Discrete Element Computation:
Algorithms and Architecture

by

Eric David Perkins

Submitted to the Department of Civil and Environmental Engineering
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Information Technology

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2001

© Massachusetts Institute of Technology 2001. All rights reserved.
Discrete Element Computation:
Algorithms and Architecture

by

Eric David Perkins

Submitted to the Department of Civil and Environmental Engineering
on May 25, 2001, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Information Technology

Abstract

The Discrete Element Method is a numerical technique used to model physical phenomena through the dynamic interactions of a large number of distinct bodies. The strength of the method lies in its ability to accurately model the behavior of inherently discontinuous media, such as granular, fractured, or powdered materials.

The major computational obstacle in discrete element simulation is the automatic detection of contacts between bodies. For large simulations, the complexity of the contact detection process is driven by the general spatial reasoning problem of neighbor searching, in which candidate intersection pairs are selected based on their proximity.

Neighbor search algorithms exist that exhibit linear scaling in the number of bodies. These algorithms rely, however, on the assumption of uniformly sized objects. Deviations from this assumption, inherent in many common physical systems, significantly degrade performance. This thesis presents a new grid-based algorithm which accommodates objects of varying size.

A new grid-based neighbor search algorithm, called CGrid, is developed to deal with objects of varying sizes. A generic formulation for any number of dimensions is presented. CGrid scales linearly in the number of bodies, and is less sensitive to object size disparity than existing linear algorithms. By combining performance and robustness, CGrid provides a reliable neighbor search solution for general simulation systems.

An architecture for simulation is presented, which is designed to support rapid prototyping and extension development. The core architecture provides an infrastructure of generic components for simulation management. The simulation object hierarchy is constructed to address the issues associated with developing extension capabilities, and supporting the wide variety of objects and behaviors which can be employed within the Discrete Element Method.

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

First and foremost I would like to thank my advisor and committee chair, John Williams. His consistent support, understanding patience, and energetic enthusiasm have been more than I could have asked for. Without him, I am sure I could never have come this far. Thanks also to the other members of my committee, professors Amaratunga and Connor, for their helpful and timely guidance on this thesis and in my studies in general.

To Julia, for taking me in when I needed it, like the family you have always been to me, and for constantly keeping me plied with coffee and good spirits. To Brian and James for putting up with my vagaries, and always being there for advice, and help.

To Ben Cook and Petros Komodromos, thanks for the discussions, without which much of this thesis would be blank. And , of course, thanks for always reminding me of the important deadlines.

Thanks to Joan McCusker for her always amiable help in dealing with MIT, and friendly support and encouragement. I can’t imagine what any of us would do without your help.

Lastly, to my parents, my unbending support and unconditional defenders for the past 26 years, you have made this and everything else possible. With love, respect, and the most profound gratitude, I dedicate this work to you.
Contents

1 Introduction  13
   1.1 The Discrete Element Method  14
      1.1.1 Contact Penalty  14
      1.1.2 Geometric Representation  16
      1.1.3 Contact Detection  19
   1.2 MIMES  20

I Algorithms  28

2 Neighbor Searching  29

3 Review of Existing Methods  32
   3.0.1 Tree Methods  33
   3.0.2 Sorting Algorithms  35
   3.0.3 Grid-based algorithms  37

4 CGrid: Neighbor Searching for Objects with Arbitrary Extents  45
   4.1 SGrid: Generalized Bucketing  45
      4.1.1 SGrid Contact Mask  46
      4.1.2 SGrid Subdivision  51
      4.1.3 SGrid: putting the pieces together  55
   4.2 CGrid: Supporting Arbitrary Extents  57
      4.2.1 Discretization of Arbitrary Extents  57
8.2.2 Identifiers .................................................. 92
8.2.3 Cloning ...................................................... 93
8.3 Containers and Lookups ................................. 95
8.4 Type Interactions ......................................... 97
  8.4.1 TypeId Convention ...................................... 98
  8.4.2 TypeInteractionManager .............................. 99
  8.4.3 InstanceInteractionManager ......................... 100

9 Simulation .................................................. 102
  9.1 SimLevel .................................................. 102
  9.2 SimLoop .................................................. 104
  9.3 Generic SimLoop ......................................... 105
  9.4 Modules .................................................. 106
  9.5 SimModule ................................................ 106
  9.6 SimulationManager ...................................... 107
  9.7 SimObjManager .......................................... 108

10 Element .................................................. 109
  10.1 Goals and Motivation .................................. 109
    10.1.1 Monolithic Elements ............................... 109
    10.1.2 Modular Elements .................................. 110
    10.1.3 Shareable Components ............................... 111
  10.2 Properties .............................................. 111
  10.3 Locale .................................................. 113
  10.4 Bounding Box .......................................... 113
  10.5 Dynamics ............................................... 114
  10.6 Contact ................................................ 115
  10.7 Geometry ............................................... 116
  10.8 Geometric Intersection ............................... 117
  10.9 Material ............................................... 118
  10.10 Actions ............................................... 119
11 Behaviors

12 Scripted Interface

12.1 General Command Guidelines ............................................. 123

12.1.1 Option Notation .......................................................... 124

12.1.2 Help ............................................................................. 124

12.2 Command Infrastructure ....................................................... 125

12.2.1 CommandLine ................................................................. 125

12.2.2 Three Phase Evaluation .................................................... 126

12.3 ArgStack ............................................................................ 127

12.3.1 PipeLine ....................................................................... 128

12.3.2 PipeDest and PipeSource ................................................. 128

12.3.3 Pipes ............................................................................. 129

12.3.4 Pipers ............................................................................. 134

12.3.5 Implemented Pipes ........................................................... 138

12.4 Benefits .............................................................................. 139

12.5 Potential Issues ................................................................. 140

13 Architecture: Contributions ..................................................... 142

A CGrid Implementation .............................................................. 143
List of Figures

1-1 Discrete Elements .................................................. 15
1-2 Granular DEM Example ........................................... 16
1-3 Contact Law .......................................................... 17
1-4 2D MIMES Superquads ........................................... 18
1-5 DFR Example .......................................................... 18
1-6 Contact Detection: Neighbor Searching and Geometric Resolution .... 20
1-7 MIMES User Interface ............................................. 21
1-8 MIMES: Unconfined Compression in a Bonded Sample ................. 23
1-9 MIMES: Particle Mixing in a Rotating Drum .......................... 24
1-10 MIMES: Standing Waves in Particles on a Shaker-Table[98]........... 25
1-11 MIMES: Bore-Hole Break-out .................................. 26
1-12 MIMES: Hydraulic Fracturing .................................... 27

2-1 Bounding Volumes .................................................... 30

3-1 Quad Tree Example .................................................. 34
3-2 R-Tree Example ....................................................... 34
3-3 DESS Example .......................................................... 36
3-4 Grid-based Methods .................................................. 38
3-5 Bucketing Example ................................................... 38
3-6 Bucketing and Sparse Simulations ................................ 39
3-7 Grid-based algorithms: over-reporting ................................ 42
3-8 Effect of object size distributions on NBS grid ......................... 43
3-9 Simulation with long boundary rectangles ............................ 44
Listings

NBS Main Loop ........................................... 40
Vector ....................................................... 83
Rotation ..................................................... 85
Shareable ..................................................... 91
Identifiers ..................................................... 92
Cloning ......................................................... 94
Lookup ........................................................ 95
NameLookup ................................................... 96
TagTable ........................................................ 96
MIMES Contact ................................................. 97
TypeId .......................................................... 98
TypeId Usage ................................................... 99
TypeInteractionManager ..................................... 100
InstanceInteractionManager ................................. 100
SimLoop ........................................................ 104
SimObj .......................................................... 104
SimBehavior .................................................. 105
Module .......................................................... 106
SimModule ...................................................... 107
SimulationManager ........................................... 107
SimObjManager ................................................ 108
Element ......................................................... 110
Properties ...................................................... 112
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locale</td>
<td>113</td>
</tr>
<tr>
<td>Region</td>
<td>113</td>
</tr>
<tr>
<td>Dynamics</td>
<td>114</td>
</tr>
<tr>
<td>Contact</td>
<td>115</td>
</tr>
<tr>
<td>ContactPenalty</td>
<td>115</td>
</tr>
<tr>
<td>PenaltyBasedContact</td>
<td>116</td>
</tr>
<tr>
<td>Geometry</td>
<td>117</td>
</tr>
<tr>
<td>IntersectionPoint</td>
<td>117</td>
</tr>
<tr>
<td>Material</td>
<td>118</td>
</tr>
<tr>
<td>MaterialInteraction</td>
<td>119</td>
</tr>
<tr>
<td>StandardBehavior</td>
<td>119</td>
</tr>
<tr>
<td>GroupBehavior (interim)</td>
<td>120</td>
</tr>
<tr>
<td>BehaviorManagerEntry</td>
<td>121</td>
</tr>
<tr>
<td>GroupBehavior (final)</td>
<td>121</td>
</tr>
<tr>
<td>BehaviorManager</td>
<td>122</td>
</tr>
<tr>
<td>CommandLine</td>
<td>125</td>
</tr>
<tr>
<td>ArgStack</td>
<td>127</td>
</tr>
<tr>
<td>PipeDest</td>
<td>128</td>
</tr>
<tr>
<td>PipeSource</td>
<td>129</td>
</tr>
<tr>
<td>ArgPusher</td>
<td>130</td>
</tr>
<tr>
<td>ArgPasser</td>
<td>130</td>
</tr>
<tr>
<td>ArgPiper</td>
<td>131</td>
</tr>
<tr>
<td>ArgPopper</td>
<td>132</td>
</tr>
<tr>
<td>DoubleArgPopper</td>
<td>132</td>
</tr>
<tr>
<td>ArgSwapper</td>
<td>133</td>
</tr>
<tr>
<td>Command System Syntax</td>
<td>134</td>
</tr>
<tr>
<td>Piper</td>
<td>135</td>
</tr>
<tr>
<td>PiperT1</td>
<td>135</td>
</tr>
<tr>
<td>PiperC1</td>
<td>136</td>
</tr>
<tr>
<td>PiperT1C1</td>
<td>136</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The Discrete Element Method (DEM) provides a powerful way to model the behavior of physical systems using collections of distinct bodies. The method is able, using large numbers of bodies, to capture behaviors which are not well modeled by methods based on continuum assumptions alone. The method finds applicability in situations where the phenomena of interest are rooted in discontinuity. Thus studies of fracturing in ice sheets [36, 32, 76], beams [47], and rock [33, 34, 49], have all been successfully undertaken with discrete element methods. Of particular interest are problems in granular materials, where the discontinuity of the system is can lead to fundamental inaccuracies in a continuum representation. [78, 75].

The goal of this work is to address some of the inherent difficulties in the computational and software-architectural aspects of the method, and to provide a high-performance, extensible framework for state of the art research in DEM applications and methods. This chapter gives an overview of the Discrete Element Method, providing insight into current methods, useful tools, and computational obstacles. Part I covers high-performance contact detection algorithms for Discrete Element computation, presenting a new algorithm that scales linearly in the number of bodies, and is appropriate for simulations with distributions of object sizes. Part II presents an architecture for DEM simulation designed with a special emphasis on extensibility, and clear design.
1.1 The Discrete Element Method

The Discrete Element Method (DEM) denotes a set of numerical modeling techniques which takes as its base assumption the discontinuity between its elements, and whose main emphasis is on the solution of contact between those bodies. In general, a DEM simulation involves a number of interacting bodies which undergo large displacements and rotations. The method has been used with rigid and deformable bodies in continuous, discontinuous[27, 95, 94], and fracturing[35, 72, 51] configurations, and supports a wide range of material laws, physical behaviors, and geometries.[57]

The method has evolved from early work in a number of disciplines including physics of particles [9, 45, 44, 37, 38], geomechanics [14, 26, 16, 17, 15, 31], and structural engineering [70, 43, 53]. Its theoretical background is founded in the Finite Element Method [11], and the Finite Difference Method. A full theoretical development of the method can be found in [57].

1.1.1 Contact Penalty

While the Discrete Element Method has been used in conjunction with continuous models to simulate internal deformation of the bodies, we are concerned with the general DEM concept, and will consider a more basic form in which each body is associated with a fixed geometric extent and an independent position and orientation (see figure 1-1). The boundary deformation is modeled using a contact penalty. In the case of continuous bodies, the penalty function represents a compatibility constraint on the continuity of the aggregate geometry. In the case of discontinuous bodies, on the other hand, this penalty function represents an approximate model of impact-induced boundary deformation in the bodies.

One promising area for application of the Discrete Element Method is the study of granular materials [63, 74, 93]. In this case, the goal is to model the material as an aggregate of primarily discontinuous bodies. The inter-body forces can be seen as modeling the micro-scale deformation properties of the material grains. Figure 1-2 shows the use of rigid elliptical discrete elements to simulate the behavior of a soil
Typically, when modeling discontinuous media such as granular materials, the contact is penalized with a linear elastic restorative force, as well as viscous damping, and a frictional shear force. Figure 1-3 shows a schematic view of the individual element-to-element contact model. In order to track the history of the frictional force, the contact penalty is resolved incrementally as follows, where \( f_n \) and \( f_s \) give the penalty force in the normal and shear directions, \( u_n \) and \( u_s \) give the normal and shear displacements, \( k_n \) and \( k_s \) are the normal and shear stiffnesses, and \( \beta \) is the stiffness proportional damping.

\[
\begin{align*}
\delta f_n &= k_n \delta u_n + \beta k_n \delta u_n \\
\delta f_s &= k_s \delta u_s + \beta k_s \delta u_s
\end{align*}
\]

The equations of motion are integrated using an explicit scheme, as the dynamic nature of the applied loads can only defined explicitly.
1.1.2 Geometric Representation

As with internal constitutive relations and contact laws, the Discrete Element Method supports a wide array of geometric representations. These range from simple spheres to arbitrarily complex polyhedral geometries.

The boundary representation, in which the geometry is defined by boundary elements (i.e. edges or facets) has been used in several discrete element simulation systems [35, 73]. This method is capable of representing complex geometries, but is inefficient in its most basic form. A naive algorithm requires $O(M^2)$ computations to find the intersection of two bodies with $M$ facets or edges. Better performance can be achieved through indexing strategies. For example, the geometry can be partitioned using a tree structure to achieve $O(M \log M)$ performance.

Williams and Pentland proposed a superquadric scheme in which element geome-
Figure 1-3: Schematic View of the Contact Law: normal and share relations

tries are defined by a superquadric function \([88, 90, 89]\). Figure 1-4 shows a collection of two dimensional superquadric elements. This method has the advantage of providing a simple test for point inclusion. That is, the value of the superquadric function, evaluated at the point of interest reveals, in its sign, whether the point is inside (positive) or outside (negative) the body. Thus, if the body surface is sampled with \(M\) points, the intersection test can be carried out in \(O(M)\) computations. Repeated evaluation of the superquadric function is, however, costly, especially if the number of test points is large (i.e. when the surface sampling is very detailed).

Drawing on the functional representation of the Williams and Pentland scheme, Williams and O’Connor proposed a discrete function representation scheme (DFR) \([85, 86, 91, 56, 55]\). In this scheme, the discriminant function is defined on a discrete grid. The evaluation of the function at a given test point is then reduced to a hashing operation to locate the appropriate grid square, and an interpolation of the corner points to evaluate the function. For a body with \(M\) facets, the DFR contact resolution requires \(O(M^{1/2})\) operations. Apart from the performance benefits over the analytical superquadric formulation, the DFR scheme is capable of representing arbitrarily complex bodies, such as the one shown in figure 1-5.
Figure 1-4: Two-Dimensional Superquadric Elements in MIMES

Figure 1-5: Discrete Function Representation Example [56]
1.1.3 Contact Detection

The DEM simulation software must automatically detect and resolve, at each timestep, all of the contacts between the bodies in the system. This process, called contact detection, is the heart of discrete element simulation, and is also its most computationally intensive proposition. Specific algorithms for contact detection are discussed in part I. Here we simply note the main challenges for contact detection.

The process of contact detection can be divided into two separate algorithms, neighbor searching and geometric resolution (see Figure 1-6). The goal of the neighbor search is to identify and list objects within a certain neighborhood or zone around a target object. The resulting list is often called the neighbor list. The geometric resolution phase compares the target object geometry against the geometry of the objects in the neighbor list in detail. The computational cost of geometric resolution depends only on the complexity of the geometric representation (for example, the DFR scheme of [54, 56] requires $O(M^{1/2})$ computations if there are $M$ facets in the geometric representation). The cost of neighbor searching, on the other hand, depends on the total number of objects in the simulation and therefore has the potential for poor scaling; in a naive implementation, where every object is checked against every other object, the cost is $O(N^2)$ operations. Because the number of objects treated in discrete element simulation is often large, neighbor searching can become a computational bottleneck. The development of efficient neighbor search algorithms is thus crucial to the overall performance of the simulation.

The separation of neighbor searching from geometric resolution allows the generic problem of finding the intersections of a set of geometric regions to be addressed without reference to the details of local geometry. The separation, which is achieved through the use of simple bounding volumes, ensures that the neighbor search algorithm will remain valid for any geometric representation and resolution scheme chosen. Indeed, once separated from the specifics of discrete element geometries and numerical schemes, neighbor searching becomes a general spatial reasoning problem. The solutions to the neighbor search problem advanced in this thesis, therefore, bear
application to a wide variety of fields including computer graphics and distributed coordination and planning.

The robustness of the neighbor search algorithm to variations in underlying geometry is essential in the development of a generalized DEM simulation system, where any number of geometric representations may be used concurrently. The implications of allowing arbitrary geometries to coexist in the system, however, must not be ignored. If the neighbor search algorithm is designed with a specific geometry in mind, its performance may degrade when it is used outside that restricted setting [61].

1.2 MIMES

The MIMES project at MIT IESL [63] began as an attempt to bring user-oriented interactivity to discrete element simulation. The combination has proved very successful, and has fostered significant research in applications as well as algorithms and computational models for DEM. By creating an environment that is interactive and graphical in interface, MIMES allowed users to interact with the simulation in new ways, and to use the tool as an experimental laboratory, for qualitatively investigating multi-body physics. A screen-shot of the MIMES interface is show in figure 1-7.

Prompted by the initial successes of MIMES, and building on the object-oriented design of the MIMES architecture, the project went on to implement a wide variety of extension capabilities, adding functionality to the base environment in new,
unplanned ways, leading to many advances in the methods of discrete element simulation.

In various versions, MIMES has included support for electrostatic forcing, time varying loads, finite-element fluid forcing, both FEM and BPM internal deformation models, rigid assembly elements, source and sink elements, and four distinct geometric types. While possible in more traditional systems, it is unique interactivity of MIMES that makes it so appealing for experimental development, and an attractive test-bed for new ideas. The ease of use and interactivity has also fostered research in models for contact, bonding, and cloning, and object sources and sinks[60]. Furthermore, the demands of a user-friendly system have prompted research into robust methods for contact detection, resulting in the high-performance algorithm developed in part I.

As a demonstration of the power and generality of such a system, several example simulations are shown in figures 1-8 through 1-12. Figure 1-8 shows an investigation of fracture during unconfined compression in a bonded sample. The elements are bonded using a point-to-point cohesion model, and are being loaded with uniform
strain. Figure 1-9 shows a simulation of particle mixing in a rotating drum, where the drum is composed of a rigid assembly of quadrilateral elements, and the particles have a distribution of shapes and sizes. Figure 1-10 shows the formation of a standing wave phenomena in a granular assemblage being subjected to vertical vibration through the boundary container [98]. Figure 1-11 shows a simulation of bore-hole break-out, a known fracture phenomena in oil-well bores. Finally, Figure 1-12 shows a simulation of hydraulic fracturing of a well-bore formation, using an extended object source to simulate the fluid loading[20].

The architecture developed in part II is intended to draw from the strengths of the MIMES project, but to avoid some of the pitfalls that were encountered. The most important aspect of the MIMES project to date, in terms of its contribution to discrete element research, has been its use as a test bed for advanced or experimental methods. While the architecture of MIMES proved to be suitable for extension development, it was not initially designed for such dramatic reconfiguration. This has, over the years, led to increase in the complexity of the MIMES core, and necessitated overhaul of some of the core infrastructure. Without a completely new core architecture, however, MIMES extension becomes increasingly difficult. Using a combination of newer technologies, and a design oriented to extension development form the outset, the new DEM Architecture proposes to take the MIMES concept of a computational laboratory even further.
Figure 1-8: MIMES: Unconfined Compression in a Bonded Sample[62]. Shading indicates breakage in the cohesive bonds.
Figure 1-9: MIMES: Particle Mixing in a Rotating Drum. Objects are shaded according to the magnitude of their velocities.
Figure 1-10: MIMES: Standing Waves in Particles on a Shaker-Table[98]
Figure 1-11: MIMES: Bore-Hole Break-out[20]. Objects are shaded to show breakage in the cohesive bonds.
Figure 1-12: MIMES: Hydraulic Fracturing[20]. Objects are shaded to show breakage in the cohesive bonds.
Part I

Algorithms
Chapter 2

Neighbor Searching

As outlined in chapter 1, neighbor searching is one of the most computationally intensive parts of a DEM simulation. This is because of the potentially large number of elements involved, and the inherent scaling difficulties of the underlying spatial reasoning problem. This chapter presents an introduction to the problems of neighbor searching. Chapter 3 presents a review of existing neighbor search methods, with some insights into their comparative benefits. In chapter 4, CGrid, a new high performance grid-based algorithm that is designed to support objects with arbitrary extent is developed. Chapter 5 presents the results of performance tests in which CGrid is compared to the NBS algorithm [48].

The interface between local geometry and global neighbor searching is the bounding volume (see Figure 2-1). By acting on the bounding volume rather than the local geometry, the neighbor search algorithm is able to treat all geometric representations in the same, simplified way. This shifts the focus to the difficult spatial reasoning problem, and makes explicit the notion that the details of local geometry have been suppressed. While any bounding volume could be used, the sphere is the most common, and the one chosen for this implementation. The sphere is represented simply by a position and radius, and is rotationally invariant, so that the extent (radius) need only be computed once for each object for the whole simulation.

The neighbor search algorithm uses the bounding volumes to determine a contact neighbor list (a conservative set of contact candidates) for each object. The complete
set of candidate pairs, \( C \) determined by the neighbor search algorithm consists of all of the pairs (target, candidate) taken from the contact neighbor lists with duplicate pairings removed. The hypothetical ideal neighbor search algorithm returns a candidate set \( C_{\text{exact}} \) that is exact for the underlying bounding volume in the sense that no pairs are included whose bounding volumes do not overlap. Since the size of the contact neighborhood does not depend on the number of objects, we can say that the size of \( C_{\text{exact}} \) is proportional to \( N \).

The obvious goal of neighbor-search algorithm design is to minimize computational costs. In order to achieve better than \( O(N^2) \) performance, real implementations employ simplifying schemes that quickly identify a conservative approximation to \( C_{\text{exact}} \). The conservative candidate set \( C \) may include candidate pairs whose bounding volumes do not intersect, but must include every pair whose bounding volumes do intersect. Any such conservative approximation \( C \) can be reduced to \( C_{\text{exact}} \) by simply applying the exact intersection test for the bounding volume each member, and in practice this step is almost always expedient. For the purpose of analysis, however, the generalized neighbor search algorithm is considered to include no calls to the bounding volume intersection check, and the geometric resolution phase is considered to begin with such a call. This division underscores the compromise, inherent in any neighbor
search algorithm, between minimizing computation time $T$, and minimizing the size of the candidate set $C$. These goals are called speed and accuracy, respectively. In most implementations of neighbor searching, and all of the implementations considered here, $C$ is proportional to $C_{\text{exact}}$ (and thus $N$), but $T$ could scale as poorly as $N$ squared. This means that for arbitrarily large simulations, speed will be the dominant consideration. Within a finite range of simulation sizes, however, accuracy must also be considered.

As discussed in [61], the underlying geometry and size distribution of the simulation objects may influence the algorithm performance. The algorithm presented here, called CGrid, was developed out previous work [61, 59] on neighbor searching for MIMES. In order to support inexperienced users and to cope transparently with arbitrary simulations, the algorithm is designed to be robust to variations in geometry and object size, and to decouple algorithm performance from rare-case objects.
Chapter 3

Review of Existing Methods

Contact detection, in general, has been long recognized as the major computational obstacle in Discrete Element Simulation [54, 73]. Much of the previous work in the area of contact detection has focused on improved geometric representation and resolution algorithms [54, 56, 88, 77, 13], or on integrated neighbor search and geometric resolution schemes [50, 71]. On current scales, where simulations regularly include thousands of objects, $N$ can be considered much larger than $M$. In this case, the neighbor search algorithm is specifically dominant. Apart from lack of generality, integrated approaches suffer from too much local detail. The increased complexity of resolving the details of contact detract from the algorithm’s ability to efficiently address the global-scale problem of neighbor searching.

Body-based methods[57] also figure widely in early references on discrete element computation. In body-based methods, the objects track a body-centered neighborhood of nearby objects. This body-centered neighborhood is assumed to contain all of the objects which can impact the pivot object in the next few timesteps. In addition to inviting the catastrophic possibility of contact omission, body based methods do not solve the spatial reasoning problem, but rather defer it for longer intervals.

The pressures of the increased simulation size allowed by modern computational resources has led to a shift away from these locally-focused methods of contact detection. Recent schemes have been proposed which better address the principal issues of neighbor searching. These include some tree-based methods, sorting methods, and
grid-based methods. Of these, grid-based methods hold the most promise for future use, since they scale linearly in simulation size.

### 3.0.1 Tree Methods

Tree-based neighbor search methods index the set of regions represented by the object bounding volumes using a tree structure. The hierarchical nature of tree should be able to provide performance of $O(N \lg N)$. Several tree-methods exist, including digital trees, hex trees, k-d trees, and R-trees. Of these, only R-trees are well suited for the representation of regional data (i.e. objects with geometric extent). Other tree methods for intersection tests are typically derived from simpler point-data methods. The extension to regions typically involves an integration of the neighbor search and geometric resolution phases, with the associated problems. Two examples are the tree-portion of the BSD algorithm of Munjiza et al.[50] and the quad-tree implementation of Wensel and Bicanic[71].

In the BSD binary tree, the data points of the underlying polygonal geometric representation are entered individually into a simple BSP (binary space division) tree. The pivot geometry is then used to query the BSP tree starting at the root. This scheme is clearly dependent on the level of detail used in the local geometry, and does not generalize easily to non-polygonal representations. The algorithm performance is best characterized as $O(NM \lg NM)$, or $O(NM(\lg N + \lg M))$, which represents a significant constant multiplier for geometric representations of any complexity.

The quad tree[71] uses a classic quad tree to subdivide space so that leaf nodes either overlap less than two objects, or are sufficiently small. In the case of the sufficiently small nodes, the overlapped objects are checked for contact with each other. This algorithm also suffers from entanglement with local geometry. The problem is clear in figure 3-1 where the tree grows inordinately complex trying to decide where, exactly, the two discs overlap.

R-trees are often used for spatial indexing problems in database applications because they are highly general, and well suited to the representation of regional data. In general, an R-Tree is a balanced tree structure in which each node has an n-
dimensional rectangle data value. The rectangle of non-leaf nodes is defined to encompass the regions occupied by all of the child nodes. In this way, the data rectangles are classified into a tree which is easily queried for data regions intersecting a given query region. Figure 3-2 shows an R-Tree for a small set of rectangles.

While very general, R-trees are difficult to build. This complexity arises from the freedom allowed in constructing the tree, or more specifically in splitting over-full nodes. Valid trees can be constructed that are highly sub-optimal. It is possible, for
example, to build a valid tree in which all of the non-leaf nodes occupy almost the entire space. In this case, the tree provides worse performance than an exhaustive search. Methods for building well-formed R-trees abound, but all require significant computation (typically quadratic in order) [29, 12, 28]. The objects in a DEM simulation are rearranged at each timestep, so the tree must be reformed at each timestep. Since it is not clear how the tree from the previous timestep could be efficiently updated, it is essentially necessary to reconstruct the entire tree every timestep. For this reason the R-Tree is not particularly well suited to DEM simulation.

The query performance for the RTree algorithm is difficult to assess since it depends on the tree quality. If, however, the tree is well constructed, an individual object query should require $O(\lg N)$ operations. Thus the overall query performance, for a well-constructed tree, should be $O(N\lg N)$. If a suitable construction or update method were developed, this level of performance could be attractive given the generality of the R-Tree formulation. For truly large simulations, however, the linear performance of grid-based algorithms will always prevail.

### 3.0.2 Sorting Algorithms

Sorting algorithms attempt to index the set of object-extents by sorting their projections along one or more axes. This approach has the distinct advantage of being straightforward, and adaptable to objects with a distribution of sizes. Unfortunately, the projections do not capture the local intersections well, and so the intermediate candidate sets derived from each axis of projection are large, and grow with increasing numbers of objects. In order to obtain a more accurate candidate set, the intermediate sets must be intersected, and this operation does not scale linearly. Sorting methods derive from point based methods, and early implementations [69] for regional objects were unwieldy. A more recent example of a highly optimized sorting algorithm is DESS.

The DESS algorithm [61] indexes both the upper and lower bounds of each object. The projected upper and lower bounds are sorted in lists for each of the primary spatial axes. The sorted lists are used to determine so-called projection candidates
for each object. The sort and rank procedure can be performed in (nearly) linear time, with limiting assumptions, but the intersection of the projection candidate lists requires $O(N^{2/3})$ operations because the lists are unordered, so their intersection must be found exhaustively. This is shown graphically in figure 3-3, where the shaded area represents the area of object extremities contained in each of the projection candidate lists for the central pivot object.

Despite the inherent drawbacks of sorting-based algorithms, DESS performs well for moderately sized simulations. For such simulations DESS is comparable to, or outperforms high performance, linear algorithms[61]. It also has the distinct advantage of being insensitive to object size. These factors have made it the method of choice in the MIMES project. For large simulations, the scaling costs outweigh the constant order benefits, and a grid-based algorithm is recommended.
Grid-based algorithms

Grid-based (also called hashing or bucketing) algorithms, which have been in use for some time in large simulations, are based on the assumption that each object can be approximated by a bounding sphere superimposed on a grid-like discretization of space. In the following discussion, the term cell is used to refer to the area of space associated with each grid-point. The term row is used to refer to a one-dimensional line of cells. In three-dimensional discussion slice is used to refer to a two-dimensional discrete slice of space. The general term bucket is used to refer to any collection of objects associated with a given grid point along a given axis, regardless of the number of free ordinates along the other axes.

If the grid spacing used is at least as wide as the widest bounding sphere, each object in the simulation can be associated with exactly one grid cell. Only objects associated with adjacent cells can then come into contact, so the fixed spatial relationships of the grid can be used as a surrogate for the transient relationships among the objects. An example is shown in Figure 3-4. Since the assignment of buckets is a simple rounding procedure that can be achieved in \( O(N) \) operations, the grid offers a way to capture the local contact relationships that is both simple and efficient.

The typical algorithm can be summarized as follows: In one pass over the objects, they can be divided into buckets according to their discrete ordinates along a given axis. Each of the lists of objects associated with those buckets can then be divided on another axis. By subdividing the pivot bucket, as well as its neighboring buckets, in each dimension, all of the contacts can be evaluated using a small number of bucket arrays for each dimension. Because each object is only visited a constant number of times, the computational cost of the whole algorithm is \( O(N) \). Figure 3-5 shows the assignment of objects into rows along the \( y \) axis (represented by the shaded boxes along the left), and subdivision of two adjacent rows into individual cells along the \( y \) axis. Note that, following the convention of NBS[48], the objects are subdivided along the \( y \) axis first, and then along the \( x \). The contact mask, visible at the center of the figure, is used to identify neighbor cells for the target cell in the target row.
Figure 3-4: The discrete grid is used to resolve the object neighborhood

Figure 3-5: Bucketing Example
The obvious advantage of grid-based algorithms is the linear scaling in $N$. The disadvantages of the grid approximation, however, make them somewhat sensitive to the conditions of arbitrary simulations. In a simple implementation, the algorithm visits each bucket in order, even if it is empty. This means that performance depends on the size of the problem domain, and not just the number of objects. This can pose problems in simulations where the domain is inherently sparse, or where it grows with time. Figure 3-6 shows an example with both isolated and regional sparsities. These sorts of sensitivities make the algorithm too difficult and unpredictable for use in arbitrary simulations.

The NBS algorithm [48] overcomes the performance degradation due to sparse simulations through careful bookkeeping, and object-based traversal of the bucket lists. The NBS loop is given in pseudo-code below. Instead of visiting each row in order as is done in the simple bucketing algorithm, NBS uses the (unordered) object
list to traverse the row array. This is done by looping over the object list, identifying
the rows associated with each object. If the given row has not yet been visited, it
is subdivided into the array of cells. Within each row, the (again, unordered) list of
member objects is used to traverse the appropriate cell array. Each bucket (row or
cell) has a visited flag, which is used to keep track of whether that bucket has already
been processed. Following this procedure assures that all of the occupied buckets are
visited, and that only occupied buckets are ever visited.

```plaintext
foreach obj in globalObjectList {
    iy = discreteY(obj);
    yBuckets[iy].append(obj);
    ybuckets[iy].visited = false;
}

foreach obj in globalObjectList {
    iy = discreteY(obj);
    if (yBuckets[iy].visited)
        continue;
    yBuckets[iy].visited = true;
    foreach obj in yBuckets[iy] {
        ix = discreteX(obj);
       xBuckets[0][ix].append(obj);
        xBuckets[0][ix].visited = false;
    }
}

foreach obj in yBuckets[iy-1] {
    ix = discreteX(obj);
    xBuckets[1][ix].append(obj);
}

foreach obj in yBuckets[iy] {
    ix = discreteX(obj);
    if (xBuckets[0][ix].visited)
        continue;
    xBuckets[0][ix].visited = true;
}
```

40
Another disadvantage of existing grid-based algorithms, which is not addressed in the NBS algorithm, is that the discretization itself is dependent on the maximum object size. Specifically, smaller objects are treated as if their potential contacts lie in a
grid-centered neighborhood three times as wide as the largest object. This decreases the accuracy of the algorithm, causing a greater degree of over-reporting (see Figure 3-7). It is especially problematic if, for example, the object sizes are distributed probabilistically (see figure 3-8). In this case, the largest object is essentially an outlier, with the vast majority of objects being significantly smaller. This would result in widespread over-reporting poor performance. If the performance degradation is significant enough, the algorithm, while continuing to exhibit linear performance, may be outperformed over some range of N by more general but higher order algorithms[61].

An even more dramatic outlier case can come from problem-specification. When compacting a sample of particles between plates, for example, it is likely that the uninformed user will specify the plates as very long rectangular elements, as shown in figure 3-9.(Note that a more expert user could take advantage of the rigid assembly
Figure 3-8: Effect of object size distributions on NBS grid

feature of MIMES to create a rigid collection of small rectangles that could be used as a single boundary). This will result in outlier objects which are as large as the simulation itself. The NBS grid is virtually useless in such a situation, since the whole simulation will be encompassed in one or two grid-squares. For such a problem, where the out-of-norm object size ratio is dependent on the simulation size, the performance will not be linear, but rather $O(N^2)$. Note that performance is also super-linear if the object size distribution is strictly defined as probabilistic.
Figure 3-9: Simulation with long boundary rectangles
Chapter 4

CGrid: Neighbor Searching for Objects with Arbitrary Extents

Large simulations require efficient scaling, but nonuniform object sizes adversely effect existing grid-based algorithm performance. The situation is particularly difficult in simulations involving distributions of object sizes, and in situations where boundary objects are extremely large (see section 3.0.3). What is needed is a grid-based algorithm that permits objects to cover more than one grid-point, effectively decoupling the algorithm performance from such out-of-norm objects. The algorithm developed in this chapter, CGrid, achieves this objective. By decoupling the algorithm from the largest objects, robustness to outliers is increased. Furthermore, by tailoring the grid-spacing to smaller objects, more pervasive over-reporting is significantly reduced, and performance is improved.

4.1 SGrid: Generalized Bucketing

The CGrid algorithm is essentially a generalization of the Bucketing algorithm that supports objects with arbitrary discrete extents. The method is developed by separating the assumption of fixed extents from the subdivision process. To that end a generalized formulation of the contact mask, as well as the subdivision process is first developed while maintaining the assumption of fixed extents. Both the contact
mask and subdivision procedure are presented in a formulation for arbitrary dimension to fully expose the subtleties of the algorithm at all levels. It should be noted that the arbitrary dimension formulation represents a significant development in its own right as only a two dimensional formulation is given in [48], and the extension to three dimensions involves a number of difficulties not represented in two. In the interest of describing the algorithm as a unit, the general Bucketing formulation is first composed into a fixed-extent algorithm called SGrid, which is then extended in section 4.2 to the complete CGrid algorithm.

4.1.1 SGrid Contact Mask

In two dimensions, The contact mask used in NBS (shown in figure 4-1) only covers two grid-spaces in the first subdivision dimension (y by convention). Thus it only references two concurrent rows of objects. Its extension to three dimensions (figure 4-2), however, proves somewhat unwieldy, covering three grid-spaces in the second dimension. Extensions on further dimensions cover three grid-spaces on every axis but the first. This increases memory costs, and introduces unwanted complexity in the inner loop. Furthermore, the mask itself becomes increasingly complex and difficult to implement.

A minor adjustment to the NBS mask yields a simpler formulation. Instead of
Figure 4-2: 3D extension of the NBS Contact Mask

Figure 4-3: SGrid Contact Mask (2D)
masking the contact for a specific pivot cell, the SGrid mask resolved contact relations *among* the current grid-square and its three lower-adjacent neighbors. A two-dimensional example is shown in figure 4-3. The NBS lookahead check (from the pivot to the lower-right adjacent cell) is simply delayed a step, and performed as a cross check between the two partially current cells immediately adjacent to the current cell. The same intersections are found using the SGrid mask, but the formulation extends indefinitely without covering more than two grid-spaces in any dimension.

Figure 4-4 shows a representative two-dimensional neighborhood, where the dark object is the pivot, and the eight lighter objects are the neighbors. The situation is, of course, schematic, and both the pivot and neighbor objects might, in a real simulation represent zero, one or several objects. The five positions of the NBS mask required to resolve all eight pairings are shown in figure 4-5. Similarly, the SGrid mask requires four positions, which are given in figure 4-6.

One subtle difference to note between the NBS and SGrid masks is the question of when to apply the mask in the case of unoccupied cells. In NBS, the rule is simple. The mask is applied at every position where the pivot cell is occupied. Because the SGrid mask does not refer to a single pivot cell, the occupied positions of the SGrid mask are defined differently. The mask is applied at every position where any cell that is current on the last axis is occupied. Examples of how the mask applications change for NBS and SGrid when some of the neighbor cells are unoccupied are shown in figures 4-7 and 4-8. Further detail about occupied buckets is given in section 4.2.3.
Figure 4-5: Five Positions of the NBS Mask in the Pivot Object’s Neighborhood

Figure 4-6: Four Positions of the SGrid Mask in the Pivot Object’s Neighborhood
Figure 4-7: Three Occupied Positions of the NBS Mask in the Pivot Object’s Neighborhood

Figure 4-8: Three Occupied Positions of the SGrid Mask in the Pivot Object’s Neighborhood
The cross-checks can be easily extended to any dimension by addressing the various positions of the mask as either current or lower-adjacent with respect to each axis, and cross-checking intersections for positions which are not labeled lower-adjacent on the same axis. Figure 4-3 shows the addressing encoded as a bit-string, with one representing lower-adjacent on each axis, and zero representing current. The general rule then reduces to a bitwise NAND of the two strings. Note also, that the bit-string label also clarifies the definition of an occupied position of the mask; a given position is considered occupied if there is an object in one of the mask buckets with a zero as the most significant bit.

4.1.2 SGrid Subdivision

The NBS subdivision algorithm, as presented in [48] proceeds as follows. In one pass, all of the objects are distributed into buckets corresponding to uniform rows of space in one dimension. Taking a given row, the objects which were assigned to that row are distributed into another array of buckets divided along the second dimension (cells). By maintaining two adjacent subdivided rows at a time, and using a local contact mask as shown in figure 4-1 on each pivot cell, half of the pivot’s contact neighborhood is always accessible, and all of the contacts can be determined in one complete pass.

In a more abstract sense, the process involves a series of self-similar subdivision stages (see figure 4-9). For convenience in the generic definitions, the SGrid and CGrid subdividers are associated with the axes in their numeric order (x, y, z). Focusing just on the current bucket, without considering the adjacent buckets, the NBS algorithm can be recast in terms of a recursive series of individual subdivision stages. To make the division more concrete, we will consider each stage to be carried out by a separate Subdivider object, which is associated with a given subdivision axis. The process reduces to the simple recursive operation: Some source set of objects is fed into a Subdivider (see figure 4-10). The objects are arranged in an array of buckets corresponding to a discretization of space along the associated axis. The objects in each bucket are fed, in turn, as source objects into the next Subdivider. At the
deepest level of recursion, the current bucket is an individual cell in space. The first object source is the global set of simulation objects, and the last object destination is the contact mask.

The adjacent buckets add a minor complication. Noting from the discussion of the mask in section 4.1.1 that only the current and lower-adjacent buckets are needed, the complete formulation can be developed as follows. The first \((x)\) Subdivider behaves as outlined above, forwarding a single set of objects corresponding to one bucket along the first axis. In the second \((y)\) Subdivider, the buckets in the current array are held over for a second run as adjacent buckets, just as, in simple Bucketing, the most recent row is held over to act as the adjacent row. (Note that the object-based bucket-array traversal of NBS prohibits reuse of the adjacent row, and thus must subdivide both current and adjacent rows). At the end of each pass of the second subdivider, the current bucket array is shifted over into the adjacent bucket array, and new objects are accepted in the current array. The second subdivider has, therefore, two sets of objects to forward at each grid-point, one current, and one adjacent. By extension,
these two sets are doubled in the third (z) subdivider, with two current arrays getting shifted into lower-adjacent positions at the end of each pass. The four arrays in the third subdivider correspond to the four possible combinations of current and adjacent on the first two axes axis. The recursion continues, doubling the number of arrays at each subdivision stage, until the contact mask is reached. Figure 4-11 shows the SGrid3D subdivision process including adjacent buckets.

As indicated in the discussion above, the contact mask can be seen as a specialized form of subdivider, in which the objects are checked for contact, rather than subdivided and forwarded. In this light, the addressing scheme used in section 4.1.1 on the contact mask takes on even clearer significance. The bit strings are used to encode a hierarchical adjacency, with the most significant bit designating current or adjacent in highest subdivider, and the lower-order bits designating adjacency in the lower
subdividers. To reiterate, at the end of a subdivision run, during the array-shift, all of the current arrays on the last axis (the first half of the set of arrays) are shifted into the corresponding adjacent arrays. For convenience, the first half of the set of arrays is referred to simply as the subdivider's current half, and the other arrays as the adjacent half. This concept extends to the mask itself, for which the first half of the mask cells (the ones with zero as the most significant bit) are referred to as the current half.

Figure 4-12 presents, more formally, the complete typical SGrid subdivider, associated with the $s$'th axis (starting at 1). At each subdivision stage, the $s$ parallel sets of input objects are subdivided along the appropriate axis. The $s$ sets corresponding to the current subdivision grid-point are forwarded to the next stage, and subdivided into the next subdivider's current half. Before the next subdivider returns from pro-
cessing, it evicts the adjacent half, and shifts the current half into adjacent positions. Control returns to the calling subdivider, which forwards the $s$ corresponding to the new current grid-point. It should be noted here that the object-based bucket traversal used in NBS could easily be accommodated in the SGrid formulation; the object list for each bucket in the current half would be have to be traversed, and the visited flag would be associated with the grid-points (and not the individual buckets). For CGrid, however, this is not feasible (see section 4.2.3, so the SGrid formulation presented assumes that the buckets are traversed in order.

### 4.1.3 SGrid: putting the pieces together

The above discussion can be assembled into a fixed-extent algorithm, called SGrid for Subdivider-Grid. Figure 4-13 presents the stage-wise operation of the 3D SGrid algorithm, showing both the schematic object volumes and the subdivider arrays.
the $x$ Subdivider, all of the objects are divided into grid-spaces along the $x$ axis. The set of objects at the first grid-point is then forwarded to the $y$ Subdivider. The $y$ Subdivider divides its input objects along the $y$ axis, and forwards the subdivided sets, one at a time, to the $z$ Subdivider. The $z$ Subdivider subdivides and forwards its objects to the contact mask.

When the processing at the contact mask is done, the objects in the first four buckets of the contact mask (the current half) are moved into the corresponding buckets in the adjacent half. This prepares the mask for the next grid-point in the $z$ Subdivider. Control returns to the $z$ Subdivider, which forwards the four sets of objects for the next $Z$ axis grid-point to the contact mask. When the $z$ Subdivider has traversed its entire $Z$ axis, the current half (consisting of the first two arrays) is shifted into the adjacent half (which is first cleared). Control is returned to the $y$ Subdivider. In the $y$ Subdivider, the two sets corresponding to the next grid-point are forwarded back up the pipeline, and the processing recurses as before. At the end of each pass of the $y$ Subdivider, the current half is shifted, and control returns to the
$x$ Subdivider. Once the $x$ Subdivider traverses the entire $x$ axis, the run is complete, the entire discrete space has been traversed, and all of the contacts are found.

### 4.2 CGrid: Supporting Arbitrary Extents

The SGrid formulation, while useful in its own right, has been developed here for the purpose of extension to objects with arbitrary discrete extent. This will require a few more simple transformations. First, the discretization model must be modified to represent arbitrary extents. The adjacency model must be then generalized further to hold-over objects for several subdivision runs. The result is an arbitrary-extent grid-based algorithm called CGrid.

#### 4.2.1 Discretization of Arbitrary Extents

NBS arranges the objects in subdivision cells by the location of their centroids. When the distance from the centroid to the bounding surface is fixed, as is the case with identical bounding spheres, the centroid provides a very efficient encapsulation of the bounding surface. The contact neighborhood around any centroid is simply the neighborhood of directly adjacent cells. For objects which cover an arbitrary number of cells, however, the centroid does not lend itself to use as a discretization point. With respect to the discrete grid, the bounding surface is taken to be a d-dimensional axis-aligned bounding rectangle. As in the DESS algorithm [61], the extent of an axis-aligned bounding rectangle can be easily encoded in the pair of upper and lower bounding points, corresponding to the corners of the rectangle.

Drawing on the treatment of bounding coordinate pairs used in [61] the objects can be arranged in grid-spaces according to the discretization of their lower bound. With these assignments, each object must be checked against any objects between its own (i.e. lower-bound) cell and the cell of its discretized upper-bound. If this method is applied to the identical spheres approximation (see figure 4-14), we can see that the resulting arrangement is identical. The shift of the discretization point to the lower extent is insignificant, because the shift is identical for every object. Thus,
the centroid-based discretization can be seen as an optimization of the bounding coordinate pair for fixed-extent objects.

4.2.2 Current and Collector

Applying the SGrid algorithm directly to objects with arbitrary extents requires that each subdivision stage maintain an arbitrary number of adjacent "halves". (Obviously, in such a situation, the term half is misleading, since the set of arrays would be divided up into an arbitrary number of corresponding adjacency levels—not just two as in SGrid). This is clearly intractable. A closer examination of the role of the adjacent pseudo-halves reveals, however, that there is no significant difference between the adjacent pseudo-halves and that the objects in each are treated much the same. This observation leads to a simplification: the (single) adjacent half of the SGrid subdivider is replaced in CGrid with a collector half. When a subdivider shifts the current half into the collector half, it does not automatically evict the already present objects, but instead scans them for objects whose upper-bound does not intersect the newly current grid-space. Note that objects with arbitrary extents may cover only one grid-square (and not overlap at all into any other). The eviction of collected objects should, therefore, take place after the previously current objects are shifted in (so that they, too, are subject to eviction). Figure 4-15 depicts a 2D
Figure 4-15: The y Subdivider Membership for CGrid2D (three steps are shown)

example over three grid-point steps of the x Subdivider. The membership of the y Subdivider is shaded, dark for current, and light for adjacent.

The extension of the SGrid contact mask to arbitrary extents is carried out exactly as with the subdividers; the collecting mechanism is applied to the mask’s adjacent half. In three dimensions, the contact mask has four current cells, and four collector cells. The same contact-rule developed for SGrid is applied to detect all of the intersections exactly once. A formal correctness proof is given in section 4.3.

4.2.3 The Occupied Bucket Queue: handling sparsity

One of the main contributions of the NBS algorithm is the object-based grid-traversal. By jumping around the grid space using the unordered object lists, NBS avoids visiting empty buckets. This makes the algorithm more robust to sparse simulations. The approach, however, is not valid for CGrid because the collector buckets depend on in-order traversal of the grid-space. The only way to avoid unoccupied buckets is to maintain a priority queue of occupied buckets, called the occupied bucket queue (OBQ). This is not as costly as it would seem, however, since the number of occupied grid-points in any dimension is much smaller than the number of objects. Furthermore, some loose continuity assumptions help to make updates of OBQ efficient.

The OBQ is an ordered list of grid-points along the subdivision axis. It indicates which grid-points have associated buckets that are occupied. In the worst-case, where
no continuity assumptions can be made, the list maintenance requires $O(N \lg N)$
operations, with the constant factor being quite small since the number of grid-
points along a given axis is much less than the number of objects. This can be
reduced to linear time as long as the simulation remains fairly reasonable; The change
between the OBQ for some pass of a subdivider, and the OBQ for the next pass is
small and constant with simulation size. The OBQ is kept during the shift/eviction
operation, and updated to include newly occupied grid-points. It is then purged of the
newly unoccupied ones. The cost of this operation is small, and is at least partially
counterbalanced by the increased efficiency of in-order traversal: in NBS, both the
current and adjacent rows need to be subdivided, duplicating the work of subdivision
once for each row. With in-order traversal, the current half can be simply appended
to the collector half, a cost equivalent to one insert to the list.

4.3 CGrid: Proof of Correctness

In the following, consider two objects, $A$, and $B$, in an $n$ dimensional space, with
axes numbered from one to $n$. Let $S_i$ designate the subdivider associated with axis
$i + 1$, for $0 \leq i < n$, and let $S_n$ designate the contact mask. The position of a given
subdivider at any time during traversal is considered to be the grid-point of its parent
subdivider’s current bucket(s). The position of the overall algorithm is considered to
be the tuple of the positions of all of the non-zero subdividers (including the mask).
Note that the zeroth subdivider has no position, and that all of the objects are always
present in the zeroth subdivider.

**Lemma 1** If $A$ is present in some $S_i$, at some position of the algorithm, then $A$ is
in every $S_j$, for $0 \leq j < i$.

In order to enter $S_i$, $A$ must already be in subdivider $S_{i-1}$. Furthermore, since $A$ is
bounded in extent, it is evicted from $S_i$ before the end of every pass of subdivider
$S_{i-1}$. Thus, $A$ must be in $S_{i-1}$ both before it enters $S_i$ and after it is evicted. By
induction, $A$ is necessarily present in every subdivider $S_j$, $0 \leq j < i$. 

60
Lemma 2 A appears in Subdivider $S_i$ at position $P$ of the algorithm if and only if $P_j$ lies within the bounds of $A$ on $j$ for all $0 \leq j < i$.

Assume that $P_k$ lies within the bounds of $A$ on $k$. Then, if $P_{k+1}$ lies within the bounds of $A$ on $k+1$, $S_k$ will forward $A$ to $S_{k+1}$ before it reaches $P_{k+1}$, and $S_{k+1}$ will not evict $A$ until after it passes $P_{k+1}$. Thus $A$ is present in $S_{k+1}$ at $P$. Induction on $k$ proves that $A$ appears in $S_i$ if $P_j$ lies within the bound of $A$ for all $0 \leq j \leq i$.

Now assume that $A$ is present in $S_i$, but that $P_j$ is not within the bounds of $A$ on $j$ for some $j < i$. If $A$ appears in $S_i$ at $P$, $A$ must also appear in $S_j$, and $S_{j-1}$ (lemma 1). Since $A$ appears in $S_j$, it must have been forwarded to $S_j$ by $S_{j-1}$ at some point earlier in the current run of $S_{j-1}$. Thus, $P_j$ must be greater than the lower bound of $A$. But if $P_j$ does not lie within the bound of $A$, it must also be greater than the upper bound of $A$. But then $S_j$ will evict $A$ before reaching position $P_j$, contradicting the initial hypothesis.

Lemma 3 If $A$ and $B$ are both in the contact mask, when the algorithm is at position $P$, then $A$ and $B$ overlap

Assume that $A$ and $B$ do not overlap. Then, there must be some axis, $i$ for which the bounds of $A$ do not intersect those of $B$. Without loss of generality, let $A$ be the object with the lowest lower bound on $i$. Since $B$ is in the mask, it must also be either current or collected in all of the subdividers $S_j$, $0 \leq j \leq n$ (lemma 1). Furthermore, if $B$ is current or collected in subdivider $S_i$ then $P_i$ the position of subdivider $S_i$, must lie between the bounds of $B$ on axis $i$ (lemma 2). But since $A$ is also in the mask, and therefore current or collected on $S_i$, $P_i$ must also lie between the bounds of $A$ on $i$. $A$ and $B$ must, therefore, overlap on $i$, contradicting the initial assumption.

Lemma 4 If $A$ and $B$, both in the mask at algorithm position $P$, are both collected in some subdivider $S_i$, then there exists some position of the algorithm, for which $A$ and $B$ are in the mask, and either $A$ or $B$ is current in $S_i$.

Without loss of generality, let $A$ designate the object with the highest lower bound on $i$. Consider the position $P'$, such that $P'_j = P_j, j \neq i$, and $P'_i$ is equal to the lower
bound of $A$ on $i$. Since both $A$ and $B$ overlap $P'$, both $A$ and $B$ are in the mask when it is at position $P'$ (lemma 2). Furthermore, since the lower bound of $A$ is equal to $P_i$, which is also the position of subdivider $S_i$, $A$ must be current in $S_i$, when the algorithm is at position $P'$.

**Lemma 5 The Contact Condition:** If $A$ and $B$ overlap, then there exists some position $P$ of the algorithm for which $A$ and $B$ are both in the mask, and for which either $A$ or $B$ is current in every $S_i$ for $1 \leq i \leq n$.

Since $A$ and $B$ overlap, lemma 2 implies that both $A$ and $B$ are in the mask at some position of the algorithm. Then, by induction on $i$, lemma 4 implies that there exists some position of the algorithm for which both objects are in the mask, and are not collected in the same subdivider.

**Lemma 6** The position $P$ in lemma 5 is unique

Assume that two distinct positions $P$, and $P'$ satisfy the contact condition. Since $P \neq P'$, then $P_i \neq P'_i$ for some $i$. Let $P$ be the position with the higher ordinate on $i$. Since both $P$ and $P'$ satisfy the contact condition, and since objects are only current for one position of the subdivider, either $A$ or $B$ must be current in $S_i$ at $P$ and the other at $P'$. Therefore, the lower bound of one (say $A$) must be equal to $P_i$, and the lower bound of the other ($B$) must be equal to $P'_i$. Now, since $B$ is in the mask at position $P$, the lower bound of $B$ on $i$ must be less than or equal to $P_i$ (lemma 2), and therefore less than or equal to the lower bound of $A$ on $i$. But since $A$ is in the mask at $P'$, the lower bound of $A$ on $i$ must be less than or equal to $P'_i$, and therefore less than or equal to the lower bound of $B$ on $i$. Thus the lower bound of $A$ on $i$ is equal to the lower bound of $B$ on $i$, and $P_i$ is equal to $P'_i$, contradicting the initial assumption.

**Lemma 7** The position $P$ of lemma 5 is visited by the algorithm

Since either $A$ or $B$ is current in $S_i$, for $1 \leq i \leq n$, $S_i$ is considered occupied at position $P_i$, and the algorithm will descend to visit the contact mask at position $P$.  

62
Correctness For every pair of intersecting objects, the contact condition is satisfied exactly once, and for every other pair, it is never satisfied.

Lemma 2 implies that only objects which intersect are ever in the contact mask together, and therefore, only intersecting objects can meet the contact condition. Lemmas 5, 6 and 7 imply that the condition is met for every intersecting pair in exactly one position of the algorithm, and that the algorithm actually visits that position.

4.4 Implementation Notes

The above discussion has been general in dimension, and made little reference to implementation details. In order to direct the reader towards a working implementation, and to better support the performance discussion in the next section, some of the choices made in the 2D C++ implementation used in MIMES are outlined.

As with NBS, the lists associated with the subdivider buckets are implemented with array based linked lists. A general description of array-based linked lists can be found in [65]. The technique is particularly suited to CGrid because the link (i.e. next) array can be shared among all of the buckets and arrays of a given subdivider because the membership of those buckets is mutually exclusive. This linearizes the memory cost for the link array. The head arrays, of course, are separate for each bucket array, but in a normal simulation, the number of grid-points along a given dimension is significantly less than the number of objects. Indeed, for an k-dimensional rectangle of closely packed objects, the memory cost of the head arrays is $O(2^k N^{1/k})$, since the number of head arrays is equal to $2^k$, and the length of those arrays will be $O(N^{1/k})$. The dominant term in the overall memory requirement is the link array, which is $O(N)$.

The OBQ is also implemented using array-based linked lists. The advantage of this method is that insertion of already present grid-points into the queue is extremely cheap; It requires only a single lookup to determine that the grid-point is already in the queue. Each queue is made up of two link-arrays (forward and backward), to
provide the functionality of a doubly linked list. This is done to facilitate insertion of new grid-points into the list.
Chapter 5

Performance

This section discusses the performance of the 2D MIMES implementation of CGrid. Raw performance is investigated using NBS as a benchmark algorithm. Sensitivity to off-normal objects is investigated using an artificial cell-size factor.

5.1 Scaling in Simulation Size

In order to demonstrate the linear scaling of CGrid, a battery of simple trial simulations was developed. The algorithm was run on a series of trial simulations in which a square block of close-packed discs is dropped on a fixed line of discs (see figure 5-1). The test was run for simulation sizes between 100 and 50000. The results of the scaling battery for CGrid are plotted against the results for NBS in figure 5-2. Due to fluctuations in the system-load, and the timing mechanism used, deviations from the otherwise linear behavior are seen for isolated runs. It is clear from figure 5-2 that NBS has the advantage for these simulations, which represent ideal conditions. It is also clear, however, that CGrid scales with the same linearity as NBS, and that the performance difference is not significant.
Figure 5-1: Test Simulation of 57757 objects with objects shaded by velocity
Figure 5-2: Performance vs. Number of Bodies
5.2 Sensitivity to Off-Normal Objects

When the objects in the simulation have varying sizes, NBS performance degrades. If the NBS cell-size is defined by some off-normal object which is $k$ times larger than the normal objects, then performance will degrade in $O(k^2)$. In CGrid, the cell-size can be selected to best suit the normal objects, but it is informative to consider the performance degradation caused by the off-normal objects. If, for instance, the simulation is made up of two widely different object sizes, there may be many “off-normal” objects. Thus, it may be useful to know how the difference between the object and cell-size influences the simulation performance on a per-object degradation basis.

In order to test the sensitivity of to off-normal objects, uniform simulations are used, with an artificially adjusted cell-size. Since the CGrid cell size can vary without regard to the actual object sizes, it is usually chosen to be the “normal” object size. By imposing an artificial cell-size on a uniform simulation, the performance loss to be expected from objects which lie off the normal object size can be approximated, while still working within the framework of reproducible, and easy to generate, uniform simulations. The results of these tests are conservatively indicative of the slowdown expected from real distributions of object sizes.

The tests were run for simulations of size 1000, and cell-size factors up to 1:20. For CGrid the tests were also run for cells smaller than the objects. The results are shown in figure 5-3. As expected, optimal cell-size for a uniform simulation is the actual object size for both algorithms. CGrid, however, shows marked improvement in the slowdown due to cell size adjustment. This is particularly true for cells that are smaller than the objects, where a ratio of 1:20 between cells and objects is associated with a slowdown of less than two. For cells that are larger than the objects, a 20:1 ratio results in a slowdown of eight for CGrid, and forty-four for NBS. The gains in large-cell performance can be attributed to the fact that CGrid takes objects which do not overlap more than one grid square into account. In NBS, these objects are checked against all eight neighboring cells, whereas in CGrid they are immediately evicted from the collector cell, and do not get checked against any objects except those
in their own cell. This effect can be seen in figure 5-4, which shows the same results on a log-log scale. It is believed that the single-cell object efficiency is responsible for relative lack of degradation below cell factors of about eight. After this point, CGrid exhibits the same quadratic scaling of NBS, but starting from a much lower base.

While CGrid is clearly not completely insensitive to object size variances, it shows dramatic increase in robustness to size variances. By carefully selecting the cell size such that the minimum and maximum objects experience the same slowdown, CGrid can expect a modest performance degradation of less than two for simulations with object size ratios up to 120.
Figure 5-3: Performance vs. Cell Factor
Figure 5-4: Performance vs. Cell Factor (log-log)
Chapter 6

Algorithm Contributions

A general formulation of Bucketing for arbitrary dimension has been presented. This formulation was extended to work with objects of arbitrary discrete extent. The resulting algorithm, CGrid, has several benefits. First, and of most value, CGrid is unconstrained by off-normal objects, since the cell-size can be selected independent of the object size distribution. This means that if highly off-normal objects are introduced, either probabilistically, or by problem design, the performance impact will be small. This represents a significant gain in the robustness of the algorithm for use in general simulation, providing greater performance reliability, regardless of input conditions. This contribution alone makes CGrid the new algorithm of choice for a user-friendly simulation system such as MIMES.

The second benefit is in the actual performance degradation seen when off-normal objects are encountered. The severe degradation seen in NBS is significantly reduced for both large and small objects. When combined with the ability to select moderate cell-sizes, this allows CGrid to deal efficiently with simulations involving widely varying object sizes. This level of robustness, coupled with an efficient linear time algorithm, ensures that any simulation which is encountered, regardless of size or composition, will perform according to the best expectations.
Part II

Architecture
Chapter 7

Introduction

The following chapters are devoted to the development of a 3D DEM Simulation architecture. The overall goal of this architecture is to provide an infrastructure for research in discrete element simulation capabilities and applications. The architecture is envisioned as providing a set of guidelines rigid enough to enforce a global consistency and robustness to change, and flexible enough to allow for real extension. The goal of this work is not, therefore, to provide particular simulation capabilities. The Architecture is furthermore not targeted to accommodate particular extensions. Instead, the range of enhancements attempted and actualized in the MIMES project is taken to be indicative of the ways in which simulation components need to be flexible.

By drawing from the challenges, and successes of extensions in MIMES, this architecture proposes to facilitate and encourage similar uses. It is expected that by providing a more extension-friendly environment will open the way for further exploration of innovative DEM capabilities.

7.1 Extension vs. Subversion

While enhancements to the MIMES simulation environment have proven successful [62, 24, 41, 20, 59, 23], it has often been through a process of architectural subversion rather than extension. The rigid assembly functionality is typical.
In order to allow a collection of elements to act as a single entity, MIMES provides a rigid assembly element type. The assembly element has all of the basic element state, as well as a member list, which holds pointers to all of the member elements. Before dynamic integration occurs on the simulation objects, the rigid assembly elements collect up the applied forces from all of their members, as well as the collective properties such as mass, center of gravity and moment of inertia. The forces are related to the collective center of gravity, and integration proceeds for all of the elements, including the assembly element with its collective properties and applied loads. The assembly then overrides the integration of the individual elements with projected motions of the collective.

This mechanism, while effective, has several drawbacks that underline the deficiencies of subversive enhancements. First, there is wasted work; The member elements should be exempt from integration, since the results of integration are ignored. Furthermore, the assembly element does not require any of the baggage that goes along with the element class; contact detection, constraints and display parameters are all nearly meaningless for this meta-element. A second drawback is that the members continue to provide support for features that have ill-defined effects. What, for example, should happen if a user decides to fix the position of a member element? And, perhaps more strangely, what will happen if a user puts one element into two rigid assemblies? These difficulties detract from the ease of use of the solution, but also from its robustness to future, unrelated extensions.

A final, and telling drawback of subversive enhancement is the incapacity for further extension. A well defined mechanism for rigid binding of simulation objects could, if sufficiently general in nature, be used to generate a wide variety of useful behaviors. For example if rigid bindings could be limited to single axes, and if elements could participate in more than one rigid binding, a series of bindings could be used to drive pistons or shaker tables, with greater predictability and ease than the current penalty-constraint based mechanisms used in MIMES. Unfortunately, since the rigid assembly system in place relies on overwriting the built-in dynamics, it is not possible to combine effects in this way.
The difficulties encountered in rigid assemblies are not limitations of the particular subversion scheme chosen, but rather limitations of the broad assumptions made in the definition element class. In this case, the assumptions are that all elements will use the same integration scheme, and that all element dynamic behavior is captured in applied forces or penalties. The notion of a rigid assembly simply does not fit this model. So, while it is possible to subvert the model, that subversion only serves to make further extension of the platform that much more difficult.

Subversive enhancements also tend to make the inner workings of the simulation increasingly opaque. Thus increasingly experienced and knowledgeable developers are required to maintain, or further extend, the functionality. As an example, because the contact detection mechanisms have been subverted for use with source and sink elements, extension of further geometry types will require a careful understanding of that process. Thus extension in MIMES has steadily raised the developer learning curve, and alienated possible extenders. The next chapter addresses the target skill set of the different classes of developers at which the DEM Architecture is aimed.

7.2 DEM Developers

The DEM Architecture is designed to minimize the effect of feature enhancement on the developer learning curve. In order to make sure that all levels of development are well-supported, the design is specifically targeted at individual classes of developers, and attempts to encourage their separate, and distinctly necessary contributions. Three classes of developers are envisioned: Core developers, extension developers, and application developers.

Core developers will be most interested in the enhancement and maintenance of the architecture itself. They can be assumed to be proficient in the languages and tools used, and are expected to be familiar with the actual class hierarchy implemented throughout the simulation, as well as the guiding principles of the overall architecture. Their knowledge of the specific extensions may be limited by exposure.

Extension developers will be most interested in the modeling aspects of some par-
ticular extension. They are not, in general, familiar with many areas of the class hierarchy, but should be expected to understand the working models for the classes which immediately impact their extension of interest. An extension developer wishing to implement a rigid assembly capability is be expected to be familiar with the dynamic integration hierarchy (Dynamics), and the generic action facility (Standard-Behavior), but is expected to know little about the contact and neighbor searching methods and facilities, for example.

Application developers will be interested in the application of the methods supported by the architecture within some problem domain. They may, within the course of their investigations, wish to implement some once-off capabilities, especially if these can be done using the scripted front-end. They are expected to be proficient in the interface language, and the provided DEM commands. They may furthermore be expected to implement simple extensions to pre-existing capabilities. For example, an application developer might wish to implement a field-potential forcing (e.g. fixed fluid gradient), and could be expected to do so by following the example of body force application (BodyForce).

Of particular importance within the university setting is the extension developer. For the DEM architecture to be a successful tool for research in DEM methods, as well as applications, it must be readily available for extension, by many investigators, and in many directions. Developers at the core level, however, require a great deal of training in the architecture itself. Furthermore, developers who are committed at the core level will be rare enough; their expertise is better utilized in the enhancement of the core architectural functions. It is not, therefore, feasible or even desirable, to expect all developers to perform at the core-level.

Application developers, on the other hand, will not have the commitment or expertise to provide extensions with lasting viability. Developing, instead, proof-of-concept, or once-off implementations, application developers will tend toward solutions which are either non-extensible or semi-subversive. While integral to its overall success, application-level development cannot support the architecture over the long term. The architecture must, therefore, be as amenable as possible to extension-level de-
development. This commitment to extensibility will help to ensure both the continued development, as well as the overall quality of the architecture as a whole.

7.3 Extensibility

Extensibility, as a goal of the DEM architecture, refers to the provisions made for extension-level development. This is perhaps the most important goal of the architecture, as extensibility is seen as the key to the long-term viability of the architecture as a simulation and research tool. Extensible tools are desirable, not just so they can keep up with advancing developments in the field, but also to promote experimentation in those development areas. This architecture is envisioned not just as a laboratory for simulation, but also, and even primarily, as a laboratory for development itself. If enhancements are to take place, it is key to the maintainability of the core that these enhancements be accommodated as much as possible, and that their development within the framework be made as easy and inviting as possible, to avoid subversion and fragmentation.

7.3.1 Generic Infrastructure

The DEM architecture, as implemented here, attempts to provide basic infrastructural components on as generic a basis as possible. These components are designed to be reusable, and inter-operable. The conventions that they require and support are documented carefully and explicitly, in order to exhaustively probe their generic applicability. The need for generic infrastructural components arises because many of the tasks of DEM Simulation require similar support structures, and involve similar tasks. The development of generic tools, specialized for the tasks of DEM simulation allows for simpler extension development.

Of course, reusable infrastructure simplifies development. It also helps to make the actual code more palatable and understandable. It should be noted that this is an important advantage in a system which is expected to be maintained and extended by many developers. It is difficult to guarantee that developers will write clear,
well commented code, but if the infrastructural components have simple and easy to understand interfaces, their use will be transparent, and will at least not obfuscate the meaning of the extension code.

Reusability in the infrastructure components also helps enforce consistency. Apart from the obvious consistency of simple duplicate usage, it is clear that extensions which make use of the same basic infrastructure components will be forced to conform to the same governing conventions. Thus the extenders are cajoled into developing interfaces which are consistent with the guidelines of the architecture.

Finally, it is worth noting that perhaps the main goal of the provision of infrastructure within the DEM architecture is to shift some of the burden of extension development off the shoulders of the extension developers, and into the hands of the core-level developers. This ensures that more attention will be paid to rare cases, exceptions, and graceful recovery, and thus promotes a greater level of robustness in the whole system, including future extensions.

7.3.2 Modularity

In the DEM architecture, a decision has been made to consistently consider the simulation and infrastructural functionality in terms of components. This decision is based on the desire to maximize extensibility. By dividing the architecture into small components, two specific goals are achieved, readability, and replaceability. The main target developers of the architecture, the extension-level developers, are not expected to have an in-depth understanding of large portions of the core architecture. Instead, they will learn the workings of the components related to their extension area.

By making the components small, and focused, the volume of code to be understood to effect a given extension is made more palatable. Replaceability is also promoted, in the sense that the self-contained components are easier to pull in and out of the architecture. The nature of the component model also promotes the explicit specification of conventional requirements and assumptions, thereby making it easier to ensure that a replacement or extension will conform to the same interface as the component it replaces or extends.
Another way in which modularity is used in the DEM architecture, is to promote multiplicity of extension. In MIMES, the element class was largely monolithic, and thus elements were constrained to be exactly one type. It was not possible, without a significant subversion of the architecture, to add new features such as sink contact behavior to the existing geometric hierarchy. In its current state, the Element is the only component in the architecture with subcomponent design. The design is considered, however, to be an important feature of the architecture, and the required support (conventions and infrastructure) is provided so that future development of other components may take advantage of the same design.

7.3.3 Abstraction

The components definitions of the DEM architecture are designed to encapsulate an abstraction of their function. Dynamics, for example, is envisioned simply as a consumer of applied loads and source of dynamic state. The architecture, at this time, provides a simple implementation of each of these abstractions, with the assumption that more complex formulations will follow. For example, implementations higher-order integration schemes, or collective dynamics, etc. are envisioned.

The goal of defining the component interfaces in abstract is to promote enhancement in the form of real extensions rather than subversions. It is only possible to extend the component definition to support new, unplanned, uses if the definition does not include implementation-specific assumptions. By explicitly defining an abstract conceptualization of a component, it is possible to make clear exactly what assumptions are being made, and what is left to implementation. In this way it is possible to identify whether a given extension will require core-level redefinition, or simply extension-level development. The abstract interface specification also generally helps to enforce and enhance the component model for the architecture, ensuring that component functions take place entirely with the component.
7.3.4 Consistency

In discussing the provisions for extensibility in the DEM architecture, consistency has been mentioned several times. Consistency of interface is a major component of the architectural strategy for extensibility and long term viability, and merits a careful examination.

By promoting consistency of interface throughout the architecture, the development learning curve is softened. This occurs on two levels. First, the base-level definitions of the core interfaces provide an infrastructure of careful definitions and motivations. The idea is somewhat like that of reusability in infrastructural components, but for the meta-information of definitions, commentary and documentation associated with the core interfaces. This means that new developers have better resources at hand for understanding the basic interfaces than they might if the various components and extensions had individually tailored interfaces.

The second way in which interface consistency softens the learning curve is by easing the transition between extension-level and core-level development. By enforcing strict consistency throughout the architecture, we make sure that the design of single components will mirror the design of the overall architecture in miniature. This provides new developers with small, palatable instances of the architecture design to learn, and ensures that the design principals and interface style will translate to the entire system. Thus each component serves as a useful introduction to the system as a whole.

Finally, it should be noted that well-enforced consistency makes core-level maintenance easier. By conforming to the conventional interfaces and techniques, extension developers can ensure that core-level developers will understand the way in which their components are intended to interact with the system, and will thus be better able to debug and maintain those interactions. without having to gain expert knowledge in the numerical workings of the extensions.
Chapter 8

Core Infrastructure

As indicated above, one of the main goals of the architecture is to provide a strong infrastructure for the development of DEM simulation components. In addition to the basic types for quaternion and vector math, and smart string like objects, infrastructure is provided for reference counting, containers, coupled cloning and polymorphic type-interaction resolution. Conventions are also developed for defining identifiers, and supporting the various infrastructural components.

8.1 Basic Types

The DEM Architecture provides some generally useful basic types for performing discrete element computation, as well as for interacting with the front-end interpreter. Here we cover the interfaces for these types, and some of the decisions for their inclusion.

8.1.1 DEM Math

Computation is at the heart of discrete element simulation, and therefore the basic math types are of great importance. The architecture defines three basic math types, Scalar, Vector, and Rotation. The Scalar is a conventional name for the basic type double, and is used to ensure that all of the DEM components use the same precision
of floating-point values. It does not seem, at this time, to be particularly useful to define a Scalar class separate from the language internal-type double, but the convention of using Scalar to specify floating point numbers would allow for that in the future. String conversion of all of the math types is handled cleanly and uniformly by the TclObj class (see below).

8.1.2 Vector

The Vector type follows the usual expectations, containing an array of 3 Scalars. The full range of operators is provided for vector-scalar operations, and addition and subtraction for vector-vector operations. Multiplication is not provided because it is unclear whether it should return the scalar or vector product. These are provided instead through methods, \texttt{dot()} and \texttt{cross()} respectively. Additionally, component-wise access is provided through the array subscript operator. A Scalar-array conversion operator is provided for use with library functions (such as those for most graphics packages) which take array arguments.

A note on equality is worthwhile. Vector math tends to bring out the imprecision in floating point math, and it is therefore essential that equality comparison be treated carefully. The DEM Vector provides two tests for equality, one with a specified tolerance, and one which uses a default tolerance. The tolerance is a limit on the length of the difference vector. In order to make the default tolerance useful, it is specified as a fraction of the average length of the arguments. This ensures that the default equality test will not balk at very small length vectors.

```cpp
class Vector {
  const static Scalar DEFAULT_TOLERANCE_FRACTION;

public:
  Vector(); // zero vector
  Vector(const Scalar i, const Scalar j, const Scalar k);
  Vector(const Vector &V);
  Vector(const Scalar s);
};
```
Vector math:
Vector& operator+=(const Vector& v2);
Vector& operator-=(const Vector& v2);
Vector operator+(const Vector& v2) const;
Vector operator-(const Vector& v2) const;
Vector operator-() const;
Vector cross(const Vector& v2) const;
Scalar dot(const Vector& v2) const;
Vector proj(const Vector& v2) const;
Vector& negate();

//normalize vector and return mag() before normalization
Scalar normalize();
Scalar mag() const;
Scalar mag2() const; //magnitude squared

Scalar& operator[](const int i) { return v[i];};
const Scalar& operator[](const int i) const { return v[i];};
operator Scalar*() { return v;};
operator const Scalar*() { return v;};

//Scalar math
Vector& operator*=(const Scalar s);
Vector& operator/=(const Scalar s);
Vector& operator+=(const Scalar s);
Vector& operator-=(const Scalar s);
Vector operator*(const Scalar s) const;
Vector operator/(const Scalar s) const;
Vector operator+(const Scalar s) const;
Vector operator-(const Scalar s) const;

//equality:
bool operator==(const Vector& v2) const {return equals(v2);};
bool operator!=(const Vector& v2) const {return !equals(v2);};
bool equals(const Vector& v2) const;
bool equals(const Vector& v2, Scalar tol) const;
8.1.3 Rotation

The interface for Rotation in the DEM architecture is intended to be generic with regard to the parameterization used in the implementation. It is expected to be used both as a transform and a rotation parameterization. There are a number of issues that make dealing with rotations in three dimensions difficult. The availability in the Architecture of a fully functional parameterization of rotation should make the handling of rotations less ambiguous, and ensure that every component in the architecture benefits from the same rigorous treatment of rotations. Rotation constructors are defined with axis/angle and orientation forms. The default constructor produces an identity rotation. Composition is achieved with the compose(...) method, as well as the multiplication operator. Rotation of Vectors is achieved with the rotate(...) method. Further, the methods invert(), inverse(), and normalize() are defined. Note that the multiplication operator composition syntax is allowed because most mathematical models of rotation transforms support multiplication as the method of composition.

```cpp
struct Rotation
{
    //rotation quaternions are always unit
    Rotation();  //no rotation
    Rotation(const Scalar angle, const Vector& axis);
    Rotation(const Vector& theta);

    void normalize();
    Rotation& invert();
    Rotation& operator*=(const Rotation& q2);
    Rotation operator*(const Rotation& q2) const;
    Vector rotate(const Vector& vec) const;
    Vector& rotateInPlace(Vector& vec) const;

    static Rotation& compose(const Rotation& a, const Rotation& b, Rotation& result);
    static Vector& rotate(const Rotation& a, const Vector& src, Vector& result);
};
```
Several alternatives exist for the parameterization of rotation. The forms considered for this architecture include Euler angles, matrix transforms, and unit quaternions. Quaternions were chosen because they are (from a user’s perspective) easy to use, because they are singularity-free, and because they are not (too) over-specified. Since rotations have often been overlooked in work on discrete elements, and because many example implementations are inexact or incorrect, a review of the comparative benefits of these options seems instructive.

Euler angles parameterize rotation as a series of three base rotations, usually about the primary axes (either local or global). One disadvantage is the sheer number of distinct ways the Euler angles can be defined (at least 12). Furthermore, the method introduces a singularity known as gimbal-locking, which occurs when two of the three base rotations align, and thus do not form a space. This occurs for every definition of the Euler angles. From a user perspective Euler angles are not particularly easy to visualize or produce, and they are difficult to apply and compose. One definition (taken from [68]) is shown in figure 8-1. The subtlety of Euler angle composition has led to erroneous implementations where simple vector addition is used. Typically it is simplest to translate the Euler angles into a matrix for use as a transform, and this is not difficult, though it requires trigonometric functions. The reverse translation (matrix to angles) is difficult, however, so the transform is used as a sort of cache.

Matrix transforms themselves are highly efficient for rotating vectors, and have no singularities. They are, however, highly over-specified, and therefore require more memory, and re-normalization. From a user perspective, matrix transforms are difficult to visualize or produce, and since extraction of angular properties from a matrix is difficult, they are not well suited to front-end use. It is worth noting that while translations and rotations can be combined into one four-by-four matrix, using homogeneous coordinates, this complicates the process of inversion. It is furthermore not particularly convenient since it is often only necessary to rotate vectors. A for-
Figure 8-1: Euler Angles \((\psi, \theta, \phi)\) defined by rotations around the local axes in the order \(z, x, z\).

mulation for integration of rotation matrices can be found in [10].

Unit quaternions bridge the feature gap between Euler angles and matrix transforms. A quaternion consists of one real (or scalar) part and three imaginary parts (which form a vector). The vector describes the axis of rotation. The scalar and the magnitude of the vector are both defined by the angle of rotation (the scalar is the cosine of the half-angle, and the magnitude is the cosine of the half-angle).

Clearly, quaternions are one degree over-specified for parameterizing rotations, and require re-normalization, but they are singularity free, and have a unique definition. Re-normalization, in this case, means ensuring the unit condition, which can be accomplished by component-wise division by the magnitude. A related alternative, the orientation vector, is not over-specified, but cannot be directly used as a transform, or composed with other rotations. Composition and vector rotation for quaternions, on the other hand, is achieved with quaternion-math multiplication.
8.1.4 Quaternions

A brief overview of quaternion math is given, with some simplified expressions for unit quaternions, and vector rotations. Further discussion of quaternions as rotational parameterization can be found in [30] and [68].

\[ q_1 q_2 = (s_1 s_2 - \mathbf{v}_1 \cdot \mathbf{v}_2, s_1 \mathbf{v}_2 + s_2 \mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2) \]

Staring with the following notation, where \( s \) is the scalar part, and \( \mathbf{v} \) is the vector part,

\[ q = (s, \mathbf{v}) \]

quaternion addition can be defined in terms of the vector an scalar parts as follows:

\[ q_1 + q_2 = (s_1 + s_2, \mathbf{v}_1 + \mathbf{v}_2) \]
Following the example of regular complex numbers, the magnitude squared is given as

\[ |q|^2 = (s^2, \mathbf{v} \cdot \mathbf{v}) \]

The inverse is then

\[ q^{-1} = 1/|q|^2 (s, -\mathbf{v}) \]

For unit quaternions (such as those used for rotations), this simplifies to

\[ q^{-1} = (s, -\mathbf{v}) \]

Rotation of a vector \( \mathbf{p} \) is carried out with the quaternion operation below, where \( s \) and \( \mathbf{v} \) are the scalar and vector components of a unit quaternion.

\[ (\mathbf{p}', 0) = q(\mathbf{p}, 0)q^{-1} \]

Which, when the scalar component is discarded, can be simplified to

\[ \mathbf{p}' = s^2 \mathbf{p} + \mathbf{v}(\mathbf{p} \cdot \mathbf{v}) + 2s(\mathbf{v} \times \mathbf{p}) + \mathbf{v} \times (\mathbf{v} \times \mathbf{p}) \]

While rotation is slightly more expensive with quaternions than with matrix transforms, composition is slightly less expensive. Neither operation requires trigonometric operations. Inversion is particularly simple, requiring a simple negation of the vector part.

From a user perspective, the quaternion is easily translated to and from the more intuitive axis/angle and orientation formats, and is even somewhat coherent in its native form. In sum, the quaternion is well suited to general use as a parameterization of rotation, as well as a transform, and is therefore used as the single provided implementation of Rotation. A convincing argument for the theoretical rigorousness of quaternions as measures of angular position, as well as velocity is made in [46].
8.1.5 TclObj

In interacting with the front-end interpreter, and through it, the user, it is necessary to use a variety of string procedures. Tcl, which has been chosen as the embedded interpreter language for the architecture, provides a rich set of string-handling procedures with TclObj. The Tcl_Obj system is capable of storing booleans, integers, and double-precision floating point numbers as well as strings. Procedures for converting among all of these types is provided, and extension types can be implemented. The Tcl_Obj system manages its own memory, and thus makes a convenient dynamic string. Finally, a Tcl_Obj can be shared using a reference counting scheme. For a complete description of the Tcl_Obj system, see [25].

To make the Tcl_Obj functionality easier to use, and to add a layer of separation between the core and the Tcl library, the TclObj class is provided as a smart wrapper for the Tcl_Obj pointer. The reference counting is automatic, and works on construction, destruction and assignment. The string representation can be appended to using the append(...) method or the left shift operator. The value of the object can be extracted into typed data using the put(...) method or the right-shift operator. When some argument is shifted into the object, it's appropriate string representation is appended to the current string. When an argument is shifted from a TclObj, an attempt is made to translate the entire current string into the argument type. If translation fails, the shift operator does nothing, and the put(...) method returns an error. Support for translation of all of the basic math types is provided, but has not been integrated into the Tcl interpreter (i.e. a vector is not a special type, but just a list of Scalars).

TclObj also has built in mechanisms for interacting with the interpreter. The method eval(...) is used to evaluate the contents of the TclObj as an interpreter script. The interpreter pointer itself can be used as a token for the interpreter result in shift operations.

A separate class, TclListObj (Tcl List Object) is used to wrap the tcl list functionality available in TclObj. The interface is extended to provide push(...), which appends an element to the list, and pop(...), which pulls off, and translates, the first element in
the list. The shift operators are modified to use `push(...)` and `pop(...)`. All arguments are translated via a temporary TclObj, so the same translations are available for Tcl List Objects.

It is worth noting that the TclObj functionality is not Tcl specific, and could be re-implemented either with a new interpreter, or even as a self-contained feature. The features that Tcl provides were used in part to save development, and in part to optimize interaction with the Tcl interpreter.

8.2 Conventions

In order to maintain consistency throughout the system, the DEM architecture defines a number of conventions. Conventions, in this sense, define the interface for compliant components. The interface is assumed to be compliant by convention, but there is no requirement that the objects inherit from specific base classes. Objects are simply required to provide a compliant interface. The use of conventions in the DEM architecture arises from the need to encourage consistency while minimizing the overhead of virtual function calls. In order to simplify development, and to supply concise definitions, most of the conventions are defined by a simple base class. Compliance is guaranteed for objects inheriting from these definition classes. For some conventions, notably the Simulation Loop convention, class definition is inconvenient, and so no definition is provided.

8.2.1 Reference Counting

The DEM architecture supports reference counted object sharing through the Sharing conventions. Objects that support reference counting can do so by complying with or inheriting from the Shareable convention definition. The required interface consists of three methods. The objects are expected to destroy themselves when the reference count drops to zero.

```cpp
class Shareable {
    : ...
}
```
To use the reference counting system, the architecture also provides the template class `Sharer<C>`, which implements a fully functional pointer to C, with calls to share and unshare inserted during construction, destruction and assignment. For convenience, `Sharer<C>` is abbreviated as `S<C>` in the rest of this text. The primary motivation for the development of the Sharer class was to simplify the use of STL containers with pointers, but they are used extensively to simplify object management everywhere. The only caveat to their use is that reference loops will not be destroyed without developer intervention. It is recommended, therefore, that developers avoid sharer loops. To provide a general coarse grained check on sharer loops in the system, the Shareable implementation provides a static count of all allocated shareables. If this number does not drop to zero when the shareable module is unloaded, an error is reported, warning the developer that sharer loops are present in the system.

### 8.2.2 Identifiers

Experience in the MIMES project has demonstrated the usefulness of several different methods of identifying objects. These include nondecreasing unique integers (UID’s), zero-based consecutive integers (table id’s), and string names. In each case, it is usually necessary or expedient to store the identifier (or a copy of it) within the object. For example, a tableId entry in the object serves as a reverse-index of the object table. These three basic identifiers are encapsulated in the identifier conventions, UID, TableId, and Name. They define both a standard typing for the actual data, as well as a standard way to address the members.

```cpp
struct UID {
    typedef unsigned int Type;
    Type uid;
};
```
struct TableId {
    typedef unsigned int Type;
    Type tableId;
};

struct Name {
    typedef TclObj Type;
    Type name;
};

Each of the identifier types conforms to a simple overall convention of defining Ident::Type to be the data type for the identifier. The identifier conventions are required for the lookup infrastructure presented in section 8.3.

8.2.3 Cloning

One of the most difficult problems with large simulations is the generation of large packed samples of heterogeneous composition. One tool which is useful in this regard is the ability to duplicate whole regions of a simulation. This facility, called cloning, has been developed in MIMES, and has been used successfully in a number of applications. Small packed samples are quickly generated, and cloned out to span a large region. Once the region is tessellated, a short consolidation of the objects is sufficient to create a fully packed sample. Figure 8-3 shows a simple cloning example.

The challenge of cloning a region of pre-packed objects is to clone the shared items such as contact constraints, point-to-point bonds, etc., so that the new copies of the region are indistinguishable from the originals. In MIMES, this challenge was met using a complicated system of book-marking tailored to the specific coupling components involved. This architecture proposes a cleaner, more general solution. At its heart is the Cloneable convention. Objects that conform to Cloneable<Type> provide a method Type* clone(S<CloneSession>) which returns a pointer to the Cloneable type. The CloneSession provides a hashtable that matches original objects to their cloned duplicates. Clones are requested via the CloneSession method Type* getClone(Type*),
Figure 8-3: Cloning packed samples to create large nearly packed regions[62]

which in turn calls the Cloneable type's clone method if it does not already have a registered clone for the original instance.

```cpp
template<class This>
struct Cloneable {
    Sharer<This> clone(Sharer<CloneSession>) {
        return new This(*this);
    }
};

struct CloneSession : public Shareable{
    template<class Type>
    Type* getClone<Type>(Type* original);
    Table::find_iterator i = table.find((Key)original);
    if (i != table.end())
```
Compared with the method implemented in MIMES, this method is very simple and clean, but requires extra memory, and performs extra work in registering clones for instances which are not shared. As a part of the generic DEM infrastructure, however, this method is more appropriate since it will always work, regardless of the particular objects involved. Furthermore, since the CloneSession table is implemented with a hashtable, there is little performance loss due to extra clone registration.

8.3 Containers and Lookups

In order to simplify development, and improve data management, the DEM architecture makes extensive use of the STL containers[66]. A few enhancements are provided to integrate the containers with the identifier conventions, and provide some functionality useful in DEM simulation. First, a general convention called Lookup is defined. The Lookup convention ensures that a standard set of type definitions are in place, and that the lookup table (of type Table) is called table.

```
template <class T, class K, class V>
struct Lookup {
    typedef T Table;
    typedef K Key;
    typedef V Value;
    typedef S<V> VShr;
    typedef V* VPtr;
    typedef typename T::iterator iterator;
    Table table;
    
    Key insert(VPtr);
    bool erase(Key);
};
```
Based on the Lookup convention, a specific lookup convention is defined for each of the three basic identifier types. The UIDLookup manages a set of objects conforming to the UID convention, and assigns them uid's as they are inserted into the table. The assignments are indexed with a hashtable. The TableLookup manages a table-indexed set of objects (conforming to the TableId convention), and makes sure that their TableId assignments always reflect their positions in the table. The NameLookup manages a hashtable mapping names to objects. A provision for assigning default names (composed of a string followed by a sequence number) to objects which are inserted without a name. The UIDLookup and TableLookup just implement the Lookup interface, but NameLookup adds a method for inserting already named objects.

```cpp
template <class C>
class NameLookup {
  :

public:
  NameLookup(Key basename);

  bool insert(VPtr p);
  bool insert(Key k, VPtr p);
  bool erase(Key k);
  VShr get(Key k) const;
};
```

Another feature extrapolated from MIMES is the ability to associate names with groups of objects. A generic facility for implementing such a feature is provided with TagTable. A TagTable extends a unique association Lookup to include the ability to create many to many mappings between object instances and Name-like Tags. The tags are mapped to the Lookup key, which is in turn mapped to the object instance.

```cpp
template <class Lookup>
class TagTable : public Lookup {
  :

public:
```
8.4 Type Interactions

A final, and important feature of the DEM infrastructure is a generalized facility for resolving two-way subtype interactions. As background, two-way subtype interactions occur when the interaction between two polymorphic objects need to be found. In MIMES, where arbitrary geometries were allowed to collide with each other, this problem was solved using a two-level virtual function call. The interaction is initially requested using the virtual method `check_contact(Element*)`, which is used to select the right second-level virtual method to call.

```cpp
struct Element {
    virtual bool check_contact(Element* e)=0;
    virtual bool check_contact(ElementSubType* e)=0;
};

bool ElementSubType::check_contact(Element* e) {
    return e->check_contact(this);
}

bool ElementSubType::check_contact(ElementSubType* e) {
    :
}
```

The two-level method has several drawbacks. Apart from the performance issues raised by a double virtual method call, the two-level method requires that all of the subtypes be declared in the parent class. This crowds the class definitions, and makes
it difficult to locate and maintain the actual methods to be applied. Furthermore, it
does not necessarily resolve the pairing \((A, B)\) in the same way as \((B, A)\).

Another solution to the problem is to hard-wire integral type identifiers into the
system. There is some evidence that this was initially tried in MIMES, and seems to
have been the method used in [19]. This method is, however, not well suited to an
evolving code, since it requires a (rather inelegant) switch statement to resolve the
type pairing. Additionally, it places the burden of type id choice on the extension
developer, who may not be aware of all of the implemented subtypes. Furthermore, it
is up to the extension developer(s) to ensure that \((A, B)\) and \((B, A)\) do the same thing.
Overall, too much work is required to effect incremental changes such as extension
for this to be a viable option for an evolving architecture.

8.4.1 TypeId Convention

The system used in the DEM architecture cuts a compromise between the two-level
virtual method call, and hard-wired type identifiers. It uses type identifiers, but sup-
plies an automated way of generating these identifiers, and a generic way of mapping
id-pairs to results. The identifiers are generated globally, so they are non-consecutive
for any given base class. The reason for this is that one class may wish to partici-
pate in more than one TypeInteraction. The system depends on following the TypeId
convention, which requires that the base type have a member called instanceTypeId.
The recommended method of fulfilling the TypeId convention is for the base class to
inherit virtually from the TypeId convention definition, and for the subtypes to inherit
from the SubTypeId\(<C>\) convention definition. This allows the sub-types to construct
their ownTypeId field using their subtype identifier. A Macro to define and initialize
the static subtype member subTypeId using the static method Typeld::New() completes
the system. Here we give the basic elements.

```c
struct Typeld {
    typedef unsigned char Type;

    //initialize using C::subTypeld
```
template <class C> Typeld(const C* c);

Type instanceTypeld;
static Type New();

};

template <class C>
struct SubTypeld : virtual public Typeld {
    SubTypeld() : Typeld(this) {}
    const static Typeld::Type subTypeld;
};

//macro to define and initialize the static member subTypeld
//using Typeld::New()
#define DefineTypeld(C) Typeld::Type SubTypeld<C>::typeld = Typeld::New();

An example usage in the base and derived class is shown:

struct Base : virtual public Typeld {
    Base() {}
    :
};

struct Child : public Base, public SubTypeld<C> {
    Child() : Base(), Typeld(this) {}
    :
};

DefineTypeld(Child);

8.4.2 TypeInteractionManager

With type id's assigned largely automatically, all that remains is to provide a simple mechanism for mapping pairs of Typeld's to interactions. Here the assumption is made that the notion of interaction can be encapsulated in an interaction class, and
that the job of the TypeInteractionManager<T,I> is to map Typeld pairs to interaction class generators, where a generator is a pointer to a function that returns an interaction class sharer. With this assumption, the interaction manager can be implemented with a simple hashtable mapping Typeld pairs to function pointers.

```cpp
template <class T, class I>
class TypeInteractionManager {
    
    public:
        typedef S<l> (FunPtr*) (S<T>, S<T>);
        S<l>& get (S<T>, S<T>);
        bool set (Typeld::Type, Typeld::Type, FunPtr);
};
```

Finally, since the resolution of type interaction is entirely managed by the TypeInteractionManager set(...) and get(...) methods, it is a straightforward matter to ensure that the interaction \((A, B)\) is the same as \((B, A)\). This is done in set, by storing the same function pointer in both entries. The policy uses slightly more memory, but wastes nor work.

### 8.4.3 InstanceInteractionManager

In some cases, interactions between instances can be reused. For example the interaction between two material instances will always be the same. On the other hand the interaction between two geometry instances is not the same. When the interactions can be reused, it is desirable to resolve the interaction only once, and then cache the result. This is done using the InstanceInteractionManager. The InstanceInteractionManager extends the TypeInteractionManager with another hashtable mapping instance pointer pairs to interaction class instances. When the requested interaction is not found, the TypeInteractionManager is used to create a new one, and it is cached for future invocations.

```cpp
template <class T, class I>
struct InstanceInteractionManager : public TypeInteractionManager<T, I> {
    S<l>& get (S<T> p1, S<T> p2) {
```
S<i>& v = table[Key(p1,p2)];
if (!v)
    v = Super::get(p1,p2);
    return v;
};

The instance interaction manager supersedes the material list functionality of MIMES, and frees the materials from zero-based indexing. Additionally it allows a lazy instantiation model where only materials which actually interact need to instantiate interactions.
Chapter 9

Simulation

This chapter discusses the conventions of simulation. At the most basic level these are the SimLevel and SimLoop conventions. These conventions define the standard interface for stepping all simulation objects through the simulation loop, as well as a mechanism for state change advertisement.

9.1 SimLevel

In a general simulation environment, it is desirable for objects to be able to advertise changes in their state in case other dependent objects hold derived state that might be invalidated by the changes. The facility for advertising state changes is supported through the SimLevel convention. SimLevel defines three distinct simulation phases: INIT, STEP, and END. Newly created, or updated objects are marked as INIT. Objects which are unchanged since the last simulation step are marked STEP. Finally, objects which are no longer relevant and are preparing for removal are marked END.

```cpp
struct SimLevel : public Shareable {
    enum Level {INIT, STEP, END};
    Level _simLevel;

    SimLevel() : simLevel(END) {}  
    Level simLevel() {return _simLevel;}  
    void setSimLevel(Level l) {_simLevel = l;}  
}
```
The SimLevel convention can be used to solve a variety of state-change issues. A typical example can be seen in the management of element mass. The mass depends on the element scale (in Locale), the geometry, and the material. If any of these are changed during the simulation, the mass must also be changed to reflect the update. This is accomplished by checking the SimLevel of the Locale, Geometry, and Material, and updating if any are not at level STEP. Another example of level-change can be found in objects which are kept up to date by some management module. In contact detection, for example, contact objects, which are left intact from one timestep to the next, must be eliminated if they have not been updated by the NeighborSearch module. This can be done by marking all contacts as ready for removal, and updating current contacts to have a SimLevel of STEP. After the contact detection phase is complete, updated contacts will be at the standard STEP level, whereas outdated contacts will still be marked END, and can accordingly be removed.

By carefully defining the SimLevel convention, and rigorously applying it throughout the architecture, we add a new level of extensibility to the simulation. If, for example, it becomes necessary to support another level, it can be added globally by modifying the SimLevel convention. One example that comes to mind is support for distributed computation. In a truly distributed system it is necessary divide the objects up between the nodes. Thus it will be necessary to support objects entering and leaving the simulation on any given node while it is running. This can be supported with the current mechanism, and indeed support for mobile objects in distributed computing has been one of the motivating factors behind the SimLevel convention. It may be convenient, however to have objects in a semi-present state on nodes which are nearby the actual owner nodes. These semi-present objects could be used for contact detection in the border-zone, but would not be fully instantiated. This could be supported with a special SimLevel level, perhaps called FOREIGN.
9.2 SimLoop

To provide a consistent interface to the procedures of looping, and to define, in a general sense, the canonical stages of a simulation, we specify the SimLoop convention. The SimLoop convention is not formally defined with a class like most of the other DEM Architecture conventions, because the arguments to the simulation methods are not specified.

```cpp
struct SimLoop {
    int simInit(...);
    int stepInit(...);
    int stepDo(...);
    int stepEnd(...);
    int simEnd(...);
};
```

The `simInit()` phase is used for one-time initialization of members at the start of a simulation (or when an object enters the simulation). The `stepInit()` phase is reserved for pre-step initialization. This might include such tasks as clearing force-accumulators. The `stepDo()` phase is where the main computation for the simulation should take place. The `stepEnd()` method handles post-step computation and cleanup. This might include force-integration, or bounding box calculation. Note that each of the SimLoop methods returns an exit status to indicate successful completion of that phase.

Formal definitions are provided for two specific sub-conventions, SimObj and Sim-Behavior. While not envisioned at the time of publication, there is no reason that other sub-conventions might not be used. Their definitions are easily extrapolated from the SimBehavior example.

SimObj implements the SimLoop for standard SimLevel objects. The definition class provides a default implementation that updates the SimLevel at the correct points in the loop. The switch to STEP level occurs at the end of `stepEnd()` to conform to the definition of the STEP level, requiring that the object be unchanged (or not unusually changed) since the last step. Note that the inheritance is virtual. This is because some parts of the architecture will inherit from both SimBehavior and SimObj.
The same interface and implementation is provided for SimBehavior, which implements the SimLoop for objects which act on an object of type Element. This is called a behavior because the object will be used to implement some kind of element action, such as body-force loading. SimBehavior is the only implemented SimLoop sub-definition that takes an argument because at this time, only elements have behaviors. If others are implemented, it might be prudent to rename SimBehavior SimElementBehavior.

```
struct SimBehavior : virtual public SimLevel {
    int simInit (S< Element >) { setSimLevel(INIT); return OK; };
    int stepInit (S< Element >) { return OK; };
    int stepDo (S< Element >) { return OK; };
    int stepEnd (S< Element >) { setSimLevel(STEP); return OK; };
    int simEnd (S< Element >) { setSimLevel(END); return OK; };
};
```

9.3 Generic SimLoop

The SimObj and SimBehavior conventions define an interface for simulation, but are not intended for use as polymorphic types. This is because, in most cases, SimLoop objects do not need to be polymorphic, and so it is desirable to avoid the overhead involved in the virtual interface. To supply polymorphic versions of the SimLoop conventions, the architecture provides the GenericObj and GenericBehavior classes. The interfaces are the same, except that the five simulation methods are virtual.

When the Generic versions of the SimLoop sub-definitions are used as polymorphic objects, they are not conventions. But they will sometimes also be used as
conventions for polymorphic SimLoop sub-definitions. For example, if the polymorphic class T conforms to the GenericObj convention, implies that T provides the five virtual simulation methods, which it may, (but need not necessarily) inherit from GenericObj.

### 9.4 Modules

An important part of the Architecture’s support for extension is the `Module` convention. The principle function of the module convention is to provide a standard way to initialize and clean up after modular capabilities. In order to facilitate dependency resolution, the `Init()` method is expected to allow multiple invocations. After a module has been initialized, subsequent calls to `Init()` should simply return success. Thus, any code that requires support for some feature need only ensure that support is present by calling `ModuleType::Init()`. The `Exit()` procedure is given as a suggestion for how to name an exit handler for use with the `Tcl_CreateExitHandler(...)` mechanism.

```cpp
class Module {
protected:
    static TclObj current;
    const static int INIT_OK = TCL_OK;
    const static int INIT_ERR = TCL_ERROR;
    static int Init() { return INIT_OK; }
    static void Exit(ClientData) {};
};
```

### 9.5 SimModule

As an extension of the Module convention, the architecture provides the class `SimModule`, which conforms to the Module and GenericObj conventions. It is used to manage capabilities which require step-wise maintenance. Everything that moves in the simulation is managed, on some level by a SimModule. Elements are managed by the `ElementManager SimModule`. The `NeighborSearch SimModule` is used to step the neigh-
bor search algorithm at each timestep. Behaviors are managed by the BehaviorManager SimModule.

```cpp
struct SimModule : public GenericObj, public Module {};
```

## 9.6 SimulationManager

The collection of SimModule instances, which aggregates all of the simulation loop management in the architecture, is managed by the SimulationManager. This central module, manages the simulation loop for the whole simulation, and times it to the program event loop. At each step of the event loop, the SimulationManager calls the stepping methods of the SimModules in the order in which they were added to the list. Modules are added using addModule(...), and can be replaced using refModule(...). The active() method is also provided to inform the caller if the simulation is currently running, in case, for example, some operation should only be carried out when it is not active.

```cpp
class SimulationManager : public Module {

public:
    void addModule(S<SimModule> m);

    void start(); //start the simulation
    void resume(); //resume the simulation
    int loop();    //do one loop of the simulation

    bool active(); //is the simulation active?

    static int Init();
    static void Exit(ClientData);
};
```
A generic SimModule for managing SimObj-compatible components is provided through the SimObjManager. This component extends a Lookup to provide simulation loop management for the member objects. The SimModule interface acts as an entry point for the SimLoop interface for the whole collection.

```cpp
template<class C, template <class> Lookup>
class SimObjManager : public Lookup<C>, public SimModule {
    public:
    int simInit () {forall(table,&C::simInit); return OK;};
    int stepInit() {forall(table,&C::stepInit); return OK;};
    int stepDo  () {forall(table,&C::stepDo); return OK;};
    int stepEnd () {forall(table,&C::stepEnd); return OK;};
    int simEnd  () {forall(table,&C::simEnd); return OK;};
};
```
Chapter 10

Element

This chapter covers the Element class, and the modular components which define its behaviors. This is the heart of the DEM simulation, and provides most of the basic discrete element functionality.

10.1 Goals and Motivation

The design of the element class departs in many ways from the design used in MIMES. In addition to the overall architecture goals of extensibility and consistency, the element class is designed for compactness. In large simulations, the per-element storage footprint can be a great consideration, as memory requirements effectively limit the simulation size for single processors.

10.1.1 Monolithic Elements

The MIMES element design can be characterized as monolithic. While composed of several independent structures, the whole element package functions essentially as a unit. This poses several problems, especially for extension. Because the element functions as one very large class, it is not possible to extend element functionality in more than one way. The basic subtypes of element are the four supported geometry types. In order to extend other features, the extension must be implemented in the
base type, separately in each subtype or through a complex subversion of the element class.

A related problem with monolithic elements is that implementation of new capabilities typically requires new data members. Thus, after ten years of extension development, the base element has twenty five vectors, fifteen scalars, and twenty integer flags. The definition includes one hundred and sixteen methods, of which forty-two are virtual.

Finally, since the entirety of element functionality resides in some part or subclass of the element class, there is little support for sharing components. Geometries could, for example, profit from sharing by drastically reducing the cost of representational complexity. In MIMES, the only component which could be considered to be shared is the material properties, and it is supported through a specialized material interface.

10.1.2 Modular Elements

The DEM Architecture adopts a modular design for Element. The class itself is essentially a table of sharers for the seven built-in components, with support for some generic extension components. The element class is Cloneable, and inherits from SimObj, and passes the SimLoop methods to its components. All of the components conform to the Cloneable convention and some form of the SimLoop convention. The generic extension components must inherit from StandardBehavior (see 10.10).

```cpp
struct Element : public SimObj, public UID, public TableId {
    typedef std::set<S<StandardBehavior> > BehaviorList;
    S<Material> material;
    S<Properties> props;
    S<Locale> locale;
    S<Region> bbox;
    S<Dynamics> dynamics;
    S<Geometry> geometry;
    S<Contact> contact;
    BehaviorList actions;
};
```
Element* clone(S<CloneSession>);
}

This design has several advantages. One clear advantage is that support for partially instantiated objects in a distributed environment is easily integrated into this framework. Another advantage is that each of the modules can be extended separately. Thus it is easy to have an element which combines specialized dynamics with a particular geometry. Furthermore, since many of the capabilities which were core-supported in MIMES can be implemented as StandardBehaviors (see section 10.10), the data members they require need not be set aside for every element. In fact in the case of capabilities which were optionally supported with some integer flag, the flag is not necessary at all; either the behavior is in the action list or it is not.

10.1.3 Shareable Components

With the adoption of modular components, and the use of Sharers, support for shared components becomes quite simple. The shared component is simply referenced in each of the sharing elements. In the current implementation, Geometry, Material, and some StandardBehavior can be shared among any number of elements.

It should be noted here that the SimBehavior interface is designed with shareable components expressly in mind; while non-shared components could have pointers to their parent elements to use when applying behaviors, the shared behaviors cannot, and so the argument to the SimLoop methods is used to pass that information. This, of course, also saves the non-shared components from storing an extra pointer.

10.2 Properties

The Properties class groups together non-dynamic properties of the element. This lends definition to the SimLevel for Properties. That is, whenever of the public items in Properties is changed, the Properties SimLevel is reset to INIT. The meaning of mass and moment of inertia is clear enough. The boundary flag is used to specify a
class of objects for which contact is ignored, a feature normally used for boundary objects which need not interfere with each other.

```cpp
struct Properties : public SimBehavior {

    enum XFixity {FREE, PLANAR, LINEAR, FIXED};
    enum RFixity {RFREE, AXIAL, RFIXED};

    Scalar mass;
    Vector I;
    bool boundary;

    XFixity xFix;
    Vector fixedDirection;
    RFixity thetaFix;
    Vector fixedOrientation;

    Properties * clone(S<CloneSession> C);
};
```

The fixity functionality in Properties departs somewhat from that of MIMES. Four states of translational fixity are supported, FREE, PLANAR, LINEAR, and FIXED. For PLANAR, the vector `fixedDirection` specifies the direction in which the element is not allowed to move. For LINEAR, `fixedDirection` specifies the only direction in which the element is allowed to move. Obviously, FREE elements are free to move, and FIXED elements cannot move. In either case, `fixedDirection` is ignored. This formulation is more general than the axis-aligned formulation supported in MIMES, since it allows fixity in any direction. The generality of the formulation does not cost much, since the fixity condition is easily applied as a projection of the dynamics along the `fixedDirection`.

A similar interface specifies rotational fixity. The `thetaFix` flag takes on values of RFREE, AXIAL, and RFIXED. In the AXIAL case, `fixedOrientation` specifies the axis around which the object is allowed to rotate. This formulation is, again, more general than the intuitive extension of the MIMES formulation to 3D, and is also achieved quite easily through projection.
10.3 Locale

The Locale component defines an element’s local frames of reference. In the DEM Architecture, elements refer to three distinct frames. In the geometry frame, the axes are aligned with the element geometry’s primary axes. The coordinates are scaled such that the element geometry has a bounding sphere of one unit. The inertial frame is a uniform scaling of the geometry frame such that it is isometric with the global coordinates. The global frame is, in turn, achieved through a rotation and translation of the inertial coordinates. The quantities required to perform these transformations are encapsulated in the Locale component. The distinction between inertial and geometry frames is made in order to facilitate sharing of geometries (see section 10.7).

```cpp
struct Locale : public SimObj {
  Vector x;
  Rotation theta;
  Scalar scale;
  int stepEnd(S<Element>); //update bbox
};
```

10.4 Bounding Box

The Element bounding box (bbox) is a bounding coordinate pair encapsulated in the Region class. The bounding box is used by the neighbor search algorithm to find contact neighbors. Normally, the bounding box is updated by the Locale during stepEnd(). The bounding box is cached separately, however, so that it can be modified in special cases. Highly non-spherical objects such as thin plates, for example, have large bounding spheres, and produce very large, and not necessarily optimal bounding boxes. This is especially true in the common case where the plates are axis-aligned. In such a case the geometry, which might be expected to be aware of the situation can update the bounding box more aggressively.
struct Region {
    Region(Vector l, Vector h) : lo(l), hi(h) {};
    Vector lo;
    Vector hi;
};

10.5 Dynamics

The integration of element dynamics is encapsulated in the Dynamics component. The component provides an interface for applying forces and moments, and public access to the vector data members for velocity, angular velocity, displacement, and angular displacement. The base dynamics class is fully specified as a placeholder; it discards the applied loads.

class Dynamics : public GenericBehavior {
public:
    Vector v;
    Vector omega;
    Vector dX;
    Vector dTheta;

    void applyForce(Vector f);
    void applyMoment(Vector m);
    virtual Dynamics * clone(S<CloneSession>)=0;
    int stepEnd (S<Element>);  //integrate motion
};

A leapfrog-step integrator is supplied in the form of StandardDynamics, and follows closely, the example of the integration scheme used in MIMES. The only difference of note is the integration of rotation, which is significantly complicated by the move to three dimensions. Much of the literature on integration of rotations concerns integration of the Euler angles, and does not, therefore, inform on the integration of our quaternion-based rotations. Some authors report integration of the rotation
quaternion by simply integrating the four components separately. This is incorrect since it ignores the unit constraint, but the error may be insignificant for small angles. The method reported in [67], which takes the unit constraint into account is used in StandardDynamics.

10.6 Contact

The Contact component encapsulates an element’s response to contact with other elements. It provides the usual Cloneable<Contact> and GenericBehavior interfaces. It also provides the static method addCandidatePair(...) to allow the NeighborSearch module to identify contact candidates.

```cpp
class Contact : virtual public Typeld, public GenericBehavior {
    protected:
        typedef std::list<S<Element>> CandidateList;
        CandidateList candidates;

    public:
        virtual Contact* clone(S<CloneSession>)=0;
        static bool addCandidatePair(S<Element> e1, S<Element> e2);
        int stepDo (S<Element>); //process candidates
};
```

The contact candidates are added to the (protected) candidate list of one of the participating elements. This element is called the owning element, and is decided by UID, so that it is always the same for a given pairing. Implementations of Contact are expected to consume or clear the candidate list themselves. Typically the candidates are used to create or update ContactPenalty objects. The ContactPenalty class is a GenericBehavior that encapsulates the penalty constraint between two elements.

```cpp
class ContactPenalty : public StandardBehavior {
    protected:
        S<Element> other;
    public:
```
int stepDo (S<Element>);  //apply forces to both elements.
;
);

The general facility for managing ContactPenalties is provided with the component PenaltyBasedContact. A PenaltyBasedContact maintains an ordered set of current contact penalties. The set is ordered on the value of other, for efficient lookup of candidates. During stepInit(...), the penalties are all marked for removal (END). During stepDo, when a new candidate is processed, the penalty set is searched for a matching existing penalty. If one is found, it is updated. Otherwise, a new one is created. The type of contact penalty created is resolved based on the types of Contact components involved, using a TypeInteractionManager. Once all of the candidates are consumed, any penalties which are still marked for removal are removed.

class PenaltyBasedContact : public Contact {
protected:
    typedef std::set<ContactPenalty> PenaltyList;
    PenaltyList penalties;
    int stepInit(S<Element>); //mark all penalties for removal
    int stepDo (S<Element>);  //process candidates
;
};

10.7 Geometry

The geometry component is designed to maximize the ability to share geometry information among objects. To this end, the element geometry is defined in a totally separate geometry frame (see section 10.3). The geometry frame is scaled such that the bounding sphere of the geometry is one unit. This means that the same geometry can be used for different sized objects. It also means that objects can be easily scaled up or down, opening up the possibility of transforming simulations between different units. Of course, another advantage of the unit bounding sphere convention is that the element geometry can be completely approximated without reference to the ge-
ometry. The geometry frame is further constrained to be centered on the center of mass, and axis-aligned with the primary moments of inertia.

The geometry must provide, within its frame of reference, the volume and primary moments of inertia. In addition, it provides a region intersection check (for use in region queries, for example).

```cpp
struct Geometry : public SimObj, public Name, virtual public Typeld {
    virtual Scalar V() const;
    virtual Vector I() const;
    virtual bool check(S<Region> r);
};
```

The geometry is both a SimObj and a SimBehavior. The SimObj interface is used by the GeometryManager, and the SimBehavior interface is used by the element. The SimBehavior stepEnd method is virtual to provide the geometry with an opportunity to update the element bbox.

Currently the Architecture supports the SphereGeometry, of which there is only one instance, since there is only one kind of unit sphere.

### 10.8 Geometric Intersection

Intersections of geometries can take many forms. Since we have implemented single point contacts, we use the IntersectionPoint to summarize the intersection between two geometries. Support for IntersectionSurfaces would be a companion extension to multi-point contact.

The IntersectionPoint is set up to reference the participating geometries at construction. The interface for the IntersectionPoint provides three data elements, the point of intersection, the normal direction for the intersection, and a scalar penalty representing the amount of overlap. These are chosen for the specific reason that they are useful for computing contact penalties. All quantities are in the global frame. The renew method is used to update the intersection, which may be kept for many timesteps, using two fresh locales.


```cpp
struct IntersectionPoint : public Shareable {

public:
    Vector point;
    Vector normal;
    Scalar penalty;
    virtual bool renew(const S<Locale> &1, const S<Locale> &2) = 0;
};
```

The subclasses of `IntersectionPoint` are doubly type-specific. Accordingly, the intersection type used when two geometries intersect is resolved using a TypeInteraction-Manager.

### 10.9 Material

MIMES included support for shared Materials through a material table, accessed by the elements using a material id. While functional, the model did not allow for extended material types. Furthermore, the types had to be addressed by number in the simulation as well as the interface. In keeping with the overall goals of the Architecture, Material is designed to support extension in both the material type and the material interaction type. The materials are identified by name, and managed by the MaterialManager through their SimObj interface.

```cpp
struct Material : public SimObj, public Name, virtual public Typeld {
    Scalar rho;
    Scalar alpha;
    Scalar alpha_rotational;
};
```

The basic material type provides three scalar quantities, density, viscous damping, and rotational viscous damping. Note that the damping properties correspond to damping on the global dynamics of the object (Raleigh damping). In MIMES Rayleigh damping was a global property. Here we have chosen to make it material dependent.
The interaction (at contact) between two materials is defined by the MaterialInteraction class. The interface provides the getForce(...) method to evaluate the force due to material collision. The getForce(...) method returns the force corresponding to a displacement of \(\mathbf{u}\), and a velocity of \(\mathbf{u}\dot{\mathbf{u}}\), with normal and tangential directions defined by \(\mathbf{n}\) and \(\mathbf{t}\) respectively. For use in calculating viscous damping, the masses are provided as arguments \(m_1\) and \(m_2\). The material interactions are managed by an \texttt{InstanceInteractionManager<Material,MaterialInteraction>}.

```c++
struct MaterialInteraction : public SimObj {
    virtual Vector getForce(const Vector u, const Vector udot, const Vector n, const Vector t, const Scalar m1, const Scalar m2);
};
```

## 10.10 Actions

A facility for adding simple generic behaviors to the elements is provided through the actions list. This list contains sharers of StandardBehaviors, a subclass of GenericBehavior that conforms to the UID convention. The UID is provided so that elements can figure out what behaviors they are referencing. StandardBehavior is not very useful by itself, since it makes no mention of how the behaviors are managed. For more information about the behavior management system, see chapter 11. Without worrying about the management issues, however, it is easy to see that the GenericObj interface provides enough flexibility for a wide variety of behaviors to be implemented. The trivial implemented example is BodyForce, which applies a body force to all of the objects it acts on.

```c++
struct StandardBehavior : public GenericBehavior, virtual public UID {
};
```
Chapter 11

Behaviors

Throughout the Architecture, we use the term *behavior* to refer generally speaking to any component which acts on or modifies an Element. Many behaviors, in this sense, can be implemented through the StandardBehavior interface (see section 10.10). The way such behaviors are applied is simply by being present in the element’s action list. This mechanism is intentionally flexible, and can support a wide range of behaviors. Most kinds of behaviors, however, will require more careful management than the action list supports. The most common type of StandardBehavior is the GroupBehavior. GroupBehaviors can be shared by any number of elements. Because the elements will call the SimLoop methods many times in one step, the SimLevel is best managed through a separate SimObj interface. This interface is used by the BehaviorManager.

```cpp
//interim definition:
struct GroupBehavior : public StandardBehavior, public SimObj {
    //modify SimBehavior interface to decouple it from the SimLevel
    int simInit (S<Element>) {return OK;};
    int stepInit(S<Element>) {return OK;};
    int stepDo (S<Element>) {return OK;};
    int stepEnd (S<Element>) {return OK;};
    int simEnd (S<Element>) {return OK;};
};
```
More complex kinds of behaviors can be envisioned which do not fit the Group-Behavior model. For example, one way to implement graphics (suitable for the VTK toolkit) would be to have each element own (through its actions list), an individual DrawElement behavior containing some element-specific state (such as a suitable VTKActor). The DrawElement instances need to be collectively managed by some DrawTarget component, in order to tell them where to draw, and when. This DrawTarget might in turn be considered as an aggregate representative of the draw behavior for user-interaction purposes. In order to support such a scheme, the Architecture defines behaviors through two disjoint interfaces, StandardBehavior, and BehaviorManagerEntry. Since The BehaviorManagerEntry acts as a sub-manager for StandardBehaviors, we call this scheme two-level behavior management.

The BehaviorManagerEntry is a GenericObj-compatible class that conforms to the UID convention, and is managed by the BehaviorManager. It must provide a mechanism for applying the behavior it controls to elements, and for removing it from those elements. This is done through the addObj(...) and delObj(...) methods. Furthermore, since all behaviors are likely to have implementation specific configuration options, the BehaviorManagerEntry must also provide choice parsers for configuration and query. For further discussion of command parsing see chapter 12. Note that the Module convention is included to provide the behavior an entry point for setting up the parsers, and initializing a creation command within the BehaviorCommand.

```cpp
struct BehaviorManagerEntry : public GenericObj,
                  virtual public UID, Module {

    virtual bool addObj(S<Element>)=0;
    virtual bool delObj(S<Element>)=0;
    S<ChoiceParser> conf;
    S<ChoiceParser> cget;
};
```

With the BehaviorManagerEntry definition given, we can modify the GroupBehavior definition to be compliant.

```cpp
struct GroupBehavior : public StandardBehavior,
```
public BehaviorManagerEntry {

    int simInit (S<Element>) {return OK;};
    int stepInit(S<Element>) {return OK;};
    int stepDo (S<Element>) {return OK;};
    int stepEnd (S<Element>) {return OK;};
    int simEnd (S<Element>) {return OK;};

    virtual bool addObj(S<Element> e) {
        return e->actions->insert(this);
    }
    virtual bool delObj(S<Element> e) {
        return e->actions->erase(this);
    }
};

Finally, the BehaviorManager itself is simply a SimObjManager.

struct BehaviorManager : public SimObjManager<BehaviorManagerEntry> { }
Chapter 12

Scripted Interface

One of the most useful aspects of the embedded Tcl interpreter is its use as a scriptable command line interface. In fact the entirety of the GUI in MIMES can be thought of as a scripted extension to the command set provided through the interpreter. Additionally, through scripts and the interpreter shell, the command-line system is often the main interface between the computational engine and the power user, since it provides a greater functionality. For this reason, an important aspect of the DEM Architecture is its command system. The architecture tries to improve on the MIMES interface in two basic ways, through supplied infrastructure, and command guidelines. The guidelines try to extend the principles of consistency and extensibility to the command interface. The guidelines, however, can only suggest possible command styles. It is hoped that by providing a comprehensive infrastructure for command implementation, future command development can be cajoled into consistency.

12.1 General Command Guidelines

One of the main difficulties with the MIMES command interface is the vast array of commands available. It is difficult for users to become familiar with the whole set of commands, and facilities for querying the command system are minimal. The guidelines propose to limit the number of commands as severely as possible through the use of hierarchical commands. An example is the object command, which is used
to create, query and modify objects.

```python
set i [object new configure -tags a]
object $i configure -x {100 100 100}
object a rotate {axis-angle 1 1 1 $alpha}
object a cget -mass
```

12.1.1 Option Notation

The above example also demonstrates another aspect of the guidelines. Command implementors are encouraged to use option-style command arguments whenever appropriate. This is so that users need not memorize the argument order for long strings of arguments. The approach can be overdone, of course, and it is not suggested that well defined argument sequences (like vectors) be defined with options.

12.1.2 Help

Another feature which made its way into some of the MIMES commands is command help. The guidelines call for all commands to support command-line help with the standard help options -help and ?. The help option should return information on the argument where it appears. In the special case of the first argument, the help option should return a general help. The help feature is particularly useful for hierarchical commands and commands that take options, since it provides an easy way for the user to find out what options are supported.

```bash
# get general help about the object command
object -help
# find out which subcommands are supported
object {} -help

# find out what options you can configure
object {} configure -help
# find out about configuring theta
object {} configure -theta -help
```

It is recommended the the typical help message follow the example.
12.2 Command Infrastructure

This section discusses the infrastructure that the Architecture provides for building interface commands. The command system is separated into two layers, the CommandLine layer, which handles the direct interface with the command arguments and interpreter, and the Pipeline, which uses the CommandLine to parse the arguments.

12.2.1 CommandLine

The two-layer design is intended to isolate the core from its Tcl interpreter. If the decision to adopt Tcl as the embedded language is ever reconsidered, the CommandLine layer can be re-implemented to work with the new interpreter. The class basically provides the pipeline with a way to get arguments, arg(), and a way to format results. The pipeline puts its results into any of the public members usageList, errorMessage, result, and helptext. When the command parsing finishes, control is passed to the completion methods, and the internal result code (CMD_OK, CMD_ERROR, or CMD_HELP) is used to determine which result to post, and what external result code (TCL_OK or TCL_ERROR) to use.
struct CommandLine : public Shareable {
    const static int CMD_OK = 0;
    const static int CMD_ERROR = -1;
    const static int CMD_HELP = -2;

    TclObj usageList;
    TclObj errorMessage;
    TclObj result;
    HelpText helptext;

    int nargs();
    TclObj arg();
    void pop();

    // check for right nargs, format error message
    bool wrongNArgs(int n, TclObj argnames);

    // check for helparg format usage
    bool argsIsHelp();

    int completeCheck(int rtn);
    int completeExec(int rtn);
};

The HelpText class is used by the command system to build the text for help messages. The class takes care of all of the formatting aspects of building a message such as the one given above. The text is built up in individual lines of varying types (ie. one-column, two-column, separator). As lines are added, the HelpText class keeps track of all of the maximum column widths. When all of the text has been added, the column width data is used to format each line according to the example.

12.2.2 Three Phase Evaluation

Commands implemented with the command system have three parse phases. First, the command is checked for syntax, and scanned for help arguments. The check
phase is carried out by the PipeLine method check(...). Once a command invocation has passed the check phase it enters the exec phase, carried out by the exec(...) method. If help is encountered during the check phase, the command immediately enters the help phase, using the help(...) method. The three-phase evaluation is used to ensure that a command either completes or fails completely. Without the check phase, some commands might parse most of their arguments, and modify the system in some way, only to find an error or help argument. The command cannot complete, but the changes have already been made. The return status is undefined. Since it is still possible for errors to occur during the exec phase, the CommandLine checks for this and issues a warning indicating that the return status is undefined. The help phase is used to facilitate the building of useful help messages, allowing the whole tail of the pipeline to add information.

12.3 ArgStack

All three parse methods take two arguments, the CommandLine, and the ArgStack. The CommandLine, as discussed above, holds the TclObj arguments to the command. The ArgStack holds a stack of typed data. The intent is to translate the string arguments, one at a time, into typed data and put them on the ArgStack. Together the CommandLine and ArgStack are referred to as the CommandPipe.

The ArgStack is a recursive template class that behaves somewhat like proper lisp lists. each item in the stack has a data item and a substack. The substack is shared so that the stack can be built up easily. The data item arg is a reference, so that it can be assigned to as well as from.

```cpp
template <class SArgs, class AType>
struct ArgStack : public Shareable {
    typedef SArgs SubArgs;
    typedef AType ArgType;
    Sharer<SubArgs> sub;
    ArgType & arg;
    ArgStack(Sharer<SubArgs> s, ArgType & t) : sub(s), arg(t) {}
```
12.3.1 PipeLine

The PipeLine is used to divide up command parsing into small reusable chunks. The goal is to be able to build commands with simple statements that reflect what the command does. For example:

- read in an object identifier
- read in a vector argument
- for each object in the identified set
  - select the object’s locale
  - select the locale’s position
  - set the position to the vector argument

Each piece of the pipeline will implement one of the lines above. Commands like this one should be able to be written in one line.

12.3.2 PipeDest and PipeSource

Obviously, the type of the ArgStack depends on what types are in the stack. The same extends to the PipeLine. The PipeLine is made up of individual items that inherit from one or both of the PipeLine interfaces, PipeSource and PipeDestination.

PipeDestination defines a PipeLine item which can receive a CommandPipe from upstream. It takes a template parameter indicating what type of ArgStack it expects in the CommandPipe. (Note that C++ does not permit virtual template methods.)

```cpp
template <class Args>
struct PipeDest : public Shareable {
    typedef Args ArgsType;
    virtual int check(Sharer<CmdLine>, Sharer<Args>) = 0;
    virtual int help(Sharer<CmdLine>, Sharer<Args>) = 0;
};
```
virtual int exec(Sharer<CmdLine>,Sharer<Args>)=0;
};

PipeSource defines a piece of the pipeline which emits the CommandPipe with a
given type of ArgStack. It takes a template parameter DestArgs, giving the type of
the argument stack that it will forward to the PipeDestination. It must be constructed
with a PipeDestination as an argument.

template <class DestArgs>
struct PipeSource {
    typedef PipeDest<DestArgs> Dest;
    Sharer<Dest> dest;
    PipeSource(Sharer<Dest> d) : dest(d) {};
};

12.3.3 Pipes

Clearly most of the PipeLine is made up of items that are both PipeSources and
PipeDestinations. A mechanism for building the pipeline without having to specify
the template parameters is developed in section 12.3.4. In brief, however, the system
relies on the PipeLine items being parametrized in DestArgs rather than Args. Note
that all of the PipeLine items will take DestArgs as their first template parameter.
Furthermore, they all require a PipeDest argument to their constructor. In the rest
of the discussion we will refer to extra parameters and constructor arguments where
necessary, with the assumed parameter and argument being DestArgs and PipeDest
respectively.

The translation from DestArgs to Args (required in order to declare both parts of
the interface) is easily accomplished with knowledge of the input-output relationship
involved, but the implementation requires a significant amount of template manipula-
tion. To make implementation of actual parsers easier, the command system provides
six subclasses of PipeDest to encapsulate the basic operations on the argument stack.
Each provides three methods pipeCheck(...), pipeHelp(...), and pipeExec(...), to be called from
their subclasses, that facilitate construction of the outgoing command pipe.
ArgPusher adds an argument to the stack. It takes no extra parameters, since the type to push is already in the DestArgs stack. The best example of an ArgPusher is PushType, which takes an argument from the CommandLine, translates it into typed data (for example a Vector), and adds it to the stack.

```cpp
template <class DestArgs>
struct ArgPusher : public PipeDest<typename DestArgs::SubArgs>, 
    public PipeSource<DestArgs> {
    typedef PipeSource<DestArgs> Source;
    typedef typename DestArgs::ArgType Type;
    typedef typename DestArgs::SubArgs Args;

    ArgPusher(Sharer<Dest> d) : Source(d) {};
    int pipeCheck(Sharer.CmdLine> c, Sharer<Args> a, Type & t) {
        return dest->check(c, new DestArgs(a,t));
    }
    int pipeExec(Sharer.CmdLine> c, Sharer<Args> a, Type & t) {
        return dest->exec(c, new DestArgs(a,t));
    }
    int pipeHelp(Sharer.CmdLine> c, Sharer<Args> a, Type & t) {
        return dest->help(c, new DestArgs(a,t));
    }
};
```

ArgPasser does nothing to the argument stack except pass it along to the next item. It is supplied for convenience. ChoiceParsers, for example, are ArgPassers.

```cpp
template <class Args>
struct ArgPasser : public PipeDest<Args>, 
    public PipeSource<DestArgs> {
    typedef PipeSource<DestArgs> Source;

    ArgPasser(Sharer<Dest> d) : Source(d) {};
    int pipeCheck(Sharer.CmdLine> c, Sharer<Args> a) {
        return dest->check(c,a);
    }
    int pipeExec(Sharer.CmdLine> c, Sharer<Args> a) {
```
ArgPiper takes the top argument in the stack and changes its type. It takes an extra parameter specifying the incoming type of the top argument (the outgoing type is already specified). There are lots of examples of ArgPipers, but perhaps the most interesting is Foreach, which takes a set as its input argument and forwards each member, in turn, as the top argument.

```cpp
template <class DestArgs, class Typeln>
struct ArgPiper : public PipeDest<ArgStack<typename DestArgs::SubArgs, Typeln>>, public PipeSource<DestArgs> {
    typedef PipeSource<DestArgs> Source;
    typedef typename DestArgs::ArgType TypeOut;
    typedef typename DestArgs::SubArgs SubArgs;
    typedef ArgStack<SubArgs, Typeln> Args;

    ArgPiper(Sharer<Dest>d) : Source(d) {}

    int pipeCheck(Sharer<CmdLine> c, Sharer<Args> a, TypeOut& t) {
        return dest->check(c, new DestArgs(a->sub,t));
    }

    int pipeExec(Sharer<CmdLine> c, Sharer<Args> a, TypeOut& t) {
        return dest->exec(c, new DestArgs(a->sub,t));
    }

    int pipeHelp(Sharer<CmdLine> c, Sharer<Args> a, TypeOut& t) {
        return dest->help(c, new DestArgs(a->sub,t));
    }
};
```

ArgPopper removes the top argument in the stack (presumably consuming it in some way). It takes an extra parameter specifying the type of the argument being
popped. No examples of ArgPopper exist currently.

```cpp
template <class DestArgs, class Type>
struct ArgPopper : public PipeDest<ArgStack<DestArgs, Type>>,
                  public PipeSource<DestArgs> {
    typedef PipeSource<DestArgs> Source;
    typedef ArgStack<DestArgs, Type> Args;

    ArgPopper(Sharer<Dest> d) : Source(d) {}
    int pipeCheck(Sharer<CmdLine> c, Sharer<Args> a) {
        return dest->check(c, a->sub);
    }
    int pipeExec(Sharer<CmdLine> c, Sharer<Args> a) {
        return dest->exec(c, a->sub);
    }
    int pipeHelp(Sharer<CmdLine> c, Sharer<Args> a) {
        return dest->help(c, a->sub);
    }
};
```

DoubleArgPopper removes the top two arguments in the stack for consumption. It takes two extra parameters specifying the type of the top two arguments. An example is Set, which assigns the top argument from the second argument.

```cpp
template <class DestArgs, class Type1, class Type2>
struct DoubleArgPopper
    : public PipeDest<ArgStack<ArgStack<DestArgs, Type2>, Type1>>, Type1>, Type2>>,
    public PipeSource<DestArgs> {
    typedef PipeSource<DestArgs> Source;
    typedef ArgStack<ArgStack<DestArgs, Type2>, Type1> Args;

    DoubleArgPopper(Sharer<Dest> d) : Source(d) {}
    int pipeCheck(Sharer<CmdLine> c, Sharer<Args> a) {
        return dest->check(c, a->sub->sub);
    }
    int pipeExec(Sharer<CmdLine> c, Sharer<Args> a) {
        return dest->exec(c, a->sub->sub);
    }
    int pipeHelp(Sharer<CmdLine> c, Sharer<Args> a) {
        return dest->help(c, a->sub->sub);
    }
};
```
int pipeHelp(Sharer<CmdLine> c, Sharer<Args> a) {
    return dest->help(c, a->sub->sub);
}
}

ArgSwapper swaps the top two arguments. It is an implementation in itself, and is used to reorder the evaluation of the object command to postpone the Foreach operation until all of the arguments have been parsed and stacked.

template <class DestArgs>
struct ArgSwapper

    : public PipeDest<ArgStack<ArgStack<typename DestArgs::SubArgs::SubArgs,
type name DestArgs::ArgType>, typename DestArgs::SubArgs::ArgType>>

    public PipeSource<DestArgs> {
typedef PipeSource<DestArgs> Source;
typedef typename DestArgs::SubArgs DestSubArgs;
typedef typename DestSubArgs::SubArgs Unchanged;
typedef typename DestSubArgs::ArgType Arg1;
typedef typename DestArgs::ArgType Arg2;
typedef ArgStack<Unchanged, Arg2> SubArgs;
typedef ArgStack<SubArgs, Arg1> Args;

ArgSwapper(Sharer<Dest> d) : Source(d) {}
int check(Sharer<CmdLine> c, Sharer<Args> a) {
    DestSubArgs* s = new DestSubArgs(a->sub->sub, a->arg);
    DestArgs * d = new DestArgs(s, a->sub->arg)
    return dest->check(c, d);
}
int exec(Sharer<CmdLine> c, Sharer<Args> a) {
    DestSubArgs* s = new DestSubArgs(a->sub->sub, a->arg);
    DestArgs * d = new DestArgs(s, a->sub->arg)
    return dest->exec(c, d);
}
int help(Sharer<CmdLine> c, Sharer<Args> a) {
    DestSubArgs* s = new DestSubArgs(a->sub->sub, a->arg);
    DestArgs * d = new DestArgs(s, a->sub->arg)
    return dest->help(c, d);
12.3.4 Pipers

Building up PipeLines is difficult if all of the argument types need to be specified. In order to make the system easier to use, temporary pipe-building objects called Pipers are used. Pipers are essentially a class encapsulation of the PipeLine Item's template. Given a PipeDest to operate on, a Piper can build a Pipe of the right type.

Pipers support three different syntaxes. The recommended syntax is the shift-assignment syntax, shown below, implementing the command outlined on page 128. The shift-assignment operator is chosen because it (like all of the assignment operators) is right-associative. As implied above, PipeLines are constructed from the endpoint backwards to the start. Using a right-associative operator allows the pipeline to be laid out in evaluation order from left to right, but to be constructed from right to left. The right-shift operator is used to support the inverted syntax, where the end of the pipe is on the left, and the command token is on the right. Finally, a method syntax (using the method cat(...)) is provided to work around a bug in gcc version 2.95, that fails when compiling either operator syntax. This syntax is highly deprecated since it requires a confusing number of parentheses.

```cpp
    cmd = Begin('"set.x") <<= PushElements() <<= Push('"x'")
    <<=Swap() <<= Foreach()
    <<=Select<Element>(&Element::locale)
    <<=Select<Locale>(&Locale::x)
    <<=Set() <<= End();
```

A different kind of Piper must be used for Pipes that take different numbers of template parameters, and constructor arguments. Currently there are pipers for Pipes that take one extra parameter, one extra argument, one extra of each, and two extra parameters. The naming convention is as follows. The class name is Piper optionally followed by Tn where n is the number of extra template parameters, optionally followed by Cn, where n is the number of extra constructor arguments. Note that in the case
of constructor arguments, the Piper must also take the constructor argument, and
must have a member of that type (with which to construct the Pipe).

The basic functionality of the Pipers is the same, the operator takes an argument
of type PipeDest<DestArgs>, where DestArgs is a template parameter of the operator. It
creates a new Pipe of type DestArgs, and returns it.

```cpp
template <template <class> class Pipe>
struct Piper {
  template <class DestArgs>
  Pipe<DestArgs> * operator<<(PipeDest<DestArgs> * d) {
    return new Pipe<DestArgs>(d);
  }

  template <class DestArgs>
  Pipe<DestArgs> * cat(PipeDest<DestArgs> * d) {
    return new Pipe<DestArgs>(d);
  }

  template <class DestArgs>
  friend Pipe<DestArgs> * operator>>(PipeDest<DestArgs> * d, Piper<Pipe> p) {
    return p.cat(d);
  }

  template <class DestArgs>
  friend Pipe<DestArgs> * cat(PipeDest<DestArgs> * d, Piper<Pipe> p) {
    return p.cat(d);
  };

  template <template <class, class> class Pipe, class T1>
  struct PiperT1 {
    template <class DestArgs>
    Pipe<DestArgs, T1> * operator<<(PipeDest<DestArgs> * d) {
      return new Pipe<DestArgs, T1>(d);
    }

    template <class DestArgs>
    Pipe<DestArgs, T1> * cat(PipeDest<DestArgs> * d) {
      return new Pipe<DestArgs, T1>(d);
    }
  };
};
```

135
template <class DestArgs>
friend Pipe<DestArgs,T1> * operator>>(PipeDest<DestArgs> * d,
PiperTi<Pipe,Ti> p) {
    return p.cat(d);
}

template <class DestArgs>
friend Pipe<DestArgs,T1> * cat(PipeDest<DestArgs> * d,
PiperTi<Pipe,T1> p) {
    return p.cat(d);
}

};

template <template <class> class Pipe, class Ci>
struct PiperCi {
    Ci ci;
    PiperCi(Ci cli) : ci(cli) {}
    template <class DestArgs>
    Pipe<DestArgs> * operator<<(PipeDest<DestArgs> * d) {
        return new Pipe<DestArgs>(d,ci);
    }
    template <class DestArgs>
    Pipe<DestArgs> * cat(PipeDest<DestArgs> * d) {
        return new Pipe<DestArgs>(d,ci);
    }
}

template <class DestArgs>
friend Pipe<DestArgs> * operator<<(PipeDest<DestArgs> * d,
PiperC1<Pipe,Ci> p) {
    return p.cat(d);
}

template <class DestArgs>
friend Pipe<DestArgs> * cat(PipeDest<DestArgs> * d, PiperC1<Pipe,Ci> p) {
    return p.cat(d);
}
template <template <class, class> class Pipe, class T1, class Cl>
struct PiperT1C1 {
    Cl c1;
    PiperT1C1(Cl c1in) : c1(c1in) {}  
    template <class DestArgs>
    Pipe<DestArgs,T1> * operator<<(PipeDest<DestArgs> * d) {
        return new Pipe<DestArgs,T1>(d,c1);
    }
    template <class DestArgs>
    Pipe<DestArgs,T1> * cat(PipeDest<DestArgs> * d) {
        return new Pipe<DestArgs,T1>(d,c1);
    }
};

template <template <class, class, class> class Pipe, class T1, class T2>
struct PiperT2 {
    template <class DestArgs>
    Pipe<DestArgs,T1,T2> * operator<<(PipeDest<DestArgs> * d) {
        return new Pipe<DestArgs,T1,T2>(d);
    }
    template <class DestArgs>
    Pipe<DestArgs,T1,T2> * cat(PipeDest<DestArgs> * d) {
        return new Pipe<DestArgs,T1,T2>(d);
    }
};

137
friend Pipe<DestArgs,T1,T2> * operator>>(PipeDest<DestArgs> * d, 
        PiperT2<Pipe,T1,T2> p) {
        return p.cat(d);
    }

template <class DestArgs>
    friend Pipe<DestArgs,T1,T2> * cat(PipeDest<DestArgs> * d, 
        PiperT2<Pipe,T1,T2> p) {
        return p.cat(d);
    }
};

To complete the syntax shown, and remove all of the unnecessary template parameters, the last layer of the pipe-building system is the helper function. Each implemented Pipe class (for example ArgSwapper, or PushType) also provides a helper function for generating the appropriate Piper. The Piper for ArgSwapper is generated with the Swap() function. The Piper for PushType( which takes a constructor argument naming the arg to be taken from the command line) is generated by Push(...).

Piper<ArgSwapper> Swap() {
    return Piper<ArgSwapper>();
}

PiperCl<PushType,TclObj> Push(TclObj n) {
    return PiperCl<PushType,TclObj>(n);
}  

12.3.5 Implemented Pipes

The following pipes have been implemented for command development at the time of publication. The pipes are given by their helper functions, with required arguments.

- **Discard()**: Discard the top arg.

- **Swap()**: Swap the top two arguments on the stack.

- **PushInternal(Type)**: Push an already existing internal type.
Push(TclObj name): Convert a command argument to internal type.

PushElement(): Convert an element identifier to a set of elements.

PushBehavior(): Convert a behavior identifier to a set of behaviors.

Foreach<Container>(): Select each member of the top argument set in turn.

Select<Type>(Type (C::*)): Select a member from a the top arg.

Set<Type>(): Set the top argument using the second argument (same type).

CastSet<Dst,Src>(): Set the top argument using a cast of the second argument.

Call<Rtn,Arg>(Rtn (*)(Arg)): Call a function with the top arg as its argument, replace with the return.

CallVoid<Arg>(void (*)(Arg)): Call a void function with the top arg as its argument, pop both.

CallMethod<Rtn,Arg>(Rtn (C::*)(Arg)): Call a method of the top arg with the second arg as its argument, replace both with the return.

CallVoidMethod<Arg>(void (C::*)(Arg)): Call a void method of the top arg with the second arg as its argument, pop both.

OptionList(TclObj name): Parse a command argument as an option, forward command pipe accordingly.

SubCommandList(TclObj name): Parse a command argument as a subcommand, forward command pipe accordingly.

12.4 Benefits

The command system presented in this chapter provides a number of benefits. The infrastructure provides a set of easy to use modular parsing elements. These parsers
make command development simpler, but also more uniform. They effectively separate the generic details of command parsing from the particular operations of a given command through a very simple command definition language. Since much of the functionality is built into the parsers, the resulting commands will have a consistent style, and will provide a consistent set of features including pre-check parsing, and help support. This will add to the command interface in usability, as well as uniformity.

The command system infrastructure also provides layers of separation between the core code and the user-interface. This means that if the interface language is changed, the command system can be reworked without major changes to the rest of the core. Along less drastic lines, it also means that individual command components (e.g. PushType, Foreach etc.) can be debugged and improved on a system-wide basis.

12.5 Potential Issues

While the command system presented provides much of what is desired in isolating the core code from the generic concerns of user-interface building, it poses some potential issues. The PipeLine implementation is not particularly efficient since it relies on virtual methods to pass the command pipe along the pipeline. The overhead is not expected to be great, however, since the commands are not expected to be very complex. Furthermore, the command interpreter is not expected to conform to the same performance standards as the core, because of the inherent performance limitations of parsing and of user interaction.

Another, potentially more serious problem with the PipeLine is that the compiler errors generated when a PipeLine is improperly constructed are uninformative. If for example, the types in the pipeline do not match, the compiler will likely report seemingly unrelated problems with the Piper operators. On the other hand, the command definition syntax is simple enough that the mere indication of a problem should be sufficient to correct typing discrepancies. Finally it is noted that the PipeLine will generate a large number of distinct types, all with virtual methods. This may incur
a certain amount of size overhead.

The problems presented by the PipeLine system arise from the difficulty of passing a structure of typed arguments, and the problems associated with automatically typing that structure. This problem is inextricably linked to the ability to provide truly modular parsers. It is our belief that the benefits of a completely contained parsing system outweigh the drawbacks.
Chapter 13

Architecture: Contributions

As discussed in chapter 7, one of the primary benefits of the MIMES environment has been its use as a development testbed for new methods of discrete element computation. This despite the fact that MIMES was originally conceived of as a numerical laboratory for application-level use. After years of extension and revision, however, MIMES has reached a point where further extension becomes difficult.

In this work a new architecture has been developed with the explicit goal of supporting the role of the simulator as a development testbed. The architecture has, therefore focused on providing minimal implemented functionality within a well-constructed framework, and built on top of a generic infrastructure. In particular the architecture has built in support for the following extension areas:

In particular, the Architecture provides support for extended element behaviors, generic component management, and extendable module management. Furthermore, through the command system, the Architecture provides a fully self-contained command-interface builder, that constructs highly usable and consistent commands from basic modular elements.

The key parts of the Architecture have been presented in detail, with special care taken to outline the design decisions involved. It is hoped that this will provide the architecture with guiding principles for future development, and further strengthen the design which the core architecture attempts to capture.
Appendix A

CGrid Implementation

The C++ implementation of CGrid is presented below. In keeping with the formulation of the algorithm, the subdividers are defined independant of dimension. The CGrid pipeline is instatiated with a single CGridSubdivider, which constructs all of its succesors. The CGridSort class provides a wrapper around the whole, and pushes the simulation objects into the pipeline.

    //provide some convenient types,
    //and some glue into the rest of the DEM code:
struct CGridParent : public AListParent {
    typedef unsigned int Size;   //array length type
    typedef unsigned int Index;  //index into various tables
    typedef unsigned int BArray; //index into Array of BucketArrays (small)
    typedef Index Bucket;        //discretized space ordinate
    typedef AList2::ItemRef Obj; //reference to an object

    static DemScalar cellFactor;

    //call the DEM local contact code:
    static void check_contact(Index o1, Index o2) {
        world->object_table[o1]--find_contact_detail(world->object_table[o2]);
    }

    //debugging code:
    static ostream& ALERT() {cout << "ALERT!!!!!!!!!!! ", return cout;};
};

//-----------------------
//CellFactor:
//This scheme is valid independent of the cell/object size ratio, but its performance is clearly
//worse for bad choices. This static member of CGridParent allows the cell size to be varied
//from the internally generated number (currently the min object size) at runtime.
//We can use this feature to develop performance profiles, and eventually we should be
//able to come up with a better estimate of the optimal cell size for a given object-size profile
DemScalar CGridParent::cellFactor = 1.0; //default is One

template <int AXIS>
class CGridAxis : public CGridParent {

public:
    Size ncell;
    DemScalar cellsize;

    typedef std::vector<Index> ArrayType;
    ArrayType upper_cell;
    ArrayType lower_cell;

    static DemScalar getSize(const Index i) {
        return (world->object_table[i]->elprops->bsphr);
    }

    static DemScalar getPosition(const Index i) {
        return world->object_table[i]->elstate->position[AXIS];
    }

    static Size table_size() {return world->object_table.length();}

    static bool ignore(const Index i) {return (world->object_table[i]->get_uid() < 0);}

protected:
    DemScalar minx, maxx;

    static DemScalar getXUpper(const Index i) {
        return (getPosition(i) + getSize(i));
    }

    static DemScalar getXLower(const Index i) {
        return (getPosition(i) - getSize(i));
    }

};
The following are used to discretize the ordinates obtained above

```cpp
Index get_cell_lower(const Index i) const {
    return (Index)floor((get_x_lower(i) - minx) / cellsize);
}

Index get_cell_upper(const Index i) const {
    return (Index)floor((get_x_upper(i) - minx) / cellsize);
}

public:

CGridAxis() {
    cellsize = minx = maxx = 1.0;
    ncell = 0;
}

void step_init() {
    int nobj = table_size();
    if (nobj == 0) {
        cellsize = minx = maxx = 1.0;
    } else {
        maxx = get_x_upper(0);
        minx = get_x_lower(0);
        cellsize = 2.0 * get_size(0);
        for (Index i = 1; i < nobj; i++) {
            if (ignore(i))
                continue;
            // get the smallest object, and use that as the cellsize
            cellsize = min(2.0 * get_size(i), cellsize);
            maxx = max(get_x_upper(i), maxx);
            minx = min(get_x_lower(i), minx);
        }
    }

    // adjust cellsize
    // (for testing hopefully we can find a way to optimize)
    cellsize *= cellFactor;

    // get limits:
    minx = minx - cellsize / 2.0;
    maxx = maxx + cellsize / 2.0;
}
```
//determine discretization characteristics:
ncell = (Index)ceil((maxx - minx)/cellsize);

//resize vectors in chunks
if (lower.cell.capacity() < nobj) {
    lower.cell.reserve(int(nobj*ARRAY.SIZE_FACTOR));
    upper.cell.reserve(int(nobj*ARRAY.SIZE_FACTOR));
}
lower.cell.resize(nobj);
upper.cell.resize(nobj);

//make the discretization table (maps objects to cell numbers:
for (Index i=0;i<nobj;i++) {
    if (ignore(i)) {
        lower.cell[i] = ncell;
        upper.cell[i] = 0;
    } else {
        lower.cell[i] = get_cell_lower(i);
        upper.cell[i] = get_cell_upper(i);
    }
}

if (DEBUG) {
    cout << "Index["<<AXIS<<""
    for (Index i=0;i<nobj;i++)
        cout << i <<"",
    cout << endl;
}

//general subdivider takes template args for the number of dimensions to remove
//and the axis on which to remove this dimension.
template <int NAXES,int THIS_AXIS=0>
class CGridSubdivider : public CGridAxis<THIS_AXIS> {
protected:
    //the next subdivider is on the next axis, and has 1 fewer axes remaining:
typedef CGridSubdivider<NAXES,THIS_AXIS> Self;
typedef CGridSubdivider<NAXES-1,THIS_AXIS+1> NextAxis;

    Bucket currentBucket;
    Sharer<AQueue> OBQ;
    Sharer<AListShared> shared;

}
AListArray array[AllBArrays];

NextAxis next_axis; // the next axis in the subdivision process

public:
void step_init() {
   shared->resize(table_size);
   CGridAxis<THIS_AXIS>::step_init();
   bool b=0;
   for(BArray i = 0; i< AllBArrays;i++)
      b = array[i].resize(ncell) | b;
   if (b) {
      std::cout << "Resizing stage " << THIS_AXIS << " Subdivider " << (array0.size()) << "\n" << endl;
   }
   next_axis.step_init();
}

CGridSubdivider() : shared(new AListShared()), next_axis() {
   // replace the occupied queues of the current arrays with a shared one:
   OBQ = new AQueue();
   BArray i;
   for(i = 0; i<AllBArrays;i++) {
      array[i].shared = shared;
      array[i].occupied = OBQ;
   }
}

void add_objects(AListArray srcArray[CurBArrays], const Bucket b, const AListShared* src) {
   ItemRef o;
   for (int i=0;i<CurBArrays;i++) {
      o = srcArray[i].begin(b);
      for (; !empty(o); o = src->next(o)) {
         array[i].push(lower_cell[o],o);
      }
   }
   purgeOccupied();
}

void purgeOccupied() {
   bool b;
   AQueue::iterator i;
   if (DEBUG) std::cout << "purging\n" << *OBQ << std::endl;
}
for(I=OBQ->begin();!empty(I);I=OBQ->next(I)) {
    b = false;
    for (Index i=0;i<AllBArrays;i++)
        if (!empty(array[i].begin(I)))
            b = true;
        if (!b) {
            if (DEBUG) cout << I << "\n";
            OBQ->remove(I);
        }
    }
    if (DEBUG) cout << "purged\n";
}

//transfer current half into collector
void transfer() {
    for (BArray i = 0; i < CurBArrays; i++) {
        array[i+CurBArrays].append(array[i]);
        array[i].clear();
    }
}

//decide if an object should be weeded from the collector bucket
bool weed_obj(ItemRef o) {
    return (upper_cell[o] < (Index) currentBucket);
}

//scan collector bucket for weeds
template <class Subdivider>
void weed(Subdivider &prev_axis) {
    ItemRef l;
    for(l = OBQ->begin();!empty(l); l = OBQ->next(l)) {
        for (BArray i = CurBArrays; i < AllBArrays; i++) {
            //check that the list really is occupied
            if (empty(array[i].begin(I)))
                continue;

            //weed out the tail (non-head) items in the list:
            ItemRef o, o_prev=array[i].begin(I);
            for(o = shared->next(o_prev); !empty(o); o = shared->next(o_prev)) {
                if (prev_axis.weed_obj(o)) {
                    array[i].shared->remove_after(o_prev,o); //remove o, and maintain o.prev
                } else {
                    o_prev = o; //keep o, and advance
                }
            }
        }
    }
}

148
array[i].tail[i] = o_prev;

//check head object:
if (prev_axis.weed_obj(array[i].begin(I)))
    array[i].remove_head(I);
}
}

//clear all of the arrays
void clear() {
    for (BArray i = 0; i < AllBArrays; i++)
        array[i].clear();
}

void subdivide() {
    BArray i;

    if (empty(currentBucket = OBQ->begin()))
        return;

    if (DEBUG) tab(cout) << "stage " << THIS_AXIS << " begin --- \n";
    if (DEBUG) report(0,2);
    while (1) {
        if (DEBUG) tab(cout) << "stage " << THIS_AXIS << " bucket(" << currentBucket << ")" << endl;
        next_axis.add_objects(array, currentBucket, shared);

        if (DEBUG) tab(cout) << "added objects\n";
        if (DEBUG) next_axis.report(0,1);

        //recurse subdividers:
        next_axis.subdivide();

        //advance to next cell:
        if (empty(currentBucket = OBQ->next(currentBucket)))
            break;

        //tell next axis to transfer objects from current to collector barrays
        next_axis.transfero;

        if (DEBUG) tab(cout) << "xferd objects\n";
        if (DEBUG) next_axis.report(1,2);
// tell next axis to kill the collector objects which don't overlap cells >= c
next_axis.weed(*this);

if (DEBUG) tab(cout) << "weed objects\n";
if (DEBUG) next_axis.report(1,2);
}

if (DEBUG) tab(cout) << "stage " << THIS_AXIS << " complete ---- \n";
// clear the collector object lists completely (since there are no more current objects)
next_axis.clear();
if (DEBUG) tab(cout) << "objects cleared\n";
if (DEBUG) next_axis.report(0,2);

ostream& tab(ostream & s) {
  for (BArray i = 0; i<=THIS_AXIS;i++) {
    s << " ";
  }
  return s;
}

void report(int a, int b) {
  for (int i = AllBArrays*a/2; i < AllBArrays*b/2; i++) {
    if (DEBUG) tab(cout) << "o["<<THIS_AXIS<<"]/"<<i<<":"<< array[i] << endl;
  }
}
};

/////////////////////////////////////////////////////////////////////////
// Specialize for last AXIS (not an axis at all, but rather a contact cell:
// if recursed correctly from above, AXES will give the number of axes total
// (which gives the log of the number of barrays)

// Note that VisualStudio does not support partial specialization
// Work around this by commenting out the template line,
// and #define'ing the DIMENSIONS

template <int DIMENSIONS>
class CGridSubdivider<0,DIMENSIONS> : public CGridAxis<DIMENSIONS> {
protected:
  Sharer<AListShared > shared;
  AList2 array[AllBArrays];
public:

void step_init() {shared->resize(table.size());}

CGridSubdivider() {
    shared = new AListShared();
    for (BArray i = 0; i < AllBArrays; i++)
        array[i].shared = shared;
}

void add_objects(AListArray srcArray[CurBArrays], const Bucket b, const AListShared* src) {
    ItemRef o;
    for (int i = 0; i < CurBArrays; i++) {
        o = srcArray[i].begin(b);
        for (; !empty(o); o = src->next(o)) {
            array[i].push(o);
        }
    }
}

void transfer() {
    for (BArray i = 0; i < CurBArrays; i++) {
        array[i+CurBArrays].append(array[i]);
        array[i].clear();
    }
}

template <class Subdivider>
void weed(Subdivider &prev_axis) {
    for (BArray i = CurBArrays; i < AllBArrays; i++) {
        //weed out the tail (non-head) items in the list:
        ItemRef o, o_prev = array[i].begin();
        if (empty(o_prev))
            continue;
        for (o = shared->next(o_prev); !empty(o); o = shared->next(o_prev)) {
            if (prev_axis.weed_obj(o)) {
                array[i].remove_after(o_prev, o); //remove o, and maintain o_prev
            } else {
                o_prev = o; //keep o, and advance
            }
        }
        array[i].tail = o_prev;
        //check head object:
        if (prev_axis.weed_obj(array[i].begin()))
        }
    )
}
```cpp
array[i].remove_head();
}

void clear() {
    for (BArray i = 0; i < AllBArrays; i++)
        array[i].clear();
}

//definitions provided below:
void subdivide();

ostream& tab(ostream & s) {
    for (int i = 0; i <= DIMENSIONS; i++) {
        s << " ";
    }
    return s;
}

void report(int a, int b) {
    for (int i = AllBArrays*a/2; i < AllBArrays*b/2; i++) {
        if (DEBUG) {
            tab(cout) << "o[" << DIMENSIONS << "]" << i;
            array[i].shared->dump(array[i].head, cout) << endl;
        }
    }
}

////////////////////////////////
//General definition for the last–axis (contact cell) "subdivision" process
//In this procedure, we have placed objects in each of the barrays, which, if arranged in an
//N–dimensional cube, indicate the barray of those objects w/ respect to the current
//cell in corresponding axes. The relation is clear if the barray numbers are expressed in binary:
//
//An example in 3–D,
// if "current in the A axis" means that the discretized lowest ordinate on that axis
// is equal to the value of currentBucket in the corresponding subdivider.
// (which in turn corresponds to the first visitation during this pass)
// and "collected on the A axis" means that the discretized lowest ordinate
// is less than the value of currentBucket in the corresponding subdivider.
```

152
// (which in turn implies a previous visitation during this pass)
//barray 000 contains objects which are current in all axes
//barray 001 contains objects which are current in x and y, but are collected in the z-axis.
//barray 010 contains objects which are current in x and z, but are collected in the y-axis.
//barray 100 contains objects which are current in y and z, but are collected in the x-axis.
//barray 111 contains objects which are collected in all axes

// Using this notation, all pairings of barrays which are not less-than-current on the same
// axis (ie. do not share any true bits) contain objects which may overlap, and have not yet
// been checked for overlap. Note that this rule applies correctly to barray 000 paired with
// itself, but that since no other barrays pair with themselves, it is desirable to
// explicitly check only 000 against itself. This simplifies the rule-based pairings (each
// barray is tested against every greater barray) and also allows the straightforward
// optimization of the self-check to implicitly eliminate duplicate pairings.

#ifndef WIN32

template <int D>
inline void CGridSubdivider<O,D>::subdivide() {
  if (DEBUG) tab(cout) << "concat-cell:"<<endl;
  if (DEBUG) report(0,2);
  Obj o, o2;
  // check self-contact for barray
  for (o = array[0].begin(); !empty(o); o = shared->next(o)) // each object
    for (o2 = shared->next(o); !empty(o2); o2 = shared->next(o2)) // each subsequent object
      check_contact(o, o2);

  // apply general rule to all other barray pairings (each barray against its successors)
  // noting that no pairings exist within the second half (> CURRENT) of the barray-array
  // (ie with 1 as the big bit),
  // we check all "CURRENT" barrays (0 to NewBArrays) against all of their successors:
  for (BArray i = 0; i < NewBArrays; i++) {
    for (BArray ii = i+1; ii < AllBArrays; ii++) {
      if (i & ii) // (the pairing rule)
        continue;
      // pair up the objects in the two lists:
      for (o = array[i].begin(); !empty(o); o = shared->next(o)) {
        for (o2 = array[ii].begin(); !empty(o2); o2 = shared->next(o2)) {
          check_contact(o, o2);
        }
      }
    }
  }
#endif
//extra-special specialization for 2D
//If the optimizer doesn't unroll the loop above,
//this might be faster since the rule is mostly false
//similar explicit definitions could (and maybe should)
//be made for any dimensionality.
inline void CGridSubdivider<0,2>::subdivide() {
  if (DEBUG) tab(cout) << "concat-cell:"<<endl;
  if (DEBUG) report(0,2);
  //check self-contact for array[0] (ie all current)
  Obj o=0, o2=0;
  for (o = array[0].begin(); !empty(o); o = shared->next(o)) {
    for (o2 = shared->next(o); !empty(o2); o2 = shared->next(o2))
      check_contact(o,o2);
    for (o2 = array[1].begin(); !empty(o2); o2 = shared->next(o2))
      check_contact(o,o2);
    for (o2 = array[2].begin(); !empty(o2); o2 = shared->next(o2))
      check_contact(o,o2);
    for (o2 = array[3].begin(); !empty(o2); o2 = shared->next(o2))
      check_contact(o,o2);
  }
  for (o = array[1].begin(); !empty(o); o = shared->next(o))
    for (o2 = array[2].begin(); !empty(o2); o2 = shared->next(o2))
      check_contact(o,o2);
}

//CGridSort:
//Front-class which conforms to the generalized sorter interface.
//it manages timestepping, and pumps the objects into the first subdivider for checking
//CGridSort:
template<int D>
class CGridSort : public Sorter, public CGridSubdivider<D> {
  typedef CGridSubdivider<D> Subdivider;

  public:
  void init();
  void step_init();
  //this essentially corresponds to a subdivide routine for the -1'st AXIS
  void find_contact_neighbors();
  //dummy function -- for future use
  void find_my_far_neighbors(Vector3D center, double radius, list * append_to_me) {};
};
template<int D>
void CGridSort<D>::init() {
    //get the user-specified cell-factor
    cellFactor = world->params->get_tclvard("CELL_FACTOR");
    if (cellFactor <= 0.0) {
        tclerr << tclendl << "WARNING: Invalid cell factor (" << cellFactor << ") resetting to 1.0" << tclendl;
        cellFactor = 1.0;
        world->params->set_tclvard("CELL_FACTOR", 1.0);
    }
    tclout << "Initializing a CGrid2 Sorter (CELL_FACTOR " << cellFactor << ")" << tclendl;
}

template<int D>
void CGridSort<D>::step_init() {
    DemScalar dtemp = world->params->get_tclvard("CELL_FACTOR");
    if (cellFactor != dtemp) {
        if (cellFactor <= 0.0) {
            tclerr << tclendl << "WARNING: Invalid cell factor (" << dtemp << ") keeping old value (" << cellFactor << ")" << tclendl;
            world->params->set_tclvard("CELL_FACTOR", cellFactor);
        } else {
            cellFactor = dtemp;
            tclout << "Reinitializing with CELL_FACTOR : " << cellFactor << tclendl;
        }
    }
    Subdivider::step_init();
}

//this essentially corresponds to a subdivide routine for the -1'st AXIS

template<int D>
void CGridSort<D>::find_contact_neighbors() {
    Bucket c;
    for (Index o = 0; o < table.size(); o++) {
        if (lower_cell[o] < ncell) {
            c = lower_cell[o];
            array[0].push(c, o);
        }
    }
    purgeOccupied();
    subdivide();
    array[0].clear();
}
Bibliography


[36] G. Hocking, J. R. Williams, and G. G. W. Mustoe. Dynamics global forces on an offshore structure from large ice flow impacts. In ARCTIC '85 [7].


[71] Oliver Wensel and Nenad Bićanić. A quad tree based contact detection algorithm. In Williams and Mustoe [96].

[72] J. R. Williams. The dimensional analysis of multiple bodies including automatic fracturing. [3].


[76] J. R. Williams and Chen A. Ice island impact on conical production structures. In Mustoe et al. [52].


[79] J. R. Williams, D. Lim, and A. Gupta. Software design of object oriented discrete element systems. [4].


