   To perform the merge in parallel, the data between pairs of processors can be merged until only one pair of processors with data remains. At this point, the two remaining processors send their data to the master processor where the final merge is performed. To demonstrate how this algorithm works consider eight processors, each with a list of octal codes, and one master processor. In the first stage of merging, process 1 sends its list to processor 2, processor 3 sends its list to processor 4, processor 5 sends to 6, and 7 sends to 8. Processors 2, 4, 6, and 8 then merge the lists they are storing. In the second stage of the merge operation, processor 2 sends its list to processor 4 and processor 6 sends its list to processor 8. Processors 4 and 8 then merge their lists, and the results are sent to the master processor for the final merge. This algorithm can be expanded to additional levels as well as work with an uneven number of processors.

This algorithm incurs a logarithmic increase in the amount of time spent performing communication operations as opposed to all slaves sending their data to the master processor in one step. However, the benefit of performing merges in parallel is that the actual time spent merging ordered lists on the master process will be decreased.

4. Exploit spatial and temporal coherence

The possibility of exploiting coherence to improve the algorithmic performance from $\mathcal{O}(n \ log \ n)$ to $\mathcal{O}(n)$ may exist. As a result of coherence, if objects move a sufficiently small distance between invocations of the collision detection algorithm, the number of objects that change position in the octree between time steps will be small. Therefore, octal codes in the linear octree and the sorted order of the octal codes will remain approximately constant between time steps. This situation can be exploited in step 3 of the algorithm when each slave sorts its list of octal codes.

Because the collision detection algorithm partitions and distributes the list of objects to slave processes in the same order on each invocation of the algorithm, the list of objects a slave receives and the order of the octal codes the slave generates will remain approximately constant as a result of coherence. Therefore, if the slave process caches the order of the octal codes it generates between invocations, the slave will be able to construct a nearly sorted list of octal codes upon each invocation. The list of octal codes can then be sorted completely using an insertion sort which, when run on a nearly sorted list, runs in $\mathcal{O}(n)$ time on average.

In the current implementation of the collision detection system, the algorithmic performance is bound to the runtime of the sort performed by the slave processes, which run in $\mathcal{O}(n\ log\ n)$ time. If this modification to the algorithm does in fact work, then the performance of the collision detection algorithm will be improved to $\mathcal{O}(n/k)$.

# Bibliography

[1] Gill Barequet, Bernard Chazelle, Leonidas J. Guibas, Joseph S. B. Mitchell, and Ayellet Tal. BOXTREE: A hierarchical representation for surfaces in 3D. *Computer Graphics Forum*, 15(3):387–396, 1996.

[2] J.L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18:509–517, 1975.

[3] P. Brunet and I. Navazo. Solid representation and operation using extended octrees. *ACM Transactions on Graphics*, 9:170–197, 1990.

[4] Jonathan D. Cohen, Ming C. Lin, Dinesh Manocha, and Madhav K. Ponamgi. I-COLLIDE: An interactive and exact collision detection system for large-scale environments. In *Symposium on Interactive 3D Graphics*, pages 189–196, 218, 1995.

[5] D. Ennis, B. Lucchesi, N. Oberlander, K. Wesolowski, F. C. Harris Jr., and P. Mousset-Jones. Surface mine truck safety training: A vr approach to pre-operational vehicle inspection. In Kadri Dagdelen, editor, *Proceedings of APCOM'99: Computer Applications in the Mineral Industries 28th International Symposium*, pages 811–818. Colorado School of Mines, October 20-22 1999.

[6] David Eppstein and Jeff Erickson. Raising roofs, crashing cycles, and playing pool: applications of a data structure for finding pairwise interactions. *Discrete & Computational Geometry*, 22(4):569–592, 1999. Special issue of invited papers from the 14th Annual ACM Symposium on Computational Geometry.

[7] Lori A. Freitag and Raymond M. Loy. Adaptive, multiresolution visualization of large data sets using a distributed memory octree. In Cherri Pancake, editor, *Proceedings of SC99: High Performance Networking and Computing*, Portland, OR, November 13-19 1999. ACM Press and IEEE Computer Society Press.

[8] Fabio Ganovelli, John Dingliana, and Carol O'Sullivan. Buckettree: Improving collision detection between deformable objects. In *Proceedings of SCCG2000: Spring Conference on Computer Graphics*, Budmerice Castle, Bratislava (SK), May 4-6 2000.

[9] I. Gargantini and J-Y Liu. Finding the voxels shared by a ray and a linear octree. In *Proc. Sixth Inter. Coll. Numer. Anal. and Comp. Sci.*, pages 179–185, Aug 1997.

[10] E. Gilbert, D. Johnson, and S. Keerth. A fast proceedure for computing the distance between complex objects in three-dimensional space. In *IEEE International Conference on Robotics and Automation*, pages 193–203, April 1988.

[11] S. Gottschalk, M. C. Lin, and D. Manocha. OBBTree: A hierarchical structure for rapid interference detection. *Computer Graphics*, 30(Annual Conference Series):171–180, 1996. this is the Proc. ACM SIGGRAPH, August 1996.

[12] Taosong He and Arie Kaufman. Collision detection for volumetric objects. In *IEEE Visualization '97*, Oct 1997.

[13] M. Held, J. Klosowski, and J. Mitchell. Evaluation of collision detection methods for virtual reality fly-throughs. In *Proceedings Seventh Canadian Conference on Computational Geometry*, pages 205–210, Qu'ebec City, Qu'ebec, Canada, August 10-13 1995.

[14] Philip M. Hubbard. Approximating polyhedra with spheres for time-critical collision detection. *ACM Transactions on Graphics*, 15(3):179–210, 1996.

[15] Thomas C. Hudson, Ming C. Lin, Jonathan Cohen, Stefan Gottschalk, and Dinesh Manocha. V-COLLIDE: Accelerated collision detection for VRML. In Rikk Carey and Paul Strauss, editors, *VRML 97: Second Symposium on the Virtual Reality Modeling Language*, New York City, NY, Feb 1997. ACM SIGGRAPH / ACM SIGCOMM, ACM Press. ISBN 0-89791-886-x.

[16] Yaohong D. Jiang. Set operations between linear octrees. *Computers & Geosciences*, 22(5):509–516, 1996.

[17] P. Jiménez, F. Thomas, and C. Torras. 3d collision detection: A survey. *Computers and Graphics*, 25(2):269–285, April 2001.

[18] G. Vanecek Jr. BRep-Index: A Multidimensional Space Partitioning Tree. *Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 35–44, 1991.

[19] D. Kim, L. Guibas, and S. Shin. Fast collision detection among multiple moving spheres. In *Proceedings of the 13th International Annual Symposium on Computational Geometry (SCG-97)*, pages 373–375, New York, June 4–6 1997. ACM Press.

[20] Yoshifumi Kitamura, Andrew Smith, Haruo Takemura, and Fumio Kishino. Parallel algorithms for real-time colliding face detection. *Presence*, 7(1):36–52, 1998. MIT Press.

[21] James T. Klosowski, Martin Held, Joseph S. B. Mitchell, Henry Sowizral, and Karel Zikan. Efficient collision detection using bounding volume hierarchies of k-DOPs. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):21–36, 1998.

[22] M. Lin, D. Manocha, and J. Cohen. Collision detection: Algorithms and applications. In *Proceedings of the 2nd Workshop on Algorithmic Foundations of Robotics*, 1996.

[23] M. C. Lin and J. F. Canny. A fast algorithm for incremental distance calculation. In *IEEE International Conference on Robotics and Automation*, pages 1008–1014, 1991.

[24] Yueh-Jaw Lin and Prabakar Murugappan. A new algorithm for determining a collision-free path for a cmm probe. *International Journal of Machine Tools and Manufacture design, Research and Applications*, 39:1397–1408, 1999.

[25] B. Lucchesi, N. Oberlander, F. C. Harris Jr., and P. Mousset-Jones. Surface mine truck safety training: Scenario setup for a vr driving simulator. In Q. Yang, editor, *Proceedings CAINE '99: the 12th International Conference on Computer Applications in Industry and Engineering*, pages 62–65, Atlanta, GA, November 4-6 1999. ISCA.

[26] Kwan-Liu Ma and Thomas W. Crockett. Parallel visualization of large-scale aerodynamics calculations: A case study on the cray T3E. Technical Report TR-99-41, ICASE, NASA Langley Research Center, 1999.

[27] N. Prewitt, D. Belk, and W. Shyy. Parallel computing of overset grids for aerodynamic problems with moving objects. *Progress in Aerospace Sciences*, 36:117–172, 2000.

[28] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, June 1984.

[29] Hanan Samet. Hierarchical representations of collections of small rectangles. *ACM Computing Surveys*, 20:271–309, 1988.

[30] B. C. Vemuri, Y. Cao, and L. Chen. Fast collision detection algorithms with applications to particle flow. *Computer Graphics Forum*, 17(2):121–134, 1998. ISSN 1067-7055.

[31] H. Wan, Z. Fan, A. Gao, and Q. Peng. A parallel collision detection algorithm based on hybrid bounding volume hierarchy. *CAD/Graphics' 2001*, 2001.

[32] A. Watt, F. Policarpo, P. Computacao, and R. Janeiro. *3D Games Real-time Rendering and Software Technology.* Addison Wesley, 2001.

[33] J. Wilhelms and A. Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11(3):201–227, July 1992.

[34] G. Zachmann. Optimizing the collision detection pipeline. In *Proc. of the First International Game Technology Conference (GTEC)*, January 2001.