A Distributed Simulation Environment for Multibody Physics

by

Jen-Diann Chiou

B.S., Civil Engineering
National Taiwan University, 1992
S.M., Civil and Environmental Engineering
Massachusetts Institute of Technology, 1996

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

August 1998

© Jen-Diann Chiou, MCMXCVIII. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

Author ....
Department of Civil and Environmental Engineering
August 14, 1998

Certified by.. ....
John R. Williams
Department of Civil and Environmental Engineering
Thesis Supervisor

Accepted by ....
Joseph M. Sussman
Chairman, Departmental Committee on Graduate Students

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
OCT 26 1998
LIBRARIES

Eng

A Distributed Simulation Environment for Multibody Physics

by

Jen-Diann Chiou

Submitted to the Department of Civil and Environmental Engineering
on August 14, 1998, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Information Technology

Abstract

A distributed simulation environment, which can be used to model multibody physics, is developed. The software design is based on the object oriented paradigm and is implemented in C++ to run on a single workstation or multiple processors in parallel. It provides facilities to set up a multibody physics simulation, including arbitrary 3D geometric representation, particle interactions such as contacts and constraints, and visualization for postprocessing.

Contact detection, the process of automatic identifying the geometric overlap between objects, is generally the most time-consuming procedure in the overall discrete element analysis pipeline. The computational cost of contact detection grows as a function of both the number of particles and the complexity of the geometric representation of each body. This thesis presents algorithms that significantly reduce the computational cost of the contact detection problem. The hashtable-based spatial reasoning algorithm demonstrates an $O(M)$ performance, where M is the number of particles in the simulation system for a restricted set of particles.

The discrete function representation (DFR) scheme is employed to model the surface geometry of complex 3D objects. DFR-based contact detection between a pair of objects exhibits an $O(N)$ running time performance, where N is the number of surface point used to represent each object. In practice this results in a significant speedup over traditional techniques.

A distributed DEM simulation environment is built on top of a set of software tools which exploit the parallelism embedded in the DEM analysis and which take advantage of a high-speed communications network to achieve good parallel performance. The goal is of reducing the entire computing time of of large-scale simulation problems to order $O(N)$ is shown to be achieveable using the algorithms described.

Thesis Supervisor: John R. Williams
Title: Associate Professor

Acknowledgments

This work would not have been possible without help of many individuals and institutions.

First, I am indebted to my thesis advisor, Professor John R. Williams, for his incessant support, advice, and encouragement.

Professors Jerome Connor, Eduardo Kausel, and Kevin Amaratunga, for their invaluable advice and insight.

I gratefully acknowledge Professors Shang-Hsien Hsieh, Ming-Teh Wang, San-Cheng Chang, and Yeong-Bin Yang, for their encouragement and help.

I deeply appreciate Drs. Ruaidhrí O'Connor and Dale Preece from Sandia National Laboratories, for their timely guidance, advice, and support.

To Mr. Petros Komodromos, for his help on preparing for the thesis defense.

To Mr. Felix Yi-Da Ho and Ms. Sylvia Chao and their family, for their support and taking care of my life here in Cambridge.

To Ms. Tina Lo and her family, for their encouragement and friendship.

To Mr. Bang-Yen Lin and his wife Kathy Chen, for their sincere friendship.

To Mr. Kenji Iwamoto and colleagues from Kajima Corporation, for giving me the wonderful working experience in Tokyo.

To Ms's Janet Tsai, Faustina Tsai, Chu-Jun Huang, Sandy Lin, and their family, for their friendship and encouragement.

To Ms. Joan McCusker, for her help and taking care of me here in MIT.

To Mr. Petros Komodromos and Ms. Katherine Treash, for their help on preparing for the thesis.

To Ms. Ching-Wen Hsu and her family, for their thoughtfulness and encouragement throughout these years.

To Ms's. Jane Chiou and Sylvia Chiou, for their love and help.

This thesis is dedicated to my parents, to whom I owe the most, for their unconditional support and sacrifice over all these years.

Contents

1 Introduction **12**

1.1 Motivation 12

1.1.1 Problem Description 13

1.1.2 Example 13

1.2 Thesis Objectives 14

1.3 Discrete Element Method 16

1.4 Computational Requirements in DEM 17

1.4.1 Object Representation Requirements 18

1.4.2 Contact Detection Requirements 20

1.4.3 Contact Resolution Requirements 20

1.4.4 User Interface and Visualization Requirements 20

1.5 Characteristics of the **DEM3D** Simulation System 21

1.5.1 Basic Functionality 21

1.5.2 Object-Oriented Technology 22

1.6 Distributed and Parallel Implementation 22

1.7 Review of Other DEM Systems 23

1.8 Thesis Outline 23

2 Algorithms for Spatial Reasoning **26**

2.1 Introduction 26

2.2 Spatial Sorting Algorithm 29

2.2.1 Introduction 29

2.2.2 Spatial Heapsort 31

2.2.3 Example 31

2.2.4 Implementation 34

2.2.5 Performance Analysis 35

2.3 Hashtable-based Spatial Reasoning Algorithm 37

2.3.1 Introduction 37

2.3.2 Hashtable Data Structure 44

2.3.3 Implementation 44

2.3.4 Extensions 46

2.3.5 Performance Analysis 46

2.4 Comparison 49

2.5 Summary 49

3 Object Representation and Contact Detection **50**

3.1 Geometric Representation 51

3.1.1 Superquadratic Representation 52

3.2 Discrete Function Representation 54

3.2.1 Description of DFR Scheme 56

3.2.2 DFR in 3D 57

3.3 Performance Analysis 60

3.4 Collision Response 60

3.4.1 Equations of Motion 61

3.4.2 Incremental Collision Resolution Scheme 64

3.4.3 Contact Table Data Structure 66

3.4.4 Constraints 68

3.5 Volume Integration 69

3.6 Summary 70

4 Implementation **71**

4.1 Architectural Framework 71

4.2 Visualization Graphics 72

4.2.1 Open Inventor 74

4.2.2 Virtual Reality Modeling Language 77
4.3 Matrix Manipulation Functions [37] 80
4.4 Simulation Description Language 82
4.4.1 Tcl - Tool Command Language 82
4.4.2 Independence of Software Modules 83
4.5 Data Abstraction 84
4.6 A Sample **DEM3D** Application Script 85
4.7 Summary 87

5 Distributed Discrete Element Simulation 89
5.1 Motivation 89
5.2 Characteristics of Parallel Computing 90
5.2.1 Shared-Memory Multiprocessors Machines 90
5.2.2 Communication Networks 92
5.3 Parallel Programming Models 92
5.3.1 The Data Parallel Programming Model 94
5.3.2 The Multithreaded Programming Model 95
5.3.3 The Message Passing Programming Model 96
5.4 The Message Passing Interface Standard 98
5.5 Development of Parallel Contact Detection Algorithm 99
5.6 Implementation Details 100
5.7 Performance Analysis 101
5.8 Distributed Discrete Element Simulation 102
5.9 Summary 105

6 Examples and Applications 106
6.1 Dynamic Impact Simulation 107
6.2 Contact Damping Simulation 110
6.3 Sandglass Simulation 110
6.4 Collapse of Embankment Simulation 110
6.5 Wave Propagation Simulation 116

6.6 Particles on the Slope 116

6.7 Fracture of a Fixed-end Beam 116

6.8 DFR Packing 120

6.9 Summary 120

7 Conclusions and Future Work **122**

7.1 Summary 122

7.2 Contributions 122

7.3 Related Applications 123

7.4 Future Work 124

A DEM3D Input Command Reference **126**

A.1 Introduction 126

A.2 Commands 126

List of Figures

1-1 DEM analysis pipeline 18

1-2 Areas of application for the discrete element method 19

1-3 Generic DEM Simulation Procedure 23

1-4 The Object Model of **DEM3D** system 24

2-1 Example of Spatial Sorting Algorithm 32

2-2 Performance of Spatial Heapsort Algorithm for 3D Particles for a Time Step 36

2-3 Example of Hashtable-based Spatial Reasoning Algorithm 41

2-4 Y-Direction Linked-List 41

2-5 Data Structure of Y-Direction Linked-List 42

2-6 X-Direction Linked-List 42

2-7 Data Structure of X-Direction Linked-List 43

2-8 Contact Detection 43

2-9 Performance Analysis of Hashtable-based Algorithms for 3D Particles for a Time Step 47

2-10 Speedup of the Spatial Hashing over Spatial Heapsort Algorithm . . . 48

3-1 Superquadrics with varying exponents 53

3-2 Inside-Outside Property of Implicit Function 55

3-3 DFR Contact Test 56

3-4 Discrete Function Representation [From [37]] (A) Traversal of Vertices (B) Physical To Address Space Mapping (C) Projection of B to A (C) Projection of Reduced A to B 58

3-5 DFR 3D Algorithmic Description [From [37]] 59
3-6 DFR Contact Detection Process 60
3-7 Performance Analysis of DFR Algorithms 61
3-8 Runge-Kutta-Nystrom Numerical Integration Steps 63
3-9 Contact Resolution Details 65
3-10 Contact Table Description 67

4-1 Software Framework of Modeling Environment 72
4-2 Open Inventor Architecture 73
4-3 Example of a Scene Graph 76
4-4 A Sample Open Inventor Scene 78
4-5 Example of a VRML Scene 81

5-1 Parallel Computer Architecture 91
5-2 Communication Networks 93
5-3 Message Passing Programming Model 94
5-4 Multithreaded Programming Model 97
5-5 Parallel Hashtable-based Parallel Algorithm 102
5-6 Parallel Efficiency of Parallel Algorithm 103
5-7 System Architecture of Distributed Modeling Environment 104

6-1 Dynamic Impact Simulation Snapshot 1 107
6-2 Dynamic Impact Simulation Snapshot 2 108
6-3 Dynamic Impact Simulation Snapshot 3 109
6-4 Contact Damping Simulation Example 1 111
6-5 Contact Damping Simulation Example 2 112
6-6 Contact Damping Simulation Example 3 113
6-7 Sandglass Simulation 114
6-8 Embankment Simulation 115
6-9 Wave Propagation Simulation 117
6-10 Particles Rolling on the Slope 118

6-11 Fracture of a Fixed-end Beam 119

6-12 DFR Packing Simulation 121

List of Tables

2.1 Index Tables Before Heapsort 32

2.2 Index Tables Following Heapsort 33

4.1 Matrix Manipulation Functions [From [37]] 82

Chapter 1

Introduction

1.1 Motivation

Recently, the discrete element method (DEM) has emerged as an attractive approach for scientists and engineers to study materials and systems at the granular and particulate level where the traditional numerical methods have been unsuccessful. For example, the fundamental aspects of the behavior of granular materials can not be accurately simulated using these traditional methods because of the assumption of material continuity inherent in their derivation.

To set up a physical experiment to study materials at the granular level is a difficult task. Using the discrete element techniques, reseachers can investigate the behavior of materials at the microscopic level. In order to analyze systems at the particle level, the simulation has to be able to deal with thousands or even millions of objects. Scientists have attempted to model how materials crack at the atomic level by using millions of particles and parallel computation [1]. There are several important research issues that must be addressed in developing a simulation capable of analyzing large number of particles, especially if the particles have complex geometric shape and internal state. This thesis identifies and address the computational issues of building a DEM simulation that can operate on today's workstation.

1.1.1 Problem Description

The number of particles which can be analyzed is limited by the available computing resources. As a result, most discrete element simulations have focused on small scale problem with hundreds or thousands of particles, often idealized in two dimensions. The goal of this thesis is to develop efficient approaches to reducing the computational complexity of the discrete element algorithms. Our goal is to be able to handle approximately one million bodies in a full three dimensional simulation.

It is generally acknowledged that the collision detection is the major computational bottleneck in DEM simulations. In order to facilitate the DEM simulation, we seek highly efficient collision detection algorithms to reduce the required computing time to an acceptable level. We will show a simple example to illustrate the need for efficient methods of contact detection.

1.1.2 Example

Suppose that we have a system composed of M objects and each object is represented with N surface points in 3D. If there is no efficient algorithm involved, then the computational complexity of correctly detecting all the geometric overlaps between particles is given by Equation [1.1].

$$Computational\ Complexity\ of\ all-to-all\ check\ =\ O(M^2N^2) \tag{1.1}$$

For instance, if we simulate a system with 10,000 particles, and each particle is described by 1,000 surface points in 3D and we have a fast computer which can perform an operation in 10^{-6} seconds, then, it will take 3 years to perform the contact detection of a single timestep as shown in Equation [1.2]. It is obvious that this kind of approach is not acceptable and we must invent more efficient algorithms.

$$M^2N^2/(time\ of\ operation)\ =\ 10^8\ seconds\ \approx\ 3\ years. \tag{1.2}$$

The spatial heapsort algorithm was developed to enhance the performance from $O(M^2)$ to $O(MlogM)$ [37]. Also, the computational cost of checking the intersection of a pair of particles is varied depending on the data structure chosen to represent the object geometry. The typical geometric representation schemes, such as polygon or surface patches, commonly used in computer graphics research, are not optimal for DEM because of their inefficiency in contact detection. Instead, we use an alternative called the discrete function representation (DFR) [37] which gives $O(N)$ performance, where N is the number of surface points. By combining the spatial heapsort algorithm and DFR, we can reduce the computing time to about 2 minutes, as shown in Equation [1.3].

$$(MlogM)(N)/(time\ of\ operation) \ = \ 132\ seconds \tag{1.3}$$

In this thesis, a hashtable-based spatial reasoning algorithm which demonstrates $O(M)$ performance is developed and implemented, which further reduce the computing time for contact detection. Using the same example described above, the computing time is reduced to only 10 seconds for a single step, as shown in Equation [1.4].

$$MN/(time\ of\ operation) \ = \ 10\ seconds \tag{1.4}$$

From the above discussion, it is apparent that for large-scale problems comprising thousands of three dimensional particles, efficient contact detection algorithms are required.

1.2 Thesis Objectives

This thesis develops a distributed computing environment for multibody physics simulation based on the discrete element method. It includes a set of algorithms that significantly reduce the computational time required in DEM simulation as well as some auxiliary functionalities, such as visualization, to provide a complete simulation environment.

The following issues are addressed in this thesis, which are central to the research of discrete element methods, especially from a computational perspective.

- The implementation of an object representation scheme to model 3D objects with arbitrary geometric shape. The discrete function representation (DFR) is a high-performance scheme particularly desirable for contact resolution.

- The development of an efficient hashtable-based three-dimensional contact detection algorithm, which demonstrates $O(M)$ performance, where M is the number of particles in the system. This algorithm is restricted to bodies of similar size. However, this restriction can be removed if large objects are divided into sub-regions.

- The enhancement of the original sequential algorithm to perform contact detection in a distributed and parallel fashion. In order to surpass the computational barrier associated with large-scale DEM problem, implementing a the parallel processing strategy has proved to be an effective approach.

- The software framework for the simulation environment. Object-oriented technology has been applied in designing the software. It provides a highly organized structure, particularly in terms of implementation and maintanence of the software.

The principal objective of this thesis is to develop a high-performance simulation environment and computational framework based on discrete element methods so that the behavior of granular materials at the microscopic level can be investigated with the minimum computing resources.

1.3 Discrete Element Method

Until recently, continuum models of materials have dominated the analysis of their behavior. Nevertheless, a number of numerical methods which start at the microscopic level have gained attention recently. These technologies include discrete element, cellular automata, lattice gas, molecular dynamics, and percolation models. They offer a complimentary view of the physics of material behavior to the traditional techniques, such as the finite element method.

Conventionally, engineers attempt to formalize their model by deriving governing differential equations to describe or idealize the behavior of the material. The assumption that the material is a continuum involves an averaging of parameters, such as density, over space and is based on the concept of a representative elementary volume (REV). This leads to a governing differential equation which includes a constitutive relationship defining the response of the material to external physical loads. To fully specify the model we also need to define the initial and boundary conditions so that we can derive a solution in terms of space and time.

However, this kind of "top-down" approach of treating the material as a continuum described by a set of governing differential equations is not necessarily sufficient to explain the behavior of granular materials. Alternatively, the "bottom-up" approach that views the material as composed of distinct bodies, provides a solution for this kind of problems. As long as we can ensure that the microscopic behavior of the material is correctly described, it is rational to conclude that the emergent macroscopic behavior exhibited in the DEM simulation is also correct.

A DEM analysis can be decomposed into four computational modules described as follows:

- **Geometry**

  Specify the object geometry, boundary conditions, and physical environment. It provides support for the generation and representation of a general class of

3D objects.

- **Contact Detection**

  Automatically identify the object pairs which overlap with each other. It provides a mechanism to automate *contact detection* between the objects.

- **Physics**

  Calculate forces and integrate the motion to update the position of objects.

- **Visualization and Data Analysis**

  Show the animation results of the analysis. Provides support for user interaction with the simulation and incorporate channels for visualization and temporal characterization of multibody simulations. Output the state of each object with time.

These items encapsulate the components of an DEM analysis pipeline which is represented in Figure [1-1]. According to Williams and Pentland [53], the major characteristics of discrete element methods can be summarized as follows:

- Simulates large displacements and rotations of disjoint bodies.
- Automatically identifies the occurence of contact between pairs of objects during the simulation process. This process is known as *contact detection.*

In Figure [1-2], we list the potential applications of DEM [23]. Generally, the DEM provides a complimentary numerical analysis approach to traditional methods, such as finite difference and finite element analysis.

1.4 Computational Requirements in DEM

The DEM simulation is a highly computational intensive procedure. Without efficient numerical algorithms, the entire process can be too expensive to tackle on today's

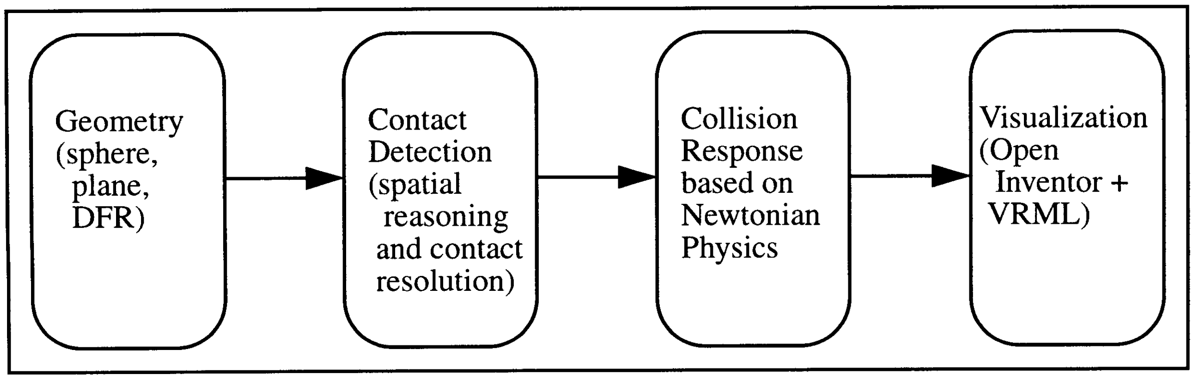

Geometry (sphere, plane, DFR)
Contact Detection (spatial reasoning and contact resolution)
Collision Response based on Newtonian Physics
Visualization (Open Inventor + VRML)

Figure 1-1: DEM analysis pipeline

computers. To make the DEM a practical tool for engineers, from the computational persepctive, the following important requirements need to be satisfied.

1.4.1 Object Representation Requirements

In the DEM simulation environment, since all physical entities are represented by discrete objects, it is essential to provide a compact and versatile object representation scheme that allows the user to define arbitrary geometric shapes of bodies. In computer graphics and computational geometry research, there are many geometry representation methods, such as, polygonal models, constructive solid geometry, implicit surface, and parametric surfaces, that have been proposed for different applications [28, 29, 34, 31]. However, the polygon-based representation is not suitable for DEM simulation because of its inefficiency in contact detection. In DEM, the object representation scheme should not only be able to describe the geometry accurately, but also needs to be helpful in collision detection.

- Offshore platforms and vessels in sea ice
  - Iceberg-bottom founded structure interaction
  - Icebreaker and tanker interaction with sea ice
  - Seabed scouring by ice features and impact on pipeline stability
  - Arching and flow studies of ice in seaways, platforms and bridge legs
  - Explosive fracture of sea ice
- Behavior of soils, rocks, and granular materials
  - Macroscopic constitutive behavior from microscopic granular structure
  - Rock pit slope stability behavior
  - Underground structure stability in jointed rocks
  - Study of earthquake mechanisms and plate tectonics
  - Liquification under dynamic loading
- Impact and explosive dynamics
  - Automobile crash simulation
  - Blast survivability studies
- Mechanical behavior
  - Metal forming
  - Interaction of machinery components
  - Analysis of linkages and chains
  - Vibration control and feedback studies
  - Fracture mechanics

Figure 1-2: Areas of application for the discrete element method

1.4.2 Contact Detection Requirements

Profiling the computational time in each phase of DEM simulation indicates that collision detection is the major computational bottleneck. As we saw in the previous example in Section 1.1.2, it is important to efficiently perform contact detection.

In addtion to the DEM, other application programs, such as CAD and analysis, often require automated reasoning about the spatial geometry of objects. For example, numerical analysis, computer animations of physically based simulation, CAD-CAM systems, path planning and control applications in robotics all require the determination and examination of multi-body interactions, [52, 55, 45, 41, 46, 18, 35, 39].

1.4.3 Contact Resolution Requirements

In order to simulate collision between bodies it is necessary to apply forces between the objects at the in contact surface. The collision response can be achieved in various ways and several algorithms have been proposed for different applications [35, 54, 4]. In this case we choose to use a Penalty Function formulation [37]. The contact resolution problem is concerned with computing the equal and opposite impulses that should be applied to the colliding objects, based on Newtonian mechanics. For particles with complex geometric shape, it is required to calculate the parameters, such as the mass, the location of center of mass, moments and products of inertia relative to the center of mass, etc. The volume integration scheme is introduced to provide this information. Here, all physical objects in the simulation are assumed perfectly rigid, although DEM simulations with deformable bodies are possible [40]. In Chapter 3, we will describe these requirements in details.

1.4.4 User Interface and Visualization Requirements

It is essential for the simulation program to provide a user-friendly interface for the engineers to enter the data, interact with the program, interprete the output, and visualize the results in an integrated fashion. In this thesis, a simulation description language is developed to ease the burden of input. Also, a versatile 3D visualization post-processor allows the engineers to not only graphically view the results in any direction, but also to provide a variety of color-coding schemes to highlight different physical properties or states of the object. For example, the velocity of the body can be linked to the object color.

1.5 Characteristics of the DEM3D Simulation System

In this section, the basic functionality and general DEM simulation procedure of running an application using the **DEM3D** simulation environment is introduced. In Figure [1-3], we list the sequence of completing a single DEM simulation procedure.

1.5.1 Basic Functionality

Performing a DEM simulation requires the specification of the following items.

- **Geometric Representation of Objects**

  For example, sphere, plane, and arbitrary shape of 3D geometry.

- **Constraints**

  Bonding, cohesion, and collision.

- **Material Properties**

  Stiffness, damping factor, friction, and so on.

- **Simulation Parameters**

  Timestep, gravity, resolution, and so on.

- **Visualization**

  Parameters for visualization, such as scale of x,y,z-direciton.

1.5.2 Object-Oriented Technology

The object-oriented technology has emerged as the mainstream paradigm in the programming community nowadays. Instead of dealing with data and function separately, the object-oriented modeling and design emphasizes object, which is a high-level abstraction of data together with operations on the data. In general, the object-oriented technology is characterized by the following three attributes:

- Encapsulation (abstract characterization of objects)
- Inheritance (code sharing)
- Polymorphism (run-time binding of operations to objects)

With the features mentioned above, the overall software design is more structured and maintenance becomes less complicated. The object model of the **DEM3D** system developed in this thesis is illustrated in Figure [1-4].

1.6 Distributed and Parallel Implementation

With the advent of powerful multiprocessor workstation and high-speed communication network, it has become advantageous to migrate from the traditional sequential programming model to the parallel counterpart. The hardware advancements provided by the powerful multiprocessor server and well-developed parallel programming software standards, such as, the message passing interface (MPI), offers a well-structured framework for integrating distributed computing resources into a simulation. These standards allow trasmitting and managing the data between a set of processors via a communication network. The characteristics of parallel and distributed computing and details of the implementation are described in Chapter 5. In Figure [5-7], we show the current configuration of the local area network, which has a high-speed 100MBits/sec Fast Ethernet switch, an SGI server, workstation, and PC server are capable of exchanging data rapidly.

```
1. Specify system parameters
2. Initialize graphics subsystem
3. Create particle objects
4. Define material properties
5. Assign system parameters
6. for each timestep
   (a) Run spatial reasoning algorithm to determine all candiate pairings
   (b) for each candidate pairing
       i. Perform the detailed check
       ii. Apply the physics based on Newtonian mechanics
   (c) Increment the timestep by 1
```

Figure 1-3: Generic DEM Simulation Procedure

1.7 Review of Other DEM Systems

In the past few years, a wide variety of DEM systems have been implemented for various applications using different representations for the individual elements. In applications areas, such as soil mechanics, rock and ice mechanics, process engineering and granular flow, mining and blasting, and physically-based modeling and animation, some DEM systems have been designed to fit their application-specific needs. An excellent review of a variety of DEM simulation systems can be found in [37].

1.8 Thesis Outline

The major software components of a discrete element system are the object representation, contact detection, and physics and visualization. Chapter 1 gives an overview to these four key modules and discuss the corresponding computational requirements

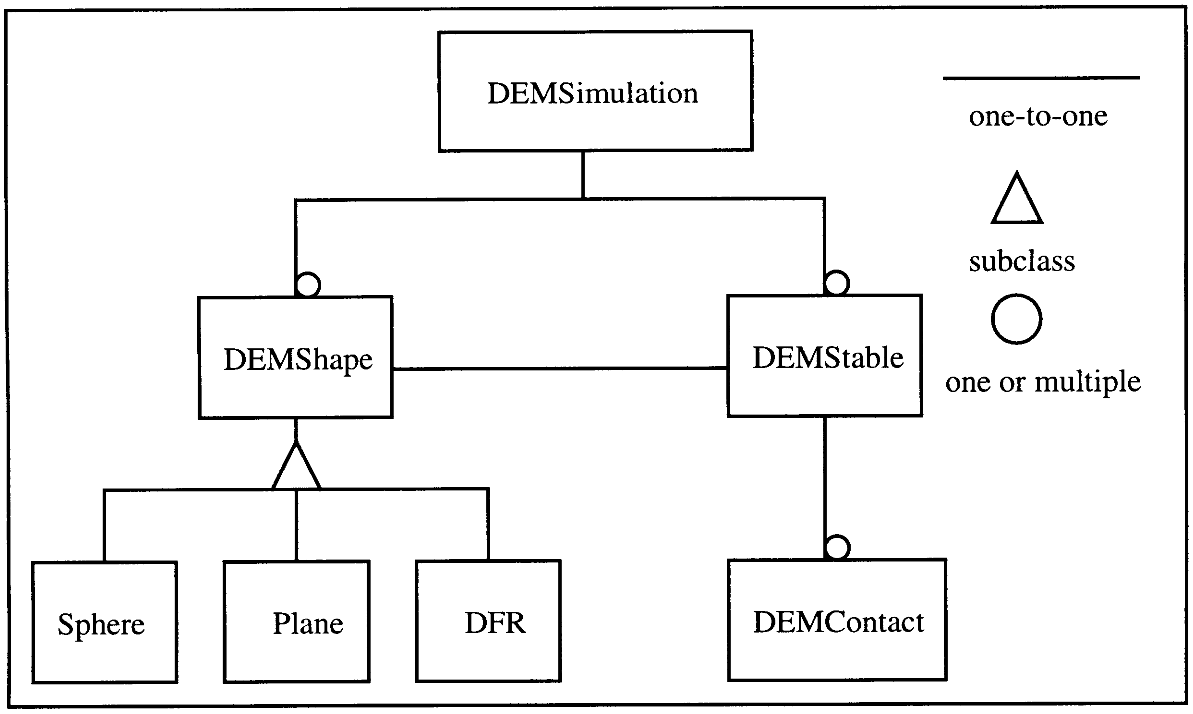


Figure 1-4: The Object Model of **DEM3D** system

of each component.

In Chapter 2, we deal with the spatial reasoning algorithms for contact detection. One is the spatial heapsort method, which has $O(M \log M)$ computing complexity. The other is the hashtable-based spatial reasoning algorithm, which exhibits $O(M)$ performance.

Chapter 3 describes object representation scheme with emphasis on the discrete function representation (DFR). The superquadratic representation is introduced first to describe the rationale behind the DFR method. The incremental collision resolution scheme and contact table data structure are also introduced to deal with the collision response problem.

Chapter 4 covers the implementation issues of the simulation environment, including visualization graphics, such as Open Inventor graphics library and Virtual Reality Modeling Language (VRML), a matrix transform library for geometric transformation operations, and a simulation description language based on the *Tcl/Tk* toolkit.

Chapter 5 presents the extension of the original system to a parallel version. A brief introduction to the parallel computing paradigm and programming models is given first. The Message Passing Interface (MPI) system, an emerging standard language for parallel computation, is used as the vehicle to perform the data communication in the distributed simulation system. The overall parallel efficiency is shown to be satisfactory.

Several sample applications, including dynamic impact simulation, contact damping, sandglass, collapse of embankment, stress wave propagation, particles rolling on a slope, fracture of a fixed-end beam, and DFR packing are presented in Chapter 6 to demonstrate the performance and capabilities of the simulation system.

Finally in Chapter 7 conclusions about the research are summarized and prospective future research directions are discussed.

Chapter 2

Algorithms for Spatial Reasoning

2.1 Introduction

As discussed in Chapter 1, the most computationally demanding process of the discrete element method-based simulation is the contact detection. It is estimated that contact detection occupies 85-90% of the overall computing time. As a result, we focus on this problem and propose algorithms which dramatically reduce the computational cost. In order to efficiently sort out the spatial relationship among the particles, we develop and implement two algorithms in this chapter aimed at significantly reducing the computational time.

Basically, the entire contact detection process can be subdivided into a spatial ordering phase and a contact resolution phase. During the spatial ordering phase, the contact detection algorithm has to figure out what particle pairings are possibly in contact with each other without having to exactly estimate where the contact point is. In other words, in this phase, as long as two particles are close based on some given criteria, we consider it is a candidate pair that requires further checking. Entering the contact resolution phase, the algorithm computes the details on the contact geometry the and resolves the physical interaction between the contact pairings based on Newtonian mechanics.

During the spatial ordering phase of contact detection process, spatial reasoning algorithm determines which pairs of particles should be considered for the further processing in the contact resolution phase. The goal at this stage is to avoid an exhaustive brute-force check of all pairings and thereby reduce the computational cost. Once the spatial ordering of the objects is complete the algorithm identifies all the possible candidate pairs of objects which may penetrate into each other based on a criterion, such as bounding box or bounding sphere overlap. The issues of the spatial ordering phase contact detection are investigated further in this chapter. The detailed checking phase is discussed in Chapter 3.

One is the spatial heapsort scheme, named after the computing method and data structure used in this algorithm. By sorting the ordinates of the particles in each dimension, the algorithm decides which pair of particles is close enough to each other and requires further detailed check. As compared with the naive all-to-all check algorithm described in Chapter 1, this method reduces computing complexity from $O(M^2)$ to $O(MlogM)$, where M is the total number of particles in the simulation system.

The other algorithm is the hashtable-based spatial reasoning scheme, have called spatial hasing, named after the data structure used in this algorithm. In the spatial hashing algorithm, the space is subdivided into a grid of cells based on the radius of the largest particle in the system and each body is assigned to a cell based on the hashing of its centroid coordinate. Independent of the particle density, this algorithm demonstrates $O(M)$ computing complexity, where M is the number of particles in the system.

The general requirements for designing high-performance contact detection algorithms are as follows:

- **Robustness**

  Robustness, in this context, means the stability of the algorithmic performance over a wide range of problems. A robust algorithm will exhibit good performance over all cases. No matter what the composition and distribution of the particles in the system, a robust algorithm should exhibit a good performance.

- **Correctness**

  A correct spatial reasoning algorithm should be able to identify all geometric overlaps within a given timestep. For simulation, the collision detection algorithm is only invoked at discrete sample times. No matter what the minimum sampling period of the collision detection system, one can choose a particle speed such that the particles entirely pass through with each other between collision checks. In our DEM simulation environment, the time step is chosen so that this cannot occur.

- **Performance**

  In theory, the performance of a spatial reasoning algorithm is a function of the number of partilces in the system. The naive all-to-all check procedure has computational complexity $O(M^2)$, where M is the number of particles. The hashtable-based spatial reasoning algorithm shows a linear relationship between computing time and the number of particles. This is a significant improvement, particularly as M grows large.

- **Parallelizability**

  A high-performance contact detection algorithm should be able to be parallelized without substantial data transfer overhead between processors or excessive extra memory requirements. In our studies, we found that there are some algorithms, for example, the hashtable-based spatial reasoning algorithm, that are highly parallelizable. This kind of algorithm always demonstrate high performance if the proper hardware configuration and software tools support for parallel computation are available.

The detailed theoretical analysis and empirical results of these two algorithms are given at the end of this chapter.

2.2 Spatial Sorting Algorithm

2.2.1 Introduction

There are several spatial sorting techniques which have been practically employed in different areas of application, such as the discrete element method (DEM) [52, 49, 47, 8], geometric modeling [32], computer graphics [42, 44, 25], molecular dynamics [43, 50, 21, 20, 15, 7], as well as geographical information systems (GIS) [44]. An excellent review of these sorting strategies is presented in [37].

Without examining the spatial coherence of the objects involved in a given environment, an all-to-all sequence of checks for potential collision needs to be performed. For a handful of objects, this could be acceptable [22]. However, if we expect to perform the simulation for thousands of objects, the order of the computational complexity dominates all other considerations and methods must be sought to reduce the work involved. Several commonly applied methods to reducing the computational complexity are reviewed below:

Cellular Subdivision

The cellular subdivision method subdivides the 3D space into equal-sized cells. Each cell is given a new coordinate number based on the size of the cell. Each particle in the system is assigned to the cell which the center of its bounding box or bounding sphere belongs to.

This method can be algorithmically described as follows:

- Construct a list in all cells.
- Add all objects contained (fully or particlally) in each cell to the list.
- Check for collision of all objects within the same cell.

However, the major limitations of this approach are as follows:

- **Spatial distribution of the objects** This static approach is suitable for problems with proportionately distributed particles.
- **Range of particle size** If the size of the particles varies greatly, then we can only subdivide the space according to the size of the smallest particle.
- **Cell resolution** This is related to the particle size. The memory requirment can be extremely high if the particle size is fairly small.

In order to surpass these limitations, dynamic data structure and adaptive cell methods can be adopted to handle certain extreme cases.

Adaptive Cell Methods

Adpative cell methods aim at avoiding the space resource costs associated with the static uniform cell approach when the particles are disproportionately distributed in the simulation space.

In this approach the simulation space is discretized by cutting planes parallel to the principal Cartesian planes, $x-y$, $y-z$ and $z-x$ respectively so as to keep approximately the same number of particles on either side of the cutting plane. This scheme is suitable for the simulation space where particles are unevenly distributed. For the uniformly distributed case, this method will suffer from excessive data maintenance overhead in comparison with the static cellular subdivision approach.

Octree Method

The Octree method is theoretically perhaps the most elegant technique to tackle the problems of spatial coherence and resolution and the tradeoff between space and time [37]. In theory, the time required to create the octree is $O(MlogM)$ and the time needed to search the tree is also $O(MlogM)$. A comprehensive and complete explanation of the quadtree and octree methods can be found in [44].

The basic idea behind the Octree method is similar to the cellular subdivision method, which is to treat the simulation space as consisting of uniform-sized rectangular cells. However, the hierarchical tree data structure, allows us to manage only those cells which contain objects. By nature, the performance of the Octree method is heavily influenced by the distribution of the particles. Irregular distribution can generate a highly unbalanced tree which can result in the worst case of constructing and searching the tree in $O(MlogM)$.

The octree method may not be so attractive for us because of the dynamic behavior of the objects in DEM environment. At each timestep, the program must reconstruct the tree and search through the tree without taking advantage of the temporal coherence with the previous frame.

2.2.2 Spatial Heapsort

In this section, the spatial heapsort algorithm with running time $O(MlogM)$ is introduced. There are two arrays required for each dimension of the problem domain. One is used to store the object identifiers, the other is used to rank the order of objects. The combination of these two arrays are referred to as a sorting table. The size of the array is equal to the number of particles. A typical sorting table is shown in Table [2.1] and Table [2.2].

2.2.3 Example

In this section, we present a simple example to clarify how this spatial sorting algorithm works. For simplicity purpose, we start with a 2D example, and then we will show how to extend to handle 3D problems.

A collection of 2D objects with similar geometry are shown in Figure [2-1]. Each object maintains a bounding box expressed in the world coordinate system. The lower bound extents are projected onto the X and Y axes as shown in Figure [2-1]. To perform the *spatial heapsort* in 2D, four arrays of integers are required, two for

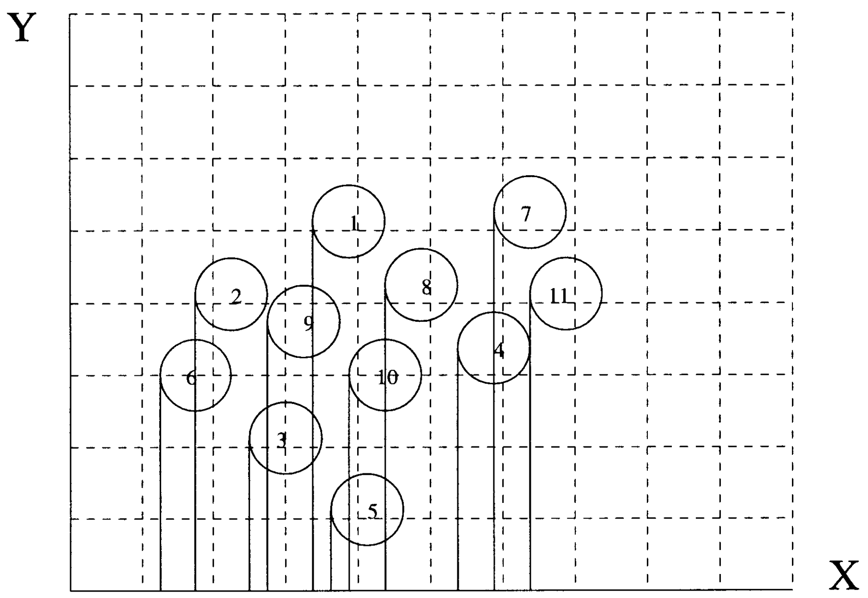


Figure 2-1: Example of Spatial Sorting Algorithm

Canonical index	1	2	3	4	5	6	7	8	9	10	11
X-dir object id	1	2	3	4	5	6	7	8	9	10	11
X-dir auxiliary	1	2	3	4	5	6	7	8	9	10	11
Y-dir object id	1	2	3	4	5	6	7	8	9	10	11
Y-dir auxiliary	1	2	3	4	5	6	7	8	9	10	11

Table 2.1: Index Tables Before Heapsort

each dimension. The first array (sort array) initially stores the object identifiers. The second array (auxiliary array) is used to store the location of each object in the first array. Each of the first arrays is initially treated as an unsorted (implicit) binary tree, which will be sorted using the heapsort scheme. The contents of the arrays corresponding to the objects in Figure [2-1] are listed in Table [2.1]:

The next stage in the algorithm is to group objects that are candidates for collision with each other. A key expedient to quickly and methodically identify candidate pairings for collision detection is to process the sorted lists object by object in ascending order along the primary axis parallel to the simulation volume. For an

Canonical index	1	2	3	4	5	6	7	8	9	10	11
X-dir object id	6	2	3	9	1	5	10	8	4	7	11
X-dir auxiliary	5	2	3	9	6	1	10	8	4	7	11
Y-dir object id	5	3	10	6	4	9	2	11	8	1	7
Y-dir auxiliary	10	7	2	5	1	4	11	9	6	3	8

Table 2.2: Index Tables Following Heapsort

equi-dimensioned simulation volume, the X axis is chosen in the absence of any other dictating factor. Each object occurring along the axis of the chosen sorted list is selected. The object selected for testing is referred to as the pivot object. By processing the sorted list sequentially, candidate pairs need only be identified once as only those objects lying ahead of the pivot object need to be considered. This approach is chosen to avoid of double check between an pair of overlapped objects.

To identify the local group of objects that are candidates with the pivot object, the following steps are performed:

1. Start at the index location of the pivot object and traverse the sorted list using a binary search. Compare the largest ordinate (extent) of the pivot in the direction of the search direction/axis with the smallest ordinate of those objects lying beyond it.

2. Stop at the index location of that object which does not have a lower-bound extent less than the upper-bound extent of the search pivot.

3. Start at the index location of the search pivot in the other coordinate directions (i.e. the Y, Z axes) and traverse the sorted list using a binary search and identify the upper and lower bounds on the indices that capture all collision candidate objects in these directions.

The two steps above, obtain the lower and upper bound indices of those objects that might be in contact with the pivot object, in each of the coordinate directions.

The final stage is the detailed contact detection between pairs of objects drawn from the the intersection of the index sets. Again, the indices from the primary search direction are used to control and identify the sequence of objects examined. For each index in this list, the index of the object is found from the auxiliary index array. This index is then used to determine if the same object exists in the candidate list of objects found along each of the secondary axis directions. If the index exists (i.e. it is an element of the intersection set of indices from all directions) a full contact detection is performed on the pair of objects.

For the example set of objects shown in Figure [2-1], using the X axis as the primary search axis, the object 4 has canonical index 9. The index bounds from the object identifier list in the X direction for objects that object 4 may be in contact with are 10 to 11 corresponding to objects {7,11}. In the secondary direction (Y axis) the index bounds from the object identifier list are 6 to 9, corresponding to objects {9,2,11,18}. The intersection of the sets of indices yields the set {11}, i.e. only a detailed contact examination of objects 4 and 11 needs to be performed.

2.2.4 Implementation

The psuedo code of the heapsort algorithm is given below.

```
heapsort_init(); /* initialize the data structure */
/* invoke the sorting routine in each dimension */
   heapsort (X); /* X-direction */
   heapsort (Y); /* Y-direction */
   heapsort (Z); /* Z-direction */
   map_rank (); /* generate the auxiliary array */
for each body { /* the pivot particle */
   neighbours (&lx, &rx, X); /* get lower and upper bound of X */
   neighbours (&ly, &ry, Y); /* get lower and upper bound of Y */
   neighbours (&lz, &rz, Z); /* get lower and upper bound of Z */
```

```
    /* multiplex neighbourhood indices from smaller index subset */
    if (((rx - lx) >= (ry - ly))) { /* if y-list is smaller */
       for (j = ly; j <= ry; j++) {
           y2x = rank[X][index[Y][j]];
           y2z = rank[Z][index[Y][j]];
   /* map Y index into X & Z indices*/
           if (((y2x >= lx) && (y2x <= rx))&&((y2z >= lz)&& (y2z <= rz))){
                       call detailed check procedure
               }
          }
    } else {   /* if x-list is smaller */
       for (j = lx; j <= rx; j++) {
  /* map X index to Y index */
  x2y = rank[Y][index[X][j]];
  x2z = rank[Z][index[X][j]];
  /* IF mapped index in X & Z index ranges AND not self */
    if (((x2y >= ly) && (x2y <= ry)) && ((x2z >= lz) && (x2z <= rz))) {
            call detailed check procedure
          }
          }
       }
```

2.2.5 Performance Analysis

In theory, there are some characteristics of the heapsort algorithm [37] listed below.

- Theoretically [48], the heapsort supposedly sort an unordered collection of M objects in $O(MlogM)$ computation time.

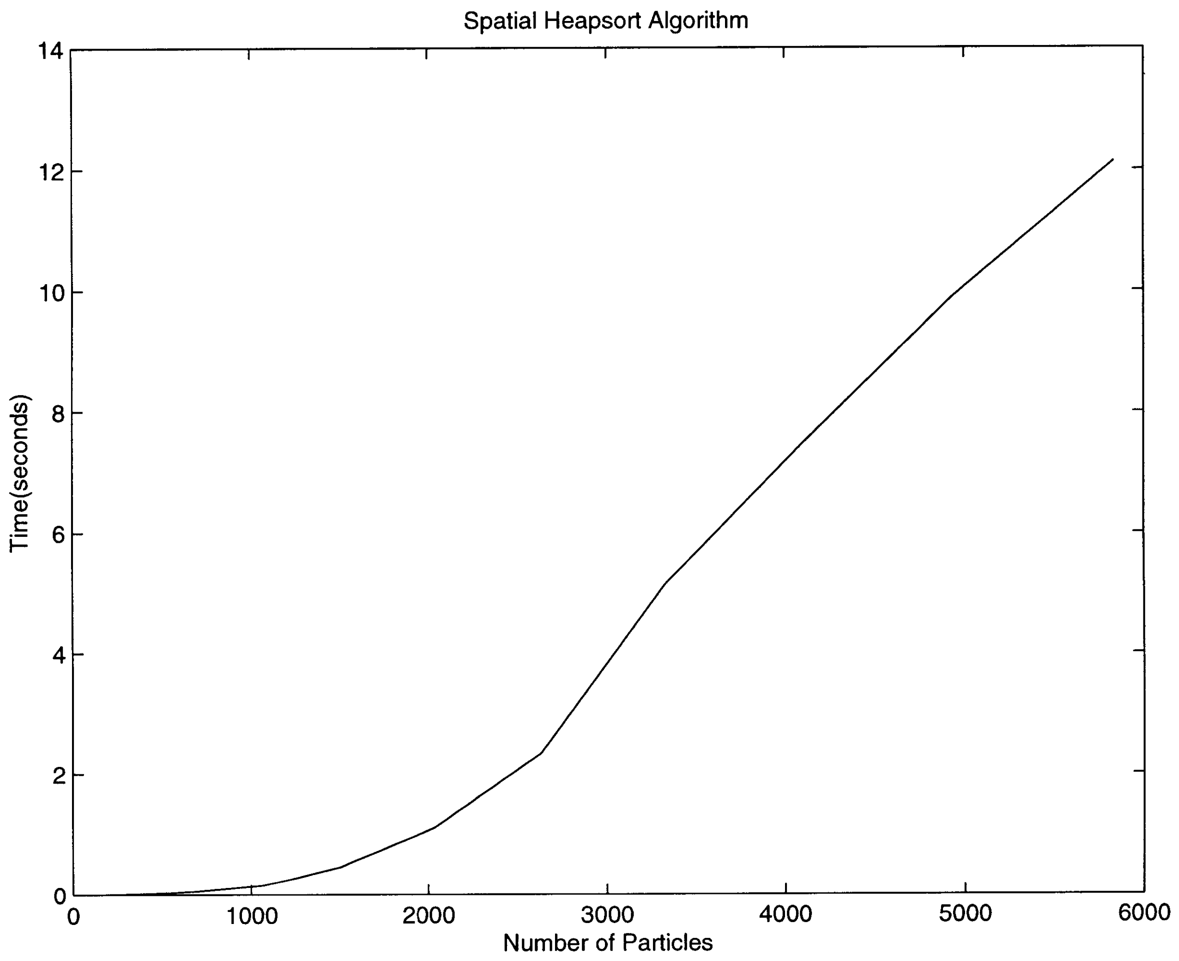


Figure 2-2: Performance of Spatial Heapsort Algorithm for 3D Particles for a Time Step

- The storage requirements are $O(M)$. No extra storage is required because the algorithm sorts in place.
- The behavior of the algorithm is stable in terms of spatial distribution of the objects to be sorted. Extreme cases will not downgrade the performance too much.
- It is straightforward to implement because no complicated data structure is required.

The empirical results of the spatial heapsort algorithm are illustrated in Figure [2-2].

2.3 Hashtable-based Spatial Reasoning Algorithm

In the following sections, a high-performance contact detection algorithm with total detection time linearly proportional to the number of particles is described in detail [36]. Furthermore, we emphasize that its performance is independent of the packing density of particles in the system.

2.3.1 Introduction

In the remainder of this section, the algorithmic description of this spatial reasoning algorithm is presented. The hashtable-based spatial reasoning algorithm is based on the assumption that each discrete element can be represented by a bounding sphere in 3D or by a circular disk in 2D. The diameter of an equivalent sphere $2R$ is obtained from the size of the largest discrete element in the system.

The space boundaries are defined by X_{min},X_{max},Y_{min},Y_{max}, as shown in Figure [2-3]. All particles are confined within the boundaries. The task here is to search all pairs of objects that are close enough to each other that we can say they are in contact in the spatial reasoning phase. After we pick up all the possible candidate pairs, the detailed check for overlap will be performed by another procedure, which is described in Chapter 3.

First, the space is subdivided into identical square cells of size $2R$, where R is the radius of the particle. Each particle is assigned with a unique integer identification number from $0, 1, 2, ..., M-1$, where M is the total number of particles. Similarly, we assign each cell an identification integer number pair $(Coord\_x, Coord\_y)$ based on the space decomposition. The $(Coord\_x, Coord\_y)$ pair, in this context, is the new coordinate for each cell in the system, as illustrated in Figure [2-3]. Equation 2.1 and 2.2 show how this new coordinate number is calculated.

$$(Coord\_x) = Integer(X - X\_min/2R) \tag{2.1}$$

$$(Coord\_y) = Integer(Y - Y\_min/2R) \tag{2.2}$$

From this criterion, we can map each object onto one and only one cell in the new coordinate system. With this new integerized coordinate, we can build a set of linked lists for each coordinate both in X and Y directions. Figure [2-4] shows the Y-direction linked list for the example in Figure [2-3]. Considering the efficient management of memory, we exclude the use of a 2D array to represent the 2D space in our implementation. Obviously, if the range of the 2D space is large, we end up having a sparse 2D matrix, which is not an economical utilization of memory resources. The same conclusion applies to the 3D situation.

First, we traverse through the entire particle set, and give each object the Y_i number. For all the objects with the same Y_i, we create a linked list for this Y_i number, and then insert the objects into the list sequentially, such as in Figure [2-4]. We use two integer arrays to represent this linked list. The first array Y_LIST contains the number of the last particle mapped to each Y_i row. It is a 1D array of size NY, where NY equals to $Y_{max} - Y_{min}$. The second array Y_LIST_N is a 1D array of size M, where M is the total number of particles. For each particle the array Y_LIST_N stores the next particle in the singly connected list. For both array, (-1) is used as a termination of a singly connected list. Therefore, if there is no element in a particular row, -1 is assigned to the corresponding element of the Y_LIST array. Similarly, each element in the Y_LIST_N points to the next particle in the list ending with a (-1).

Example

In this section, we demonstrate a simple example to describe the rationale behind this algorithm. Starting with an example in 2D, the extension to 3D can be derived in a similar fashion.

For the example shown in Figure [2-4], there is not particle located in row 0 of the cells. Therefore, the singly connected linked list for this row, where $Y_i = 0$, would be empty. The emptiness is represented by setting the $Y\_LIST[0] = -1$. For particle 5 mapped onto row 2, we set $Y\_LIST[1] = 5$. Similarly, for row 4, the last particle mapped onto it is particle 10, thus $Y\_LIST[3] = 10$. The next particle on this row is 9, thus $Y\_LIST_N[10] = 9$; the next particle is 6, thus $Y\_LIST_N[9] = 6$; and the last particle is 4, thus $Y\_LIST_N[6] = 4$, and $Y\_LIST_N[4] = -1$, as shown in Figure [2-4]. All Y_i lists are marked new at this point.

Secondly, by looping over all particles, a new Y_i list is traversed and then marked as old. Each particle from this list is placed onto a corresponding $((X_i, Y_i))$ list based on the integerized coordinate X_i. A $((X_i, Y_i))$ list actually contains all particles with integerized coordinate $((X_i, Y_i))$. In addition, all singly connected lists $((X_i, Y_i))$ contain all particles from the list (Y_i) and are represented by two array of integer numbers. The first list is a 1D list (X_LIST) of size NX, where NX is the number of cells in X direction, that is, the total number of columns of cells. The second list is a 1D list (X_LIST_N) of size M, where M is the total number of particles in the system. Figure [2-6] and [2-7] illustrate the structure of the linked-lists and relationship between particles.

Take the list Y_3 in Figure [2-4] for example, containing particles (4,6,9,and 10). Thus the corresponding $X\_LIST[4] = 6$, $X\_LIST[6] = 9$, $X\_LIST[9] = 10$, and $X\_LIST[10] = -1$. In list Y_3, there are no particle having an integerized coordinate $Coord\_x$ equal to (0,2,6, or 7), thus the singly connected lists (X_0, Y_3), (X_2, Y_3), (X_6, Y_3), and (X_7, Y_3) are empty. Therefore, the corresponding $X\_LIST[0]$, $X\_LIST[2]$, $X\_LIST[6]$, and $X\_LIST[7]$ are all assigned with (-1), as shown in Figure [2-7].

Contact Detection

After constructing the data structure described above, we can proceed to perform the contact detection by checking all the particles in the neighbouring cells. Take the cell (X_i, Y_i) for example, we should check all the particles in cells (X_i, Y_i), (X_{i-1}, Y_i), (X_{i-1}, Y_{i-1}), (X_i, Y_{i-1}), and (X_{i+1}, Y_{i-1}). Starting from the $(Y_{i=0})$ list, each time we select two lists (Y_i) and (Y_{i+1}) to do the check. As we sweep through from the beginning to the end, we should be able to pick up all the possible contacts between particles. In other words, we do not necessarily to check all the surrounding eight cells, which will result in some duplicate pairs because of the double check. For instance, we should only check the neighbouring cells, (X_4, Y_3), (X_3, Y_3), (X_3, Y_2), (X_4, Y_2), and (X_5, Y_2), for the pivotal cell (X_4, Y_3), as shown in Figure 2-8]. Moreover, checking with the neighbouring cells is only performed for the non-empty cells.

Apparently in this scenario, there's no loop running over all the cells, which implies the performance of the algorithm is independent of the number of cells. Furthermore, it indicates it is also independent of the packing density of the particles.

A variety of different 3D simulation problems have been tested against the theoretically predicted performance in Section [2.3.5].

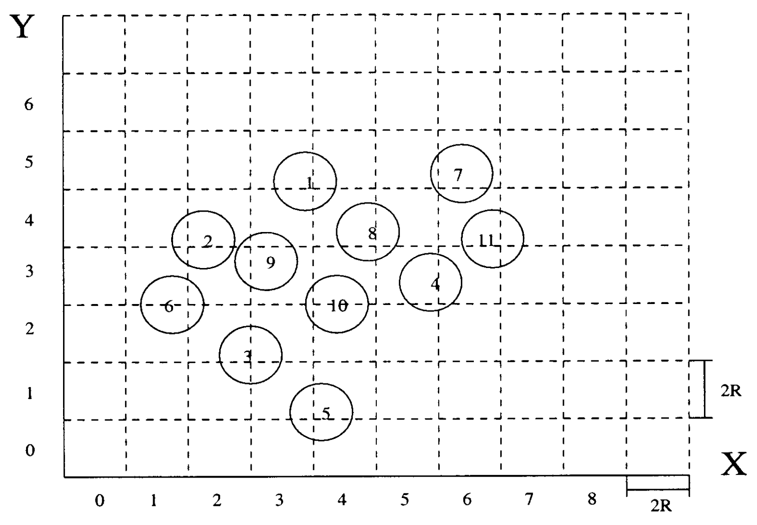


Figure 2-3: Example of Hashtable-based Spatial Reasoning Algorithm

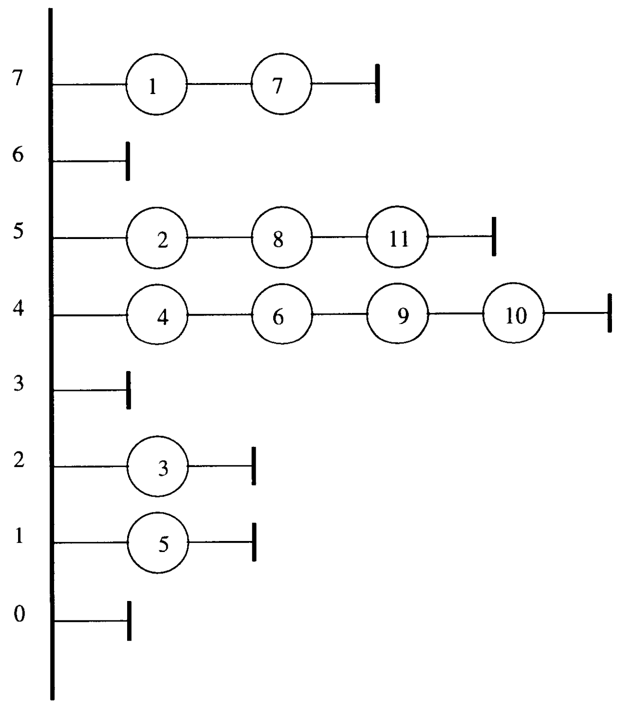


Figure 2-4: Y-Direction Linked-List

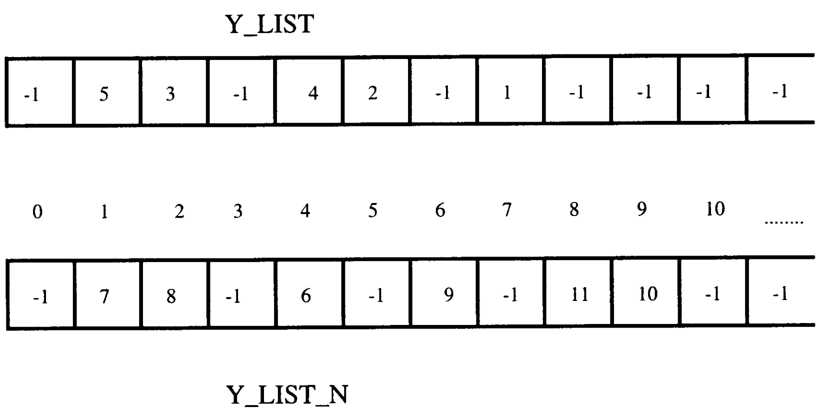


Figure 2-5: Data Structure of Y-Direction Linked-List

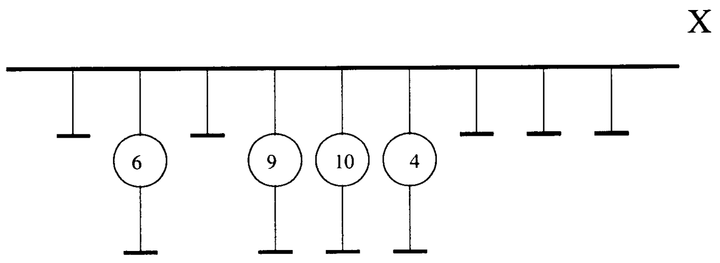


Figure 2-6: X-Direction Linked-List

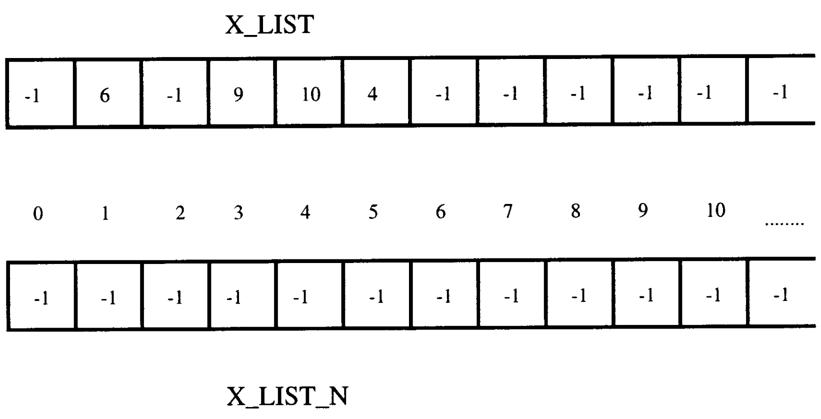


Figure 2-7: Data Structure of X-Direction Linked-List

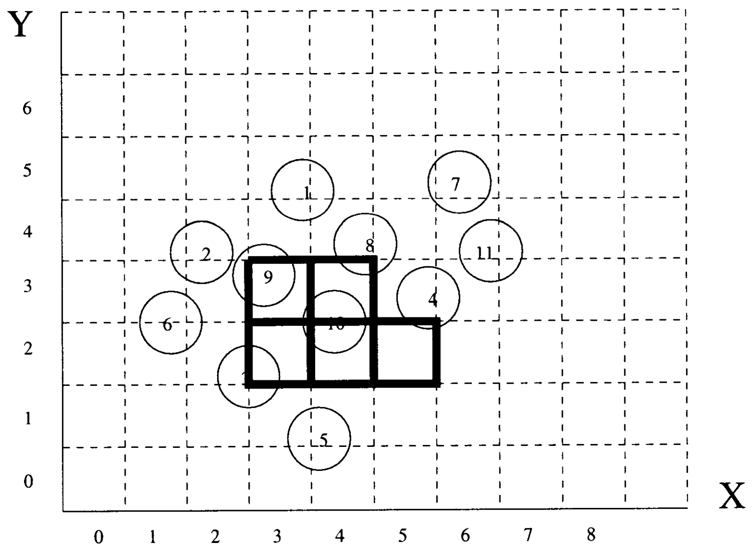


Figure 2-8: Contact Detection

2.3.2 Hashtable Data Structure

In order to reduce the amount of memory requirements, we need to develop a compact data structure to minimize the memory usage. Without an efficient memory management subsystem, the overall performance of the algorithm could deteriorate to an unacceptable level in practice.

In this algorithm, we use the linked-list (e.g., $Y\_LIST, Y\_LIST_N, X\_LIST, X\_LIST_N$) to store and manage the required information. The layout of the linked-list is illustrated in Figure [2-5] and Figure [2-7].

2.3.3 Implementation

Based on the data structure described above, we now go through each step of the algorithm to explain the implementation details.

The pseudo code for this algorithm is shown below.

```
/* Sort objects into bins in each dimension. */
function hashspace_sort ()
{
  /* 1. build the Coord_Y lists */
  for each particle {
    Coord_y = hash(centroid,radius);
    Y_LIST->insert(obj,coord_y);
  }
  for each particle {
    if(Y_LIST[i] != -1) {
     /* 2. set up X_LIST lists for Coord_Y and Coord_Y-1 */
      while(Y_LIST->head[i]){
     /* 3. loop over Y_LIST  */
     Coord_x = hash(centroid,radius);
     /* Current disk at IX in NEW list */
     X_LIST(i)->insert(Y_LIST->head[i],Coord_x);
        /* all objects in Coord_Y are in X_LIST(i) lists */
      }
      while(Y_LIST->head[i-1]) {
         /* 4. loop over Coord_Y-1 list */
        Coord_x = hash(centroid,radius);
        X_LIST(i-1)->insert(Y_LIST->head[i-1],Coord_x);
         /*   all objects in Coord_Y-1 in X_LIST(i-1) list  */
      }  /* we now have all Coord_ix lists set up   */
      /* loop over all objects in Coord_Y */
      iyob = Y_LIST->head[i];
      while(iyob != -1) {
       ix = hash(centroid,radius);
        check for contact between (coord_y,coord_x),
        (coord_y,coord_x-1),(coord_y-1,coord_x),
        (coord_y-1,coord_x-1),(coord_y-1,coord_x+1) cells.
if(iyob->neighbor_list->number_of_members > 0) {
  call detailed check function for each pair of objects
}
iyob= Y_LIST->next[i]; /* get next element in Y_LIST  */
      }  /* end of while(iyob != -1) loop */
    }
  }
}


/* Sort objects into bins */
```

```
function  hash (float position,float radius) {
    return (int)(position/(2.0*radius));
}
```

2.3.4 Extensions

The major limitation of the original algorithm is that the particle size has to be similar so as to capture any contact correctly. Nevertheless, we can extend this algorithm to support simulation systems which contains different sizes of particles by doing some preprocessing work. Before the simulation program calls the procedure, we can loop over all the particles to obtain the largest radius among them. Then, we use this largest radius to decompose the space into equal-sized cells. Since we use the largest radius to subdivide the space, we will not lose any possible collision between particles. However, if the range of the size of particles is unevenly wide, then the performance of the algorithm could be possibly downgraded to the worst case, which is $O(M^2)$, where M is the number of particles in the system.

2.3.5 Performance Analysis

This algorithm works equally well for both dense and loose density distribution of particles in the simulation system, particularly with CPU time being independent of packing density and little increasing memory usage with decreasing packing density. The performance of this algorithm is illustrated in Figure [2-9]. Empirical results show that a 10 times decrease in packing density results in only 2 times increase in memory usage in a 2D problem [36].

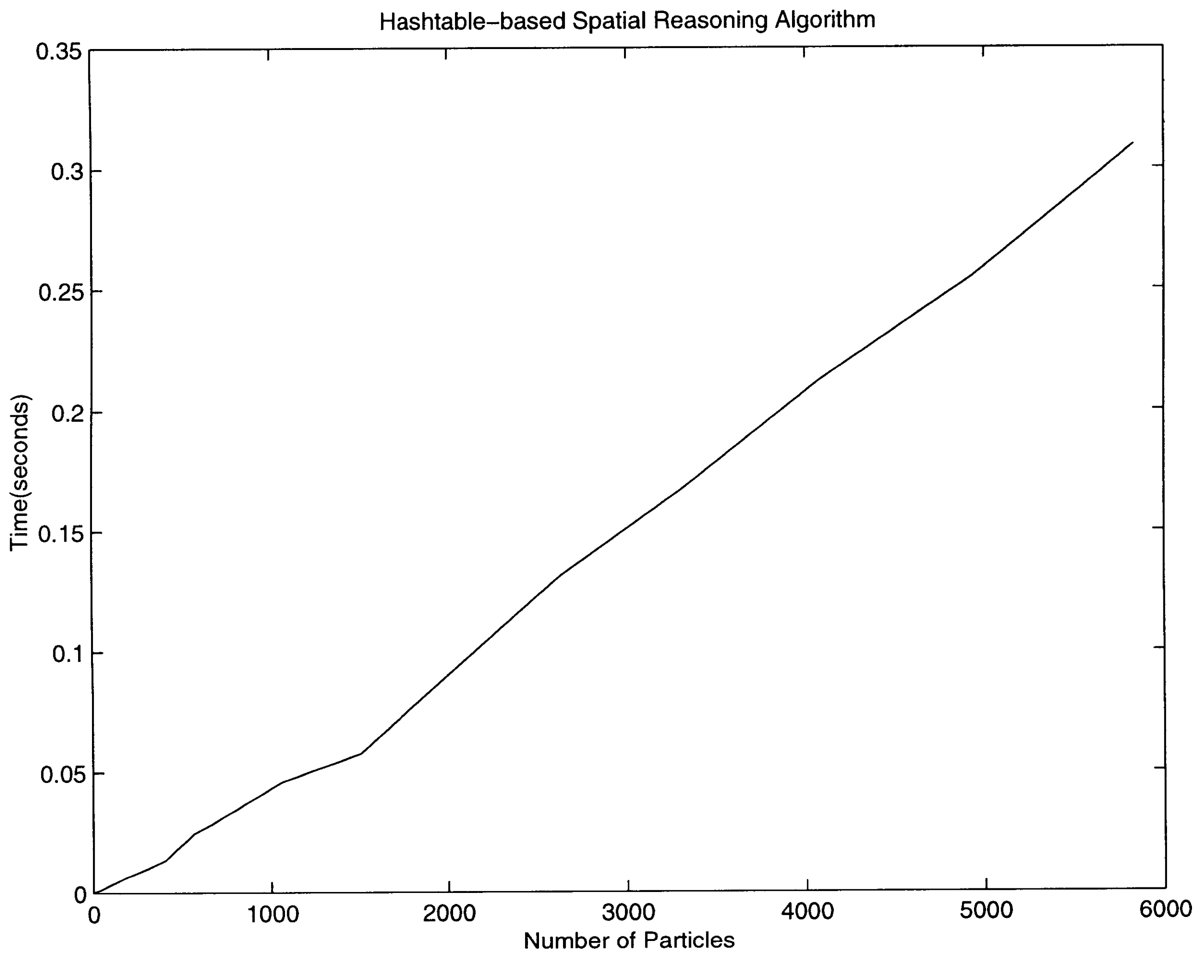


Figure 2-9: Performance Analysis of Hashtable-based Algorithms for 3D Particles for a Time Step

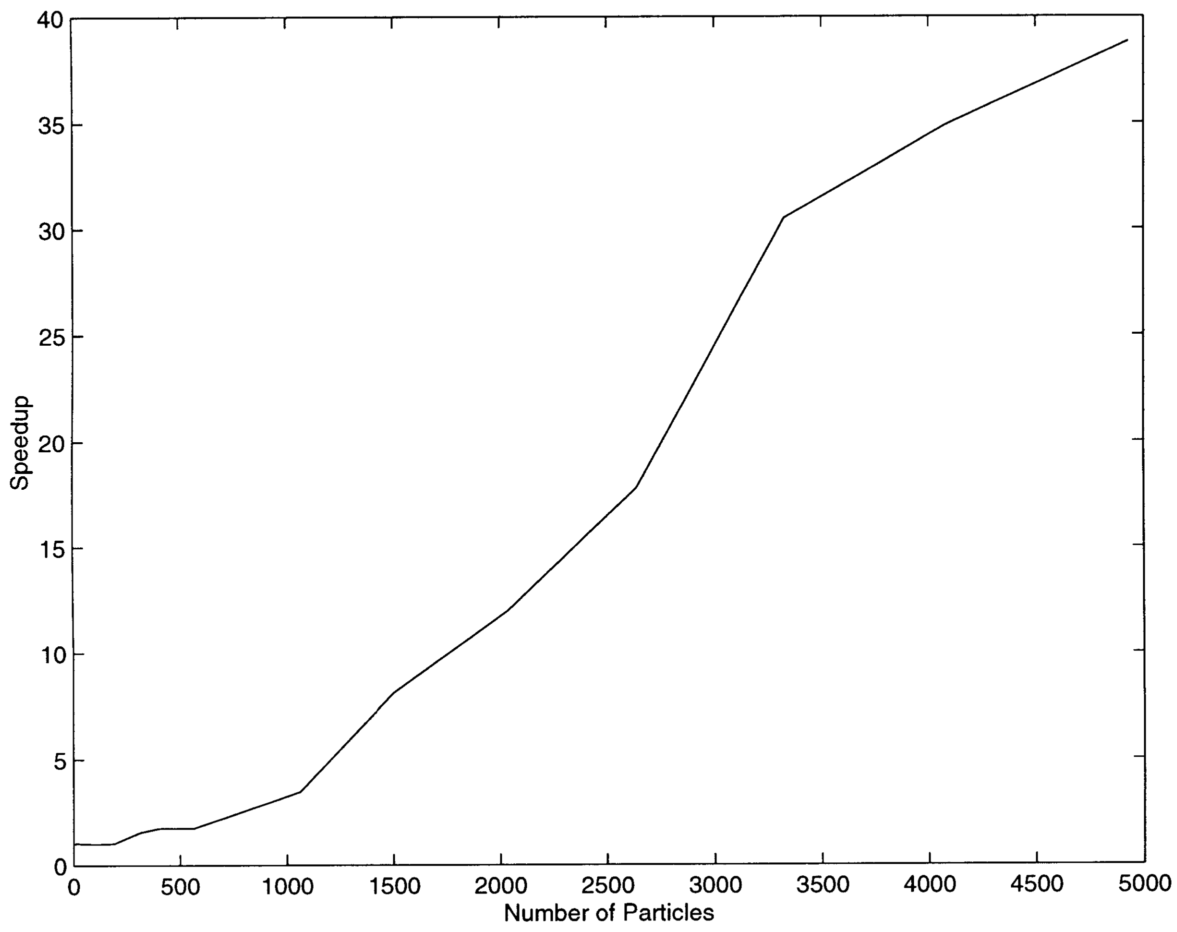


Figure 2-10: Speedup of the Spatial Hashing over Spatial Heapsort Algorithm

2.4 Comparison

The empirical results shown in Figure [2-2] and [2-9] exhibit the performance difference between these two algorithms. In Figure [2-10], we compare the two methods by testing the same problem with different number of particles. The hashtable-based spatial reasoning algorithm performs much better than the spatial heapsort algorithm. As the number of particles increases, the corresponding speedup grows, too, giving a speedup of 40 times for 5,000 particles.

2.5 Summary

In this chapter, we demonstrate two spatial reasoning algorithms for contact detection. In summary, the derivation of these algorithms are based on exploring the spatial relationship between particles.

Temporal coherence is the property that the application state does not change significantly between time steps, or frames. The objects move only slightly from frame to frame. The slight movement of the objects translates to geometric coherence, because their geometry, defined by the vertex coordinates, changes minimally between frames. The underlying assumption is that the time step is small enough that the objects do not travel large distances between frames.

Note that some other sorting schemes, such as insertion sort or bubble sort, may be suitable for discrete element simulation because the temporal coherence makes it likely that each list is almost sorted. Both sorts operate in nearly linear time for such a list, as the number of interchanges is small.

Chapter 3

Object Representation and Contact Detection

In this chapter, we will discuss the geometric modeling concepts which lead to the development of the contact detection algorithm described later in this thesis. How to represent the geometric object has long been an important research topic in areas like computer graphics and computational geometry. However, in physically-based simulation, the geometric representation not only has to describe the arbitrary shape of 3D geometry accurately, but also needs to facilitate the contact detection process. Moreover, The DEM community has in general avoided the use of traditional representation schemes, such as polygon and surface patches, because of the excessive cost of performing contact detection. In the following sections, we introduce a versatile object representation scheme called the discrete function representation (DFR) which requires only $O(N)$ computational cost to perform the contact resolution phase of contact detection. The force generation at the contact due to the geometric intersection between bodies is described in later sections.

3.1 Geometric Representation

The following types of geometric representation have been proposed as follows:

- **Polygonal Representation**

  Polygons are the most commonly used geometric models in computer graphics and simulation. They are simple and versatile in representing a wide variety of complex objects. Also, it is widely supported by graphics hardware accelerators.

- **Constructive Solid Geometry**

  Constructive Solid Geometry or CSG assemble complex objects from simple primitives such as blocks, spheres, cylinders, cones, etc. Operations such as intersection, union, and difference are provided. This method is particularly suitable for mechanical design because user can intuitively manipulate the objects in correspondence to the design process.

- **Implicit Function Representation**

  Implicit surface are defined using implicit functions. They are defined with mapping from space to the real numbers and the implicit surface are the loci of points where $F(x, y, z) = 0$. Such a function defines clearly what is inside the model, where $F(x, y, z) < 0$, and what is outside the model, where $F(x, y, z) > 0$.

- **Parametric Representation**

  Parametric surfaces are mapping from some subset of the plane to space. Unlike implicit surface, parametric surfaces are not generally closed manifolds. Therefore, unlike CSG and implicit surfaces, they do not represent a complete solid model, but rather a description of surface boundary.

In DEM analysis, these traditional representation methods are not efficient for performing contact detection.

3.1.1 Superquadratic Representation

Before we discuss the discrete function representation scheme, we introduce the superquadratic representation, which led to the development of DFR.

The mathematical description of superquadric volumes is presented below. The general equation of a three dimensional superquadric is given by:

$$\left[\frac{x}{a}\right]^{\epsilon 1} + \left[\frac{y}{b}\right]^{\epsilon 2} + \left[\frac{z}{c}\right]^{\epsilon 3} - 1 = 0 \tag{3.1}$$

where a, b and c determine the principle axes lengths and ϵ_i are exponents which change the shape of the surface. When $\epsilon_i = 2$ the equation of an ellipsoid is obtained, and for equal axis lengths, a sphere. Figure [3-1] shows an aggregate of sample superquadrics with varying parameters. Superquadrics have been applied to computer aided design, computer vision and computer animation [55, 54].

Inside-Outside Property

A particularly useful aspect of volumetric expressions is the so called 'inside-outside' property. To determine if the point (x,y,z) is inside or outside the body we substitute the point into the function below:

$$F(x, y, z) = \left[\frac{x}{a}\right]^{\epsilon 1} + \left[\frac{y}{b}\right]^{\epsilon 2} + \left[\frac{z}{c}\right]^{\epsilon 3} - 1 \tag{3.2}$$

The inside-outside tests yield:

$$\begin{aligned} F < 0.0 \quad & point\ inside\ surface \\ F = 0.0 \quad & point\ on\ surface \\ F > 0.0 \quad & point\ outside\ surface \end{aligned} \tag{3.3}$$

Now if F, is greater than zero the point is outside the body, F is equal to zero the point is on the surface, and if F is less than zero the point is inside the body, as

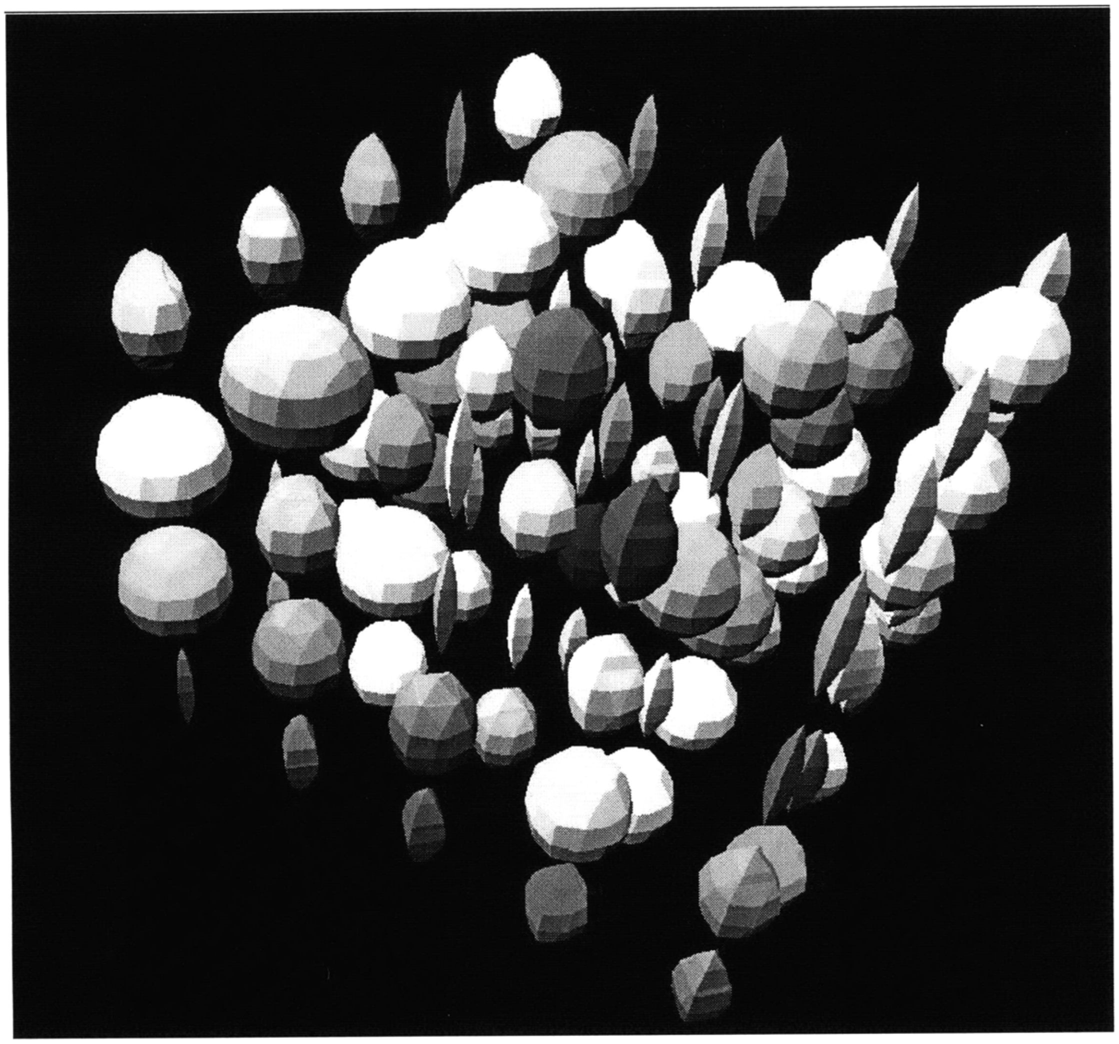


Figure 3-1: Superquadrics with varying exponents

the example shown in Figure [3-2].

Furthermore, we note that for $\epsilon = 2$, $|F|$ is the square of the distance of the point from the surface. In general Equation [3.2] provides a measure of the distance of the point (x, y, z) from the surface of the superquadric, $F(x, y, z) = 0$. It is estimated that 80% of solids can be represented by superquadric functions. Solids which cannot be represented directly can be derived either by combining several superquadrics or by superquadrics expressed in higher dimensions, called hyperquadrics, [19]. In [40], the superquadric particles have been extensively used for simulating the behavior of granular materials.

Using the explicit form of the superquadric equation in Equation [3.2], a *discrete function representation* for contact detection is derived, where volumetric objects are sliced into an ordered set of uniform cubes (*voxels*). Cubes that intersect the object's surface are then used to approximate this surface by computing the location of the intersection points of the surface with the cube edges. These points are then used to form the facets of a tessellated surface. The marching cube algorithm [30] is employed to derive the approximate surface.

3.2 Discrete Function Representation

In this section, we describe an algorithm called Discrete Function Representation which can efficiently identify the details of interference between two bodies. It was originally developed by O'Connor [37].

The DFR scheme provides a generic solution for representing arbitrary geometric object in 3D. We can impose a grid with a specific resolution on the finite-boundary object as shown in Figure [3-2]. At each intersection point of the grid, we can compute and assign a value to the point according to the object geometry. In other words, we represent the geometry by a set of pre-computed quantities on the grid, rather than using continuous curves. To obtain the boundary of the geometric object, we

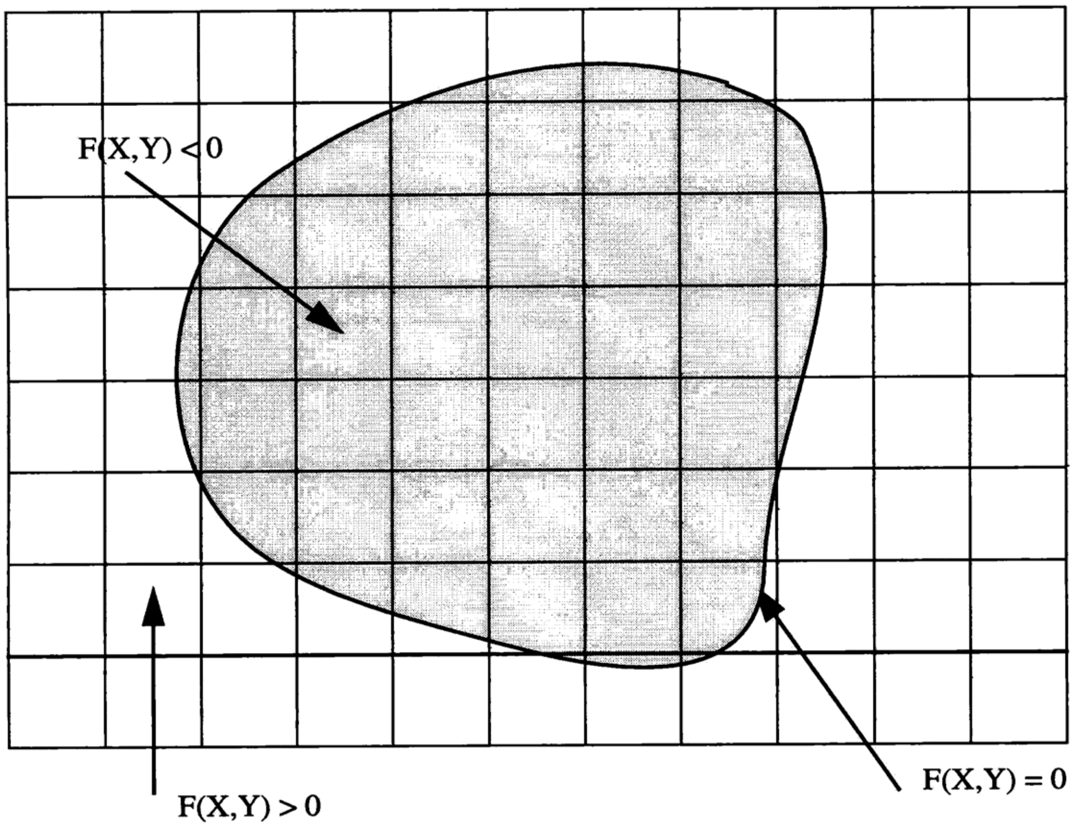


Figure 3-2: Inside-Outside Property of Implicit Function

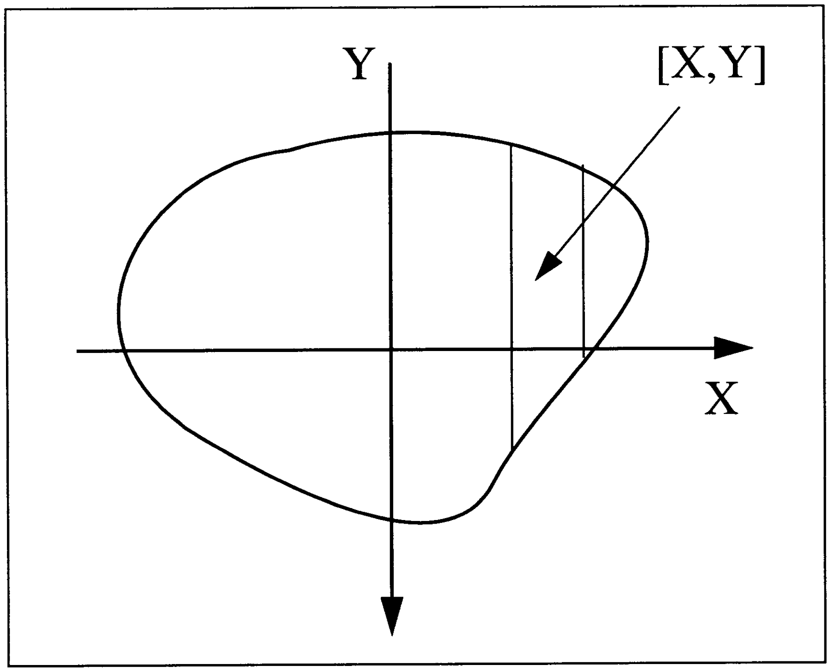


Figure 3-3: DFR Contact Test

can interpolate between the sample points to get a better approximation. In the next section, we show a simple example to illustrate the idea.

3.2.1 Description of DFR Scheme

Figure [3-4] shows a simple example of two DFR objects. The DFR scheme treats the boundary of an object as a pair of single valued functions, for example, $y = f(x)$. The function is then sampled at a particular resolution, say dx, to get the discretized coordinates. Since the boundary points are uniformly sampled, we can find an efficient mapping from the coordinate to the actual memory address space in each dimension. This allows us to retrieve the data quickly.

To perform contact detection between the two objects **A** and **B**, the geometry of

B is transformed to the reference frame of **A**. Bounding box checks are performed and the minimum and maximum extent of the overlap region in X-direction is calculated, as shown in Figure [3-4]. Using the X-coordinate to storage space map, the points in the overlap region are retrieved and marked as possible interacting points. The retrieved points in each object are checked to see if they lie inside or outside the contacting object. To perform an inside-outside check for a point of object **A** in **B**, the point from object **A** is transformed into the reference frame of object **B**. The X-coordinate of the point indicates the panel of object **B** that the point lies in. A check is then performed to see if the point indeed lies in this panel by comparing the Y-coordinate, as illustrated in Figure [3-3].

3.2.2 DFR in 3D

In this section, we extend the DFR scheme described above to 3D space. The discrete bounding hull data structure, which comprises of slice, prism, and cell, is used to enclose the 3D space. Slice is a 2D array composed of prisms and prism is a 1D array composed of cell, which is the basic building block in the discrete space. A ragged 3D array of cell indices called stencil and a set of offset/run-length (ORL) arrays are used to described the object layout, such that each cell in the discrete bounding hull corresponds to a unique location in the 3D stencil. The detailed procedure is given in Figure [3-5] (detailed derivation can be found in [37]). Figure [3-6] shows the hierarchical process of DFR contact detection.

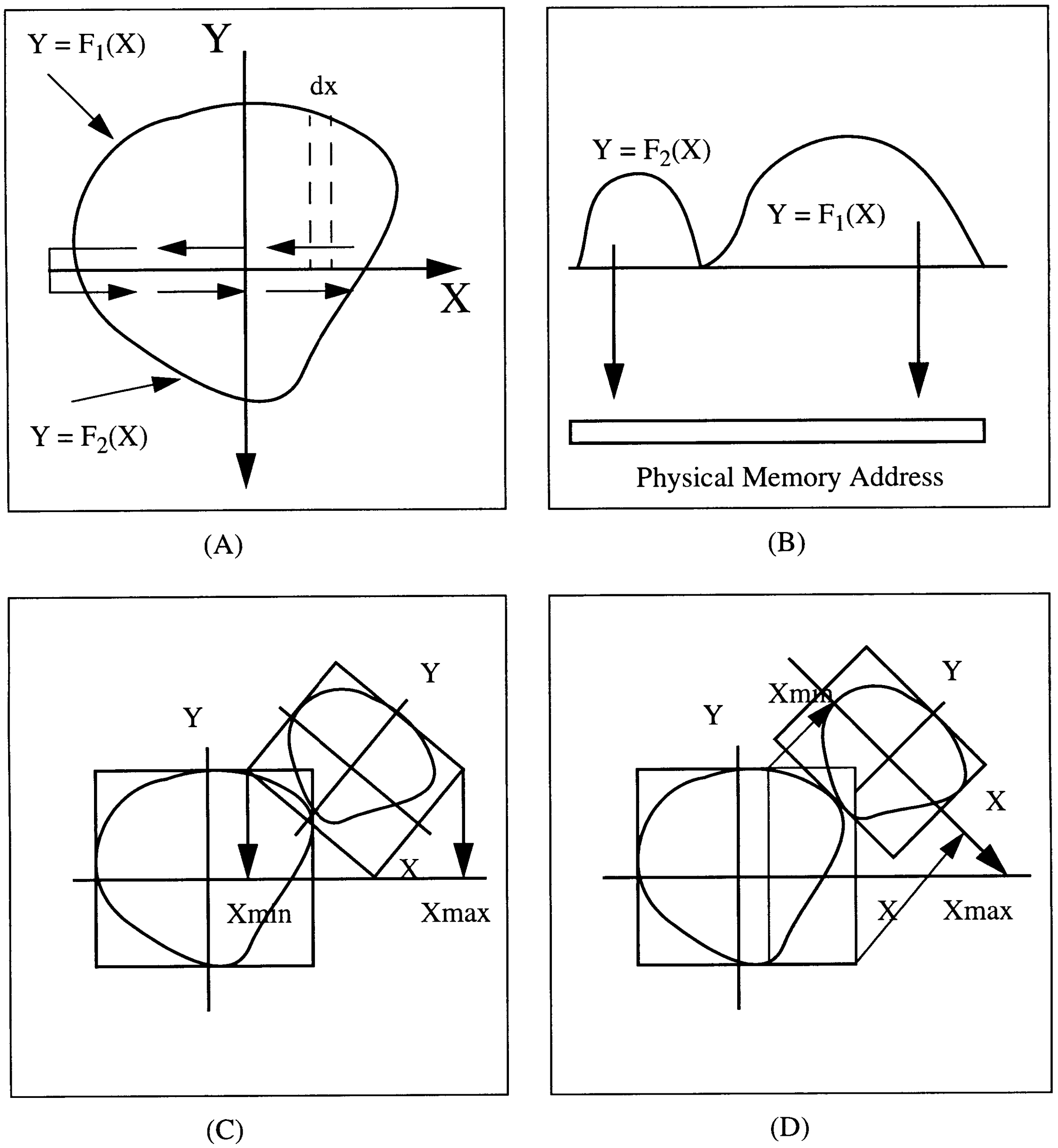


Figure 3-4: Discrete Function Representation [From [37]] (A) Traversal of Vertices (B) Physical To Address Space Mapping (C) Projection of B to A (C) Projection of Reduced A to B

1. Transform a copy of bounding box of each object into frame of the other object. Intersect each transformed bounding box with each local bounding box. If intersection points occur, then calculate the zone vector pair description and go to the next step, otherwise exit.

2. Transform the zone coordinates into the discrete coordinate system of the object to obtain the zone index set.

3. Perform cell index filtering by iterating over the zone index set.

    (a) Map discrete space coordinate of cells to offset/run-length indices.

    (b) If the index triple maps to an offset/run-length index with a valid cell descriptor table entry then continue to the next step, otherwise exit.

    (c) For each edge in the referenced cell of the source object.

        i. Transform the edge into the frame of the target object.

        ii. Clip the edge to each target cell that it spans.

        iii. For each intermediate edge

            A. Map real space coordinate of edge to cell coordinates.

            B. If the cell coordinate matches a valid offset/run-length index trip then perform the geometric intersection operations, otherwise exit.

Figure 3-5: DFR 3D Algorithmic Description [From [37]]

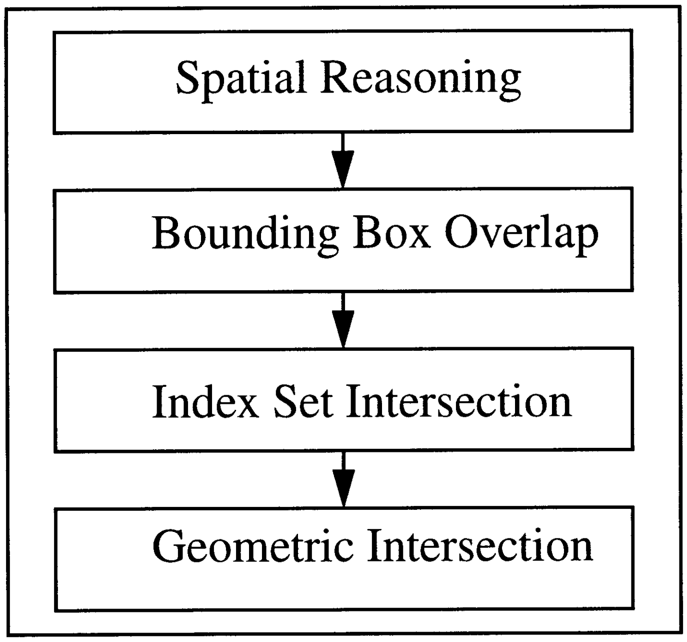


Figure 3-6: DFR Contact Detection Process

3.3 Performance Analysis

The performance of the DFR algorithm is illustrated in Figure [3-7]. It shows that for a pair of DFR objects intersects with each other, as we increase the resolution of representing the object, we can observe an $O(N)$ relationship between the number of surface points and the required time to perform the contact resolution. O'Connor [37] noted that in practical applications the performance was actually better than $O(N)$ and was closer to $O(\sqrt{N})$. Figure [6-12] shows an example of packing the DFR particles into a rectangular box.

3.4 Collision Response

In discrete element analysis, the geometry of particles must be accurately represented and their physical properties defined. Once the characteristics of individual bodies are known, a mathematical model is constructed which represents an idealization of

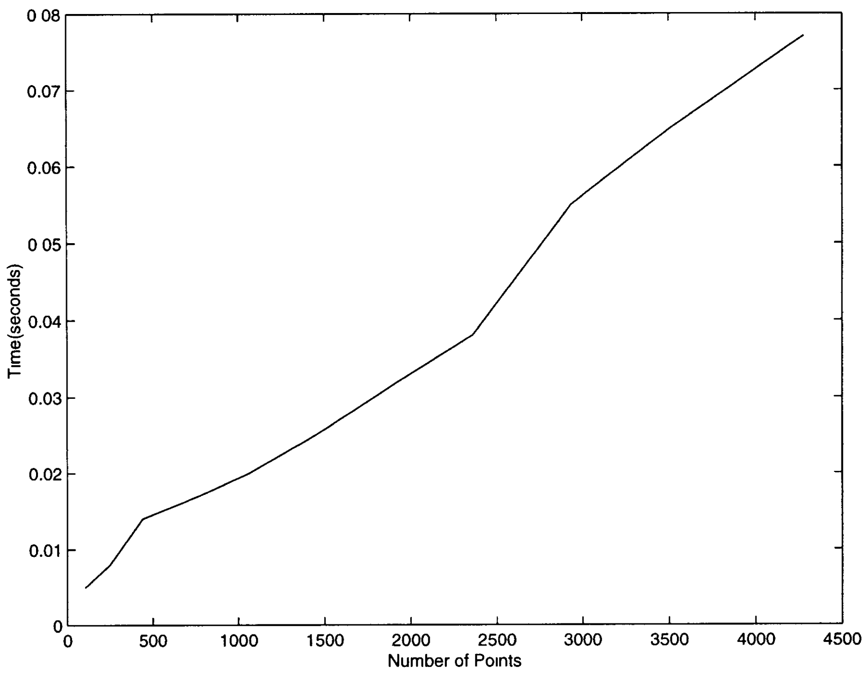


Figure 3-7: Performance Analysis of DFR Algorithms

the actual physical system. The discipline of multibody dynamics is concerned with the study of the relation between the motion of physical systems and the forces causing its motion. Classical dynamics usually deals with macroscopic models invoking the continuum hypothesis, according to which the physical properties are continuously distributed in space occupied by a physical system. With the advent of digital computers, a number of methods of dynamical analysis have been developed and employed to overcome this difficulty. Efforts were undertaken to generate and solve equations of motion for multiply contacting objects [3, 9].

In next section, we describe how we handle the parameters such as forces, moments, and constraints for the multibody system.

3.4.1 Equations of Motion

The linear and rotational rigid body motion of a discrete element are governed by the following equations:

$$M\ddot{u} + C\dot{u} + Ku = F \tag{3.4}$$

$$I\ddot{\theta} + C\dot{\theta} + B\theta = T \tag{3.5}$$

- M = mass matrix
- K = stiffness matrix
- I = moment of inertia
- B = torsional stiffness
- C = damping matrix
- F = external forces
- T = torques

The goal here is to calculate the updated position, velocity, and acceleration of the particle based on the external forces and torques applied.

We note that the computation of geometry and contact detection is performed with respect to the local coordinate system. The advantage of doing so is because the global moments of inertia do not have to be recalculated at each time step. Modal analysis techniques also benefit greatly from maintaining these quantities with respect to a local frame, [55, 54].

The solution to these equations over time is obtained numerically using a fourth order numerical method, the Runge-Kutta-Nystrom scheme. Adopting the explicit scheme can avoid of assembling and inverting large stiffness matrices, which is the major disadvantage of the implicit scheme. However, a drawback to adopting this scheme is that the magnitude of the time step must be determined to ensure the stability.

$$
\begin{aligned}
k &= \frac{dt}{2} \\
A_t &= k\, f(t_t,\ u_t,\ \dot{u}_t)r && (3.8) \\
\beta_t &= k\, [\dot{u}_t + \frac{A_t}{2}] \\
B_t &= k\, f(t_t + k,\ u_t + \beta_t,\ \dot{u}_t + A_t) && (3.9) \\
C_t &= k\, f(t_t + k,\ u_t + \beta_t,\ \dot{u}_t + B_t) && (3.10) \\
\delta_t &= dt\, [\dot{u}_t + C_t] \\
D_t &= k\, f(t_t + dt,\ u_t + \delta_t,\ \dot{u}_t + 2C_t) && (3.11) \\
u_{n+1} &= u_t + dt\, [\dot{u}_t + \frac{[A_t + B_t + C_t]}{3}] && (3.12) \\
\dot{u}_{n+1} &= \dot{u}_t + \frac{[A_t + 2B_t + 2C_t + D_t]}{3} && (3.13)
\end{aligned}
$$

Figure 3-8: Runge-Kutta-Nystrom Numerical Integration Steps

Equations [3.4] and [3.5] are rewritten to formulate the explicit scheme as:

$$\ddot{u} = M^{-1}\{F - C\dot{u} - Ku\} \tag{3.6}$$

$$\ddot{\theta} = I^{-1}\{T - C\dot{\theta} - B\theta\} \tag{3.7}$$

Details of the Runge-Kutta-Nystrom numerical integration scheme are shown in Figure [3-8], (see [27, 11] for detailed derivation). The computation of the position and velocity at time $(t + dt)$ are estimated from evaluating Equations [3.6] and [3.7] at four stations [3.8, 3.9, 3.10, 3.11]. Equations [3.12] and [3.13] specifies the updated position and velocity of the body whose motion is being analyzed.

The value of dt in these equations is the time step for the integration determined from:

$$dt \ \leq \ dt_{critical} \ = \ \frac{T_n}{\pi} \tag{3.14}$$

$dt_{critical}$ is the critical timestep and T_n the smallest period of the system being analyzed. Typically a value for $dt = \frac{T_n}{10}$ is often taken as a satisfactory estimate, [6].

3.4.2 Incremental Collision Resolution Scheme

Collision response refers to calculation of the inter-body forces and torques required to maintain dynamic equilibrium of the system of bodies being simulated. These components are derived from the linear and rotational motions of each object. The net effect one object has on another, i.e. reactive or frictional forces, are transmitted at the object-object interface, here referred as the contact region (or contact volume in the 3D case). The contact region is considered to be made up from the points of contact, as shown in graphically in Figure [3-9]. The points describing this region are those obtained in the contact detection stage described in section [3.2]. The frame of reference for these points is arbitrarily chosen as being one of the local coordinate systems of the objects involved in the collision. The required normal and shear forces are determined using an incremental penalty force scheme as explained in the next section.

The **DEM3D** simulation system generates a contact object whenever the contact detection procedure captures an overlap between an pair of particles. Each contact object contains the detailed information required to resolve the normal and shear forces that occur due to a collision between two objects.

Specifically the C++ language data structure describing the contact object abstraction is listed below:

```
class DEMcontact
{
  DEMVector        vl[2];  /* lists of contact points                 */
  int              np[2];  /* number of points in each list           */
  DEMshape     * pair[2];  /* pointers to objects in contact pair     */
  DEMMatrix     * f2f[2];  /* frame to frame transformation matrices  */
```

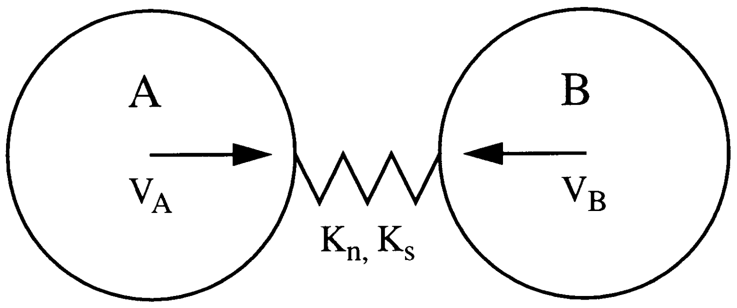


Figure 3-9: Contact Resolution Details

```
  DEMVector     force[2]; /* incremental force                      */
  DEMVector    torque[2]; /* incremental torque                     */
  int              cycle; /* phase flag                             */
  DEMcontact      * next; /* point to next object in contact list   */
  DEMcontact      * prev; /* point to previous object in contact list */
};
```

The deformation is modeled as the compression of spring between objects in contact. It can be viewed as a penalty function approach with the spring stiffness representing the penalty constant. Usually the spring stiffness is chosen based on material properties. As the object moves close to each other the spring compress generating the contact force. The shear force is described by Coulomb's friction law. The contact force is incrementally updated at every time step using Equations [3.15] and [3.16]. Suppose V_{AB} is the relative velocity and $\vec{n}$ is the normal vector and $\vec{s}$ is the direction of shear force at the contact point. Then the total normal force F can be written as:

$$F_t = F_{t-1} + K_n(V_{AB} \cdot \vec{n})dt \tag{3.15}$$

and the shear force S is

$$S_t = S_{t-1} + K_s(V_{AB} \cdot \vec{s})dt \tag{3.16}$$

such that

$$|S_t| < \mu F_t \tag{3.17}$$

where K_n is the normal stiffness, K_s is the shear stiffness, μ is the coefficient of friction and dt is the time step.

It was already noted that all points describing the geometry of an object are expressed with respect to that object's local coordinate system. However, points stored by the contact object are expressed with respect to the frame of that object in which the points are in contact with, not the frame of the object whose geometry they describe, just as what we discussed in Section [3.2].

3.4.3 Contact Table Data Structure

In order to implement the incremental collision resolution scheme described in the previous section, we need a special data structure to keep track of the creation and consequent existence or removal of a contact object between two objects because the magnitude of the time step used to compute each increment may not be removed immediately. In fact the two bodies may remain in a perpetual state of contact. A data structure called contact table that contains a reference to all occurrences of contact between objects in the simulation environment is utilized to support this scheme. For convex objects, there will always be a unique contact pairing which allows for certain efficiency in the implementation of the contact table. The structure

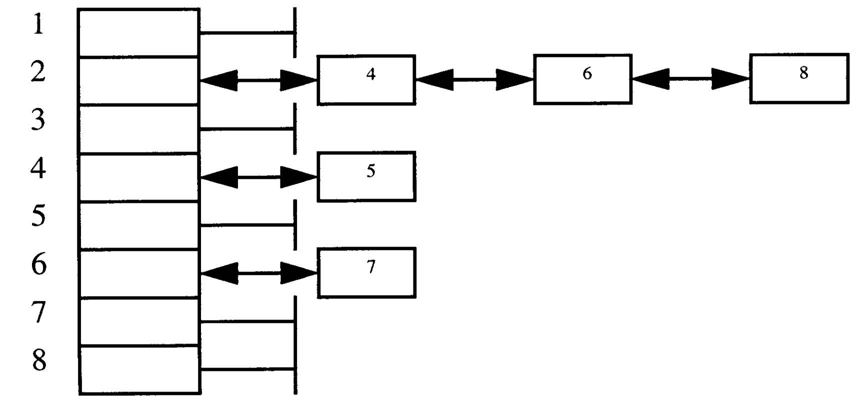


Figure 3-10: Contact Table Description

of the table is described in the following steps:

1. Create a contact table of length M, where M is the number of objects in the simulation system. Each table entry stores a pointer to doubly linked list of contact objects and a counter of the number of contact objects in the list.

2. Each table entry is uniquely associated with each object by its index, i.e. objects in the simulation are contiguously numbered, with the assumption that objects removed during the simulation will either be replaced by new objects or will be of such a small net number that the table will not need to be adjusted or rearranged.

3. Contact objects are inserted into the table by the lowest index of the object pair. For instance, if object 2 is in contact with object 6, and objects 7 is in contact with object 6, then the contact objects describing these occurrences are inserted into contact table in entry 2 and 6, as shown in Figure [3-10]. Contact objects are inserted at the the tail of each list so that they can be processed in a FIFO (first in first out) fashion at each iteration. The FIFO ordering reflects the natural ordering and sequence the occurrence of object-object contact instances.

At each iteration of the simulation, when a pair of objects are determined to be candidates for contact detection, the contact table is first consulted to see if there is a previous reference to their contact. This is simply done by checking if there is a contact list attached to the contact table entry corresponding to the lowest index value of the two objects being processed. If the list exists, then check the list using FIFO precedence for an existing contact object for the object pair. If there is none, create a new instance of a contact object and insert it in the list attached at the table position given by the lowest index of the object pair. Otherwise, retrieve a reference to the existing contact object and accumulate the contact reactions by appending a new increment of reaction forces and torques to the existing values, paying attention to the orientation of the existing vector components. By sequencing through all objects at every iteration contact reactions will incrementally build up and fall off in a smooth transition allowing the objects to move apart consistently but avoiding some of the control problems associated with less sophisticated penalty methods.

Additionally, at each iteration of the simulation a sweep is made through the contact table to identify and eliminate any contact objects containing zero valued accumulated forces and torques. This criteria is based on the assumption that the objects with zero relative motion at an interface and reaction forces equal to zero are no longer in contact.

3.4.4 Constraints

In the **DEM3D** system, constraints are not treated as independent objects. Instead, we deal with the constraints when we solve the contact response. For instance, if we specify a bonding force F between two objects, we can artificially create a contact object for these two objects and specify the threshold value F for the incremental collision resolution scheme. If the accumulated normal force is less than the threshold value, then bond will bind these two objects together. On the other hand, if it surpasses the threshold value, then the bond will be broken and then this artificial contact object will be deleted, too. An example for the bonding constraint is presented in Figure [6-11].

3.5 Volume Integration

In order to correctly compute the collision response of arbitrary shape of complicated 3D geometry represented by DFR, we need to calculate several required parameters describing the mass distribution of the rigid bodies, such as the total mass, the location of the center of mass, and the moments and products of inertia about the center of mass, so that we can apply the equations of motions to correctly compute the physical properties of the objects.

In our implementation, we adopt the approach of volume integration proposed by Mirtich [33]. The principal quantities in rigid body dynamics are a particle's linear momentum **L** and angular momentum **H**.

$$\mathbf{L} = M\dot{\mathbf{u}} \tag{3.18}$$

$$\mathbf{H} = \mathbf{J}\dot{\theta} \tag{3.19}$$

In Equation [3.18], $\dot{u}$ is the linear velocity of the center of mass, and in Equation [3.19], $\dot{\theta}$ is the angular velocity of the body, respectively. **J** is the 3 X 3 mass matrix containing the moments and products of inertia of each dimension. In order to correctly apply the equations of motion for the particle, the quantities M, J, and $\dot{u}$ must be determined. Using the algorithm developed by Mirtich [33], we can compute the volume integrals in $O(N)$ computing time, where N is the number of vertices of the body.

3.6 Summary

In this chapter, the critical issues in geometric object representation and contact detection are reviewed. The DFR method is employed to represent the arbitraily shaped 3D object and provide an efficient solution toward collision detection with only $O(N)$ computational complexity, where N is the number of surface points used to represent the object geometry. In order to simulate the collision response between objects, the incremental collision resolution scheme is adopted to resolve the response between a pair of contacted objects. We also use the contact table data structure to manage the creation and removal of contact objects between time steps.

Chapter 4

Implementation

In this chapter, we discuss the implementation issues of arising during the development of the **DEM3D** system. From the software perspective, we can separate the whole simulation system into two parts. The core module, which defines object geometry, performs spatial reasoning, contact detection, and contact resolution, is fundamentally application and machine independent. It provides the basic functionality required for all kinds of DEM simulation. The user interface module, which takes input data defined specifically for a particular problem and graphically presents the simulation results, is more application and machine dependent. In designing the **DEM3D** system, we try to separate the core module from the application dependent part as much as possible so that we can obtain high portability for running the program on different platforms.

4.1 Architectural Framework

In Chapter 1 we note that the DEM simulation is basically comprised of four modules - geometry representation, contact detection, physics, and visualization. In order to integrate these components together, a graphical toolkit and a simulation description language based on *Tcl* are employed. The *Tcl* language has proven to be an effective

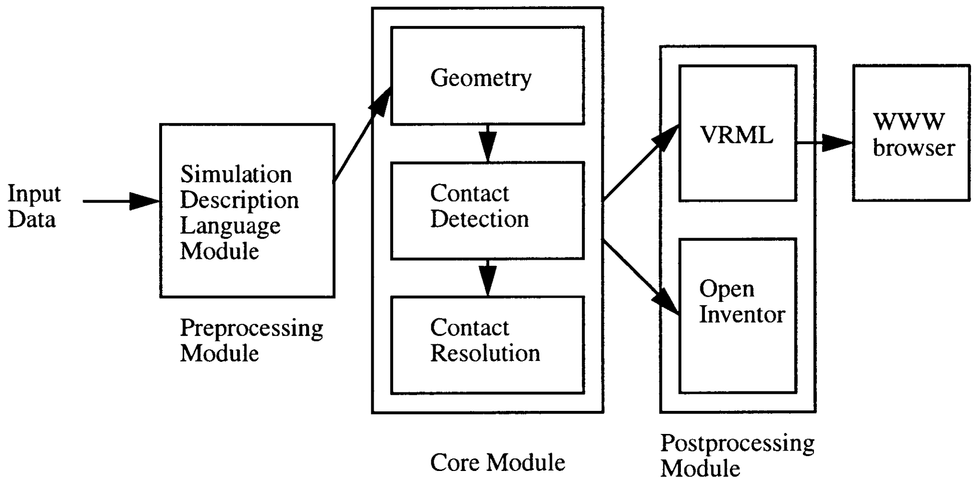


Figure 4-1: Software Framework of Modeling Environment

tool in integrating the functionally different software components together. The entire architectural framework of the modeling environment and the relationship between different components is shown in Figure [4-1].

4.2 Visualization Graphics

In 3D discrete element simulation, it is essential to provide a good visualization tool for the user to investigate the interactions between particles at the microscopic level of detail. Without a visualization tool to digest the large amount of data generated by the number-crunching code, the results of the simulation would become dauntingly difficult to interpret. In the **DEM3D** system, we employ two set of tools to perform the visualization task, one is Open Inventor, the other is the Virtual Reality Modeling Language (VRML). Details of both are given in the following sections.

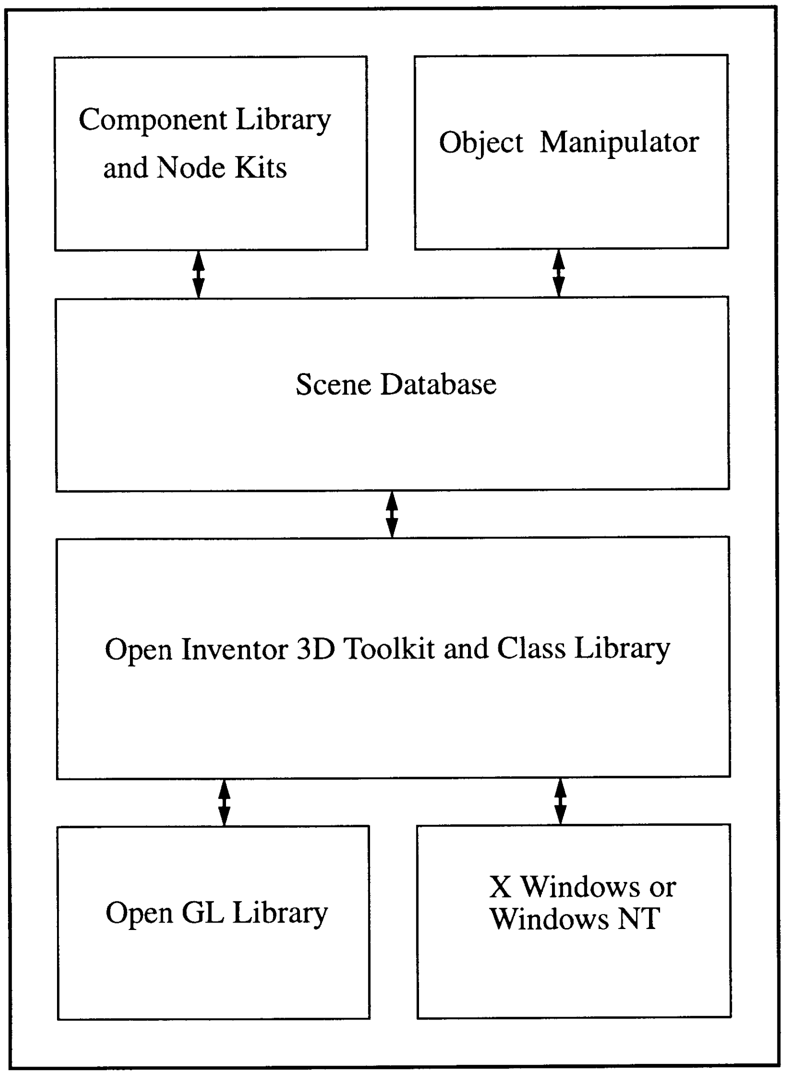

Component Library and Node Kits
Object Manipulator
Scene Database
Open Inventor 3D Toolkit and Class Library
Open GL Library
X Windows or Windows NT

Figure 4-2: Open Inventor Architecture

4.2.1 Open Inventor

Open Inventor [51] is a versatile C++ class library developed by Silicon Graphics Inc. It offers a comprehensive solution to interactive graphics programming problems. It presents a programming model based on a 3D scene database that dramatically simplifies graphics programming. It includes a rich set of objects such as cubes, polygons, text, materials, cameras, lights, trackballs, handle boxes, 3D viewers, and editors that speed up the programming time and extend the 3D programming capabilities.

The major features of Open Inventor are summarized as follows: [51]

- built on top of OpenGL standard
- defines a hierarchical structure for representing 3D objects
- introduces a simple event model for 3D interaction
- provides animation, object picking, and high-quality rendering
- is a cross-platform 3D graphics development system

In the Open Inventor environment, programmers deal with 3D objects, rather than drawing primitives. All information about these objects - their shapes, size, coloring, surface texture, location in 3D space - is stored in a scene database. This information can be used in a variety of ways. The most common use is to display, or render, an image of 3D objects on the screen. Instead of manipulating the low-level graphics primitives, for instance, like OpenGL, programmers create objects and encapsulate the information of graphical properties and associated operation functions into these objects. The whole programming model is well defined in an object-oriented fashion. We do not have to access the frame buffer to perform the rendering. By traversing the hierarchy of scene database, we can manipulate and interact with the 3D objects in a scene. A different path of traversal will result in a different scene viewed on the screen.

Key Components

A few major components of Open Inventor are listed below: [51]

- Nodes (shape,transformation, light, texture, etc.)
- Engine (boolean operation, animation, etc.)
- Node Kits (xwindow, device, viewer, etc.)
- Event Manipulator (event, sensor, error, etc.)
- Action (callback,pick,render, etc.)

The relationship between these components is illustrated in Figure [4-2].

Scene Database

The node is the basic building block used to create a three-dimensional scene database in Open Inventor. Each node holds a piece of information, such as a surface material, shape description, geometric transformation, or light source. An ordered collection of nodes is referred to as a scene graph. This scene graph is stored in the Open Inventor database. After the scene graph is built, the user can apply a number of operations, such as rendering, picking, computing a bounding box, and writing to a external file. A typical structure of a scene graph is shown in Figure [4-3].

Example

Figure[4-4] shows a typical 3D scene represented in Open Inventor. In this example, note that we apply the texture mapping onto the objects in order to clearly see the rotation of the particles.

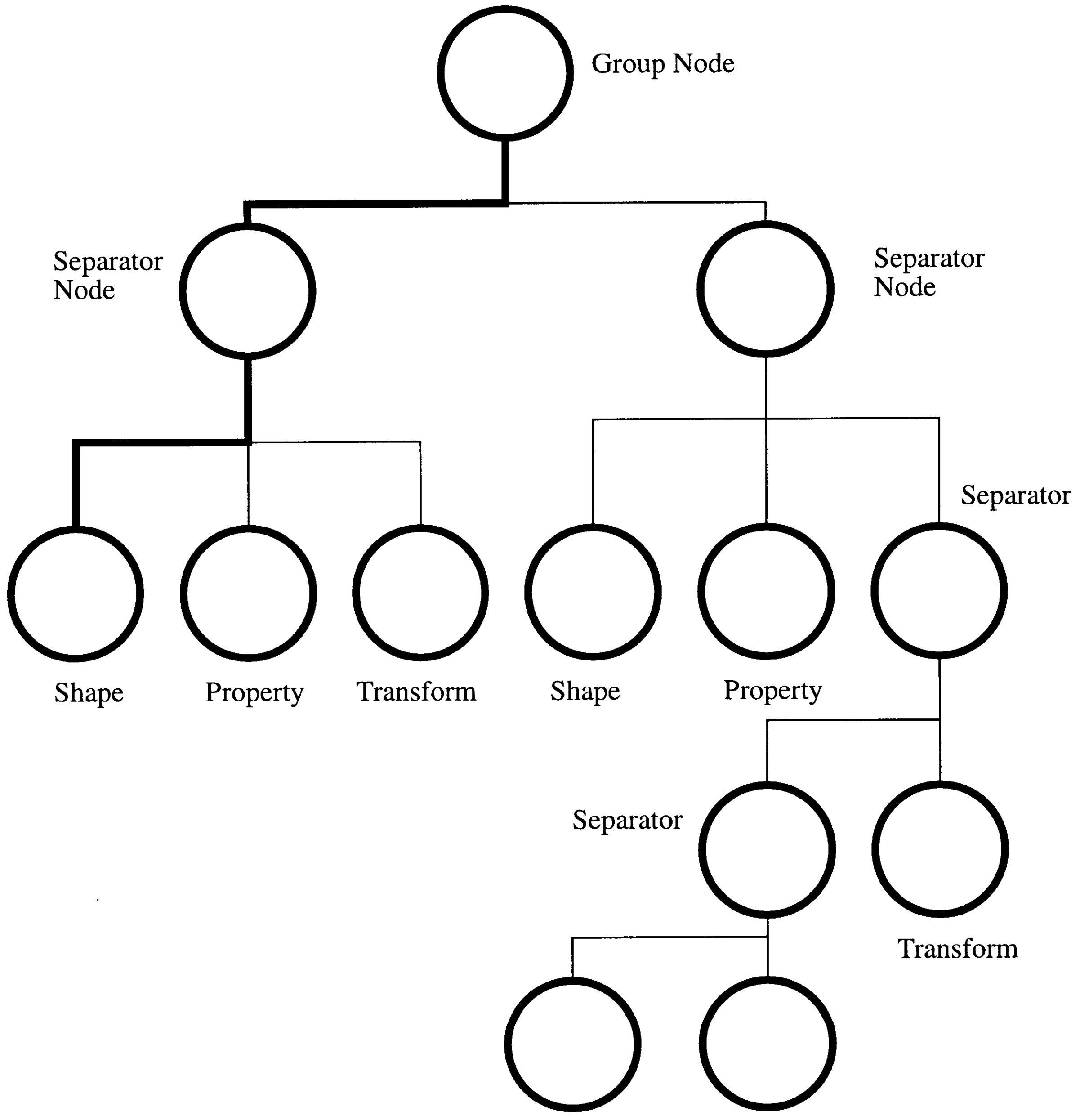


Figure 4-3: Example of a Scene Graph

4.2.2 Virtual Reality Modeling Language

The Virtual Reality Modeling Language (VRML) [12] is a file format for describing interactive 3D objects and worlds. VRML is designed to be used on the Internet, intranets, and local client systems. VRML is also intended to be a universal interchange format for integrated 3D graphics and multimedia. It may be used in a variety of application areas such as engineering and scientific visualization, multimedia presentations, entertainment and educational titles, web pages, and shared virtual worlds. VRML is capable of representing static and animated dynamic 3D and multimedia objects with hyperlinks to other media such as text, sounds, movies, and images. VRML browsers, as well as authoring tools for the creation of VRML files, are widely available for many different platforms. VRML supports an extensibility model that permits new dynamic 3D objects to be defined, thereby allowing application communities to develop interoperable extensions to the base standard. There are mappings between VRML objects and commonly used 3D application programmer interface (API) features.

In the following example, we present a simple template of the VRML format we use in the **DEM3D** system. Note that we use the PROTO command to create a prototype module for general DEM object. Based on this template command, we can instantiate an object by assigning the parameters of origin, radius, color, positions, and so on.

```
#VRML V2.0 utf8
NavigationInfo {
type ["EXAMINE", "ANY"]
}
PROTO DEMObject [ field SFVec3f origin 0.0 0.0 0.0
                  field SFColor thecolor 1.0 1.0 1.0
                  field SFFloat theradius 1.0
```

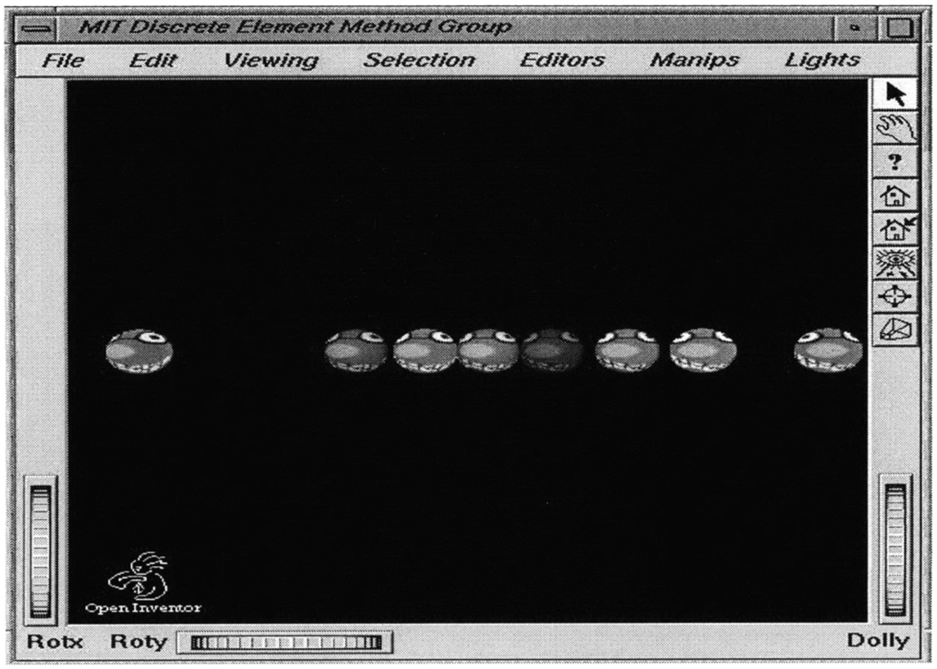


Figure 4-4: A Sample Open Inventor Scene

```
                  field MFVec3f position [] ]
{
  Transform {
     children [
       DEF TRSFRM Transform {
          children DEF DEMSphere Shape {
            appearance Appearance { material Material
            { diffuseColor IS thecolor } }
              geometry Sphere {radius IS theradius }
              }
              translation IS origin
           }
    ]
}
```

```
DEF Clock TimeSensor {
  cycleInterval 20.0
    loop TRUE
  },

DEF DEMObjectPath PositionInterpolator {
    key [
     0.0 0.05 0.10 0.15 0.20
     ...
   ]
 keyValue IS position
}

ROUTE Clock.fraction_changed TO DEMObjectPath.set_fraction
ROUTE DEMObjectPath.value_changed TO TRSFRM.set_translation

}

DEMObject {origin 13 2.05478 -60
           thecolor 0.193298 0.563568 0.00125122
           theradius 1

           position [
                     13 2.055 -60
                     13 2.055 -58
                     ...
     ]
}
```

By using the VRML representation, the simulation core module can be entirely separated from the visualization part, that is, these two modules can be executed on two different machines at the same time. For example, we can run the core computing module on a high-power server and view the results on a low-end personal computer by a web browser. The major advantage is to free the server from handling the graphics rendering task and concentrate on the number-crunching part. In practice, it will reduce the overall computing time by sixty percent. On the other hand, it provides the flexibility for the user to see the simulation results remotely without depending on a particular type of machine to perform the graphics rendering. From our experience with the large-scale problem, the size of a generated VRML file could be substantially huge. As a result, it requires further improvement by using a more compact VRML format or data compression scheme to facilitate the usage.

4.3 Matrix Manipulation Functions [37]

The mathematics of multibody dynamics are effectively described in terms of homogeneous vector and matrix expressions. To provide consistent descriptions of the various transformations a library of vector and matrix operations was implemented. The naming convention for the function names is composed of three fields as shown in the generic prototypes.

The first field of the prototype is a name field used to signify the type of object being operated on. It is either *tm* for transformation matrix, or *v* for vector. The optional suffix on the name field, *v[a]*, indicates that the contents of the source object in the parameter list is changed by the function's operation. The letter *a* is used to denote when this occurs. The *operation* field describes the operation applied to the parameters. For example, *v_scale_3d*, scales a 3D vector argument to the function. The suffix name field *dimension* denotes the dimensionality of the operation, 2D or

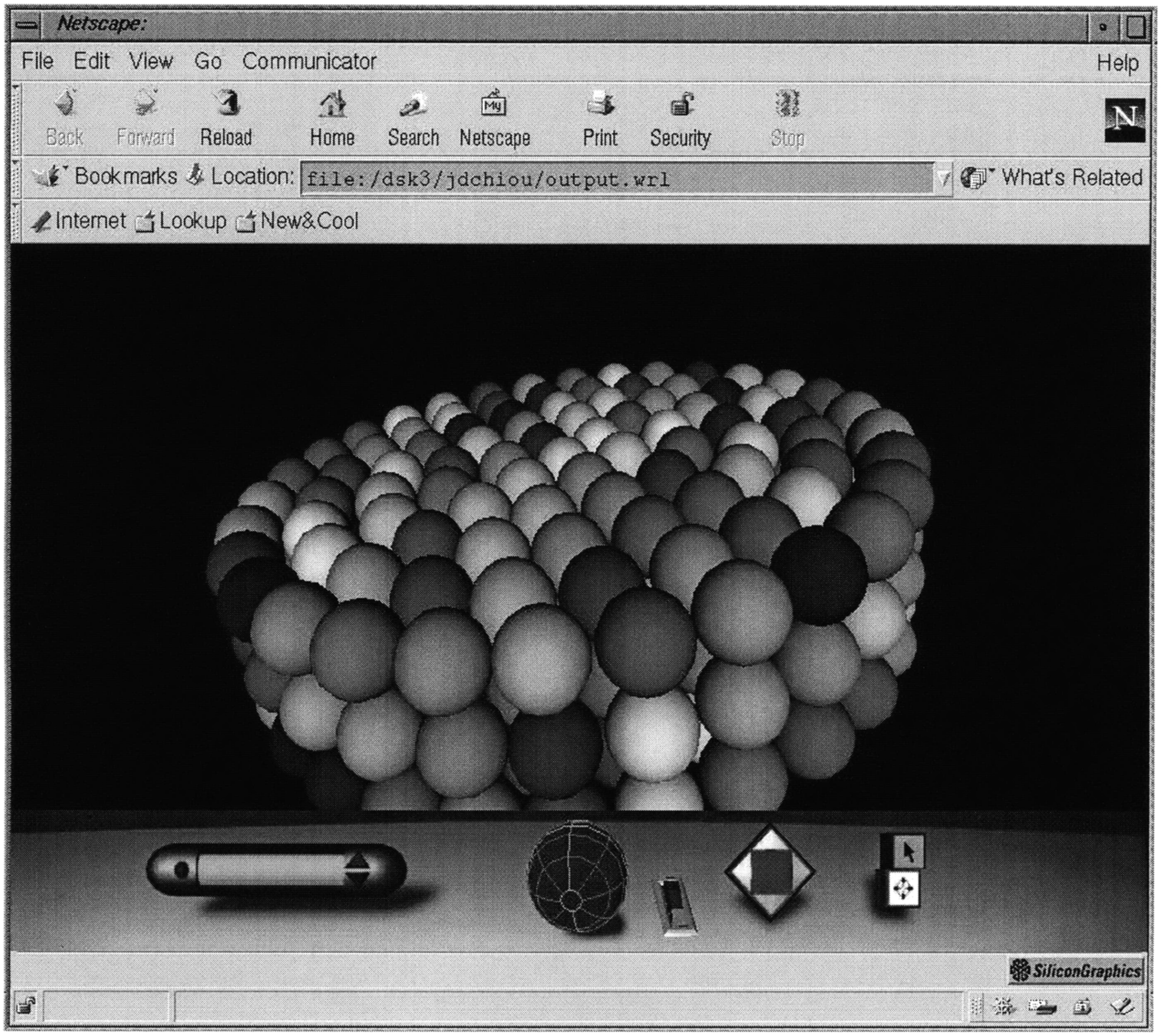

Netscape:
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Stop
Bookmarks Location: file:/dsk3/jdchiou/output.wrl
What's Related
Internet Lookup New&Cool
SiliconGraphics

Figure 4-5: Example of a VRML Scene

Function Name	Purpose
v_copy_3d	copy a vector
va_fill_3d	fill vector with values
v_add_3d	add two vectors
v_sub_3d	substract two vectors
v_scale_3d	scale a vector
v_dot_v_3d	dot product of two vectors
v_x_v_3d	cross product of two vectors
vl_alloc_3d	allocate memory for a vector list
v_len_3d	calculate the length of a vector
v_norm_3d	normalize a vector
tm_copy_3d	copy a transformation matrix
tm_mult_3d	multiply two transformation matrices
tm_rotate_3d	rotate a transformation matrix
tm_translate_3d	translate a transformation matrix
tm_scale_3d	scale a transformation matrix
tm_transpose_3d	transpose a transformation matrix
tm_inverse_3d	invert a transformation matrix
v_dot_tm_3d	post-multiply a vector by a matrix
v_orient_tm_3d	orient a vector by post-multiplying by a matrix

Table 4.1: Matrix Manipulation Functions [From [37]]

3D. The major routines contained in the library are listed in Table [4.3].

4.4 Simulation Description Language

In this section, we describe the input interface of the **DEM3D** system - the simulation description language. It is built on top of the *Tcl* toolkit. The detailed syntax of the commands is listed in Appendix A.

4.4.1 Tcl - Tool Command Language

Tcl is a *C/C++* library of routines containing a parser, interpreter, and a command language which supports variables, flow control constructs, and user definable procedures together with lexical scoping facilities. The top level behavior and syntax of the Tcl command interpreter is similar to that of a UNIX shell. Commands are issued through a sequences of strings separated by blanks and terminated by a newline or a semi-colon. The first field or string in such a sequence is the command name, all subsequent fields are considered to be arguments. Comments are denoted by the (%) character and the backlash character denotes line continuation.

Tcl provides a powerful suite of string manipulation commands, along with the ability to construct higher level abstraction such as lists (of strings), embedded lists, and association lists. Lists are denoted with a *begin* - *end* syntax using curly braces, e.g. a b c. Variables are instantiated dynamically, but with local scope, using the keyword/command *set*. For example:

```
modify penalty 100.0
```

assigns the argument penalty to refer to the value 100.

As in most other interpretive languages, operations on the data (strings) are performed by the application of primitive operators such as *expr*, which denotes an expression to be evaluated and the dollar sign ($), which signals the evaluation of a variable. For example:

```
set x_center 2
set x_radius 3
set x_max [expr $x_center+$x_radius]
```

sets the variable x_max to the result of the expression $x_center+$x_radius, i.e. the string "5". Here the square bracket operator [] forces the interpreter to evaluate the expression $x_center+$x_radius before setting the value of the variable x_max.

4.4.2 Independence of Software Modules

Efforts have been made to maintain independence between the different software modules. This characteristic is desirable from the point of view that the developers can manage the concurrent development of the enrivonment in a platform indepedent manner, and bind the interface component of the system when the platform can support it. As with most graphical interface libraries or utilities, the developer is forced to adhere to the application programming interface set up by the interface developers. This often leads to application code that is intimately linked to the interface modules. In other words the application ends up requiring a knowledge of the interface syntax and structure. From the software development side, it leads to the interface code being deeply embedded within the application code. We have tried to integrate these software modules with minimum interdependence of each other.

4.5 Data Abstraction

The **DEM3D** environment is described in terms of a collection of objects and a simulation description language to operate on these objects. The top level data abstraction is the *DEMsimulation* object. This object is used to encapsulate the entire interface to the **DEM3D** system, in which other objects can be dynamically created, manipulated and removed over the course of a simulation. Reference to the container is sufficient to gain access to any component of the active simulation environment.

The major data abstraction objects in the **DEM3D** system are:

- **DEMsimulation** - A container to encapsulate the system as a hierarchical set of reference to all the primary data abstractions in the system and the simulation parameters describing the physics.

- **DEMshape** - A common container to store a description of the **DEM3D** objects and the base class for the derived classes like DEMsphere, DEMplane, and DEMdfr.

- **DEMstable** - A container for the sorting tables described in Chapter 2.

- **DEMmatrix/DEMvector** - Floating point primitives for numerical calculations.
- **DEMbbox** - Floating point primitive for clipping operations.
- **DEMcontact** - A container for the contact objects described in Chapter 3.
- **DEMco_table** - A container for the contact tables described in Chapter 3.

The global container contains references to the primary **DEM3D** types as shown in Figure [1-4]. Instances of these types are created dynamically and they collectively make up the simulation description. The creation and control of the system is coordinated either interactively through the graphical user interface or in the form of a command language interpreter and scripts.

4.6 A Sample DEM3D Application Script

When the **DEM3D** system is first invoked, a shell is created with a command level interface to the application. We list a sample script in simulation description language as follows:

```
#
# Input file for the DEM3D system.
# Jen-Diann Chiou
# Intelligent Engineering Systems Laboratory
# Massachusetts Institute of Technology
#
# set the origin
#
modify startx 10.0
modify starty 10.0
modify startz 0.0
#
# set x,y,z scale
#
modify xscale 1.5
modify yscale 1.5
modify zscale 1.8
#
# set x,y,z resolution
#
modify resolx 0
modify resoly 0
modify resolz 0
#
# simulation parameters
#
modify timestep 0.001
modify penalty 600.0
modify damping 0.0
modify bsphr 1.0
modify rho 1.0
modify fields 0.0 0.0 -1.0
modify timesteps  500
modify physics 1
modify contact 1
modify sort 1
modify cohesion 0
modify DAMP 0
modify FRICTION 0.1
#
# create plane, sphere, and dfr objects
#
add plane 10.0 10.0 4.0 0.0 0.0 0.0 0.0 0.0 0.0 12.0 12.0 FIXED
add sphere 13.0 10.0 19.0 0.0 0.0 0.0 0.0 0.0 0.0 UNCONSTRAINED
add dfr 13.0 10.0 30.0 0.0 0.0 0.0 0.0 0.0 0.0 2.0 2.0 \
7.25 7.25 7.25 -0.99 UNCONSTRAINED
#
# start the simulation
#
start
```

```
  DEM3D% source input.tcl;
  DEM3D% start;
```

The script can be refined with additional graphical user interface commands and more complex Tcl syntax, but the basic structure is applicable to most **DEM3D** simulation scripts. The code show here forms the core of the scripts used to run many of the applications that are the subject of Chapter 6.

4.7 Summary

In this chapter, we discuss the implementation issues of the **DEM3D** simulation system. The C++ programming language is employed to implement the software based on the object model shown in Figure [1-4]. Powerful visualization tools, such as Open Inventor and the latest VRML standards, are used to render the output from the simulation program. A simple simulation description language based on the

Tcl toolkit is built to simplify the tedious input task and properly integrate different software modules together.

Chapter 5

Distributed Discrete Element Simulation

In this chapter, we present a distributed computing system for discrete element simulation. The major motivation of developing a distributed DEM simulation environement is to increase the scalibility of the computer systems. A highly scalable system provides a high-performance advantage because we can seamlessly integrate distributed computing resources via the high-speed communication network. The scalably distributed system offers an economical solution, in particular, for the large-scale simulation problems, which requires tremendous amount of CPU time, memory and storage.

5.1 Motivation

One of the major bottlenecks of DEM simulation has been the requirements of tremendous amount of computing resources to achieve satisfactory performance. In order to remove this bottleneck, we resort to parallel computing, which provides good scalibilities for large-scale computing projects.

Over the past few years, high-performance computing has been transformed from the vector processing and massively parallel processing to distributed shared-memory architectures. The distributed shared-memory cluster architectures can combine the performance benefits of massive parallel computing with the flexibility of shared-memory multitasking for a wide variety of problems. Figure [5-7] shows the overall MIT IESL cluster's existing computer and network configuration .

Based on the available parallel machines and distributed network, we implement and test the parallel algorithms specifically designed for spatial reasonsing, as described in later sections.

5.2 Characteristics of Parallel Computing

With the increasing availability of powerful parallel computers, parallel computing is no longer synonymous with arcane programming techniques and expensive cost. The generic parallel computer architecture is shown in Figure [5-1]. In this section, we focus on the shared-memory multiprocessors machines, which is the most popular parallel computer architecture nowadays.

5.2.1 Shared-Memory Multiprocessors Machines

The parallel machine that we have in the MIT IESL is a four-processor SGI Power Challenge L-series server. The basic hardware configuration of this powerful server is shown below:

- Processor 0-3: 196 MHZ IP25
- CPU: MIPS R10000 Processor Chip Revision: 2.6
- FPU: MIPS R10010 Floating Point Chip Revision: 0.0
- Secondary unified instruction/data cache size: 1 Mbyte
- Data cache size: 32 Kbytes

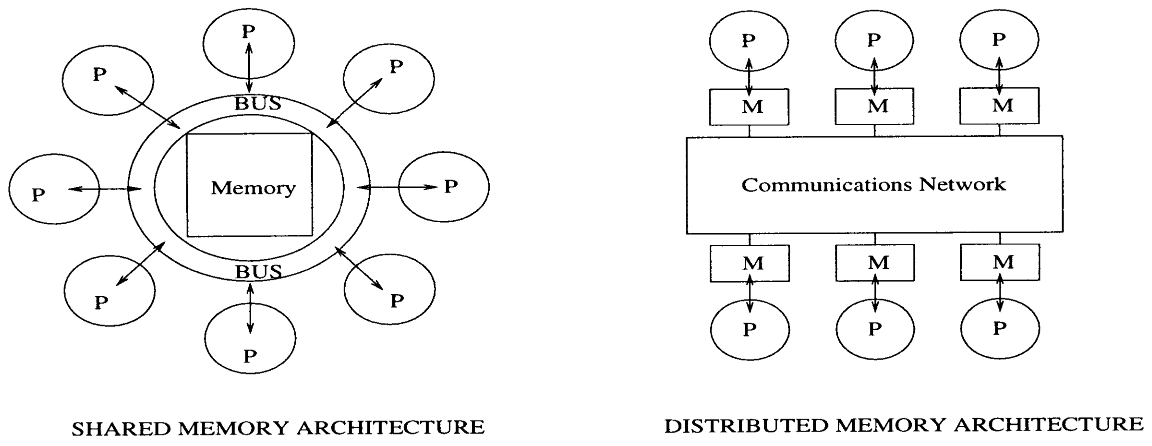


Figure 5-1: Parallel Computer Architecture

- Instruction cache size: 32 Kbytes
- Main memory size: 256 Mbytes, 2-way interleaved

We learned that an efficient program for a shared memory multiprocessor machine, like the current SGI Power Challenge, should follow certain rules:

- *Task Identification.* The programmer identifies a series of tasks, to be performed in a particular order, along with synchronization conditions. Each task, in effect, consists of bringing a data context into the cache memory, performing the extensive sequence of arithmetical operations on it using the cache as a scratch workspace, and moving final results back to the main memory.
- *Memory Referencing.* The tasks are most efficient if they reference very little data in main memory, keep the entire task computation within the cache workspace, and restrict all main memory references to a series of reads or writes of many contiguous memory locations at a time.

- *Processor Synchronization.* The multiprocessing is most efficient if there are very few synchronization restrictions that can cause processors to wait idly. Storing both an old and a new copy of the primary data in main memory often helps eliminate synchronization restrictions of small jobs.

5.2.2 Communication Networks

The fast progress of VLSI technology has brought a strong impact on the digital communication revolution. High-speed local area network with bandwidth up to 100M bits/sec is becoming affordable and popular in the research laboratories. In MIT IESL, we installed a Fast Ethernet-based (100Mbit/sec) switch as the backbone of the local area network. All the SGI servers, PC servers, and workstations are directly connected with each other via the switch. In setting up the distributed simulation environment, the efficiency of the high-speed communication network is a key component because the processors require high bandwidth channel to talk to and synchronize with each other. Figure [5-2] illustrates three typical network topology: full-interconnected crossbar switch, bus network connection, and mesh network connection.

5.3 Parallel Programming Models

From the software design perspective, three major parallel programming methods have been proposed: the data parallel model, the multithreaded model, and the message passing model. Depends on both the hardware configuration and application-specific requirements, programmers should choose the most appropriate programming model to follow [13, 2, 24]. Firstly, the data parallel model exploits the parallelism by executing the same instruction on a large amount of significantly independent data, which is distributed stored across a few computers. This model is easy to understand and implement, and also suitable for many practical engineering applications.

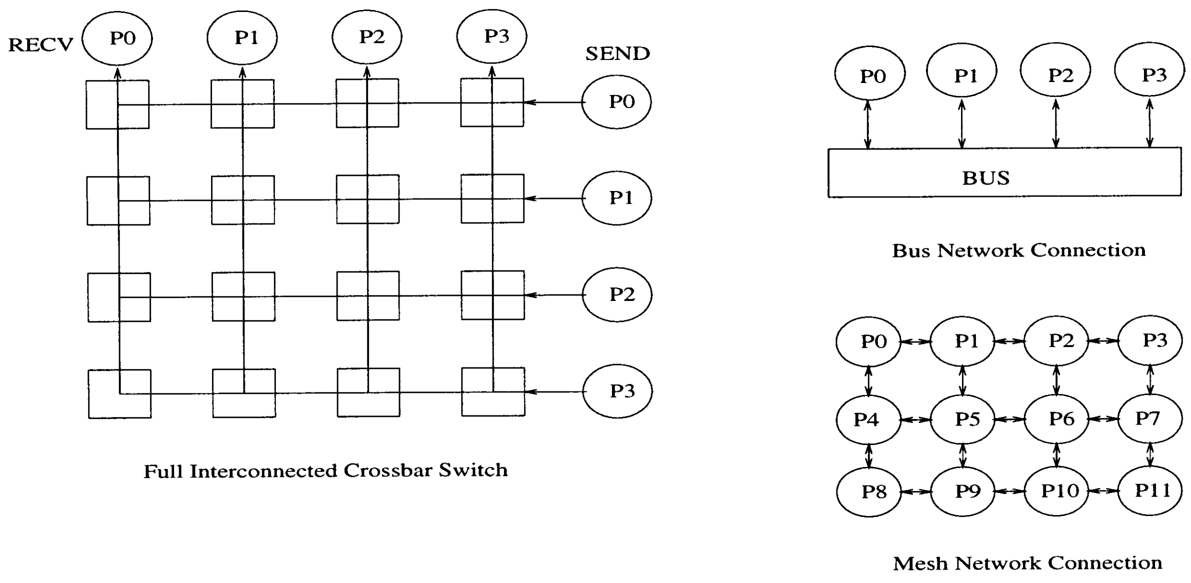


Figure 5-2: Communication Networks

The multithreaded model [17] provides the easiest way of programming among these three methods. Most of the detailed data transfers among processors are handled by complex system procedures, which are hidden from the programmer's perspectives. Message passing is a paradigm used widely on certain classes of parallel machines, especially those with distributed memory. Although there are many variations, the basic concept of processes communicating through messages is well understood. Over the last ten years, substantial progress has been made in casting significant applications in this paradigm. Each vendor has implemented its own variant. More recently, several systems have demonstrated that a message passing system can be efficiently and portably implemented. It is thus an appropriate time to try to define both the syntax and semantics of a core of library routines that will be useful to a wide range of users and efficiently implementable on a wide range of computers.

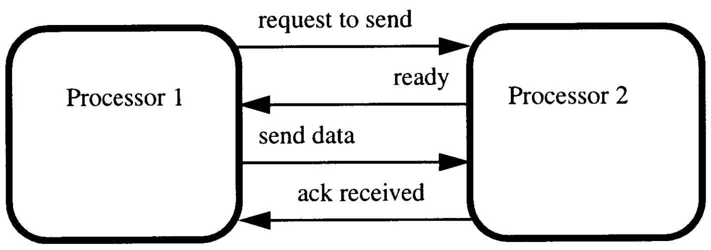


Figure 5-3: Message Passing Programming Model

5.3.1 The Data Parallel Programming Model

One of the most successful parallel programming models is the data parallel programming paradigm. This model is useful for taking advantage of the large amount of data parallelism which is available in many scientific/numerical applications. The embedded data parallelism is exploited by performing the same operation on a large amount of data and distributed across the processors over a communications network. The main characteristic of the data parallel language is the extension of data types to a standard programming language like Fortran or C. Parallel data values consist of a collection of scalar data values. The parallel language contains predefined operations on parallel variables that either operate on the parallel variable element-wise or operate on the parallel value as a whole.

There are two main virtues of the data parallel programming model. One is that data-parallel codes are fairly easy to write and debug. Just as in a serial program, the programmer sees a sequential flow of control statements. The values making up a parallel value are automatically spread across the machine, although typically the programmer does have the option of influencing how data is stored. Any synchronization or communication that is needed to perform an operation on a parallel value is automatically added by the compiler or runtime system. Secondly, it is easy for a programmer to understand the performance of the program. Given the size of a parallel value to be operated on, the execution time for an operation is fairly predictable. Since the execution of each operation is independent of the others, the execution time for the program as a whole is predictable as well.

Although the data parallel paradigm is popular, it has two significant drawbacks. One is the limited range of applications for which data parallel is well suited. Applications with data parallelism tend to be static in nature, the control flow of a data-parallel program is mostly data independent, and the program's data layout and load balancing can be done at the compile time. Many applications are more dynamic in nature and do not have these characteristics. To run programs in parallel, these dynamic applications need to exploit control parallelism by performing independent operations at the same time.

The other drawback is that the data parallel programs tend to be inefficient. Even when a data parallel program gets a good speedup, if one scales the program down to one processor, and compares it to a sequential program, the performance may be disappointing. This phenomenon occurs because the data parallel paradigm is not always a good model for taking full advantage of the sequential processors that make up most of today's parallel machines.

5.3.2 The Multithreaded Programming Model

In order to execute unstructured programs, we need a system which can take advantage of control parallelism. Data-parallel models present the user with a single thread of control. Models based on message passing increase this to one thread of control per processor. To take full advantage of control parallelism, we must virtualize the number of threads of control so that whenever the program discovers several independent tasks, those tasks can be executed in parallel, each with its own thread of control. When using such a multithreaded programming model, the runtime system must schedule these tasks and dynamically spread them across the machine in order to balance the computation load.

The most ambitious of the multithreaded languages are the implicitly parallel languages. In these language the programmer expresses his algorithm at a high level without any mention of parallelism. As a result, a sophisticated compiler automatically breaks the program up into a fine-grained multithreaded program. In this model every memory reference and every interprocedural communication is a potential nonlocal, long latency operation, which leads to small thread lengths and frequent communication. Executing efficiency under these conditions requires a platform with cheap thread creation and scheduling, as well as high-bandwidth, low-overhead communication infrastructure.

More common are the explicit multithreaded languages, in which the user must explicitly specify what can be done in parallel. These systems provide the programmer with a means to create, synchronize, and schedule threads. In order to reduce the overhead of the program, thread creation, synchronization, and scheduling are typically done by user-level runtime system code, without the involvement of the native operating system. Since the user can cheaply and dynamically spawn off tasks as the arise, these system make it easy for the user to take full advantage of the control parallelism inherent in many programs.

5.3.3 The Message Passing Programming Model

Another common paradigm for writing parallel programs is message passing. Message-passing models present the programmer with one thread of control in each processor, and these processors communicate by sending and receiving messages. This model is a good representation of the actual implementation of current parallel machines.

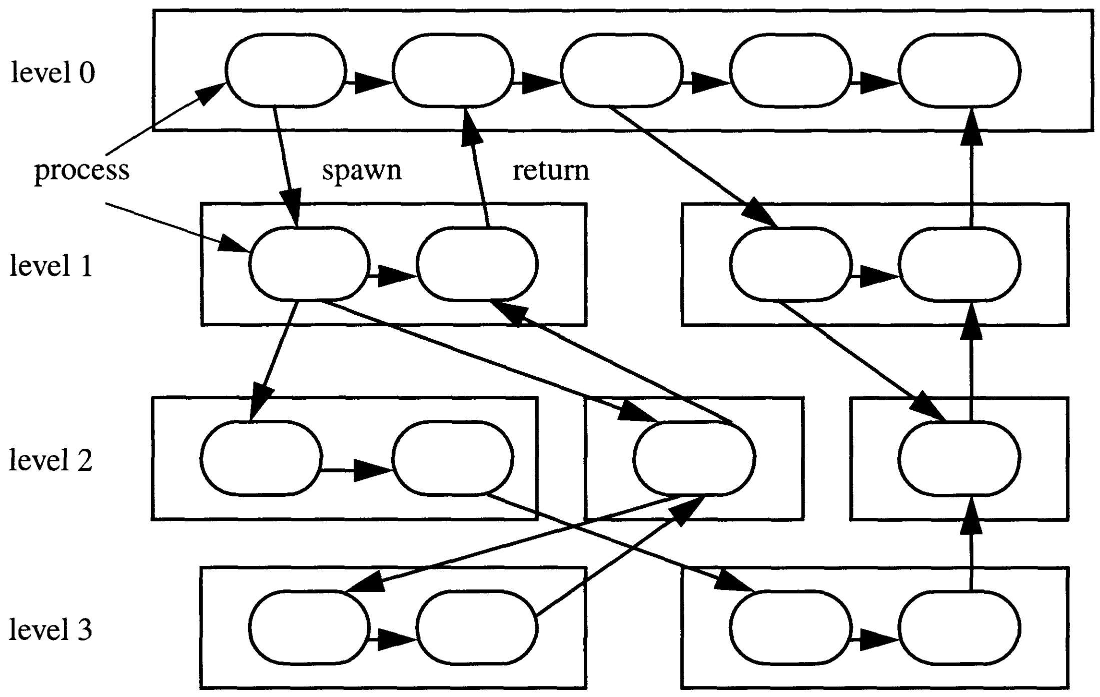


Figure 5-4: Multithreaded Programming Model

Since this model is close to the hardware, a good programmer is able to write efficient codes, just as a good assembly language programmer is able to write assembly code that is more efficient than code written in a high-level language. The major drawback of this model is the same as the drawback of programming in assembly language: writing a large program at such a low level can be overwhelming. The user must answer all the low-level questions himself, namely questions such as how to partition the program's data, when to perform communication, and how to load balance the computation. Not only must the user make all these decisions, but he must then write all the protocols necessary to carry them out. For most nontrivial programs the user spends more time writing protocols than writing the actual application. Basically, there are three strategies for message-passing programming.

The simplest message-passing model is blocking. The sender issues a send request and the receiver issues a receive request. Whichever processor issues its request first blocks and sits idle until the other processor issues its command. At that point, communication begins. Only after communication completes can the processors continue executing. It can be difficult to program well in this model, because inefficiencies occur unless both of the processors involved in a communication issue their requests at the same time. Moreover, this style of programming is prone to deadlock.

To make programming simpler, many systems implement a second type of message passing: "asynchronous" message passing. In this model, when a processor performs a send, the send executes immediately, regardless of whether or not a corresponding receive has been issued, and the sending processor can continue executing. The system uses buffers (often on both the sending and receiving side) to hold the message until it is requested by the receiver. Asynchronous message passing eases the programmer's job, but adds significant overhead to each communication due to the copying and buffering that system invisibly performs.

The third strategy for message passing is called "active messages", which reduces this overhead by providing asynchronous message passing without the automatic buffering. An active message contains a header which points to a handler, which is a piece of user code that specifies what to do with the data in the message. The user can specify many handlers, typically one for each message type. When a message arrives, rather than having a generic system-defined routine handle the message which will typically copy the message into a buffer, the system instead executes this user-defined handler to process the arrived messsage. Active messages eases the task of writing message-passing codes because it allows a programmer to write programs using low-overhead, asynchronous message passing and because the paradigm of having the message itself know how it should be handled turns out to be quite useful in practice.

5.4 The Message Passing Interface Standard

The goal of the Message Passing Interface [14], simply stated, is to develop a widely used standard for writing message-passing programs. As such, the interface should establish a practical, portable, efficient, and flexible standard for message passing.

- Design an application programming interface (not necessarily for compilers or a system implementation library).
- Allow efficient communication: Avoid memory-to-memory copying and allow overlap of computation and communication and offload to communication co-processor, where available.
- Allow for implementations that can be used in a heterogeneous environment.
- Allow convenient C and Fortran bindings for the interface.
- Assume a reliable communication interface: the user need not cope with communication failures. Such failures are dealt with by the underlying communication subsystem.
- Define an interface that can be implemented on many vendor's platforms, with no significant changes in the underlying communication and system software.
- Semantics of the interface should be language independent.
- The interface should be designed to allow for thread-safety.

In our implementation, we adopt the tools provided by Argonne National Laboratories [10] for the Unix-based machines and Mississipi State University [16] for Windows NT-based machines.

5.5 Development of Parallel Contact Detection Algorithm

The hashtable-based spatial reasoning algorithm, as we described in Chapter 2, is very suitable for parallel implementation. In other words, the data dependence between different objects is insignificant. Not only can we implement the parallel algorithm in a straightforward way, but we also achieve a satisfactory performance based on the current hardware configuration.

First of all, as illustrated in Figure [5-5], we can subdivide the entire 3D space into a few separate regions and then apply the contact detection algorithm independently in each region with little boundary overlap. This implies that there is little data dependence in each processor. For instance, if we have 8 quadrants in 3D space, we can create eight processes in a multiprocessor machine, run a process in each processor and exchange the data via shared memory. Since there is little data dependence in each subprocess, the parallel efficiency becomes satisfactory if we can minimize the network communication overhead.

5.6 Implementation Details

In the following section, we explain the implementation details of the parallel version of the hashtable-based spatial reasoning algorithm.

```
MPI_Status status;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&size);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
timestart = MPI_Wtime();
for(k=0;k<timesteps;k++) {
  int * subiylist;
  hashspace_reinit(demsim);
  hashspace_sort1(demsim,1);
  while(iylist[pointer] != -1) {
```

```
pointer++;
      }
      send_count = pointer/size;
      recv_count = pointer/size;
      subiylist = (int *) malloc((send_count+1) * sizeof(int));
      MPI_Scatter(&iylist[0],send_count,MPI_INT,&subiylist[0],recv_count,
                  MPI_INT,0,MPI_COMM_WORLD);
      subiylist[send_count] = -1;
      hashspace_sort2(demsim,subiylist,1);
      size1 = demsim->nps/size;
      start = rank*size1;
      end = (rank+1)*size1-1;
      hashphysics(demsim,start,end);
      free(subiylist);
      MPI_Gather(&subiylist[0],send_count,MPI_CHAR,&iylist[0],recv_count,
                 MPI_CHAR,0, MPI_COMM_WORLD);
    }
    timeend = MPI_Wtime() - tstart;
    MPI_Barrier(MPI_COMM_WORLD);
```

5.7 Performance Analysis

In parallel computation terminology, we define the parallel efficiency E_p in Equation [5.1].

$$E_p = S_p/p, where S_p = T_{serial}/T_p) \tag{5.1}$$

In Equation [5.1], p is the number of processors, T_{serial} is the total time of running the program on a serial machine, and Tp is the total time of running the program on

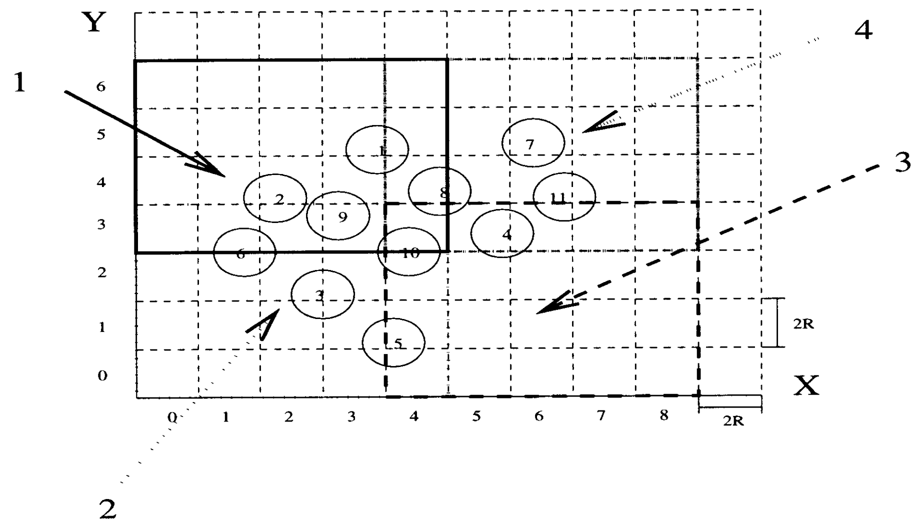


Figure 5-5: Parallel Hashtable-based Parallel Algorithm

p processors. In other words, the parallel efficiency can be defined as the speedup per processor.

The empirical parallel efficiency of the hashtable-based spatial reasoning algorithm is illustrated in Figure [5-6]. Using Equation [5.1], the parallel efficiency of our implementation ranges from 65% to 70%.

5.8 Distributed Discrete Element Simulation

Figure [5-7] shows the system architecture of the distributed simulation environment in MIT IESL. With the availability of a high-speed network switch, we can extend the parallel algorithm to be executed on different machines and platforms through message passing mechanism. From the parallel computation perspectives, this kind of distributed server cluster provides a highly scalable architectural framework for integrating computing resources on the network.

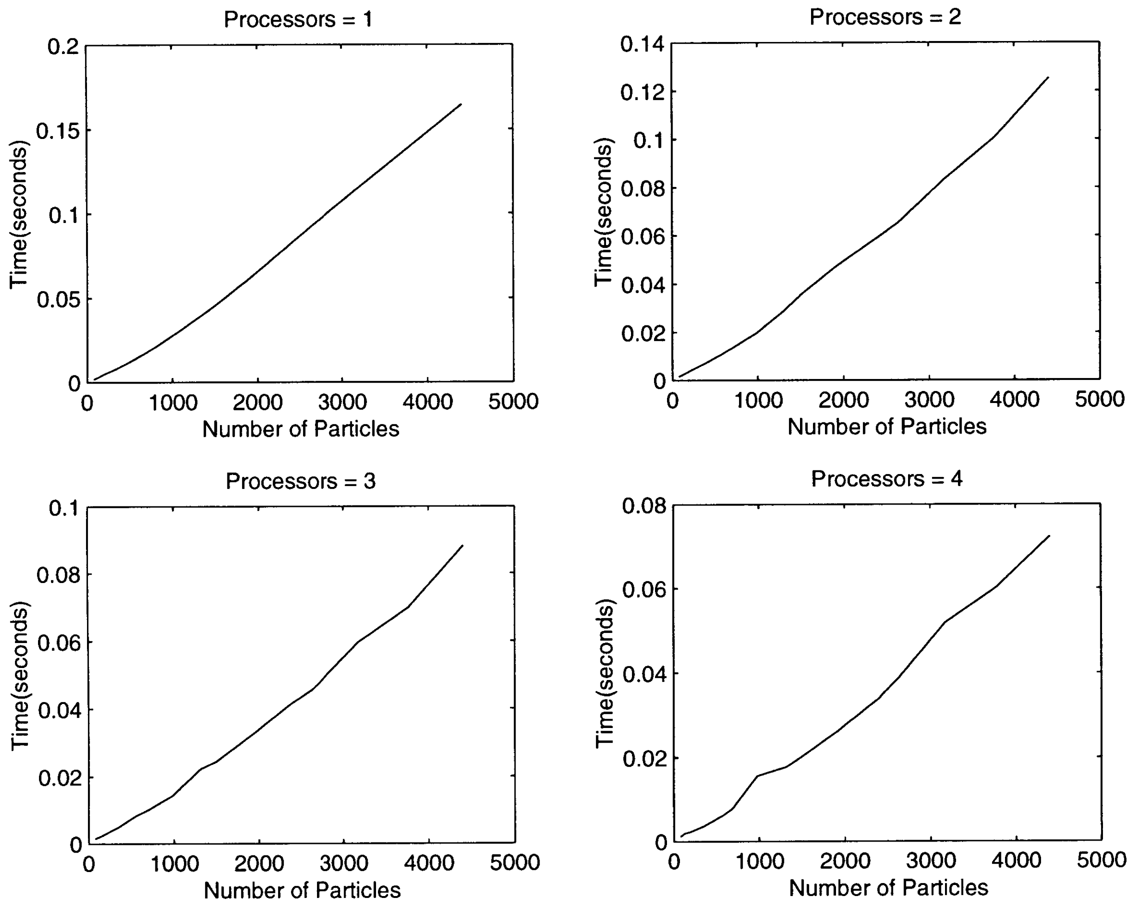


Figure 5-6: Parallel Efficiency of Parallel Algorithm

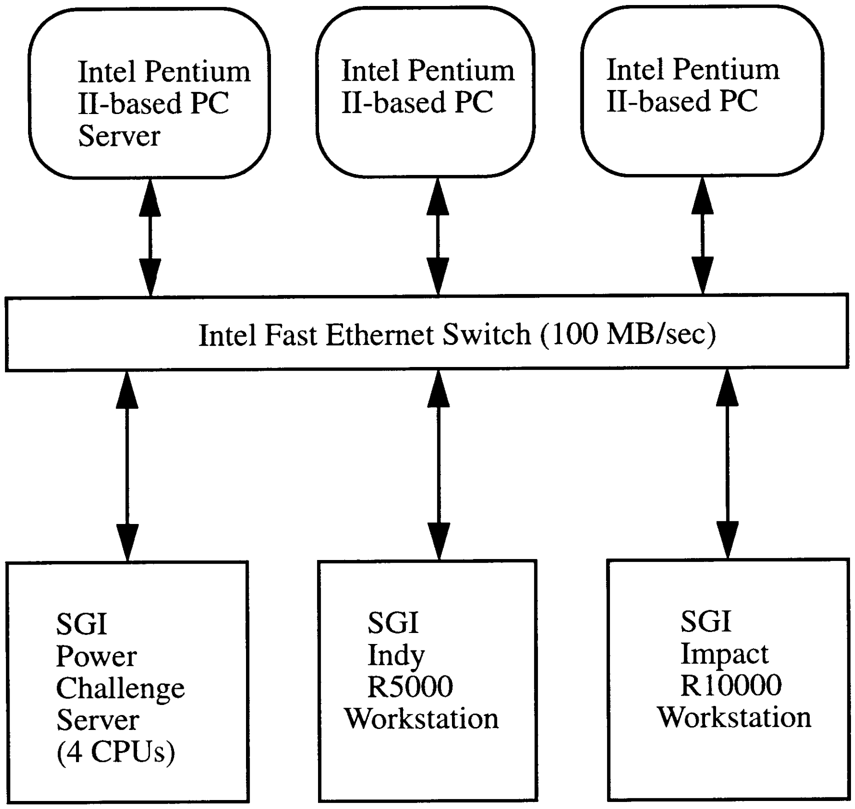

Intel Pentium II-based PC Server
Intel Pentium II-based PC
Intel Pentium II-based PC
Intel Fast Ethernet Switch (100 MB/sec)
SGI Power Challenge Server (4 CPUs)
SGI Indy R5000 Workstation
SGI Impact R10000 Workstation

Figure 5-7: System Architecture of Distributed Modeling Environment

5.9 Summary

This chapter explores the parallel computation possibility in discrete element simulation. We present the development and implementation issues of the parallel hashtable-based spatial reasoning algorithm. Also, we show the system framework of incorporating the concept of distributed computing into the DEM simulation environment. The overall parallel efficiency in our implementation is about 65-70 percent, which is generally considered as a satisfactory result for in parallel computation.

Chapter 6

Examples and Applications

This chapter describes the following sample applications of the simulation system.

- Dynamic Impact Simulation
- Contact Damping
- Sandglass
- Collapse of Embankment
- Stress Wave Propagation
- Particles Roll on the Slope
- Fracture of a Fixed-end Beam
- DFR Packing

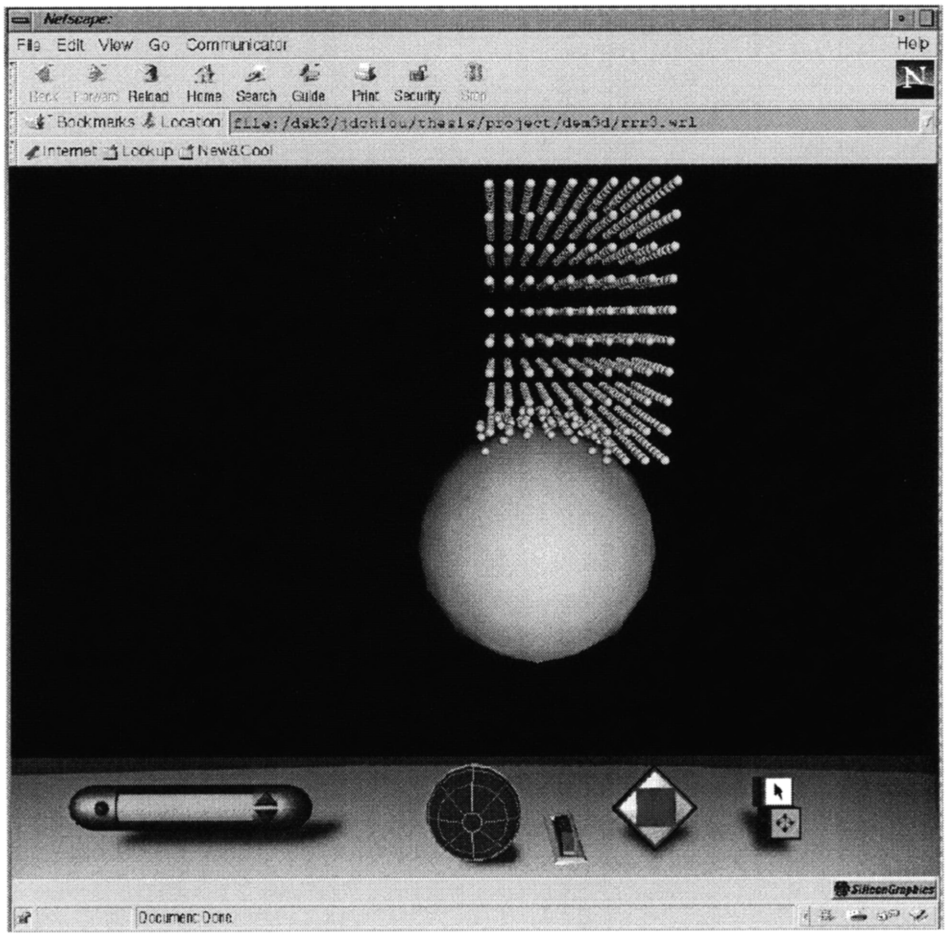

Netscape:
File Edit View Go Communicator Help
Back Forward Reload Home Search Guide Print Security Stop
Bookmarks Location file:/dsk3/jdchiou/thesis/project/dem3d/rrr3.wrl
Internet Lookup New&Cool
SiliconGraphics
Document: Done

Figure 6-1: Dynamic Impact Simulation Snapshot 1

6.1 Dynamic Impact Simulation

The first example in this chapter simulates the dynamic impact problem as shown in Figure [6-1],[6-2], and [6-3]. We let 2,000 particles fall under gravity onto a fixed larger sphere. This simulation can be run on an SGI MIPS/R10000 workstation in 2 minutes for 3,000 time steps and then rendered using the VRML within a World Wide Web browser. We have been testing the number of particles up to 250,000 on the SGI Power Challenge with 4 MIPS/R10000 processors. It takes approximately 30 seconds for each time step. This implies that we can simulate relatively large-scale simulation problems in less than a few hours of CPU time.

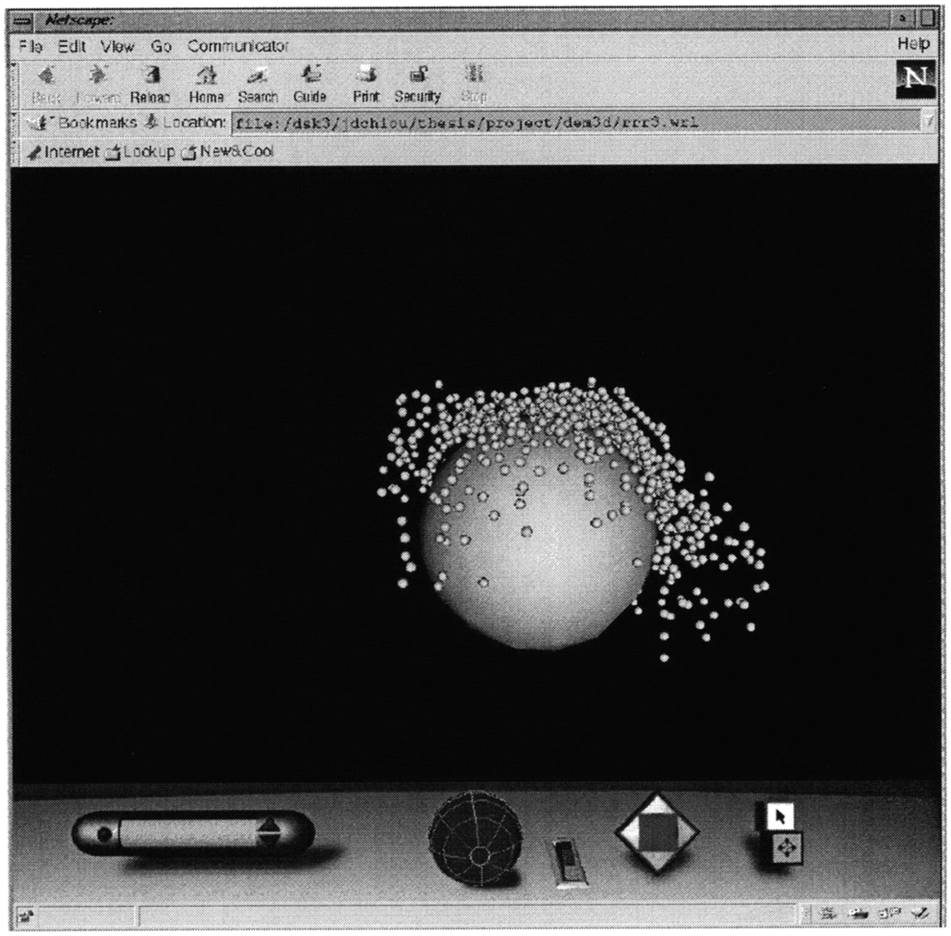

Netscape:
File Edit View Go Communicator
Help
Back Forward Reload Home Search Guide Print Security Stop
Bookmarks Location: file:/dsk3/jdchiou/thesis/project/dem3d/rrr3.wrl
Internet Lookup New&Cool

Figure 6-2: Dynamic Impact Simulation Snapshot 2

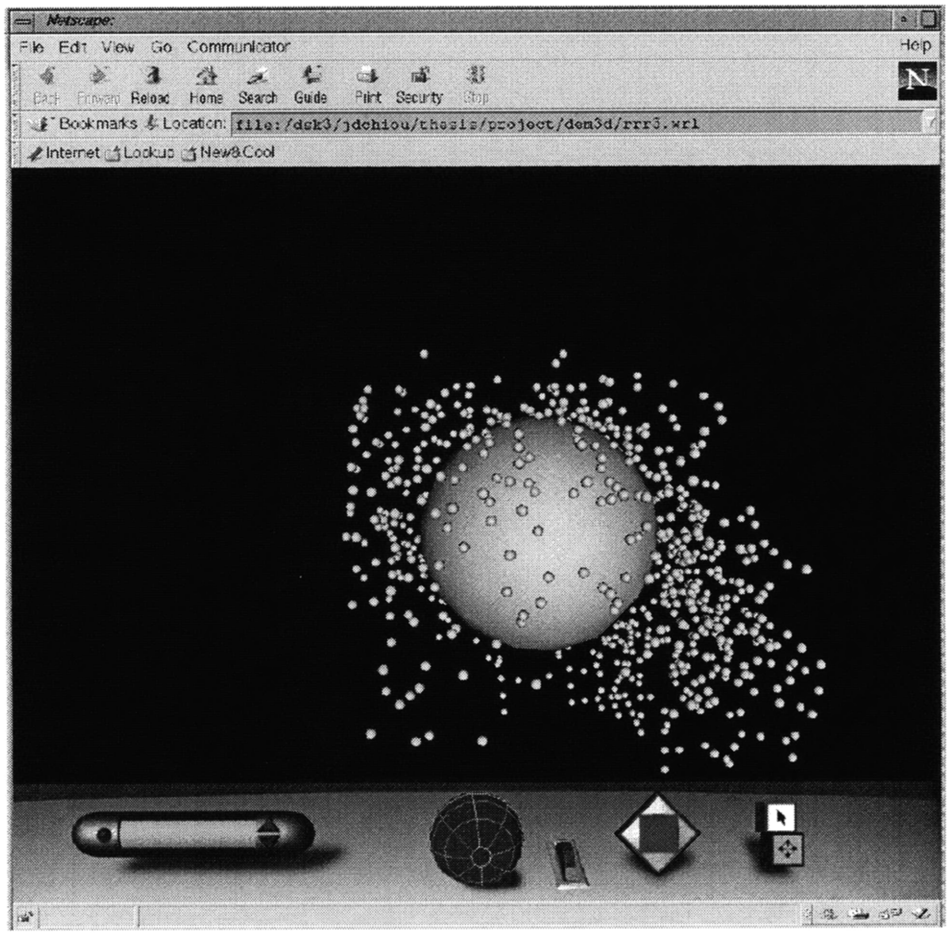

Netscape:
File Edit View Go Communicator Help
Back Forward Reload Home Search Guide Print Security Stop
Bookmarks Location: file:/dsk3/jdchiou/thesis/project/dem3d/rrr3.wrl
Internet Lookup New&Cool

Figure 6-3: Dynamic Impact Simulation Snapshot 3

6.2 Contact Damping Simulation

The second example simulates the phenomenon of contact damping. We assume that damping only occurs when two objects make contact with each other. Interestingly, we see the results reflect the theoretical derivation of damping theory correctly. Figure [6-4] shows that all the particles will be pushed to the left-hand side and reach the equilibrium if we put two fixed objects at both ends and initiate the simulation by giving the rightmost unconstrained particle an initial speed to the left direction. Similarly, in Figure [6-5], we see all these particles are squeezed to the right-hand size. Figure [6-6] shows that if we give two objects initial speeds in the left-hand and right-hand directions respectively, half the particles stay at each side. This is an interesting phenomenon that we observed during the course of simulation.

6.3 Sandglass Simulation

The sandglass simulation is a classical DEM application. We configured 3,000 particles as shown in Figure [6-7]. As the program runs, the particles flow freely through the valve, and then densely consolidate at the lower region of the container. The simulation was run for 8,000 iterations taking approximately 4 CPU hours on an SGI R10000-based workstation.

6.4 Collapse of Embankment Simulation

In this example, we simulate the collapse of an embankment. 5,000 particles are used to construct a wall with a fixed boundary at the left side and the bottom. As we see in the Figure [6-8], the failure of the free boundary of the embankment occurs on the right hand side. The simulation was run for 5,000 time steps taking approximately 2.5 CPU hours on an SGI R10000-based workstation.

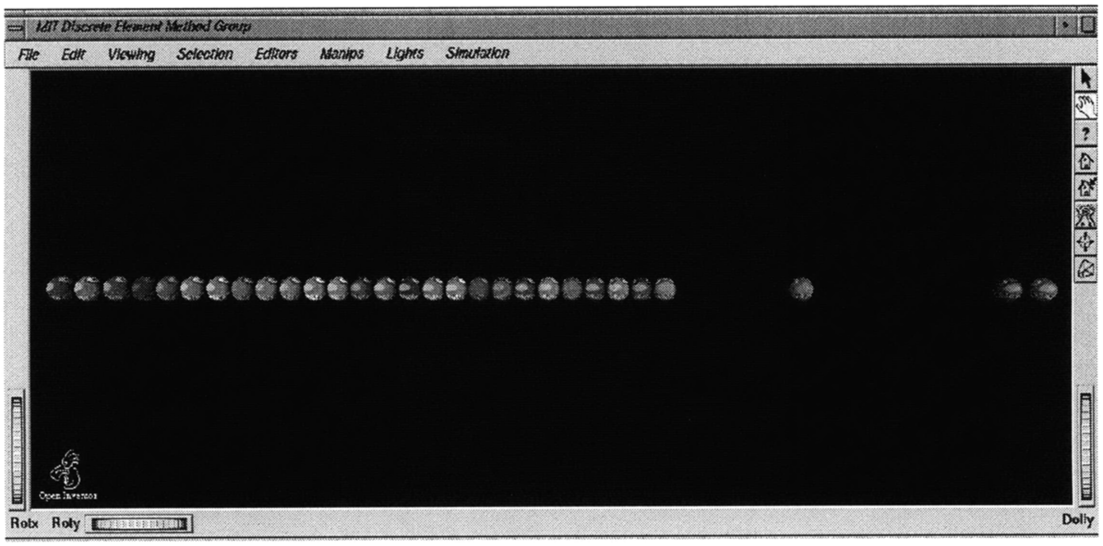


Figure 6-4: Contact Damping Simulation Example 1

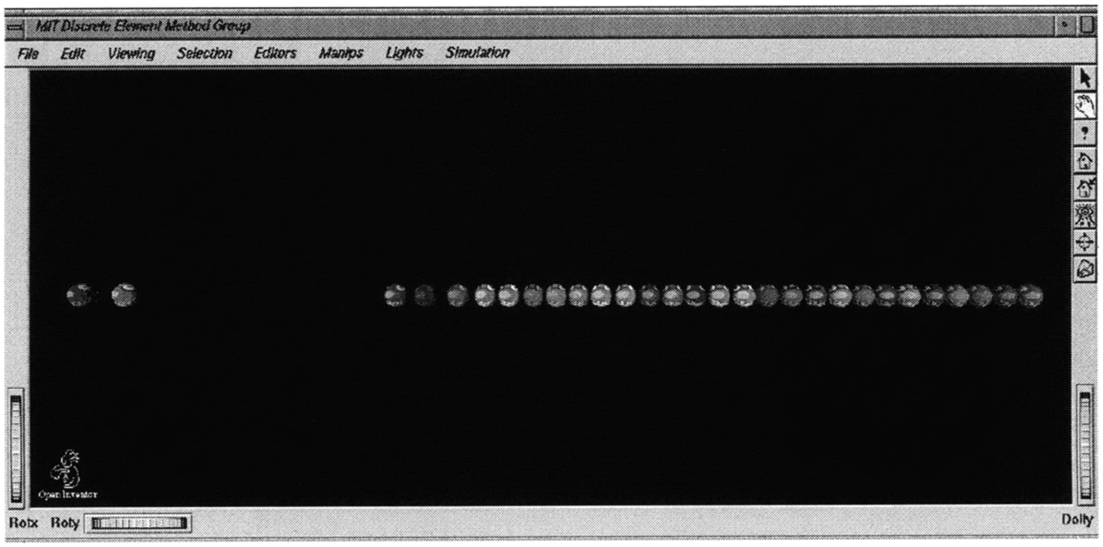

MIT Discrete Element Method Group
File Edit Viewing Selection Editors Manips Lights Simulation
Rotx Roty
Dolly

Figure 6-5: Contact Damping Simulation Example 2

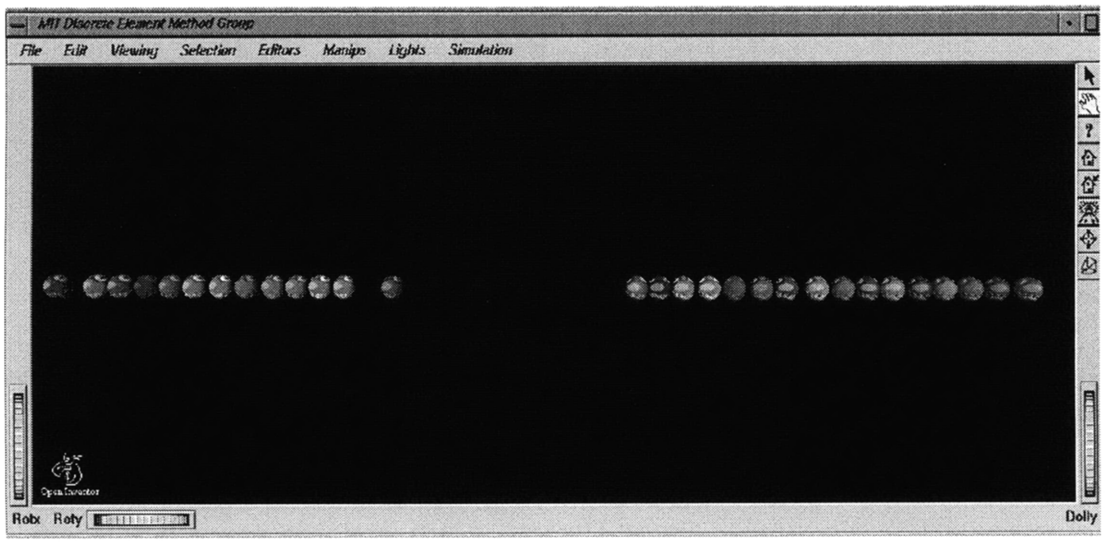


Figure 6-6: Contact Damping Simulation Example 3

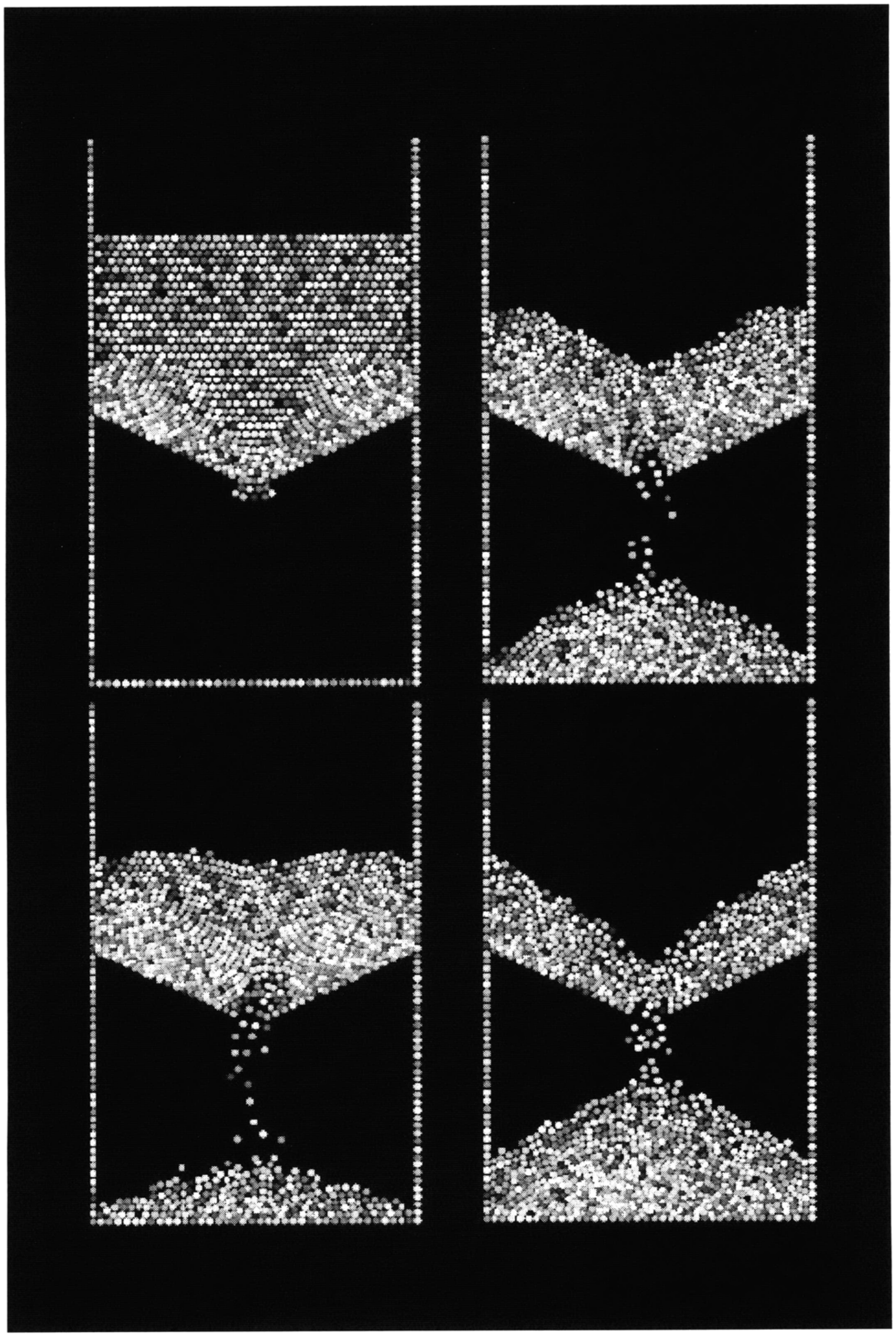
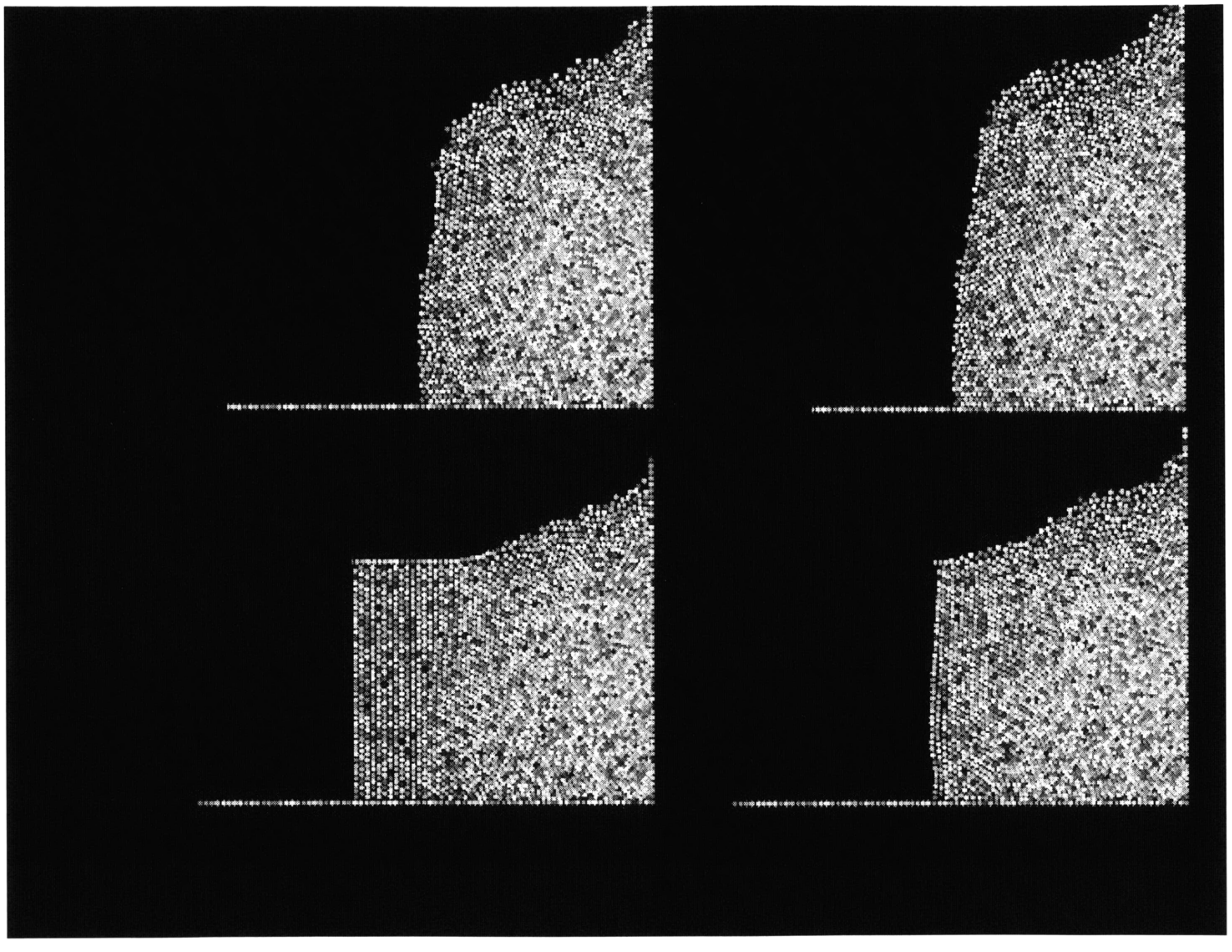


Figure 6-7: Sandglass Simulation



6.5 Wave Propagation Simulation

In this example, 10,000 particles are surrounded by fixed boundaries on the left, right, and bottom sides. We start the simulation by dropping a particle from the top with a particular velocity. The particle moves downward and hits the particle bed. The collision generated the stress wave that propagate through the entire particle bed. The wave reaches the fixed boundary and then reflected back. We can clearly see the reflected waves interact with each other and then generate more complicated wave phenomenon, as shown in Figure [6-9]. The simulation was run for 5,000 iterations taking approximately 6 CPU hours on an SGI R10000-based workstation.

6.6 Particles on the Slope

In this example, 10,000 particles are dropped from the top to hit the sloped channel box at the bottom. Particles then flow through the slope and accumulate at the bottom of the box due to the application of gravity, as we see in Figure [6-10]. The simulation was run for 8,000 iterations taking approximately 10 CPU hours on an SGI R10000-based workstation.

6.7 Fracture of a Fixed-end Beam

In this example, we demonstrate the capability of binding particles with a particular cohesive force. 5,000 particles are bonded together and both ends of the beam are fixed. Three particles in the middle of the beam move downward at a fixed velocity to break the bonds. We observe the fracture behavior of the beam from the animation shown in Figure [6-11]. The simulation was run for 3,000 iterations taking approximately 6 CPU hours on an SGI R10000-based workstation.

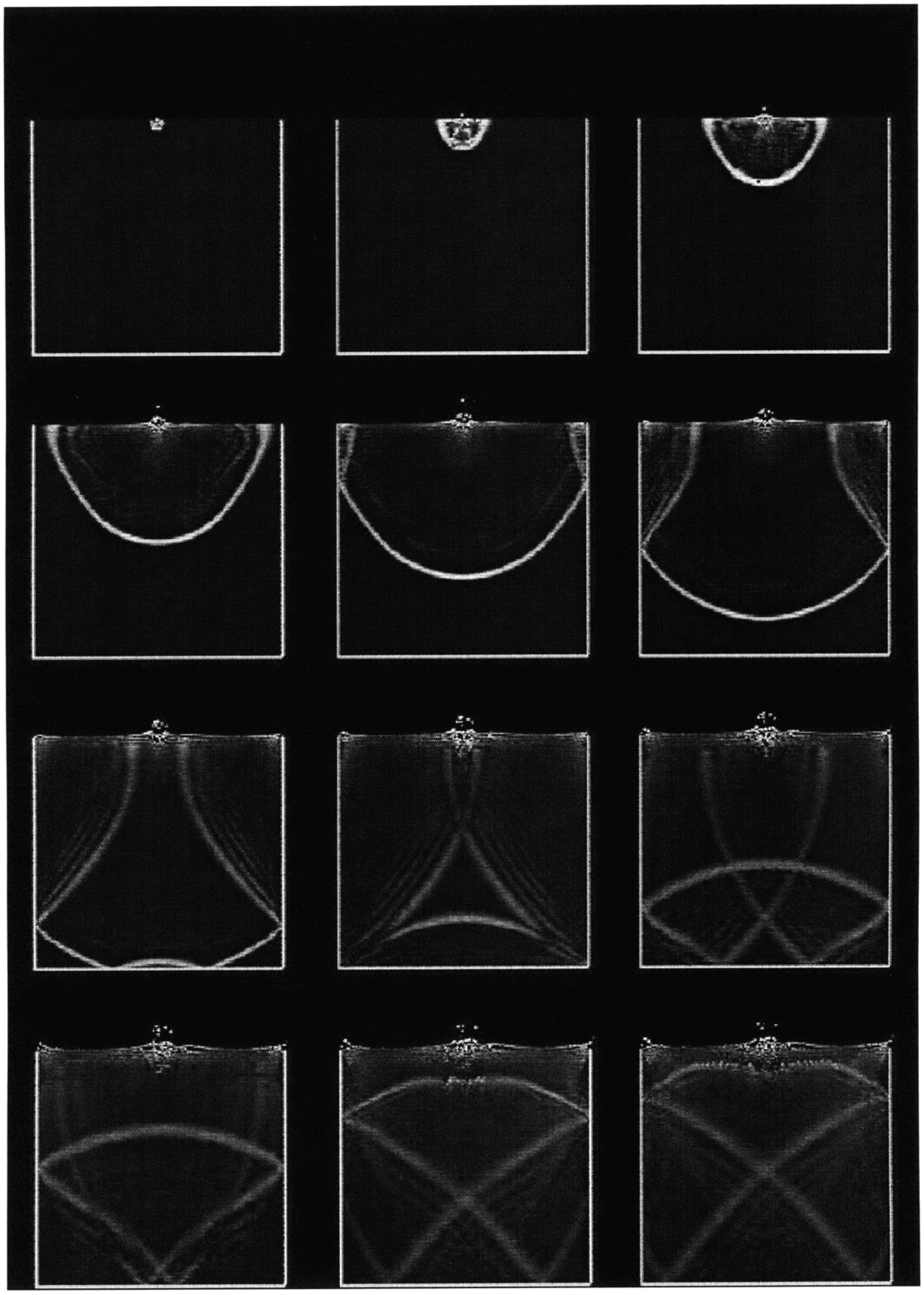


Figure 6-9: Wave Propagation Simulation

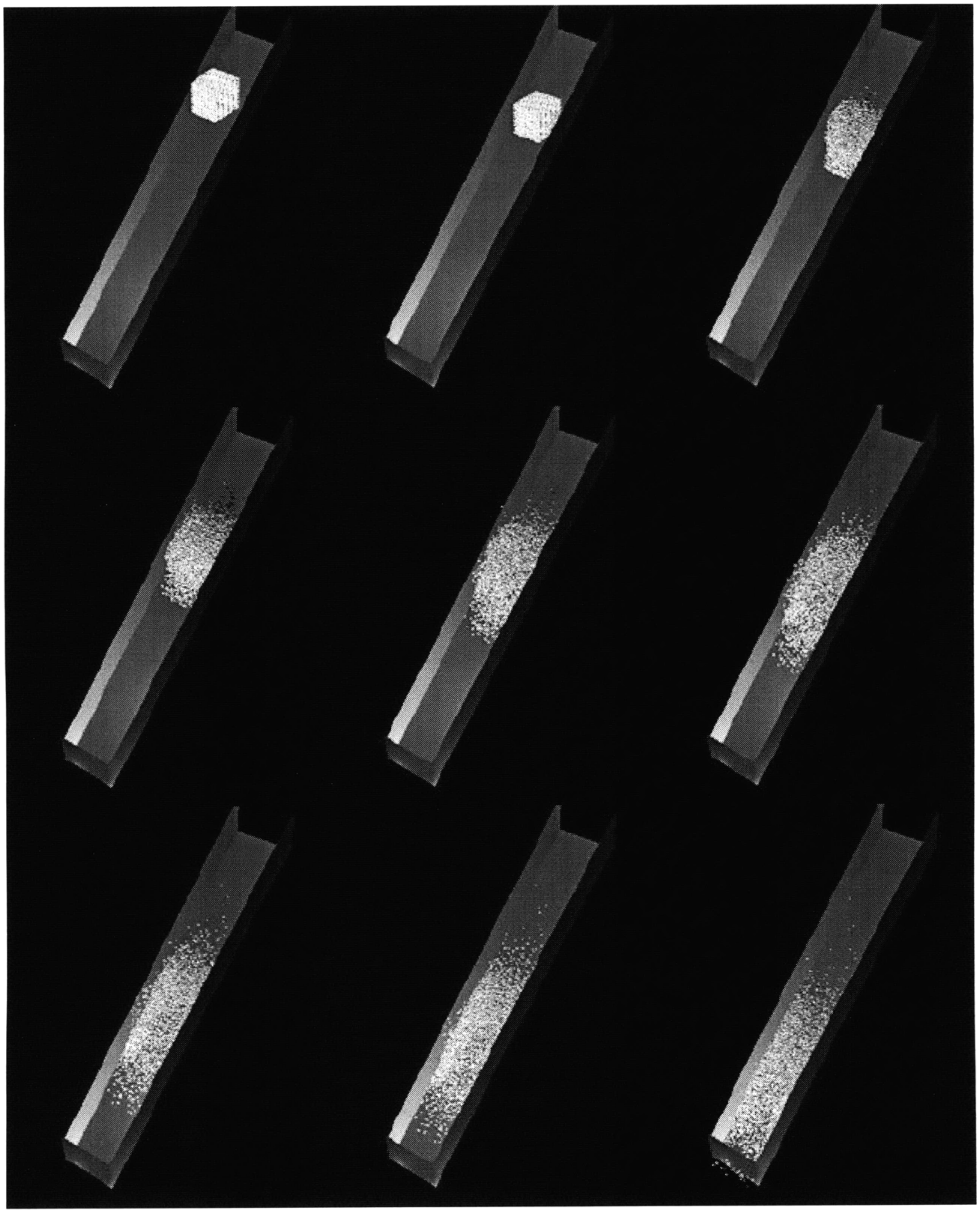


Figure 6-10: Particles Rolling on the Slope

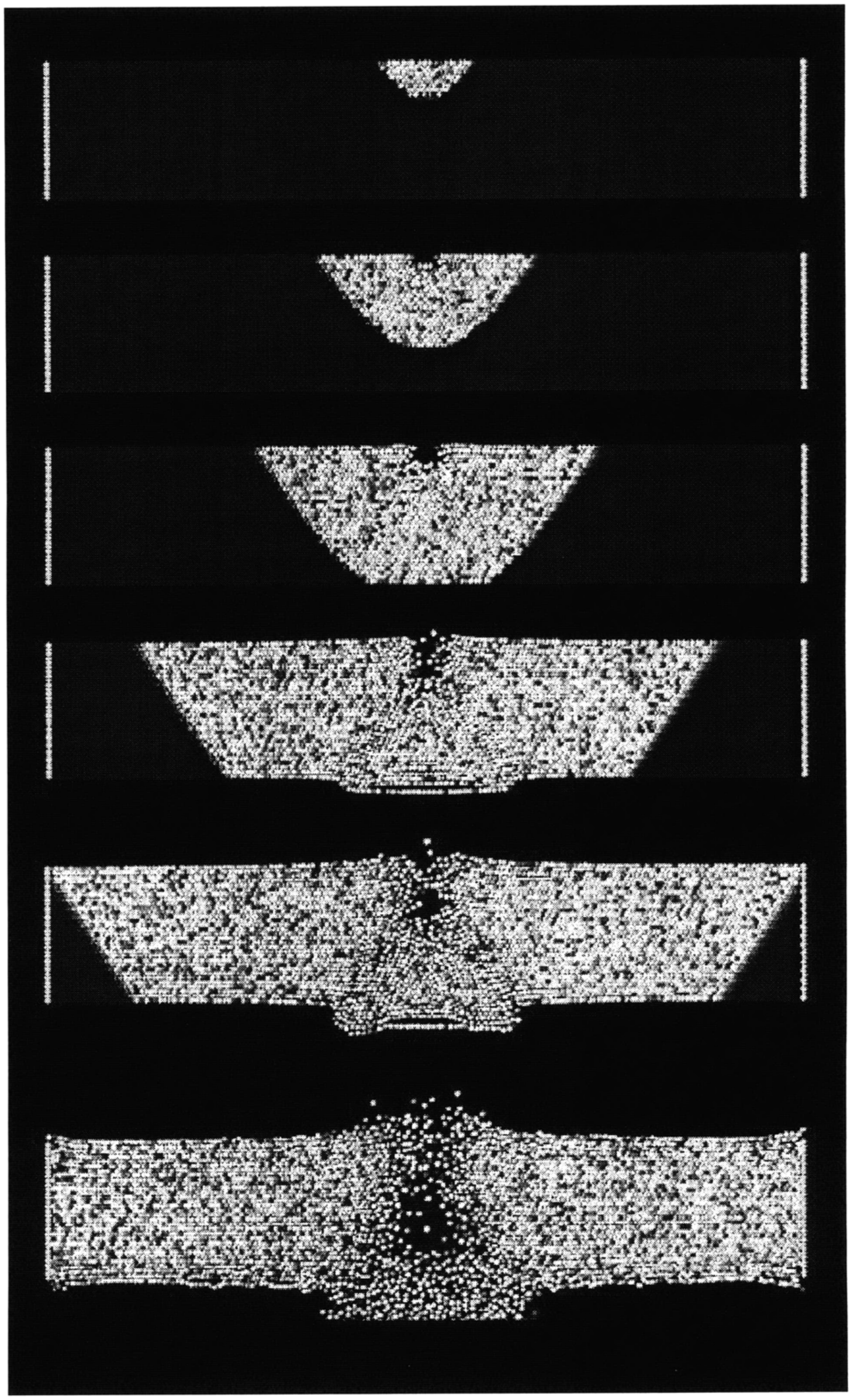


Figure 6-11: Fracture of a Fixed-end Beam

6.8 DFR Packing

In this example, we show the packing of 3,000 arbitrarily shaped of 3D DFR particles with different sizes. Particles are dropped from the top with the application of gravity and then densely packed in the cubic box, as illustrated in Figure [6-12]. The simulation was run for 5,000 iterations taking approximately 8 CPU hours on an SGI R10000-based workstation.

6.9 Summary

In this chapter, we present eight examples to show the performance and capabilities of the simulation system. As a matter of fact, there are many other applications we can possibly simulate using the system, as long as we can specify the boundary condition of the system and define the geometry of the particles correctly. Compared with the performance results listed in [37], it is apparent that the algorithms we described in Chapter 2 have significantly improved the overall performance of the simulation system, with better visualization capability in 3D.

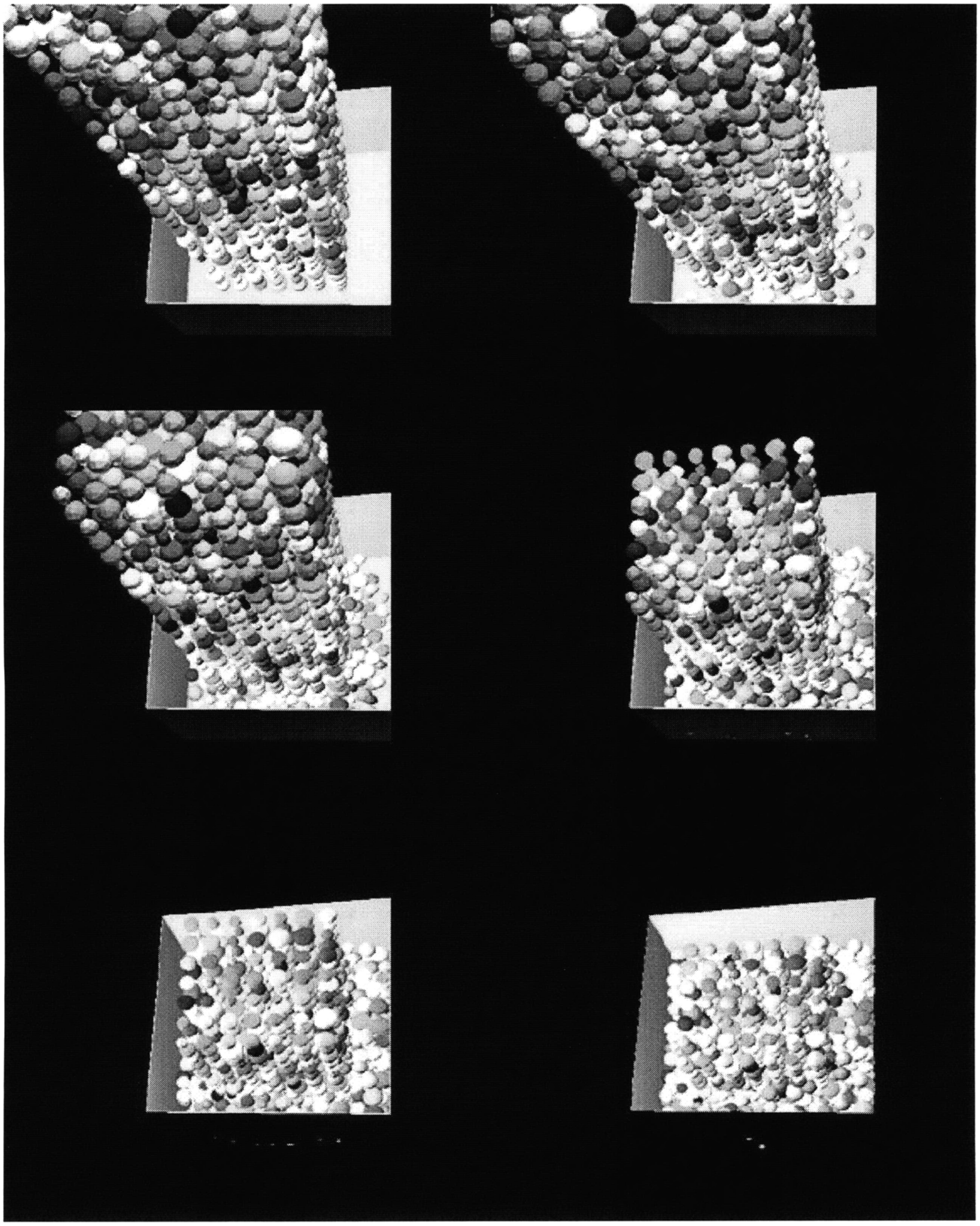


Figure 6-12: DFR Packing Simulation

Chapter 7

Conclusions and Future Work

7.1 Summary

In this thesis, we present a distributed DEM simulation system for multibody physics. The major research task here is to reduce the computational cost of the DEM simulation process. We implement two new algorithms aimed at accelerating the contact detection procedure, which is generally considered the most time-consuming process in the overall DEM simulation pipeline. Two 3D visualization tools, a language designed for simplifying the input, and distributed and parallel computation are also described in detail.

7.2 Contributions

In this thesis the following contributions have been presented:

1. A high-performance hashtable-based contact detection algorithm for 3D objects, as described in Chapter 2.

2. The implementation and test of the DFR object representation scheme, as described in Chapter 3.

3. An object-oriented software system with a versatile 3D visualization module and a simulation description language, as described in Chapter 4.

4. Distributed and parallel implementation, as described in Chapter 5.

The contact detection problem has long been an active research topic in physically-based modeling, such as robotics [28] and physically-based simulation [5]. However, the use of polygonal reprensentations is generally considered inappropriate for the DEM analysis if we take the performance requirement into consideration. The DFR scheme addresses this problem and provides a significant improvement in the treatment of polygonal surface representations, as described in Chapter 3. More importantly, the DFR introduces the concept of multiresolution representation. Users can always find a good tradeoff between the computing efficiency and accuracy depending on the application requirements.

Furthermore, through the use of parallel and distributed computer architectures, and development of parallel algorithms, the scale of DEM simulation has been dramatically increased, as we described in Chapter 5.

The ability to perform numerical experiments more rapidly has a twofold significance in terms of understanding the dynamics of granular materials. Firstly, it enables us to hypothesize and model the large-scale problem. Secondly, it facilitates the simulation of systems containing a large number of objects that were previously not feasible because of the CPU time it would require. This means the scaling effects of the number of objects in the system can be evaluated at higher resolutions.

7.3 Related Applications

Contact detection has been applied in a variety of different forms across a broad spectrum of computer modeling applications. Some of the existing and potential applications are listed as follows:

- Geometric Modeling - CAD interference tests.
- Manufacturing - Packaging optimization.
- Physically-based Simulation - Real-time simulated environments.
- Virtual Reality/Environment.
- Behavior of Granular Materials.
- Engineering Design - Rapid Prototyping.

7.4 Future Work

For future development of the **DEM3D** system, we propose the following subjects:

Deformability of Particles In the **DEM3D** system, we assume all particles are rigid-body. Nevertheless, in many practical engineering problems, this assumption may not reflect the real situation because particles actually deform under stress [40]. As a result, it would be beneficial to take the deformation into consideration for the DEM simulation in future research.

Visualization In 3D DEM simulation, it is important to provide a good visualization tool so that the user can visually investigate the detailed interaction between particles in a microscopic level. However, the data generated by the simulation system could be overwhelming when we deal with millions of particles. We need to build an efficient pipeline to concurrently digest the data when the simulation is running or develop a highly compact data representation scheme to reduce the amount of data for postprocessing.

Fluid Interaction In many practical engineering applications, particles are always surrounded by fluid. We realize that the interaction between the particle and fluid can not be ignored in many cases. More research work should be conducted to investigate this problem [26].

Appendix A

DEM3D Input Command Reference

A.1 Introduction

In this appendix, we list the commands of the simulation description language described in Chapter 4. The language is designed based on the *Tcl* script language [38].

A.2 Commands

- **ADD**
  - SPHERE x,y,z,x_vol,y_vol,z_vol,x_rot,y_rot,z_rot, CONSTRAINT
  - PLANE x,y,z,x_vol,y_vol,z_vol,x_rot,y_rot,z_rot,x_size,y_size, CONSTRAINT
  - DFR x,y,z,x_vol,y_vol,z_vol,x_rot,y_rot,z_rot,CONSTRAINT
  - BONDING particle1 particle2

    where

    * x,y,z is the center of the particle

  * x_vol,y_vol,z_vol is the velocity in x,y,z-direction.
  * x_rot,y_rot,z_rot is the angular velocity in x,y,z-direction.
  * CONSTRAINT can be UNCONSTRAINED, FIXED, or

- **MODIFY**
  - STARTX // starting point in X direction
  - STARTY // starting point in Y direction
  - STARTZ // starting point in Z direction
  - RESOLX // resolution in X direction
  - RESOLY // resolution in Y direction
  - RESOLZ // resolution in Z direction
  - DELTAT // *dt*
  - TIMESTEP // number of timesteps
  - PENALTY // stiffness of the spring
  - COHESION // cohesive force
  - FRICTION // friction coefficient
  - DAMPING // damping coefficient
  - RHO // material property
  - FIELDS // fields in x,y,z-direction, respectively
- **START** - start to run the simulation

Bibliography

[1] Farid F. Abraham. Portraits of a crack: Rapid fracture mechanics using parallel molecular dynamics. *IEEE Computational Science and Engineering*, 2(4), April-June 1997.

[2] Selim G. Akl. *Parallel Computation : Models and Methods.* Prentice Hall Press, Englewood Cliffs, New Jersey, USA, 1997.

[3] Farid M.L. Amirouche. *Computational Methods in Multibody Dynamics.* Prentice Hall, Prentice Hall, New York, 1992.

[4] David Baraff. Curved surfaces and coherence for non-penetrating rigid body simulation. In *Proceedings of ACM SIGGRAPH*, volume 24, 1990.

[5] Ronen Barzel. *Physically Based Modeling For Computer Graphics - A Structured Approach.* Academic Press, Inc., Harcourt Brace Jovanovich, San Diego, California, USA, 1992.

[6] Klaus-Jurgen Bathe. *Finite Element Procedures.* Prentice Hall Inc., Englewood Cliffs, New Jersey, USA, 1996.

[7] David M. Beazley and Peter S. Lomdahl. Large-scale molecular dynamics on mpps. *SIAM News*, 28(2), 1995.

[8] William J. Bouma and George Vanecek Jr. Collision detection and analysis in a physically based simulation. In *2nd Eurographics Workshop on Animation and Simulation*, Vienna, 1991.

[9] Raymond M. Brach. *Mechanical Impact Dynamics - Rigid Body Collisions.* John Wiley and Sons, John Wiley and Sons, New York, 1991.

[10] Patrick Bridges, Nathan Doss, William Gropp, Edward Karrels, Ewing Lusk, and Anthony Skellum. *Users' Guide To MPICH, A Portable Implementation of MPI.* Argonne National Laboratory, 1995. Available via anonymous `ftp` at: `info.mcs.anl.gov/pub/mpi/mpich.tar.Z`.

[11] L. Collatz. *The Numerical Treatment of Differential Equations.* Springer Verlag, 1966.

[12] VRML Consortium. *VRML97 - Virtual Reality Modeling Language Specification.* VRML Consortium, USA, 1997.

[13] John N. Tsitsiklis Dimitri P. Bertsekas. *Parallel and Distributed Computation - Numerical Methods.* Prentice Hall, Prentice Hall, New York, 1989.

[14] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 1.0*, 1994. Version as of June 1995 available via anonymous `ftp` at: `ftp.mcs.anl.gov/pub/mpi/mpi-1.jun95`.

[15] Leslie F. Greengard. *The Rapid Evaluation of Potential Fields in Particle Systems, ACM Distinguished Dissertation 1987.* MIT Press, 1988. ACM Distinguished Dissertation Series 1987.

[16] Mississipi State University Group. Mpi implementation for windows nt. *Mississipi State University Engineering Research Center Seminar*, page http://www.erc.msstate.edu/labs/hpcl/index.html, March 1998.

[17] MIT Supercomputing Technology Group. Cilk-5.1 reference manual. *MIT Laboratory for Computer Science*, page http://theory.lcs.mit.edu/ cilk, March 1998.

[18] James K. Hahn. Realistic animation of rigid bodies. *ACM Computer Graphics*, 22(4):299,306, 1988.

[19] J.A. Hanson. Hyperquadrics: Smoothly deformable shapes with convex polyhedral bounds. *Computer Vision, Graphics and Image Processing*, 44:191,210, 1988.

[20] Yu Hu and S. Lennart Johnsson. A data parallel implementation of hierarchical n-body methods. Technical Report TR-26-95, Parallel Computing Research Group, Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts, 1994. Submitted to Journal of Supercomputing Applications.

[21] Yu Hu and S. Lennart Johnsson. Implementing o(n) n-body algorithms efficiently in data parallel languages (high performance fortran). Technical Report TR-24-95, Parallel Computing Research Group, Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts, 1994. Submitted to Journal of Scientific Programming.

[22] Gen hua Shi. *Block System Modeling by Discontinuous Deformation Analysis.* Computational Mechanics publications, Southhampton United Kingdom and Boston USA, 1993.

[23] Inc. Intera Technologies. *CICE User's Manual.* Intera Technologies, Inc., Lakewood, Colorado, 1986.

[24] Christopher F. Joerg. *The Cilk System for Parallel Multithreaded Computing.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1996.

[25] Dieter Lasser Josef Hoschek. *Fundamentals of Computer Aided Geometric Design.* AK Peters, Wellesley, Massachusetts, 1993.

[26] Justin T. Klosek. The integration of fluid dynamics with a discrete-element modelling system : Algorithms, implementation, and applications. Master's thesis, IESL, Massachusetts Institute of Technology, 1997.

[27] Erwin Kreyszig. *Advanced Engineering Mathematics.* John Wiley and Sons, 5th edition, 1983.

[28] M. Lin and S. Gottschalk. Collision detection between geometric models: A survey. *Proceedings of IMA Conference on Mathematics of Surfaces*, 1998.

[29] Ming C. Lin. *Efficient Collision Detection for Animation and Robotics.* PhD thesis, Department of Electrical Engineering and Computer Science , University of California at Berkeley, 1993.

[30] W.E. Lorensen and H.E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics, Proceedings of SIGGRAPH*, 21(4):163,169, July 1987.

[31] D. Manocha M. Ponamgi and M. Lin. Incremental algorithms for collision detection between solid models. *IEEE Transactions on Visualization and Computer Graphics*, 1997.

[32] M. Mantyla. *An Introduction to Solid Modeling.* Computer Science Press, USA, 1988.

[33] Brian Mirtich. Fast and accurate computation of polyhedral mass properties. *Journal of Graphics Tools*, 1(2), 1996.

[34] Brian V. Mirtich. *Impulse-based Dynamic Simulation of Rigid Body Systems.* PhD thesis, Department of Electrical Engineering and Computer Science , University of California at Berkeley, 1996.

[35] Matthew Moore and Jane Wilhelms. Collision detection and response for computer animation. *ACM Computer Graphics*, 22(4):289,297, 1988.

[36] A. Munjiza and K.R.F. Andrews. Nbs contact detection algorithm. *Department of Engineering, QMW, University of London, London, U.K.*, 1998.

[37] Ruaidhrí O'Connor. *A Distributed Discrete Element Modeling Environment - Algorithms, Implementation, and Applications.* PhD thesis, Department of Civil & Environmental Engineering, Massachusetts Institute of Technology, 1996.

[38] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison Wesley, Reading, Mass, USA, 1994.

[39] Alex P. Pentland. Computational complexity versus simulated environments. In *ACM SIGGRAPH Computer Graphics, Volume 24 Number 2*, pages 185,192, March 1991.

[40] Nabha Rege. *Computational Modeling of Granular Materials.* PhD thesis, Department of Civil & Environmental Engineering, Massachusetts Institute of Technology, 1996.

[41] Craig W. Reynolds. Flocks, herds, and schools, a distributed behavioral model. *ACM Computer Graphics*, 16(4):25,34, 1987.

[42] David Rogers. *Procedural Elements for Computer Graphics.* McGraw Hill, 1985.

[43] John K. Salmon, Michael S. Warren, and Gregoire S. Winckelmans. Fast parallel tree codes for gravitational and fluid dynamical n-body problems. *International Journal of Supercomputing Applications*, 8(2), 1993.

[44] Hanan Samet. *The Design and Analysis of Spatial Data Structures.* Addison Wesley, 1990.

[45] Peter Schroeder and David Zeltzer. Path planning inside bolio. Course notes on Synthetic Actors: The Impact of Artificial Intelligence and Robotics on Animation, ACM Computer Graphics SIGGRAPH, 1988.

[46] Colin P. Stark. An invasion percolation model of drainage network evolution. *Nature*, 352:423,425, August 1991.

[47] Jeffrey W. Swegle. Search algorithm. Technical report, Sandia National Laboratories, Solid and Structural Mechanics Dept., Albequerque, New Mexico, 87185, 1993. External Distribution Memo.

[48] Charles E. Leiserson Thomas H. Cormen and Ronald L. Rivest. *Introduction To Algorithms*. MIT Press, McGraw-Hill, 1990.

[49] John M. Ting, M. Khwaja, L. Meachum, and J. Rowell. An ellipse-based discrete element model for granular materials. *International Journal for Numerical and Anlytical Methods in Geomechanics*, 17:603,623, 1993.

[50] Michael S. Warren and John K. Salmon. A parallel hashed oct-tree n-body algorithm. In *Proceedings of Supercomputing '93*, Los Alamitos, IEEE Comp. Soc., 1993.

[51] Josie Wernecke. *The Inventor Mentor - Programming Object-Oriented 3D Graphics with Open Inventor*. Addision-Welsley, Reading, Massachusetts, USA, 1994.

[52] John R. Williams. Contact analysis of large numbers of interacting bodies using discrete modal methods for simulating material failure on the microscopic scale. *International Journal of Computer Aided Engineering - Engineering Computations*, 5(3), 1988.

[53] John R. Williams, Grant Hocking, and Graham G.W. Mustoe. The theoretical basis of the discrete element method. In *Proceedings of the 1985 Conference*

on Numerical Methods in Engineering, Theory and Application, pages 897–906, 1985.

[54] John R. Williams and Alex Pentland. Superquadric object representation for dynamics of multi-body structures. In *Proceedings of ASCE Structures*, San Francisco, CA, 1989.

[55] John R. Williams and Alex Pentland. Superquadrics and modal dynamics for discrete elements in interactive design. *International Journal of Computer Aided Engineering - Engineering Computations*, 9(2), 1992.