Spatial Reasoning for Generalized N-Body Physics - Discrete Element Algorithms

by

Eric David Perkins

Sc.B., Engineering (1997)
Brown University

Submitted to the Department of Civil and Environmental Engineering
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Civil and Environmental Engineering

at the

Massachusetts Institute of Technology

May 1999

Signature of Author [illegible]
Department of Civil and Environmental Engineering
May 7, 1999

Certified by [illegible]
Professor John R. Williams
Thesis Supervisor
Professor of Civil and Environmental Engineering

Accepted By [illegible]
Andrew J. Whittle
Chairman, Departmental Committee on Graduate Studies

Spatial Reasoning for Generalized N-Body Physics - Discrete Element Algorithms

by

Eric David Perkins

Submitted to the Department of Civil and Environmental Engineering
on May 5, 1999 in partial fullfillment of the requirements
for the Degree of Master of Science in Civil and Environmental Engineering

ABSTRACT

Discrete element simulation solves Newton's dynamic equilibrium equations for a large set of constantly moving objects. In order to simulate the interaction of these objects, it is necessary, at every timestep, to find and resolve their spatial relationships. For large simulations this process is very computationally intensive, and dominates the cost of the whole simulation. It is desirable, therefore, to develop algorithms for processing this spatial intersection query as efficiently as possible. Many methods have been developed recently, both specifically for discrete element contact detection, and for spatial queries in general. Here we develop a general statement of the Discrete element contact problem, and describe an interface to a generalized contact detection module. A range of methods is then discussed and implemented. From these reults we develop improved methods that attempt to maximize performance over a wide range of problems. Finally, the implementations are compared using the MIMES 2D discrete element simulation environment.

Thesis Supervisor: Prof. John R. Williams
Title: Professor of Civil and Environmental Engineering

Acknowledgments

I would like to thank my advisor, Professor Williams for his advice and support with my research, and Dale Preece at Sandia National Laboratory for his enthusiasm, and support for my efforts. I would also like to thank my family and friends for their help and understanding.

1 Introduction

The Discrete Element Method simulates the behavior of material based on the simple interactions of many distinct elements or particles. The method is especially powerful is the study of granular materials (9), where specialized physics is required to treat the material as a continuum, and the analysis becomes very complex.

On the particle level, the individual interactions are fairly simple. Figure 1 shows a small portion of a discrete element simulation. Contacts between particles give rise to reaction forces, the system evolves, and the contacts between particles are recalculated. The complexity of the simulation arises, instead, from the number of particles and the globally dynamic nature of their interactions; with N particles, there are N squared possible contacts, where N can range from 1,000 to 1,000,000 particles, or more for very large simulations.

At any given timestep, many of the contacts are be broken, and many new contacts are added. It is therefore, necessary at every timestep, to reduce the N squared set of possible contacts to the exact set of contacts. This is a globally complex process involving the evaluation of N squared possibilities. Depending on geometric representation, it can be locally complex as well[1]. It is this process, its definition, development and optimization, which will form the focus of this paper.

[1] More general or complex geometric representations such as polygonal or polyhedral

We will define, more formally, the process of contact detection, and develop an implementation independent interface to a generalized contact detection module. Using this model, and proceeding from basic geometric concepts, we will develop a variety of interface-compatible implementations of current contact-detection methods. Drawing on the development of these methods, we will propose some improved methods. Finally we will run some benchmark problems using the interface-compatible modules in a 2D discrete element simulation system.

2 Motivation

The development of contact detection algorithms is motivated by the real-world need of discrete element simulations. The power of the discrete element method is its ability to work with many objects to simulate the behavior of discontinuous systems, such as granular materials. Discrete element simulation proceeds in a cyclic time-stepping manner. The steps of the cycle are shown in figure 2. At the beginning of the simulation, the object geometry and initial conditions are specified. The first step of the simulation cycle is contact detection. The process of contact detection is performed in two steps, first global, then local contact resolution. In the first phase, the set of all possible contacts is reduced to a set of likely contacts corresponding, roughly, to the geometric neighborhood around each particle. Once the neighborhood for a particle is defined, the second phase is entered. In this phase, a more expensive, exact method is

elements can lead to difficult geometric intersection problems of their own. See ???.

used to determine whether or not any two particles are in contact[2]. We will call the first phase *global* contact detection (GCD), because it deals with reducing the contact search area from the globally complex N squared set to a constant neighborhood around each particle (order N). The global versus local delineation of contact detection also separates the local details of exact geometry and geometric representation from this process of set reduction; because we limit the goal to finding only an approximate, conservative neighborhood of particles, it is possible to use simplified bounding volumes, which can be easily applied to any geometric representation scheme. The division of phases allows us to deal with the global complexity of many bodied simulations separately from, and without regard to, the local complexities of geometric intersection and contact resolution. After the exact methods of local contact detection are applied, the final set of contacts passes through a resolution. During this phase, the contact forces are determined, and applied to each object. Body forces and constraint forces are also applied during the resolution phase. The final step of the cycle is the integration phase. A direct integration of the dynamic equilibrium equations is performed, and the state of every object is determined for the next timestep. The state of the objects at the end of each cycle can then be output to a visualization module.

As indicated above, the complexity of global contact detection is proportional to the number of objects squared. All other aspects of the time stepping cycle are linearly proportional to the number of objects. Since large sets of objects are often used, this N

[2] This second step is often actually composed of a series of increasingly exact and increasingly expensive tests, rejecting particle pairs which can not be in contact.

squared term becomes very important, and motivates the optimization of the GCD process. The underlying focus of this research is to optimize the performance of MIMES; a user-friendly, interactive, 2D discrete element simulation environment developed at MIT (9). Two important aspects of this system have influenced the algorithms presented; interactivity, and generality. Because the system is intended to be used interactively as a hands-on computational laboratory, it is important to maintain a computational speed that will allow the simulation to run in real-time or near real-time. This motivates a speed optimization. The system is also intended, however, to be general enough to handle whatever problem the user defines, with as little user interaction as possible. This motivates the development of very general schemes, which make as few assumptions about the problem space as possible.

3 Global Contact Detection

Optimizing methods for GCD can be very beneficial to performance, especially in larger simulations where the scaling of the algorithm becomes crucial; in a fairly large simulation even the smallest N squared terms in the GCD algorithm will begin to drag down the performance. To pin down exactly the process we wish to optimize, and to facilitate the comparison of various implementations, we will define a general interface to the GCD class. This class is essentially a spatial query processor that will be optimized to process the n-to-n contact detection query, but also must be able to perform arbitrary range searches to find objects in a given area. The performance of the GCD implementation is be evaluated based on the following two criteria.

1. The final list of candidate contacts should be as small as possible. This is essentially a requirement on the accuracy of the method; we want to avoid expensive contact resolution for objects that are not likely to be in contact.

2. The algorithm should be fast. The whole point of GCD is to reduce the number of operations from the simple N squared case. Because we wish to deal with large sets of objects, the speed of the method is largely determined by how it scales.

3.1Generalized GCD interface

The GCD class will access an integer-keyed global ObjectTable which stores all of the objects in the simulation. Within each object it will have access to a simplified bounding volume. For our algorithms we will use bounding spheres, which are represented by the object's centroid and a maximum radius. It will be given a procedure to call to resolve the contact-pair candidates it finds. It is assumed that the GCD class will be implemented so that this call will be made only once on each contact-pair candidate. The public interface will consist of the following methods:

3.1.1Init(ObjectTable)

This procedure will make a one-time-per-simulation initialization of the GCD class. At the most basic, it will inform the GCD class of the location of the global ObjectTable, which lists all of the simulation objects in an arbitrary numbered order. This step will also often include memory allocation. In most cases the GCD class will need some sort of table in addition to the object table provided. If the size of this table is determined ahead of time, some efficiency may be gained by reusing the same memory space each timestep, rather than freeing and reallocating.

3.1.2StepInit()

This procedure will be called at the beginning of every iteration. This is where the GCD class will do any global calculations necessary for its neighbor finding phase. Some type of sorting is a common component of contact detection algorithms, and usually would be carried out here. Any portion of the neighbor finding algorithm that does not need to be done for each call to FindNeighbors should be done here.

3.1.3FindNeighbors(Action)

This procedure will find all neighbor object-pairs, and perform some action on them. The procedure is defined to be active. During one call to the method, all contacts are found, and processed in line. This is not strictly possible for all methods; (e.g.,those methods that can not be guaranteed to check a contact exactly once.) All passive methods, however, can be called in this active manner by setting aside memory for candidate lists, and revisiting them in a second pass to process them. By defining the process to be active, methods that can uniquely determine contacts can avoid the use of unnecessary storage, and time. In practice, the FindNeighbors routine might be hard-coded to call the specific contact detection routines rather than using the generalized callback, since it is not likely to be called for any other purpose.

3.1.4FindObjects(Region, Action)

This procedure implements the range search capability, and underlines the importance of the GCD class as a query interface to the object table. It should execute the specified action on any object that overlaps the target region.

3.2Simple GCD Implementations

To further illustrate the GCD interface, we will briefly describe two simple implementations, pairing, and bounding volume check.

3.2.1Pairing

The idea behind pairing is simply to reduce the full set of possibilities to unique pairs. This is done by looping over all objects and again looping over all objects with id's greater than the current id. The Init() and StepInit() methods are not required to do anything, and the FindNeighbors() method performs the loop described, returning all pairs. The FindObjects() method would simply check all of the objects against the region.

The computation spent per candidate contact-pair is negligible, and so the speed performance can be said to be instant. The accuracy of the method is extremely poor, having only reduced the N squared set by half. This will necessitate many executions of the potentially expensive local contact detection scheme, and result in slow execution. Note that even if the local contact detection scheme can reject impossible candidates fairly quickly (with a bounding volume check, for example), its execution will incur some unnecessary overhead.

3.2.2Bounding Volume Check

The bounding volume method builds upon the pairing method. As noted above, it is possible to reject impossible candidate pairs using a bounding volume check. In our interface definition, access to such a bounding volume was specified. In the bounding volume method we make use of this access within the inner loop of the FindNeighbors() method, and return only candidate pairs whose bounding volumes overlap.

The accuracy of the bounding volume method is, of course, much better than that of the pairing method. In fact the accuracy of the bounding volume test is the best accuracy we expect from any GCD, since the GCD is by definition independent of local geometry. The speed of the bounding volume method, however, is poor, since it requires the evaluation of a bounding volume for each unique pair.

In the next section we discuss some more useful methods for GCD, and outline their implementation.

4 Review of Current Methods

The process of GCD is essentially the repeated evaluation of a specialized spatial query of non-point data. To be precise we perform a region overlap test on the entire object table, for each member of the table. Spatial region queries are significantly more complex than point data queries, as there are many more overlap cases. Several authors have addressed the problem of non-point overlap queries, and a variety of methods exist for this purpose. There are three basic types of methods, Sorting methods (7), Hashing methods (6), and Tree methods (3). Each method has its own advantages and disadvantages, and it is not possible to identify one method that performs better for all cases. An overview of each type is given. Implementations are developed for Sorting and Hashing. To keep our specific goals in mind, we approach the methods using the GCD class description, and limit our consideration to 2D space; All of the methods, however, are applicable to more general spatial query problems, and 3D Space. For the purpose of speed and accuracy evaluation, we will consider a roughly square problem domain uniformly packed with N similarly sized objects.

In the algorithmic discussions that follow, the target object refers to a given object in the set, and the matching object refers to any object being tested for contact candidacy with the target. A diagram of all the 2D overlap cases for a target is given in figure 3. Note that the cases are numbered from 1 to 18, A and B. The cases marked A represent exactly the same contact as those marked B with the target and matching objects reversed. Thus, if a full set of case numbers (A or B but not both) is detected correctly by an algorithm, the algorithm is guaranteed to match each contact only once. For algorithms which detect the doubly full set of contacts, it will be necessary to reject directed pairs by object id.

4.1Sorting

Some of the first methods developed specifically for the purpose of contact detection were sorting methods. The basic idea is first to sort the entire table of objects on a number of axes, and then to search on those lists to limit the neighborhood of the target object. The concept is essentially derived from similar methods for point data, and its extension to non-point data creates a number of difficulties, which will become clear in the implementation development.

4.1.1SpatialSort GCD

SpatialSort is a typical implementation of Sorting for GCD, and was the original contact detection scheme used in MIMES. We outline what steps are performed in each of the interface methods, and a summary is presented of the overall performance of the algorithm.

4.1.1.1Init(ObjectTable)

The SpatialSort class will require an array of keys to the object table for each axis, which will be sorted by ordinate, and we will use corresponding rank arrays to index the sorted list by object id. These arrays are allocated in Init().

4.1.1.2StepInit()

As indicated, the first step of SpatialSort is to sort all of the objects. The objects are sorted by their lower extremity. This sorting can be done using a number of standard sorting techniques, such as heap sort, or insertion sort. The relative performance of these methods depends on the actual lists being sorted. Heap sort performs evenly on all lists in N log(N) time (unlike quicksort which performs very poorly on almost sorted lists), and is a good candidate. Noting, however, that the same lists will be re-sorted every timestep, objects can be assumed to move very little between timesteps. It may be possible, therefore, to get better performance using an insertion sort, which approaches order N time for nearly sorted lists. Once the lists are sorted in each axis, we index the lists by object number.

4.1.1.3FindNeighbors(Action)

In SpatialSort, we keep track of only the lower extremity of the objects in the list, and must search to find the upper bound of a given object. To begin, we will consider each axis independently. In order to find the region enclosed by the target object, it is necessary first to search for the upper extremity in the index of lower extremities. This can be done using a binary search, which takes order log(N) time for each object. Once this range is determined, we have the axis neighbors, since any matching object whose lower bound is not in the range, will contain the target object's lower bound within its own range. In 2D this is not the case, as the overlap cases are more complex. We can simplify the full 2D case diagram given earlier by ignoring the matching object's upper bound. This simplified diagram, shown in figure 4, represents the information available to the target object in the SpatialSort loop. Each simplified case is marked with the full 2D case numbers which apply. Examining the diagram, it is clear that not all valid contacts fall under case V. A typical missed case is 11 (A or B), which is shown in figure 5. Several methods can be used to detect these missing cases:

1. Split the process into two passes. In the first pass, collect all the possible contacts in each axis for every object in two linked lists (one for each axis). In the second pass intersect the lists for each object, and eliminate duplicate pairings with object ids.

2. Again split the process into two passes. In the first pass, the 1D range as determined above is stored for each target object. When storing the range for the target, the stored range of each candidate in the target's range is extended (if necessary) to include the target. In this way, a complete range is determined by the end of the pass. The second pass checks each object in one axis range against the other for every object. Again, duplicate pairings are eliminated by id.

3. One Pass method. Note that case 11, and all of the other missing cases correspond to either case II or IV on the simplified diagram. In addition, taking either case II or IV with case V gives a complete set of contact cases on the full diagram. It is therefore sufficient to develop a full range list for only one axis, if that axis is used for every object. The process can be described as follows: For each object a, find the range of lower extremities in one axis. In the other axis, search the index for the first object b that has an upper extremity greater than the lower extremity of a. This

establishes the full range (as used in the other methods) for the y axis, and the partial range in x. This exactly determines all of the contact cases.

Each of the above methods has some drawbacks, but method 2 is the most desirable from a performance standpoint. In the theoretical problem domain, each of the neighbor lists is approximately root(N) long. In (1), the linked lists will take up storage proportional to N root(N), and the searching and maintenance of the linked lists incur hidden N squared costs. For example, every time an object is added to a given list (which happens about N root(N) times), the list must first be searched to make sure the object isn't already a member (and the list length is proportional to root(N)). Method (2) requires no list maintenance, since the lists are stored as ranges, but the ranges include more objects than the linked lists in (1) so the accuracy is degraded for objects of varying size (see figure 6). Method (3) requires only one pass, and less storage than (1) or (2). In addition, only one of the ranges is enlarged as in (2), so the accuracy is better, but a linear search is required to achieve that range, thus introducing an N squared term.

4.1.1.4FindObjects(Region,Action)

In order to find all of the objects that overlap the given region, an upper limit is first established by binary search. The lower portion of each list must then be exhaustively tested for overlap with the region.

4.1.2Analysis of SpatialSort

If method 2 is used to create the axis neighbor lists, SpatialSort scales essentially with order N root(N), returns a candidate list which is nearly as accurate as the bounding volume on which it is based, for uniformly sized objects. In the original implementation of MIMES contact detection, method 3 was used.

4.2Hashing

Hashing, sometimes called binning or bucketing[3], is another method of simplifying the spatial query process. In this method, the problem domain is broken down into a static integer grid. Each hash grid point can be thought of as a bucket containing objects. Each object can be assigned to a bucket using a simple rounding scheme. Once the objects are all assigned to buckets, the connectivity of the grid, which is simple and predetermined, is used to get a neighborhood for each object.

4.2.1Hashing Algorithm

4.2.1.1Init(ObjectTable)

The first step in setting up this method is to determine the size and spacing of the grid. In order for each object to be uniquely locatable in a single hash grid, the grid squares must be at least as large as the largest object. An initial pass through all of the objects determines the maximum size. The start and end of the grid must also be determined, so that the hash grid array can be set aside during initialization. Because the objects

[3] A complete presentation of a similar hashing algorithm, called NBS is given in (6).

will be located in only one grid square, a compact, array-based, linked list can be used. For an explanation of array based linked lists see (10). We only require three such arrays since we can evaluate the grid using only one column, and two rows at a time. Note that if, at any time during the simulation, the objects move outside the limits defined during initialization, the method will fail.

4.2.1.2StepInit()

Before any searching is done, the objects are allocated to the hash grid for one axis (we will use Y). Each Y grid will contain about root(N) objects.

4.2.1.3FindNeighbors()

Taking one Y grid at a time, all of the objects in that grid are allocated to the first X hash grid array (X1). The other X hash grid array (X2) is initialized to empty. The grid squares in X1 should each contain a small number of objects that does not increase with increasing N. Each object in each grid square in X1 has possible neighbors in the same square and in the adjacent ones. Since we only need to check a given contact-pair once, only the previous grid square is considered. Care should be taken to eliminate duplicate matches between objects in the same square. Once the first row is finished, X1 and X2 are switched, and all of the objects in the second Y grid are allocated to the free X hash grid array (now X1). Now each object in each grid square also has neighbors in the three adjacent squares in X2 (see figure 7). The process continues, switching and clearing the X arrays each time, until all of the contact pairs have been found.

4.2.1.4FindObjects(Region,Action)

In order to find all of the objects within a region, the grids must first be intersected with the region. First, the upper and lower grid points in each axis are determined for the region. The axis ranges need to be expanded by one grid point to account for possible object overlap. Then, for each grid in the correct range of Y (which has already been hashed in StepInit()) we hash the grid objects into X1. Each grid X1 within the correct range, is then considered a candidate hit. If the region is not square, it may be necessary to do an additional check to determine if the candidate, in fact, overlaps the region. The speed of this process depends on the result size M, and is of the order root(N*M).

4.2.2Analysis of Hashing

Provided the array clearing is implemented carefully, this algorithm runs in order N time, which makes it ideal for very large problems where SpatialSort scales poorly. It too, however, has its drawbacks. Because the final candidates are based on the connectivity of the grid, the accuracy is somewhat less than that of SpatialSort. This can be seen in figure 7. Even for equally sized discs, the neighborhood around a given disc is a square essentially twice as large as the disc. Another factor reducing the accuracy is object uniformity. As indicated in the Init() procedure description the grid squares must be large enough to contain the largest object (allowing it to overlap into less than half of the next bucket). This means that if the objects are not sized equally, the neighborhood grows even larger, and the accuracy begins to seriously degrade (see figure 8). It should be noted that the algorithm is still order N, but that the constant on the front of the N term becomes large.

4.3Trees

The final group of methods is tree methods. There is a wide variety of tree-based methods that can be employed for spatial queries (1,2,3,4). In essence, these methods all attempt to subdivide the problem domain using a tree structure. One type of tree, frequently used in spatial databases to process intersection queries, is the R-tree. The idea behind the R-tree, and all of its variants, is to subdivide space into a hierarchy of nested n-dimensional quadrilaterals. The tree itself is very similar to a B-tree. A full description of the original R-tree algorithm can be found in Guttman (4). Several variants of the R-Tree exist (1,2). Many of these are simply alterations in the tree-construction algorithm which attempt to build a more optimal tree. The RTreeSearch algorithm described here is fairly general and could be adapted to any of the R-tree variants available.

4.3.1RTreeSearch GCD

Here we present the overview of an R-tree-based GCD. This implementation is representative of tree implementations in general. The performance of a given tree method is, however, dependent on the details of the tree structure and the efficiency of query.

4.3.1.1Init(ObjectTable)

In our implementation, the tree is constructed and destroyed every timestep, so there is no need to build the tree in the Init() procedure. We simply locate the ObjectTable, and return.

4.3.1.2StepInit()

Here we build the tree. Looping over the ObjectTable, we enter all objects, one at a time, in the tree. For R-trees, objects are entered at the leaf nodes, where they are stored as a key to the ObjectTable, and their associated axis aligned bounding rectangle, which we approximate (conservatively) with the axis aligned bounding rectangle of the bounding disc. Each time a leaf node is added, the changes propagate up the tree, inducing splits where necessary.

4.3.1.3FindNeighbors()

To find all of the contacts, we enter the tree at the root with a query region corresponding to the bounding rectangle of each target object in turn. Each branch of the tree that overlaps the query-region is descended, and when a leaf is encountered, its object is a contact candidate. Duplicate object pairings can then be eliminated by object id, and the remaining matching objects are returned.

4.3.1.4FindObjects(Region, Action)

The general query is implemented in exactly the same way as the FindNeigbors query, except that the query region is specified by the bounding rectangle of the supplied Region, rather than the target object.

4.3.2Analysis of RtreeSearch

Tree-based methods, and especially R-tree based methods have two distinct advantages that have made them very popular in more static spatial databases. First, they perform well on object sets with large size variations. Second, they scale very reasonably; typically, a well-formed tree can be descended in log(N) time. This means that the contact detection query is evaluated in N log(N) time. The main disadvantage of R-tree based contact detection in Discrete Element Simulation is that it requires a significant amount of processing time to build the tree, and it must be destroyed and rebuilt every timestep. The constant allocation/de-allocation requires a lot of memory turnover, which can slow down the processing. In addition, the tree itself can take a significant amount of memory. The accuracy of the R-tree method is the same as SpatialSort.

5 Improved Methods

The previous section on current methods described several methods of GCD and explained the relative advantages of each. In this section we will explore ways to eliminate some of the disadvantages of these methods, while preserving their better qualities.

5.1Double-Ended Spatial Sorting (DESS)

Spatial sorting has several advantages that make it a dependable way to approach contact detection in an interactive environment such as MIMES; when method 3 is used for neighbor list intersection, the performance is unaffected by object size distribution. In addition, the problem domain does not need to be enforced, as it does in hashing. Much of the cost incurred in spatial sorting is due to the inability to quickly locate the region occupied by a given object in the sorted index. This is because we can sort the objects on only one extremity at a time, and have to search the list to find the other extremity. We propose a new method of spatial sorting called Double Ended Spatial Sorting (DESS), which correlates the location of both upper and lower extremities of the objects in each axis. This is done simply by entering both extremities in the sorted index table. Thus each object will have two entries on each axis. In order to differentiate between ends, a side table is kept for each axis index, indicating with a positive or negative one, whether a given entry is an upper or lower bound. The sorted list is itself indexed with two separate lists ranking the objects by upper and lower bound. This allows the immediate lookup of the bounding rectangle of any object in the simulation, eliminating the need for a binary search, and immediately establishing the exact axis neighbor list.

Since the DESS index contains both extremities, the full case diagram given earlier applies explicitly. This diagram is used to ensure that each contact is reported only once during the FindNeighbor loop. Since it is not possible, without searching for each target objec, to find all of its neighbors (e.g. case 16B), the table searching algorithm must be designed to match exactly half the cases. In addition, specialized sorting routines need to be written to differentiate between upper and lower bounds. The actual algorithm is fairly simple and requires no storage, since any needed overlap information can be retrieved through lookup. The cost incurred by doubling the size of the list is minimal for larger simulations, especially if the objects can be considered to remain essentially sorted between timesteps. The method also has the advantage of consisting entirely of integer operations once the lists have been sorted.

5.1.1DESS Algorithm

The DESS algorithm as implemented for MIMES is presented here. It should be noted that the algorithm is easily implemented for 3D, by adding another axis.

5.1.1.1Init(ObjectTable)

As in SpatialSort, this procedure allocates the memory for the index lists, and populates them with the indices of the objects in the ObjectTable. As indicated in the method overview, we require one index for each axis (of length 2N). The objects will also be ranked in two lists for each axis, one for the lower bound rank, and one for the upper bound rank. If the sorting is to be done with an insertion sort, a one time presort using some other method, like heap sort is recommended.

5.1.1.2StepInit()

At the beginning of each timestep the index lists are resorted. The sorting algorithm is specially designed to sort the index table, and keep the corresponding side table concurrent. The rank lists are then repopulated by enumerating the index, and updating the appropriate rank list values, based on the side table.

5.1.1.3 FindNeighbors(Action)

First, we define a master axis (e.g. X). We will run through the sorted index of that axis sequentially. From the case diagram, it is clear that columns 1 and 6 are not contact cases, and will not appear in the X axis neighbor lists. In addition, the column 3 cases will also not appear. In order to find the reverse of the cases in column 3, we must match the entirety of column 4. To split the cases in any reasonable manner, then, we must match only columns 4 and 5, which represent a full set of contact cases. The algorithm works as follows: Taking each lower bound on the master axis as a target object, we look up the upper bound, and both bounds in the other axes to be considered. Each matching object in the range on the master axis is then checked to see if it lies outside any of the other axis ranges. If not, it is a candidate contact. Noting that it is possible to match the same object for each extremity, all matching point entries that are not lower bounds are skipped. Conferring with the 2D case diagram, it is noted that this procedure matches exactly cases 8B, 9B, 10A, 11A, 14B, 15B, 16A, and 17A (columns 4 and 5). Since this represents a complete set of positive case numbers (A or B but not both), all contacts will be detected exactly once.

5.1.1.4FindObjects(Region,Action)

As in SpatialSort, the general range search is not very efficient, since objects that completely surround the query range have no points within the range. This means that the index must be linearly searched from the bottom through the upper bound of the query range. This upper bound, however, can be obtained with a binary search, without losing any positive hits.

5.1.2DESS Method Analysis

The DESS method reduces the overhead associated with spatial sorting methods, and transforms the contact detection problem into an integer arithmetic space. The order of the algorithm remains N root(N), but all floating point computations have been moved out of searching loop, which is the only N root(N) portion of the algorithm. By contrast, the original SpatialSort algorithm required many floating point evaluations within the searching loop. In addition, the only other phase that is not order N is the sorting phase, which approaches order N for nearly sorted lists. The overhead of double sorting is not significant since it only represents a doubling in the sort time.

5.2Striping DESS

Having streamlined the sorting and range-finding portion of spatial sorting in the DESS formulation, it is appealing to try to improve the method further. The major remaining bottleneck of the DESS process is the range intersection phase. In this phase the intersection of the 1D range lists is found for each object. Because the 1D neighbor lists are given as ranges, it is only necessary to evaluate the members of one list against the other ranges. Each range is about root(N) elements wide, and there are N ranges to evaluate. This makes the overall algorithm N root(N). If, however, one of the lists could be reduced to constant length, then the elements of that list could be checked against the other ranges much more efficiently.

In the Hashing algorithm, the problem space is reduced dimension by dimension, allowing the use of only a handful of hash grid arrays. A similar procedure could be used to search the DESS problem space. Instead of sorting all of the lists at once, The sorted master axis list would be used to subdivide the slave axis lists into 'stripes.' Each stripe would contains a section of the physical space divided so that the last slave axis neighbor list within that stripe will be of constant length (see figure 9).

For two dimensions, this makes the stripes some constant multiple of root(N) wide. Each of the members in a given stripe is added to its own list for the next axis. If we are working in three or more dimensions, this 'stripe' can be further reduced in a similar fashion. Once the final slave axis is reached, the neighbor range for each object in that axis will be much shorter than the other neighbor ranges, and can be evaluated much faster.

In order for Striping to work properly, it is necessary to duplicate some objects between stripes so that objects that belong properly to one stripe can be in contact with other objects that really belong to the next stripe. This is done by keeping track of the highest upper bound of the objects as they are added to the stripe. Once all of the objects are added to the stripe list the objects in the border zone defined by that upper bound are added, allowing contact to be correctly evaluated for all of the original stripe objects. The next stripe is then started at the proper end of the previous stripe. Note that objects wholly included in the border zone will be included in two successive stripes, which can result in duplicate contacts being reported. Proper checking should be done to ensure that each of these is only reported once. To avoid too much wasted computation, the stripes should be kept to a reasonable size when compared with the border zone. It is important to note that for the reduced dimension stripes, the object lists will no longer be pre-ordered, and so it may be more efficient to use a quicksort algorithm for sorting.

6 Performance

GCD classes were fully implemented for all of the methods outlined above. Two implentations of DESS and Hashing were developed. In DESS, the algorithm was implemented with heap sort and in DESS2 insertion sort was used. This was done to determine how much the object rearrangement between timesteps would affect the performance of the sorting phase. In the original Hashing algorithm, the linked list arrays were be cleared completely for each row, resulting in a very small N root(N) term. In Hashing2 this term is eliminated. The difference between the performance of Hashing and Hashing2 illustrates the effect of even small higher order terms. The contact detection phase was altered within MIMES to be compatible with the GCD definition, and to allow run-time specification of the GCD method.

A series of benchmark tests was run on the various methods. The tests are designed to scale to include any number of objects. Each test was run twice for 500 timesteps.

6.1Test 1

Test 1 is a square sample of discs that is fixed around the exterior. The discs are given alternating initial velocities to ensure object rearrangement and supply a fair amount of contact creation and deletion. This test essentially emulates the theoretical problem used in the above discussions and is ideally suited to the Hashing algorithm. The results of the tests are presented in figure 10. The slope of the curves in this log-log graph indicate the approximate power in N of the algorithm performance, this data is shown against N in figure 11. As expected, the Hashing2 GCD method approaches linear time for large N, and the DESS, DESS2, and Hashing GCD methods all approach N root(N) time. The performance of DESS2 clearly indicates that object rearrangement is insignificant enough to neglect in the choice of sorting algorithm. It is worth noting that the actual performance of the DESS2 and Hashing2 algorithms is very similar (within an order of 2) for less than 12000 objects, and that the performance of DESS2 is only insignificantly worse than that of the original Hashing algorithm. The performance of SpatialSort is worse even than expected, and indicates that another intersection method should be used. Finally, it is clear that StripeSort suffers from some hidden scaling problems which may be related to the increasing size of the border zone with increasing N.

6.2Test 2

Test 2 is similar to Test 1, but with discs of varying size. The test was run with size ratios of 1, 2 and 4. The timings for each method were interpolated to 800 objects for each size ratio. The time variance for these ratios is given in figure 12. From this chart, it can be seen that all of the DESS-based algorithms perform better for samples with varying object size. This better performance (due to increased accuracy), outweighs the better time performance of the Hashing algorithms for average sized simulations. Test results for greater object size ratios were not obtained for either Hashing algorithm, as the smaller objects were not completely contained by the larger ones. The existence of objects outside the pre-set problem domain resulted in failures in the Hashing algorithms. Full test results for DESS2, up to a size factor of 32, are given in figure 13.

6.3Conclusions

From these tests, it can be seen that the DESS formulation of GCD performs nearly as well as Hashing for the theoretical problem of equally sized objects for moderately sized problems. Noting that 2D problems do not usually include as many objects as 3D problems, this performance can be said to be acceptable for the 2D MIMES environment. In addition, it has been shown that this algorithm is a more robust solution to the contact detection problem in general, as it does not require a fixed problem domain, and exhibits no dependence on object size ratio.

7 Future work

7.1DESS as an integer space

As mentioned in the method description, one advantage of the DESS method is that, with relatively little work, it transforms the real-valued problem space onto an integer grid of fixed size. The bottleneck of the DESS method is not in this transformation, but rather in the searching of this space. Any searching method that applies to general space could, in theory, be applied to the DESS space. One example is tree-based searching. A search tree, generally speaking, has a speed performance of order N log(N). If an R-tree was used to search the integer DESS space, some computational and memory savings could be achieved over the real-valued R-tree implementation, and the overall method might scale better than the basic DESS algorithm. The DESS-Rtree GCD class would append tree construction to the DESS StepInit() method, and replace the FindNeighbors(Action) and FindObjects(Region,Action) methods with the methods from RtreeSearch.

1. N. Beckman, H. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles", *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, Atlantic City, May 1990

2. S. Berchtold, D. Keim,and H. Kriegel, "The X-tree: An Index Structure for Higher-Dimensional Data", *Proc. 22nd VLDB Confereance*, Mumbai (Bombay), India, 1996

3. J. Bonet and J. Peraire, "An alternating digital tree (ADT) algorithm for 3D geometric searching and intersection problems", *Int. J. Num. Meth. Engng.,* **31** (1991)

4. A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching", Proc. *ACM-SIGMOD Int. Conf. on Management of Data*, 1984

5. G. Mustoe and J.R. Williams, *Proc. 1st US Conf. on Discrete Element Methods.* Golden, CO (1989)

6. A. Munjiza, K.R.F Andrews, "NBS contact detection algorithm for bodies of similar size", *Int. J. Num. Meth. Engng.,* **43** (1998)

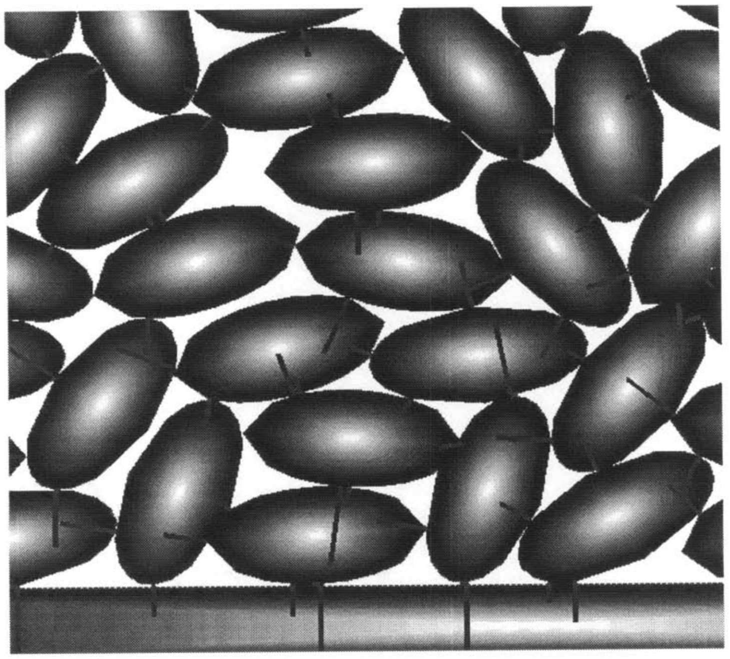
7. R. O'Connor, *A Distributed Discrete Element Modeling Environment - Algorithms, Implementations, and Applications*, ScD. Thesis M. I. T, Jan 1996

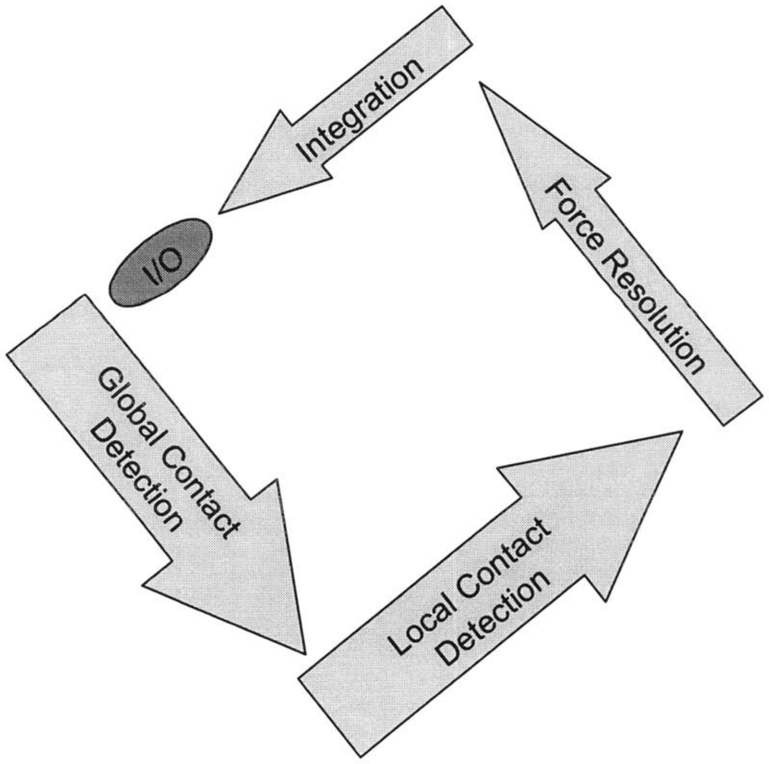
8. R. O'Connor, M.J. Gill and J.R. Williams, "A linear complexity contact detection algorithm for multi-body simulation", *Proc. 1st US Conf. on Discrete Element Methods* (1993)

9. N. Rege, *Computational Modeling of Granular Materials*, PhD. Thesis M. I. T, Feb 1996

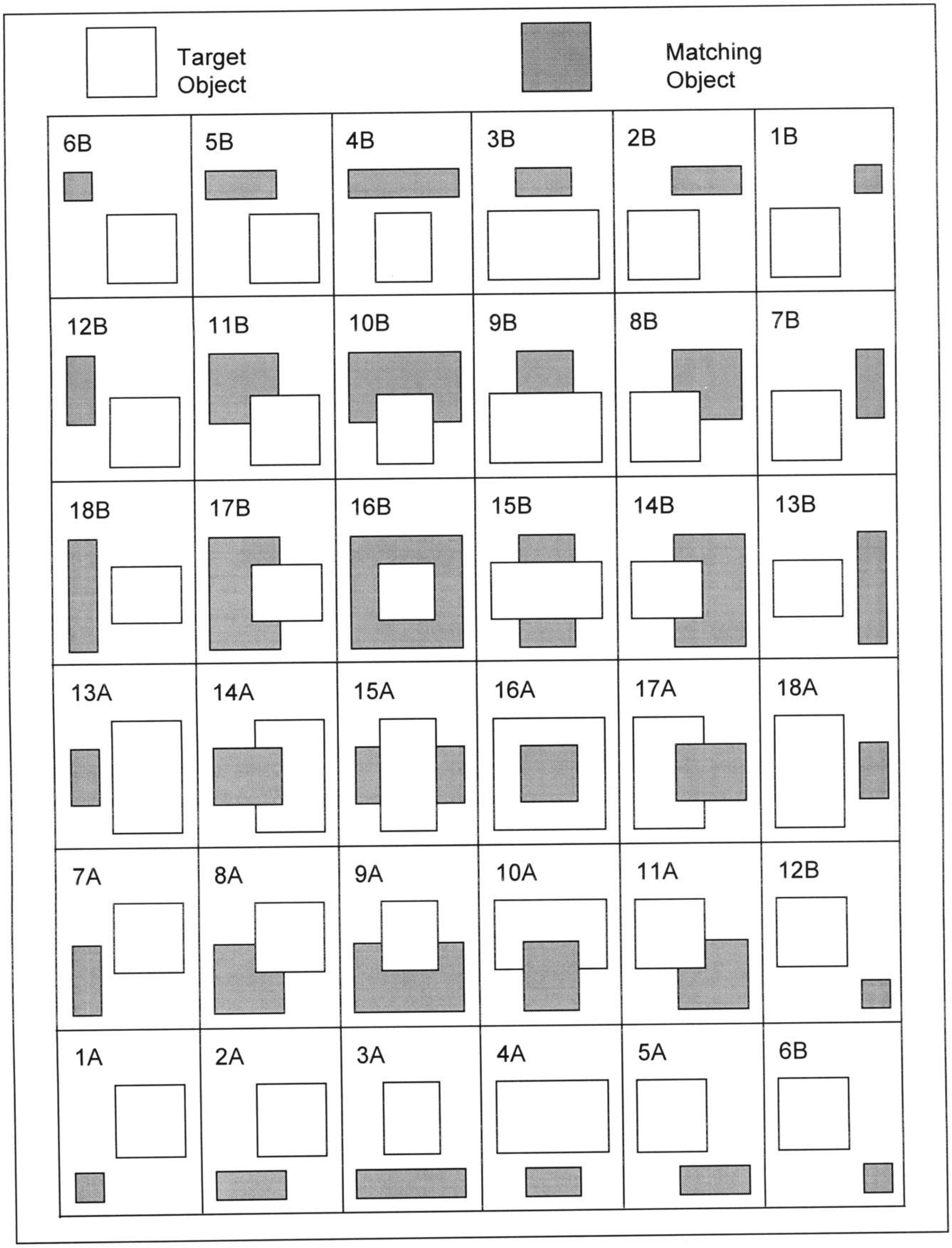
10. R. Sedgewick, *Algorithms in C++*, © 1992 Addison Wesley

11. J.R. Williams and G. Mustoe, *Proc. 2nd US Conf. on Discrete Element Methods.* MIT, MA (1993)

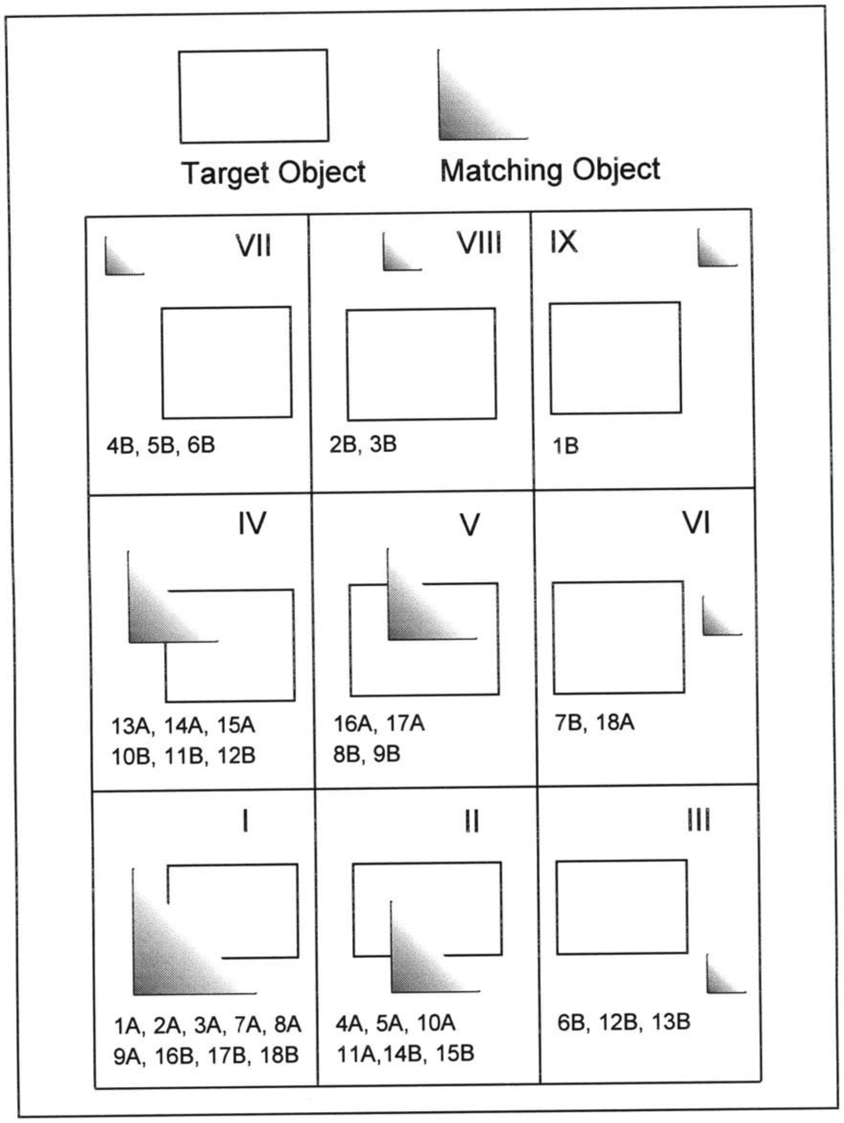

Discrete Elements in Contact
figure 1



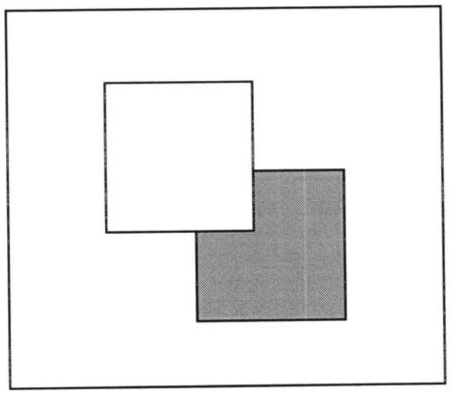
Discrete Element Simulation Cycle
figure 2



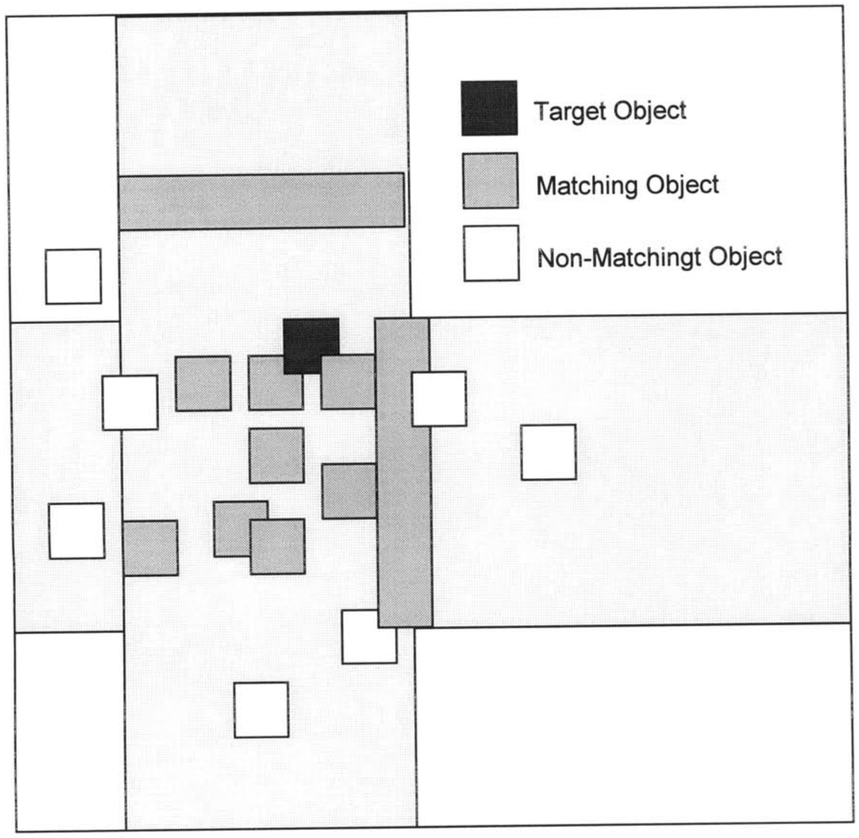
2D Contact Cases figure 3



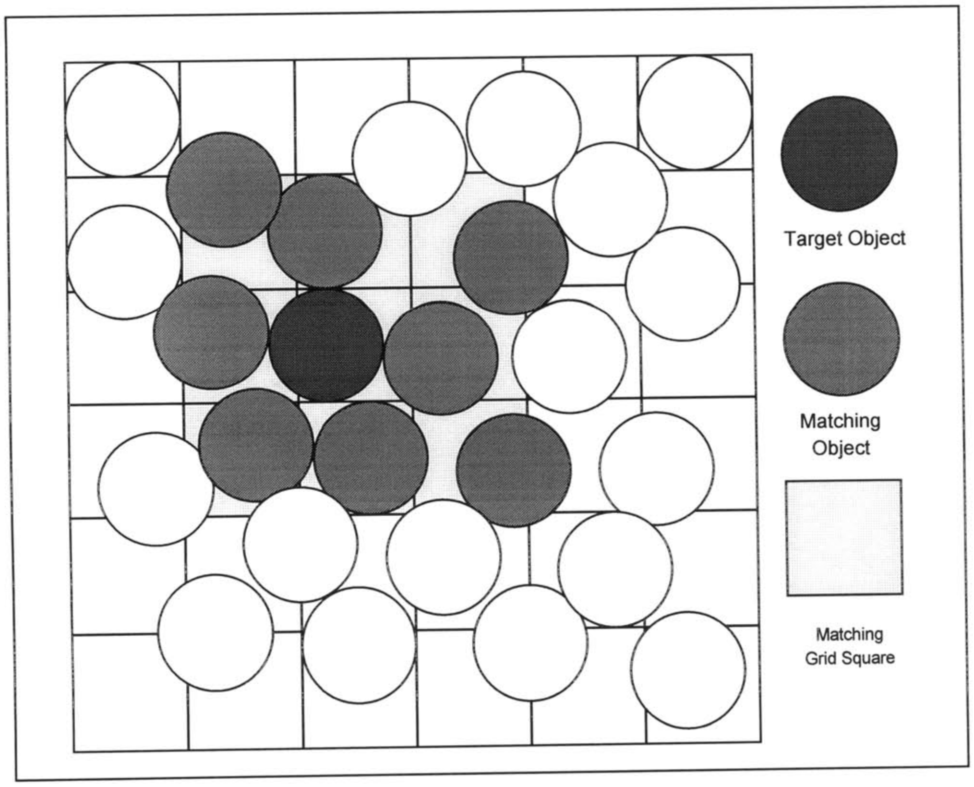
SpatialSort Simplified Contact Cases
figure 4



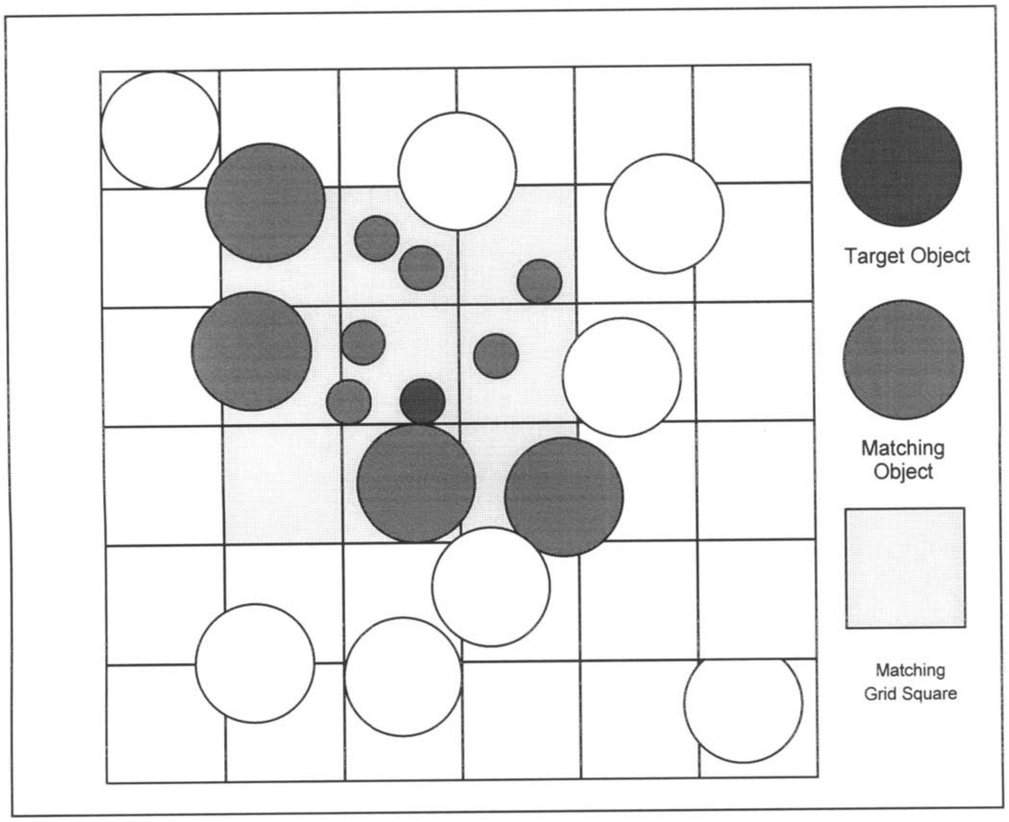
Contact Case 11
figure 5



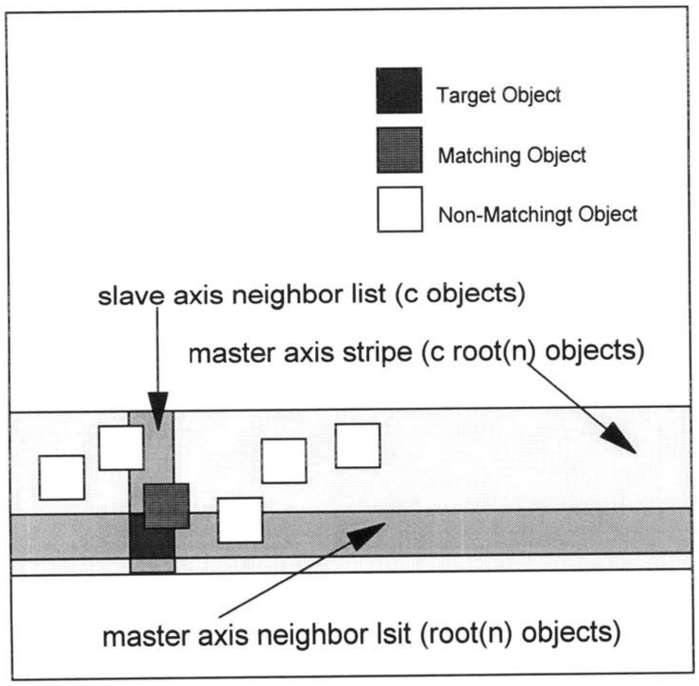
Neighbor List
Expansion
figure 6



Hashing Example
figure 7



Hashing Example
figure 8


Target Object
Matching Object
Non-Matchingt Object
slave axis neighbor list (c objects)
master axis stripe (c root(n) objects)
master axis neighbor lsit (root(n) objects)

StripeSort Example
figure 9