A Distributed Discrete Element Modeling Environment - Algorithms, Implementation and Applications.

by

Ruaidhrí M. O'Connor

BA, BAI Civil Engineering
Trinity College Dublin, 1989

SM, Civil Engineering
Massachusetts Institute of Technology, 1992

SUBMITTED TO THE DEPARTMENT OF
CIVIL AND ENVIRONMENTAL ENGINEERING
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE
DEGREE OF
DOCTOR OF SCIENCE IN CIVIL AND ENVIRONMENTAL ENGINEERING

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January 1996

© 1996 Massachusetts Institute of Technology

Signature of Author: ______________________
Department of Civil and Environmental Engineering
October 2, 1995

Certified by: ______________________
John R. Williams Associate Professor,
Department of Civil and Environmental Engineering
Thesis Supervisor

Accepted by: ______________________
Joseph M. Sussman
Chairman, Departmental Committee on Graduate Studies

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

FEB 26 1996

ARCHIVES

LIBRARIES

A Distributed Discrete Element Modeling Environment - Algorithms, Implementation and Applications.

by

Ruaidhrí M. O'Connor

Submitted to the Department of Civil and Environmental Engineering
on October 2nd, 1995 in partial fulfillment of the
requirements for the Degree of
Doctor of Science in Civil and Environmental Engineering

ABSTRACT

Discrete Element Methods (DEM) are a set of numerical techniques specifically developed to model the behavior of discontinuous systems such as granular media. Each grain of material is modeled as a discrete object with a geometric representation of its surface topology and a description of its physical state (position, orientation, body forces, etc.). The simulated system evolves over time by tracing the trajectory of each object under various physical regimes such as Newton's laws.

A fundamental component of DEM simulation is the automated identification of collisions between objects. This process is known as *contact detection*. The computational cost of contact detection increases as a function of both the number of objects being simulated and the complexity of each object's surface geometry. These factors severely limit the scale and resolution of simulation that have traditionally been performed. To simulate physically significant numbers of objects, simple primitives such as discs and spheres are used as a means to reduce the total running time. Unfortunately the simplicity of these objects permits only the gross behavior of granular media to be captured.

This thesis presents algorithms that significantly reduce the computational complexity of the contact detection problem. Improvements are obtained in two areas, namely, the development of a new geometric representation for 3D objects and the use of parallel computation.

A scheme called the *discrete function representation* (DFR) is derived to model the surface geometry of complex 3D objects. DFR based contact detection between a pair of objects exhibits an $O(N)$ running time, where N is the number of surface points used to represent each object. In practice this results in a speedup of up to 2 orders of magnitude over traditional techniques on sequential machines.

On parallel machines a *message passing* scheme is developed to distribute the contact detection task over a number of processors. Application of the algorithm to large scale 3D simulations achieves a parallel efficiency of over 95% on a dedicated machine reducing the required CPU time from several *weeks* to several *hours*.

Thesis Advisor: John R. Williams
Associate Professor, Department of Civil and Environmental Engineering

Contents

1 Introduction **19**

1.1 Motivation 20

1.2 Thesis Objectives 21

1.3 Discrete Element Methods 22

1.3.1 The Continuum View 23

1.3.2 DEM - A More Comprehensive Perspective 24

1.4 Computational Barriers in DEM 25

1.4.1 Full Enumeration Example 25

1.4.2 Complexity Measures and Real Time Bounds 29

1.4.3 Spatial Distribution and Locality 30

1.4.4 Scale and Resolution 31

1.5 Other DEM Systems 33

1.5.1 Parallel Implementations 36

1.6 Defining a Framework 39

1.6.1 The Prototype Simulation System 39

1.7 Thesis Outline 41

2 Sorting for Contact Detection **47**

2.1 A Top Down Approach 47

2.2 Spatial Sorting Algorithms 48

2.3 Spatial Sorting Using Heapsort 53

2.3.1 Heapsort Algorithm 55

2.4 2D Spatial Heapsort Example 56

2.4.1 Building the Sorting Table 56

2.4.2 Searching For Candidate Pairings 58

2.5 Summary 60

3 The Discrete Function Representation **61**
3.1 Object Representation . . . 62
3.2 The 2D-DFR Scheme . . . 65
3.2.1 Storing the 2D-DFR Data . . . 66
3.2.2 2D-DFR Contact Resolution . . . 67
3.2.3 Timing Comparison with the Cyrus-Beck Method . . . 72
3.2.4 Summary . . . 73
3.3 The 3D-DFR Scheme . . . 75
3.3.1 Overview of DFR For Arbitrary 3D Geometries . . . 75
3.4 The 3D-DFR Data Structure . . . 79
3.5 3D-DFR Based Contact Detection . . . 85
3.5.1 Algorithm Description . . . 86
3.5.2 Algorithm Steps . . . 91
3.5.3 Point Classification Timing Tests . . . 93
3.6 Summary . . . 95

4 Contact Detection **97**
4.1 Theory and Practice . . . 98
4.1.1 Object Types . . . 98
4.2 Enumeration of the Contact Pairs . . . 99
4.2.1 Spheres and Spheres . . . 100
4.2.2 Planes and Spheres . . . 101
4.3 Force Resolution and State Update . . . 103
4.4 Contact History Lists . . . 105

5 Implementation **109**
5.1 Dynamics of Motion . . . 110
5.1.1 Numerical Integration Scheme . . . 111
5.2 Visualization Graphics . . . 112
5.2.1 Visualization in Hardware . . . 112
5.2.2 Visualization in Software . . . 113
5.3 Matrix Transform Library . . . 115
5.4 User Interface - Abstraction and Control . . . 116
5.4.1 Tcl - Tool command language . . . 117

5.4.2 Tk - Tool kit 118
5.4.3 Application Independence 118
5.5 Abstract Data Types 120
5.5.1 A Templateχ*mal* Application Script 123

6 Parallel Computing **125**
6.1 Parallel Computing Applications 125
6.2 Parallel Computer Architectures 126
6.2.1 Distributed Memory Machines 128
6.2.2 Communication Networks 128
6.3 Parallel Programming Models 132
6.3.1 The Data Parallel Programming Model 132
6.3.2 The Message Passing Programming Model 135
6.4 The Message Passing Interface Standard 137
6.4.1 Message = Data + Envelope 138
6.4.2 Point To Point Communication 139
6.4.3 Collective Communications 140
6.4.4 Matrix Transpose Example Using MPI 141
6.4.5 Summary 149

7 Distributed Discrete Element Simulation **151**
7.1 Parallel Sorting 152
7.1.1 A Simple Parallel Sorting Algorithm 153
7.1.2 Parallel Sorting Tests 154
7.1.3 Evaluation of Parallel Sorting 158
7.2 Distributed Discrete Element Simulation 159
7.2.1 Implementation Details 159
7.2.2 Parallel DEM - Scaling and Limit Considerations 161
7.2.3 Base Case: Spheres - Space/Time Bounds 164

8 Applications **167**
8.1 Sandglass - A 2D Hopper-Flow Simulation 168
8.2 2D Deposition and Compaction 171
8.2.1 Particle Layout And Compaction 171
8.3 3D Geometric Models of Permeable Solids 174

8.3.1 Overview of Method 176
8.3.2 Object Representation 177
8.3.3 Particle Layout And Compaction 179
8.3.4 Void Space Casting - Isosurface Casting 180
8.3.5 Summary 182
8.4 3D Embankment Collapse 183
8.5 3D Shock Wave Propagation 183
8.6 Contributions and Future Directions 186
8.6.1 Related Applications 187
8.6.2 Future Work 187

List of Figures

1.1 All To All Checks 26

1.2 Point Classification 27

1.3 Bounds on Resolution For Different Numbers of Objects 32

1.4 Analysis Pipeline 40

1.5 Chapter Hierarchy Plan-View 44

1.6 Chapter Hierachy Front-View 45

2.1 Various Sorting Strategies 49

2.2 Binary Tree Representation of a Heap 54

2.3 Heapsort Example 57

2.4 Index Intersection for the Heapsort Example 59

3.1 Point Classification For Various Boundary Representations 63

3.2 Deposition Simulation 65

3.3 DFR Steps a) Traversal of Vertices b) Physical To Address Space Mapping 66

3.4 Simple Contact Bounds Test For Trapezoidal Region 70

3.5 Plot of Time Vs. Point Sampling (N) 73

3.6 Sample Geometries 76

3.7 a. Object Geometry b.Sampling Grid 77

3.8 Example Cell Classifications: a) Single Vertex b) Multiple Vertices . . 78

3.9 a) Tessellation b) Mesh Discretization 79

3.10 3D-DFR Elements and Example Object (Discrete Sphere) 80

3.11 3D Ragged Array Storage for Discrete Sphere 81

3.12 a) Cell Code Assignment, b) Discrete Bounding Hull (2D) 82

3.13 a) Discrete Bounding Hull, b) Internal Cells With Embedded Mesh . 84

3.14 3D-DFR Contact Detection as a Hierarchical Process 85

3.15 Zone from Sutherland-Hodgman Clipping 86
3.16 Zone From Bounding Box Overlap 87
3.17 Zone Index Set From Clip Zone 88
3.18 Discrete Edge - Edge Contact 90
3.19 3D Point Classification Test Times For DFR and Cyrus-Beck 94
3.20 3D Point Containment Speedup Using DFR Over Cyrus-Beck 95

4.1 Sphere-Sphere Contact Details 100
4.2 Construction of Sample Point for Sphere Plane Intersection Test . . . 102
4.3 Contact Force Resolution 103
4.4 Contact Object C Structure 105
4.5 Contact Table Description 106

5.1 Application Independence 120
5.2 Example View of Graphical User Interface 123
5.3 A Template Application Script 124

6.1 Simple Models of Parallel Computer Architectures 127
6.2 Communication Networks 129
6.3 Serial Code for Matrix Transpose 133
6.4 Data Parallel Code for Matrix Transpose (Using C*) 134
6.5 Message Passing Communication Operations 136
6.6 All-To-All Transpose Pairings For Row 0 142
6.7 An All-To-All Transpose Algorithm 144
6.8 MPI Code for All-To-All Transpose Operation 145
6.9 Scatter Gather Swap Pattern, L = pass 146
6.10 All-to-All Vs. Scatter Gather Transpose Timings 149

7.1 Simple Parallel Mergesort on 8 Processors 152
7.2 Graph of Sorting Time for N = 8.. 1,000,000 Keys 156
7.3 Closeup of Sorting Times for N = 10,000.. 40,000 Keys 157
7.4 Closeup of Sorting Times for N = 100.. 2000 Keys 158
7.5 CPU Time [sec] Vs. Number of 2D Objects, N 162
7.6 Detail (for network) of CPU Time [sec] Vs. Number of 2D Objects, N 163
7.7 Scaling of CPU Time Vs. Number of SP2 Processors (10,000 2D objects) 164
7.8 CPU Time [sec] Vs Number of Objects on Different Machines 166

8.1 Simulation of Particulate Flow 169
8.2 Parallel Update Configuration Space 173
8.3 Particulate Material Composed of a) Discs b) Superquadric Elements 174
8.4 Porous Monolith and 2D Section 175
8.5 Porous Model Generation 176
8.6 Superquadric Ellipsoids with varying exponents, ξ, η 178
8.7 3D Spheres Packing 179
8.8 3D Particulate Packing (10,000 grains) 180
8.9 Various Views of a Porous Material 181
8.10 Failure of Cut Embankment 184
8.11 Shock Wave Propagation in 3D Particulate Fluid 185

List of Tables

2.1 Sorting Table Before Heapsort 58

2.2 Sorting Table After Heapsort 58

3.1 Operation Cost 71

3.2 Timing Results 72

3.3 Typical Speedup Factor 74

3.4 Point Classification Times and Speedup 93

4.1 Cross Reference of Intersection Tests Performed 99

5.1 Runge-Kutta-Nystrom Numerical Integration Steps 111

5.2 gl Emulation Routines 114

5.3 Example Vector-Matrix Routines 115

5.4 χmal Objects 122

6.1 Processor Pairings for 8×8 Matrix Transpose 143

7.1 Sorting Time for N Keys 155

7.2 Sorting Time for N Keys on 8 SP2 nodes 158

7.3 CPU Times on IBM RS/6000 320H and 590, and Sun4 Workstations 160

7.4 CPU Times for SP2 and CM-5 and network of Sun Workstations . . . 161

7.5 CPU Times 3D Spheres Test 165

8.1 Function Call Sequence 170

8.2 Simulation Profile 170

Acknowledgements

I would like to acknowledge and thank many, many people who have helped and supported me over the years. The pages of this thesis only reflect a small part of what I have learnt here and what I hope to learn in the future. These acknowledgements are neither alphabetical or chronological and will undoubtedly remain incomplete. My intention is simply to recognize some of the groups of people that have influenced me both personally and academically.

To Hugh, Tom, Jane, Michelle, Liza and Russell for never questioning. Daragh for always being there. Natasha, this flight of fancy may have obscured me, I hope I have not become invisible. Steve McDowell and David Watts whom I miss but gradually learnt to understand why. To Ronan, John, Sara and Michael ("On Silly Balls" didn't fly with the publishers), Steve Crane, Steve Tee, Gervaise, Alida, Siobhan Fahey, Judy Hobson, Liam Pender, Gordon White, Oira Quinn and Julian Rafter, Tom McCabe, Cliodhna Callan, Kieran Murphy, Graine Murphy, Barry Henry, Suzanne, Eimer, Maibh Keavney, Niallo Carrol, Alan Fogarty, Eoin Condon, Conrad, William Walsh, Eoin Murphy, Fergal Kelly. Anne, I hope some of my random comments made sense.

To Professor Simon Perry and Dr. Niamh Harty from Trinity College, for your transatlantic advice, support and openess. I have learnt more than I can ever let you know or thank you for. This journey became real because of you both.

I find fatalistic footprints throughout my life, and see how they lead to the current time. To Brian O'Connor, Eoin Lynch, Fergal O' Donoghue, Ray Knox, Conor Casey, David Healy, Niall Smith, Brian MacMahon, Gary McCarthy, John McEntaggart, Leonard Ryan, Stephen Murphy, Hugh Francis, Anthony Quinn, SP, Taffy, Killer, James, Willie, Manfred and Kevin. Fathers Hannon, Divine, Hally, Kilbride, Smith and the many teachers and priests at Blackrock College for structure. Andrew O'Kelly and Jim Doran for the adventures. Jane, Norma, Jim, Niamh, Phillip, Wilmer, Mark, Ian, Ivor, Mark, Michael, John and George. Paul Hewson told me stories, I fumbled the dance.

An escape to an island from an island was an intense 6 weeks in the jungle with Danny Ryan, Peggy Ryan, Daniel and Susan Vogel, Joan McHugh (the first time I met you, but not the last), Susan Sarle, Bob Clevenger, Jan Fontana, Caroline deGeoffrey, Kevin Woosely, Mathew Koenig, and all at 4th & Broadway and No. 1 Picadilly Circus.

To the many wonderful people I have met in MIT and the Building 1 environs. Jesus and Maru Favela, Nabha Rege (my DEM partner, technical advisor and calming influence), Patrick Kinnicutt for friendship and kindness, Nestor Agbayani, Allan, Gorti Sreenivasa (a lucky man), Shamim, Ashu, David Lim, Kazumi Iwane, John Thomas, Kevin Amaratunga, Itunumi Savage, Ashok, Anil, Salal, Tom Grayson, Antulio and Amalia, Shenoi, Josh Elliot, Maria Santagata, Mike Geer, Fenny and Jen-Diann. Master-builder, Bob Beretta, for his insights and insight. To Daniel Higgins for helping me understand and continue. Jonathan Richmond supplied tea and Christmas cards for much of my stay.

Joannie McCusker (Donnie and Ryan), for your friendship (and cookies), thank you for looking after me. Professors Nigel Wilson, Steve Lerman, Jerome Connor, Robert Logcher, and Dr. Duvvuru Sriram, all of whom have made the MIT experience persist. To Professor Ladd for giving a mad Irishman a chance.

To Cynthia Stewart and Jessie Carty for keeping me afloat, their gentle reminders on things past due and votes of confidence throughout the years. I would have registered in Harvard if it wasn't for both of you. Elaine Healy, Muriel Fredricks, Stephanie George, Pat Dixon and Carol McIntyre for making Head Office a less intimidating place. Lisa Magnano, for your kind spirit and enthusiasm. Paula, Maria and Mary Grace from Building 1. Trond Kaalstad has provided me with unending trust, confidence and a special friendship through the many ups and downs. I am eternally grateful. To Joe Brenner for listening while I talked. Danielle, making coffee wouldn't have been as much fun without you. To Mary Elliff for bridging the gaps when I was trying to schedule my meetings. Donald Knuth for the constraint based page layout system from hell.

To David Zeltzer for allowing me take his course on advanced 3D graphics and introducing me to the very special people in the Snake Pit. Steve Drucker and Steve Pieper for having the patience, their support and friendship. David Chen (for 3d), David Sturman (for the helping hand), Michael Johnson (for opening a mystery door to the Connection Machine), Mike McKenna (for the T-shirt) and Margaret Minsky for her unconditional support of naive exploration. I thank you all for your unwitting help in getting me through the spring. I learnt by accident, 3 days after defending my thesis, that Martin Friedmann was no longer thrashing down Massachusetts Ave. I only met you in passing, I see your shadow, it's still burning. Thank you for your spirit. I trust you, it works. I'm sorry I missed you.

```
char*p="char*p=%c%s%c;main(){printf(p,34,p,34);}";main(){printf(p,34,p,34);}
```

Mauricio Daher, Jim Culbert and Gary Cutbill for trusting me with passwords and showing me the ropes. Graham Mustoe for encouragement and helping keep the show running. Majors John Gill and Martin Lewis of the Air Force Office of Scientific Research who managed the research project that initially supported some portions of this work. To everyone at IS, DCNS, AC, FL, SIPB, and previously MIT Project Athena, thanks for having such as great space to work in. John, Lou, Anne, Dot and Carol were incredibly helpful and supportive when I needed them most.

Elizabeth Menhenett (you understand me better than I do), Ronke Olabisi (it's your beat they are marching to), Jane Lee, Brian Kelly, Mimi Huang, Dionne Chapman, Sonia Ensenat, Aileen Lee, Vincent Cobalt, Soonkyu Shin (SK), Renu Kurien (Bostwick), Deana Rafizadeh (the wicked which from Atlantis), Michelle Hoo-Fatt, Wona Chung (no, not Mary), Elizabeth Kader, Mellisa Kwok, Christina DeMello. Karen Pfautz for taking care of Ronke. Alex Worden for solace and confusion. Ralph Santos for literary nuances in C/Paris. Chris Teixeira for sympathetic industrial noise. I have met many special people during an extended tenure in MIT Course 1.00, thank you all for your unique perspectives on the world. Alden Jones for elevating work. Jim and Donna for getting me home from Mexico. Lisa Cullen for smiles.

Nabha Rege and Daniel Higgins helped critique several draft chapters, pulling many loose thoughts together and throwing looser ones away. I thank you both for the time and encouragement you have given me. To David and Laura Ford for their friendship and love over the years, Central Square was always a warmer place to live because of you. Joanne and Dale, Kennedy Biscuit Lofts will never be the same, many thanks for all the cheese and jazz. Jessica for coffee and colour crush. Christiana, see you in New Mexico. Tommy and Shiela, Ann, Michael, Conor, Manfred and Andrea, Felissa, Deirdre, and Annette, for your care and affection.

To my thesis committee, who braved my organizational and scheduling "methods" over the past 2 years. Professor Eduardo Kausel, Dr. John Germaine and Professor Mark Jakiela. Thank you for your support, confidence, patience and insight.

To my advisor, and committee chairman, John Williams and his wife Rita for their special friendship and care during the cold winters.

To my parents, Michael and Zoë, and my sisters and brothers, Orla, Turlough, Síana, Etain and Darragh, my grandmother Darinka (Dinky) Roche, and grandaunt Jig Durley. Thank you all for making this possible and seeing me through to places I never expected to reach.

Chapter 1

Introduction

The Sandcastle

When a child builds their first sandcastle, it stands with simple pride until the tide reclaims the structure as its own. The next sandy edifice tends to be more complex than the first, with towers and turrets, and a gate to enter by. The features are carved with a finger, a stick, or better yet a razor shell, already dried and lying on the beach. Through trial and error the child learns to control the cascading flows of sand, the collapsing arches, the sliding walls, the crumbling watch-tower falling into repose. The details become increasingly complex. The definition of features place sand grain with sand grain, balanced precariously upon one another. The slightest movement of the hand and the secret tunnel disintegrates, the walls slope into a pile.

Can the collapse be anticipated? Can it be avoided? Where does it all start? Which grain of sand is overcome with the need to move, initiating the collapse? Does the entire structure disintegrate spontaneously, or does the failure start with a single grain and propagate to all of the others?

These are not simply questions of a child. They are fundamental questions about a complex physical system of great natural, civil and theoretical importance. Water is cleansed by sand, cities are built on sand, its behavior forms a prototype in the study of critical dynamics [3, 88]. Industrial processes concerned with powders and grains, require an understanding of these particulate systems support external forces, how fluid flows through the the bulk material and how the bulk material itself may flow. The various modes of behavior depend on the distribution of particle size, shape, packing density and other phase relations. In this thesis I describe a modeling system in which to examine the effect of these characteristics on particulate systems.

Dr. Terzaghi, 1925

The problems of foundations are as old as any with which the structural engineer has to deal. However, structural engineering became during the last one hundred years an elaborate and very intricate science, while in the vast field of earthwork engineering nothing has been produced except a set of pile-driving formulæ and an academic theory of earth pressure against retaining walls. Symbolically speaking these theories fit into the frame of modern civil engineering as Trinity Church fits into Broadway. The structure is very venerable, but its foundations are just strong enough for supporting a one-story building. Hence, before erecting a more elaborate structure, one has to dig deeply below the old foundations and to provide a more substantial substructure.

This has been done by careful study of all those physical factors which cause the difference between the soils the engineer has to deal with and the ideal material invented by the authors of our classical earth-pressure theories. These factors can be assembled into three groups: first, the forces acting at the points of contact between the soil grains; second, the strain effect produced in soils by external forces; and third, the effect of time on the development of strain. - Dr. *Charles* Terzaghi, 1925. [112]

1.1 Motivation

The work of Karl (*Charles*) Terzaghi during the early years of this century marked the birth of modern soil mechanics. The *study* he espoused in the preceding quotation continues today in a variety of forms, both theoretical and experimental, using old methods and *new*. While considerable progress has been made in the treatment of soils as continua, this perspective is only appropriate when the soil truly behaves as a continuum.

There is a need to understand critical phenomena whose root cause lies at the granular level. These phenomena determine not only *when* an embankment or foundation fails but *how* the failure evolves, how the body of material moves from one metastable state to another. To obtain an understanding of these processes it is necessary to examine the physics of the material, not as a continuous body of matter, but

at the microscopic level as a *particulate system*. It is at this microscopic scale that the observable macroscopic phenomena first become manifest. It is through the physical interactions of the sand grains that intermediate scale structures are formed. These structures combine to form still larger structures that span multiple characteristic lengths and ultimately emerge to define the macroscopic behavior. But how do these strucures form? How do local perturbations grow to influence the global behavior of the material? Can it show us how a body of sand reacts to internal fluctuations, or to external processes in the environment? These are some of the questions that motivate the study of granular materials.

1.2 Thesis Objectives

To understand the behavior of granular materials we need some way to observe and measure the detailed interactions occuring at the microscopic level, without influencing it. The goal of this thesis is *to provide a means to investigate the granular microscape, undetected.* We do this by creating an idealized numerical model of the granular system, in which we can directly control the environment and both observe and interrogate the objects that populate this model as it evolves over time.

We look to computer simulation, and specifically to a class of multibody techniques known as *Discrete Element Methods*, as a framework in which to construct this environment. Granular materials are modeled as collections of discrete objects that move independently of one another. Forces can be applied to objects, globally as with gravity, and locally on the individual objects through short range potentials and surface contact. Full knowledge of an object's geometry and physical state are available to us. The goal is then to develop models that are then sufficiently large (in some statistical sense) to answer the questions we pose.

The principal computational limitation of DEM technology is the excessive time required to simulate physically significant numbers of objects. The problem is compounded by the need to represent the individual objects with sufficient accuracy so as to capture the salient characteristics of the individual grains and better reflect the behavior of the material being modeled. Traditionally, as a means to reduce the computational time, there has been a trade-off between the number of objects modeled and the level of detail of each object's representation.

This thesis presents a number of methods that significantly reduce the computa-

tion time required in DEM simulation. This is achieved in a number of ways:

1. The development of a new object representation scheme to model complex 3D geometries.

2. The derivation of an efficient contact detection algorithm based on the object representation scheme.

3. The use of parallel computation to distrbute the simulation over a number of processors.

The development of a computational environment in which to simulate granular materials is the overall theme of this thesis. It serves as the framework in which to implement and apply the algorithms presented in this work. The objective of this work is then summarized as follows:

- Define and implement a computational framework in which to *efficiently* simulate the physics of granular systems containing *large numbers* of *complex* objects.

While the nature of the experimental apparatus developed in this work may not have been familiar to Dr. Terzaghi, I believe it would have appealed to him nonetheless.

1.3 Discrete Element Methods

To study the behavior of materials we rely on mathematical modeling, an idealisation. There are essentially two possible directions from which to formulate this idealization, from above as a continuum or from below as a collection of representative particles (atoms, particles, grains, etc.). The appropriateness of the top down view depends on whether the behavior of the material can be characterized macroscopically using a *representative element volume* to adequately capture the behavior at the microscale. In other words, the material should behave in such a manner that the effects of a local peturbation at microscopic level does not manifest itself at the scale length of interest.

1.3.1 The Continuum View

The model most commonly applied to analysing a material is that of a governing differential equation, based on a *continuous* idealization of the material. The governing differential equation describes a constitutive relationship between characteristic properties of the material (at a chosen scale) and how it responds to different external processes (force, temperature, etc.). To fully specify this model we also need to specify boundary and initial conditions that define a unique solution in space and time.

For a real material, the simplest model that is assumed is usually linear-elastic. Experimental characterization is then used to obtain measures of the intrinsic properties. For instance, in engineering mechanics the deformation of a material under an imposed load is often of interest. A constitutive relationship in this case determines the strain induced in the material due to an applied stress. The bulk properties obtained for elastic materials would then be the modulus of elasticity, Poisson's ratio and the *voids* ratio. Of course this assumes that an elastic model is appropriate in the first place. This is the simplest case, modern soil mechanics extends this analysis to far more complex non-linear and non-elastic models. While these latter techniques are better suited to the study of the underlying behavior, they still fail to capture the microscopic behavior. To understand why, one needs to look at the assumptions made in chosing these models. The fundamental assumption is continuity. The consequence of this assumption is the ability to characterize the material using bulk properties.

What do these properties tell us about the material? They are averages. They describe how the bulk material behaves macroscopically by averaging interactions occuring at the microscopic level. This is consistent with the assumptions made for the mathematical model of the material. But in essence this is treating the material as a *continuum*. Depending on the scale at which one considers the material this assumption may not be valid. One merely needs to lift up a handful of sand and watch it flow through your fingers to realise that it is not continuous at all, it is a discontinuous system of particles. The continuum model in this case does not, or simply cannot, tell us what is happening at the microscopic level. The averages do not pertain to the behavior of the individual grain, the geometry of the grain, which grain rubs against some other grain, or which of them form intermediate structures that spontaneously appear and disappear. Fluctuations in the material's composition

become smeared over the representative volume and are lost.

To examine a granular material, at the scale of the individual grain, requires that experimental measurements are performed at the microscale. This places limitations on what can be investigated. A probe small enough to be placed in a sample will tend to disturb the local behavior that it is intended to capture.

1.3.2 DEM - A More Comprehensive Perspective

Discrete Element Methods (DEM) are a family of related numerical techniques used to simulate physical systems exhibiting discontinuous material and geometric behavior. The premise underlying these methods is that the physical system of interest can be modeled as a collection of discrete objects that interact through surface contact. The surface geometry of an object is represented using either mathematical functions such as a sphere, or as a tessellation of facets that bound a closed region. In particular, a discrete element method has the following characteristics (after Williams *et al.* [130], Cundall and Hart[25], Pande, Beer and Williams [93]):

1. Permits large displacements and rotations of disjoint bodies.

2. Automatically identifies the occurence of contact between pairs of bodies as the simulated system evolves. This process is known as *contact detection.*

Discrete element methods provide a means to model discontinuous systems such as granular soils at a more fundamental level of idealization, that is, as a collection of discrete objects. These techniques have found application in a wide range of engineering problems arising in areas such as soil, rock and ice mechanics, constitutive modeling, transport processes including suspensions, and in manufacturing processes such as packaging. Similar techniques are used to simulate so called N-body systems, such as molecular dynamics, fluid dynamics and celestial mechanics, which consist of interacting charges, vortices or point masses. Discrete element methods extend the techniques used to solve N-body problems by including the ability to analyse the details of discrete surface interactions in the form of inter-body contacts.

1.4 Computational Barriers in DEM

A fundamental component of DEM simulation is the automated identification of surface contacts between objects. This is referred to as *contact detection*. Contact detection is known to be the principal computational cost in DEM simulation, scaling as a function of both the number of objects being simulated [127, 40, 15] and the complexity of each object's surface geometry, [42, 81, 94, 37, 132].

Simulations containing relatively small numbers of objects quickly overwhelm the largest and fastest of current day computers. This is due to the number of *possible* interactions an object takes part in. Chosing which objects to examine, how they affect one another and how these interactions take place, requires considerable ingenuity. The complexity of these models is felt most in terms of the time it takes to calculate the state of the objects populating the system. This forms a computational barrier on the scale and resolution of what can be modeled. The impact of this barrier becomes more evident in the following examples. Cost for representative operations in the subsequent description are either obtained from real simulations or have been measured using independent benchmarks.

1.4.1 Full Enumeration Example

Consider a system of objects such as those shown on the left of Figure 1.1. In the most general situation no information about the shape and spatial distribution of the objects is known *a priori*. Contact detection then requires that each object in the system must be checked against every other object. For a system containing M objects, the exhaustive enumeration of the pairings is said to require *of the order of* M^2 contact tests and is denoted by $O(M^2)$. This says that the algorithm has a *worst case* performance requiring a number of operations that aymptotically approaches M^2 operations, as M becomes very large.

Pairwise Contact Tests

Each test for contact between a pair of objects determines whether the boundaries of the objects intersect and, when this is the case, captures a geometric description of the region of overlap. The cost of the intersection test greatly depends on how each object's boundary geometry is represented. For arbitrary geometries such as

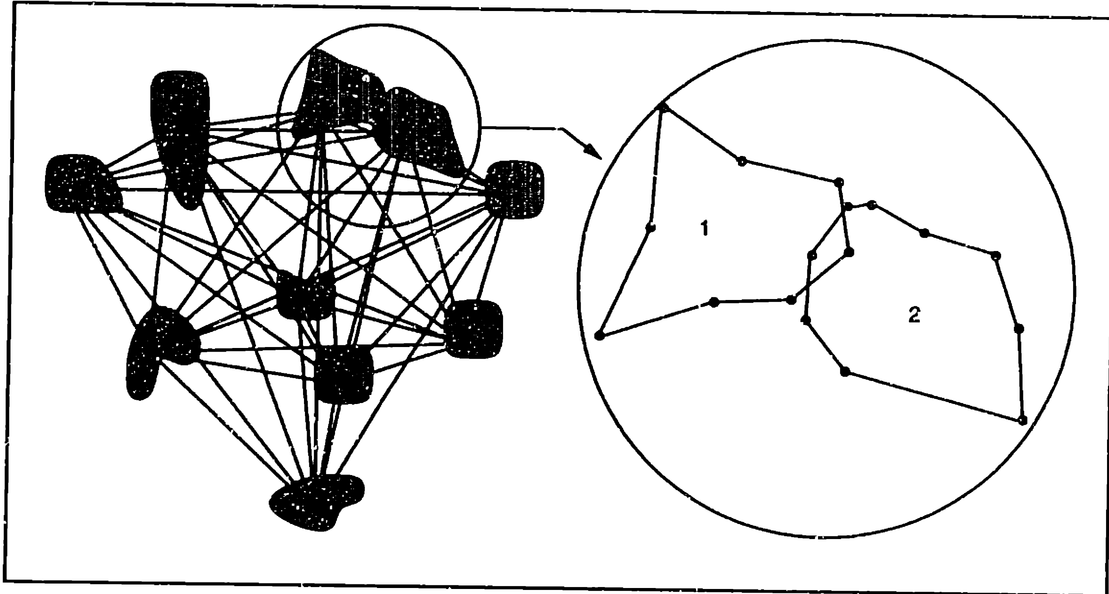

Figure 1.1: All To All Checks

those shown on the right of Figure 1.1, polygonal representations are traditionally used. Once again since no information regarding the shape or extent of the objects is known *a priori*, each edge from one object must be checked for intersection against *every* edge from the other object.

For *convex* polygons the *all-to-all* set of edge intersection tests is equivalent to testing every point on the boundary of one object to see if it lies inside the boundary of the other object. Testing points in this manner is referred to as *point classification.*

Point Classification

To test if a point lies inside a polygonal region it must be checked against *every* edge of the boundary. A simple analogy helps to show why this number of checks are necessary. Consider a square room with one-way mirrors on each wall and assume that the reflective face of each mirror is on the inside of the room. If a person is standing *inside* the room they can confirm this fact by checking for a reflection of themselves when they look to the left (mirror 1), in front (mirror 2), to the right (mirror 3), and behind (mirror 4). This idea is shown in Figure 1.2.a. Note that it is necessary to see a reflection in *all* mirrors to guarantee that they are inside. If the

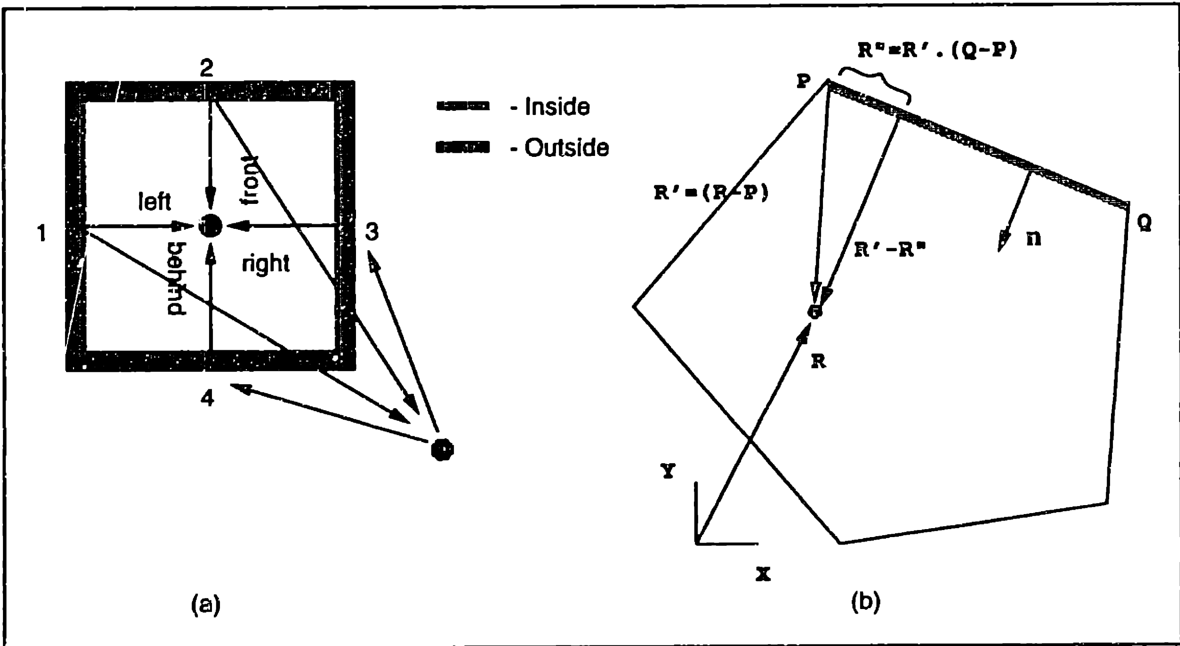

Figure 1.2: Point Classification

person is outside the room, say below the lower righthand corner they will only see a reflection of themselves in mirrors 1 and 2. In this case the *lack* of a reflection in *either* mirror 3 or 4 is sufficient to indicate that the viewer is *outside* the room. Again this is shown in Figure 1.2.a.

Mathematically the inside-outside tests are performed by replacing the inward facing mirrors with inward pointing edge normals. For example, consider the polygonal region shown in Figure 1.2.b, and the point $\vec{R}$, both described with respect to a common coordinate system. The test if the point $\vec{R}$ lies on the interior of the region, the following contruction is performed. For a given edge PQ, form the vector joining the end point $\vec{P}$ with the test point $\vec{R}$ as $\vec{R}' = \vec{R} - \vec{P}$. Next the component of $\vec{R}'$ parallel to edge PQ is obtained by projecting $\vec{R}'$ onto the vector $\vec{PQ} = (\vec{Q} - \vec{P})$. The component, denoted $\vec{R}''$ is calculated from $\vec{R}'' = \vec{R}' \cdot \vec{PQ}$. The component of $\vec{R}'$ perpendicular to edge PQ is then formed as $\vec{R}' - \vec{R}''$. Finally, if perpendicular component points in the same direction as the inward pointing edge normal then $\vec{R}$ is said to be on the inside of the region with respect to edge PQ, that is, $(\vec{R}' - \vec{R}'') \cdot \vec{n} \geq 0$. This is equivalent to saying that the angle between the vectors is less than or equal to 180°. To confirm that $\vec{R}$ is inside the complete region it is necessary to perform the same construction with *all* of the other edges describing the boundary of the region.

The requirement that all boundary edges be checked to guarantee containment also extends to classifying lines against convex polygons. Each check is more complicated but the same principals used for point classification apply. Finding that one edge intersects the boundary of the other object simply confirms that contact might have occured but is not sufficient to fully determine what portion of the line is contained.

The main point here is that the component operation of the classification test is relatively simple, it is the cummulative cost of the *all-to-all* requirement that becomes the computational bottleneck in pairwise contact checking. Two polygonal objects represented with 100 boundary edges would require 20,000 edge classification tests in the worst case.

A Unit Operation Cost

One of the most common operations that occurs during contact detection is the calculation of a distance between two objects (in the point classification example just described, the expression $(\vec{R}' - \vec{R}'') \cdot \vec{n}$ is equivalently calculating the distance of point $\vec{R}$ from the edge PQ, to within a squareroot operation). The objects might be composed of points, lines or planes but the basic notion of a distance calculation remains. The simplest of these operations is to calculate a *point-to-point* distance. On a modest workstation[1] the time to calculate a *point-to-point* (T_d) was calculated in approximately $T_d = 10^{-5}$ seconds. The maximum number of distance calculations possible in 1 second of CPU time is then $1/T_d = 10^5$ operations. The cost of other distance measures, such as *point-to-edge* or *point-to-plane*, are considered to differ only by a small constant factor.

Total Cost for the Naive Algorithm

The simplified examination of the point classification test allows us to define a bound on the cost of the pairwise contact test. If both objects are represented using N boundary edges then $O(N^2)$ checks are necessary to correctly determine whether they are in contact. Combined with the full enumeration of the object pairings leads to a *worst* case running time T(M) given by Eqn 1.1.

$$T(M) = O(M^2N^2) \tag{1.1}$$

[1]The times used in this section were obtained using an IBM RS/6000 320H (20 MHz) workstation.

Furthermore, Eqn. 1.1 is technically a tight bound on the running time for the algorithm in that this order on the number of operations are *always* required.

While a simplification, it is often easier to appreciate the implications of this complexity measure when some reasonable cost per operation is associated with each of the terms. A conservative time to use for each *operation* is the unit operation cost described earlier. This was said to require 10^{-5} seconds. If the model contains $M = 10^5$ 3D objects, each with $N = 10^2$ faces, then the cost of a complete set of contact detection tests for the entire system requires 10^9 seconds. This is equivalent to over 30 **years** of dedicated computer time. Clearly this would be infeasible.

1.4.2 Complexity Measures and Real Time Bounds

O-notation provides an asymptotic upper-bound on the order of growth of a given algorithm. It describes how a particular algorithm will perform in a *worst case* situation. Formally it is defined as follows [115]:

Definition 1 *For a given function $g(n)$, $O(g(n))$ corresponds to the set of functions $O(g(n)) = \{f(n)$: there exists positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$.*

$O-$*notation* essentially singles out the highest order term from a function containing all of the components that contribute to the actual running time. For example, an algorithm with a running time of $T(M) = c_0 + c_1 M + c_2 M^2$ would be described as having a *worst case* running time of $T(M) = O(M^2)$ as the function $T(M)$ will be dominated by the M^2 term as M becomes increasingly large.

For problem sizes of practical interest the running time may in fact be dominated by the constants that are ignored in the asymptotic analysis. For instance, if $c_1 > c_2$ then for some range of M the cost might be better approximated by the $c_1 N$ term. This is an important factor if one wishes to make reasonable predictions about expected running times of a given algorithm, for a known input size M.

A more insidious problem with O-notation (or more correctly, its misuse) is that it is often used to report a single measure for a *worst case* running time without qualifying whether this is dependent on the input. This becomes important when a worst case running time only occurs for a small number of input sets, whereas the average performance of the algorithm is significantly better. This occurs when the performance of the algorithm depends on some characteristic of the input set such

as the monotonicity of the ordering or the distribution. The standard example for this arises in the analysis of the Quicksort algorithm which has a worst case running time of $O(M^2)$ when the input set is pre-sorted but it can be shown probabilistically to require *on average* $M \log M$ operations to complete [115]. For these reasons it is preferable to qualify the expected average in some manner.

1.4.3 Spatial Distribution and Locality

The main problem with the preceding *naive* algorithm is that it does not take into account the physics of the system. For instance, if the objects only interact through surface contact, objects lying distant from one another will not affect each other. This suggests that the spatial distribution of the objects should be examined as a means to reduce the number of pairwise checks. This process is referred to as *spatial sorting*. It attempts to organize the system of objects in such a way that only objects close to one another need to be checked for contact. The cost of spatial sorting scales with the number of objects in the system. In Chapter 2 a number of algorithms are described that require on average $M \log_2(M)$ operations to complete, where M is the number of objects.

While the worst case running time for the simulation algorithm remains as being $O(M^2N^2)$, the *average* running time will be closer to that of Eqn. 1.2.

$$T(M) = O(M \log_2(M) + \rho M N^2) \tag{1.2}$$

The parameter ρ is an empirical measure of the number of objects that are in contact with one another. In the DEM literature this term is often referred to as the *coordination number*. In general the value of ρ depends on both the size, shape and spatial distribution of the objects. So in the worst case $\rho = M$ which reduces Eqn. 1.2 to that of Eqn. 1.1. However, in practice the average value of ρ will be much smaller than M. A simple example would be a system of single sized spheres which have a coordination number of $\rho \leq 12$ when optimally packed, [47].

The running time described by Eqn. 1.2 is considered to be representative of a discrete element simulation algorithm for large numbers of objects, M, where each object is represented using N surface facets. The $M \log_2 M$ term denotes the cost of the sorting component of the algorithm and $\rho M N^2$ corresponds to the contact tests between pairs of objects.

Using the unit operation cost for the contact test (10^{-5} seconds) and a cost of 10^{-4} seconds for each sorting operation Eqn. 1.2 is used to obtain an approximate measure of the improvment gained by the implementing the revised algorithm. The time per sorting operation was obtained empirically and applies only to this size of problem. For $M = 10^5$ and $N = 10^2$ the total time required is approximately 1.4 days of dedicated computation. This is a significant savings over the full enumeration algorithm in the previous section, but still much too large. The cost reflects the time required to perform a *single* iteration of the simulation. In practice several thousand iterations will be necessary to simulate the physical system over a useful time scale.

1.4.4 Scale and Resolution

The complexity measure for a general contact detection algorithm does not give a tight upper-bound on the running time. In other words, the running time depends on the input. This has both positive and negative implications in that the average running-time is generally lower than the worst case. It is generally not possible to quantify how much better the average case will be. However, in the absence of more precise measures (a probabilistic analysis) some reasonable empirical conditions can be developed to define practical bounds the problem. This was the case when ρ was introduced in Eqn. 1.2.

Now, if an upper-bound on the time available to perform a simulation is chosen, the scale and resolution of a simulation can be approximately determined. In this section a set of practical simulation sizes are developed in terms of an equivalence class relating the number of objects being simulated (scale) and the accuracy with which each object is represented (resolution). The set of results will make it possible to predict improved limits on the number of objects, of a given resolution, that it is feasible to simulate. This will serve to guide the development of more efficient algorithms later in this work.

Some Working Limits

To find a set of approximate bounds on the scale and resolution of possible simulations, a reference simulation is defined with limits on the the total amount of computation time available and the number of iterations required. These limits are chosen to be:

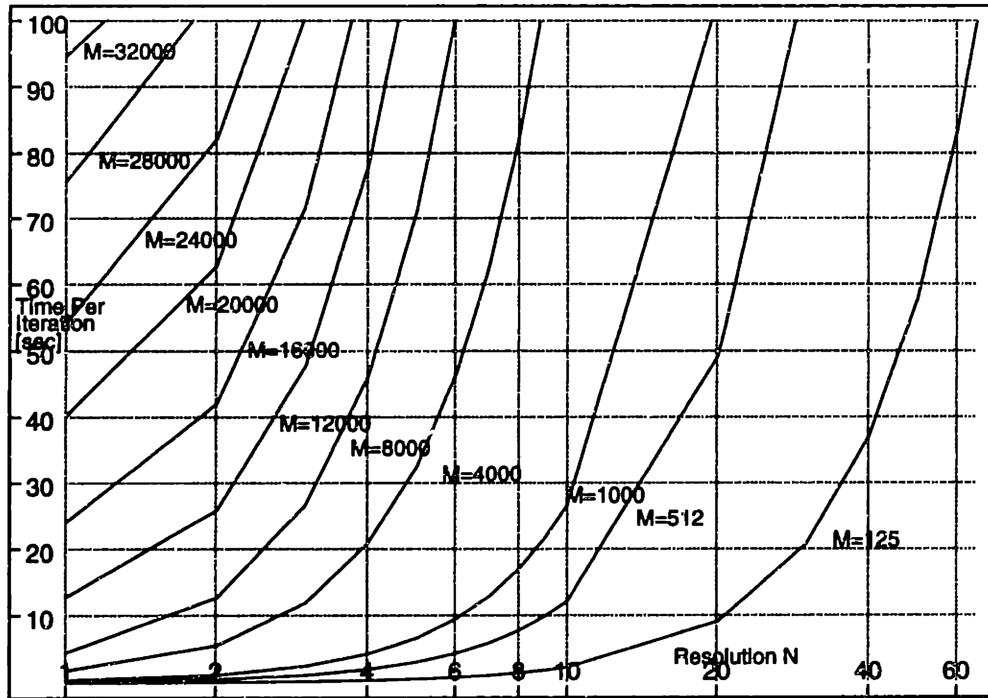

Figure 1.3: Bounds on Resolution For Different Numbers of Objects

CPU Time	10 days
Number of Iterations	10,000

The limits are based on the use of a sequential computer running at sustained rate, near the capacity of the machine, and with adequate memory to store the problem description (geometry, state data, etc.). The effective CPU time *available* per iteration based on these values is $T_i = 86.4$ seconds. Equating the average running time expression of Eqn. 1.2 with the time per iteration, T_i, gives:

$$T_i = M\log_2(M) + \rho M N^2 = 86.4 seconds \tag{1.3}$$

Eqn. 1.3 can then be used to define an initial intercept with the time axis. $T_i = 86.4$ is used as an upperbound on the time available from which to calculate feasible values for the number of objects and the resolution of each object that can be simulated, that is $T_i = M\log_2(M) + N^2$. The number of objects, M, is first obtained for a base case simulation consisting of spheres, giving a reference point on the time axis. Spheres are considered the simplest primitive that we can use in DEM

simulations and define the initial intercept with the time axis.

Using the unit operation cost of $T_d = 10^{-5}$ seconds per distance calculation the cummulative time for different object resolutions is then calculated assuming a coordination number of $\rho = 10$. The corresponding bounds on the resolution of each objects is graphed in Figure 1.3. Each curve corresponds to a fixed number of objects M. The intercept with the vertical axis corresponds to the time required to simulate M objects with a resolution $N = 1$, that is, spheres. Each curve then spans a portion of the resolution axis corresponding to the number of surface facets that it is feasible to represent. For example the $M = 1000$ curve shows that this number of spheres requires 0.4 seconds per iteration to simulate. Then, if the objects were modeled as polyhedra, up to approximately $N = 20$ surface facets could be used to represent each one. This is the bound on the resolution for this number of objects.

Using computational cost as the perspective of most interest, a review of other DEM systems appearing in the literature is now appropriate. This is the topic of the next section.

1.5 Other DEM Systems

Different applications use both different representations for the individual elements and can support different numbers of objects. Overall the goal is to capture representative characteristics of the system being simulated. In this section I summarize the various applications of discrete element methods described in the literature. In the context of this work computational aspects of the applications are noted according to generality (2D, 3D), type of object representation used (disc, sphere, ellipse, polygon, etc.) and where possible a measure of the scale of simulation performed. In some cases a run-time for the DEM simulations are also quoted. Unfortunately most authors fail to futher qualify this cost with an indication of how many iterations are also performed. Without this extra information the run-time costs cannot be usefully compared and are included here primarily to gain an estimate of how long the typical classes of experiments took to complete.

Soil Mechanics

Ting uses several hundred 2D ellipsoids to model laboratory experiments on sands, [118, 119, 120]. Ng works with similar numbers of 2D ellipsoids to examine fabric anistropy [87, 131] and recently described the use several hundred 3D ellipsoids to investigate the sensitivity of cyclic behavior to particle shape [85] and fabric anistropy in [86]. Dobry and Ng [26] provides a useful literature review in which they examine the spectrum of DEM applications, algorithms and the scale and scope of existing software. Hakuno *et al.* [44, 45, 43] use several thousand discs to model a variety of physical processes including slope failure, retaining wall behavior, pile penetration, sand liquefaction and fracture of cemented materials. Ishibashi *et al.* [59] provide experimental validation for assemblages of glass spheres and numerical models containing approximately 1000 spheres. Kuhn and Mitchell develop quasi-static and creep models for granular soils using approximately 1000 spheres [68]. Trent and Margolin [123] examine cementitious geologic materials using a system of several hundred discs and a parametric model for intergranular bonding. Issa and Nelson [60] model geologic materials using polygons to describe arbitrarily shaped 2D objects. They cite computational resources as limiting the number of objects (200-300) used in their work.

Rock and Ice Mechanics

Williams *et al* [129, 84], Worgan and Mustoe [137], Hocking *et al.* [33], use several hundred tetrahedral 3D elements to model ice floes in Arctic environments. Barbosa and Ghaboussi [5] describe the use of quadrilaterals elements to model jointed rock masses. They quote a cost of 30 minutes on a high end workstation to model less than 100 elements. Shi investigates similar systems using a scheme he calls *discountinuous deformation analysis* (DDA) [57, 56]. The method uses a global stiffness matrix formulation and an implicit time-stepping scheme to update the system. To ensure kinematic compatibility, the global system must be solved repeatedly for each time step, perhaps negating the value of large time steps permitted by the implicit scheme. Most DDA simulations reported contain only small numbers (100s) of elements, typically 2D quadrilaterals. This would appear to be a constraint of the global matrix formulation which scales extremely poorly for systems containing large numbers of elements.

Process Engineering and Granular Flow

An important process in chemical and manufacturing industries is the transport of granular materials, particularly in the form of a powder suspensions. Walton investigates high speed chute flows using several hundred inelastic spheres [125]. Thornton [116] uses spheres to model process engineering problems such as hopper flow and powder deposition, and has also investigated agglomeration processes using several thousand spherical elements [117]. Hogue develops a system that supports arbitrarily shaped 2D objects [52]. Simulation of a short bin flow experiment containing less than 100 2D objects required 2 hours of computation on a Next workstation.

Mining and Blasting

Preece investigates rock blasting mechanisms using several thousand disc elements in the *distinct motion code* (DMC), [110, 98, 99, 97]. He quotes four seconds of simulated time using 4000-5000 discs required 420 CPU seconds on a CRAY XMP/416 for the simulations performed [110]. In other applications he lists 1960 spheres as requiring 5300 CPU seconds on a SUN Sparc-2 [99] and 1521 discs required 1191 CPU seconds on a SUN Sparc-10 [97]. Munjiza *et al.* use several thousand 2D (triangular) elements to model bench blasting phenomena [82, 83]. In both cases they cite CPU time as the factor limiting the scale of model examined.

Physically Based Modeling and Animation

In recent years there has been a great interest in developing computer animation systems that incorporate some level of physical realism, [95, 81, 7, 42, 78, 69]. These sorts of application involve modeling small numbers of complex objects as opposed to the large scale systems discussed above. Multibody techniques are used in real-time environments such as medical simulation using small numbers of complex 3D objects [19] and in Computer Aided Design (CAD) [134, 133, 128]. Goyal *et al.* report a 3-D simulation system in which they model small numbers of convex polyhedra [37, 38].

Summary

From this brief examination of current DEM implementations, two points are noted. The first is the widespread use of simple primitives, namely discs and spheres. The

second notable observation is the size of the simulation performed. Again, for all but a few cases, the number of objects used in typical simulations number in range of hundreds to the low thousands. Both items reinforce the notion that the computation time in DEM analysis is exececssively large. For much the same reasons the type of analysis performed is usually limited to treating 2D problems.

1.5.1 Parallel Implementations

References to parallel DEM are reasonably common, documented implementations are significantly fewer in number. Ghaboussi *et al.* describe a DEM system that utilizes neural networks to identify contacts [36]. Meegoda and Washington develop a parallel version of Cundall's TRUBAL code for spheres [77]. They present times for 200 spheres on a Connection Machine CM-2 with which they obtain a speedup of 5 over simulations run on a VAX 8800. The representative parallel efficiency was not mentioned. Hustrulid at the Colorado School of Mines has performed timing tests for up to 625 discs on a Transputer with a parallel efficiency of less than 50%. It is not clear whether the parallelization would only become more beneficial for larger numbers of objects.

A related, and important class of parallel multibody techniques arise in the field of Molecular Dynamics (MD). In these simulations the objects are usually simple point charges or masses. The computational complexity arises from the multiplicity in the number of particles. As described earlier, for a system of N particles, traditional multibody techniques require $O(N^2)$ operations per iteration. In recent years a number of powerful *hierarchical tree-codes* have been developed to significantly reduce this cost. The most notable development is perhaps the Rohklin-Greengard algorithm, [40]. The cost of their algorithm is shown to require $O(N)$ operations. The cost hidden in the coefficients of the complexity expression will however be quite high. These costs arise in the evaluation of the multi-pole expansions of net charge assoicated with clusters of particles distant from one another. For short-range problems (as typified in DEM applications) the advantage of the $O(N)$ algorithm may not be forthcoming.

Distributed N-Body codes have been implemented for particle systems consisting of point masses or charges in a variety of areas including astrophysics, fluid dynamics, materials and chemistry, [126, 103, 138, 55, 54]. The cutoff for parallel over sequential

methods may be as high as 70,000 particles due to the serious problems associated with minimising communication between processors. As the particles are of small extent, foe example point masses, it is generally straightforward to subdivide the particle system by regular tree subdivision of the problem domain. For DEM applications the spatial extent of an object (the space occupied by the geometry describing the object) may span the subspace associated with a partition in the tree. This problem is discussed in more detail in Chapter 7 where an adaptive space subdivision is performed by sorting the objects along each Cartesian axis, and then building rank tables much like the tables of Salmon et al, [126].

Cellular Automata

A particularly attractive means to model granular media are cellular automata. Cellular automata (CA) are mathematical models of dynamical systems in which space and time are discrete. The state of each site is described using a minimal amount of information (typically a few bits of memory). Each site on the space-time grid evolves according to a set of rules which specify one of a finite number of discrete states. Hopcroft and Ullman [53] use the control mechanism of an elevator as an intuitive analogy of a finite state automaton (corresponding to a single site of the space-time grid). To determine the future state (location) of the elevator one examines the current location, the current direction of motion (up-down) and some number of outstanding requests for the elevator. These items of information are then combined in the form of a rule to determine the next state (destination) of the elevator.

Cellular automata can be thought of as a generalisation of the elevator analog in which *all* sites in the space-time grid are updated *synchronously.* The state of each site in the space-time grid is derived from a finite number of previous states and the current state of a fixed number of nearest neighbor sites. This is similar to finite difference schemes used to solve partial differential equations. Toffoli examines these analogs in [122]. The behavior of a physical system modeled in this fashion will be exact in the discrete case and arise out of statistical mechanical measures in the continuous case.

These methods are computationally elegant in that they are naturally manifest in the form of massively parallel computer hardware while the compactness of their representation allows models containing several million sites to be modeled using con-

ventional amounts of memory. The *cellular automata machine* (CAM) developed by Toffoli and Margolous is perhaps the most elegant realization of such an architecture, [121]. They develop a broad collection of techniques and applications for this machine, some previously described in more technical detail in the references below. Their treatise is particularly readable, giving a practical introduction to the physical modeling using CA and elicit many constructive directions for non-specialists to follow. Hillis' Connection Machine [50, 51] presents a universal CA machine that additionally incorporates non-local communication capabilities.

Although CA have been a theoretical interest in mathematics and computer science since the 1940s, a broader understanding of their complexity and application was introduced in the early 1980s, most notably by Wolfram [135]. A common application of CA in physical modeling has been in fluid dynamics, made popular by researchers such as Brosl Hasslacher [48], Salem *et al* [136] and Margolus [75]. More recently Molvig *et al.* [79] and, Teixeira [111], develop a powerful class of multi-speed lattice gas automata that capture macroscopic continuum hydrodynamics *exactly*.

Discrete materials such as granular soils have also been modeled with some success. Bak and Creutz develop a 2D cellular automaton model to examine critical dynamics of dry sands in [3]. Rothman examines lattice gas models of 2D fluid flow in porous media in [102], and Olson and Rothman [92] model binary fluid flow in both 2D and 3D. Chen *et al.* dicuss 3D lattice gas flow models in porous media similar to sandstones in [20]. An appealing characteristic of CA techniques in the context of modeling flow in granular media lies in the ability to define very complex boundary conditions at the pore scale which are not possible using volume averaged techniques. Doolen [27] contains a useful cross section of lattice gas automata research and applications. Gutowitz [41] provides similar coverage of the research directions being pursued in developing cellular automata theory.

The principal limitation of current CA based techniques is due to the complexity involved in using them to model heterogenous systems. Because the resolution of the representative elements is uniform, it is difficult to model systems that span several scale lengths.

1.6 Defining a Framework

This thesis describes the development of a multibody simulation environment that will allow us to examine the behavior of a model material, at the microscopic level. Granular materials are represented as a system of complex, discrete objects, that interact through surface contact. New algorithms are developed to permit large scale simulations to be performed at resolutions considered intractable using traditional simulation methods.

The computational complexity of DEM is in some way complementary to the physical characterization required to effectively simulate granular media. Physical characterization requires that the system being modeled contains a sufficient number of objects to be representative in some statistical sense. Within this representative sample the shape of the individual particles must be represented to some level of accuracy or resolution. They are thus both computational requirements and modeling requirements and are summarized as:

1. Generality - That arbitrary geometries are representable.

2. Scalability - That large numbers of objects can be modeled.

This is the complex side of the simulation system and leads to the principal contribution of this thesis: *To reduce the computational complexity of multibody simulation*[2].

1.6.1 The Prototype Simulation System

The collective system developed in this work is called χ*mal*[3]. The intended use of the system is the examination of the behavior of granular materials, such as sands. The environment is likened to a simulated laboratory in which to perform *experiments* on samples of a material represented as collections of discrete objects. Each object has its own geometric and physical description which defines how it will interact with other objects. Experimental apparatus is described using objects whose

[2]This should not be confused with the complexity of the physical system being simulated. The real system is only computationally complex through mathematical perspective. Reducing the complexity of the mathematical perspective does not imply that results from the system will be more correct, it simply means hypothesized models for the physics of such systems can be tested more rapidly. This is still an important goal.

[3]Pronounced *chi*-mal, from the German for *many*.

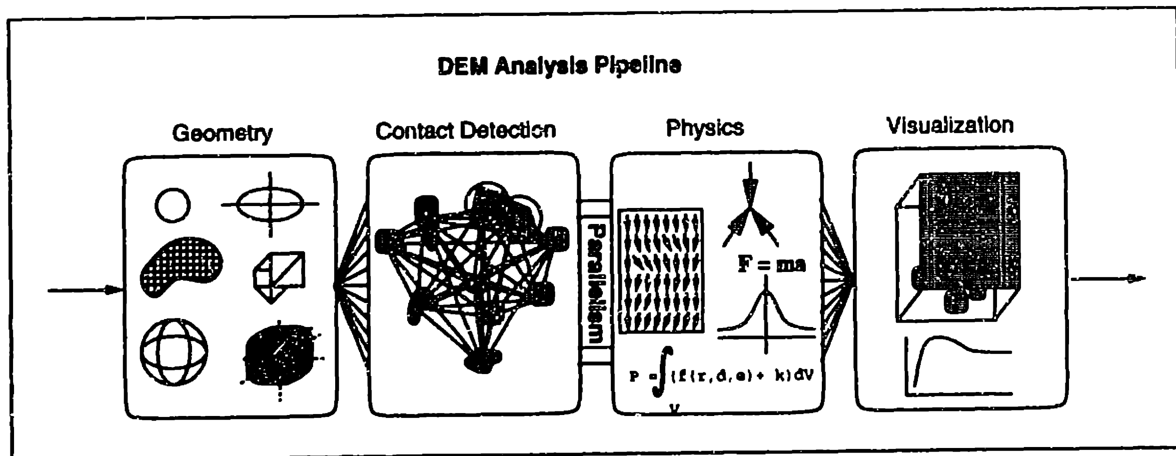

Figure 1.4: Analysis Pipeline

geometry describe containers, loading plates, valves, supports and so on. Experimental measurements are made by examining displacements of the boundaries and by interrogation of individual objects in the sample.

To implement the simulated laboratory, the following list of specifications are used:

1. Provide support for the generation and representation of a general class of 3D objects.

2. Provide a mechanism to automate *contact detection* between the objects.

3. Incorporate channels for visualization and temporal characterization of multi-body simulations.

4. Provide support for user interaction with the simulation.

These items encapsulate the components of an DEM analysis pipeline which is represemted in Figure 1.4. It is assumed for generality that this takes place in a parallel computing environment, implied by the fanning out of the pipeline through the contact detection and physics phases.

1.7 Thesis Outline

The components of the χmal system are described in the following chapters of this thesis. Figures 1.5 and 1.6 describe mappings between the various levels of the system and the chapters that describe them. The shaded regions indicate the areas in which this research has focused and contributions have been made.

Chapter 2 deals with sorting strategies used to spatially organize objects in a multibody simulation. Sorting the objects is performed in an attempt to reduce the amount of interbody coupling that needs to be accounted for in the numerical scheme. Essentially it seeks to reduce the number of detailed contact tests by identifying objects that are distant from one another. The most commonly used methods are reviewed on the basis of generality and complexity. Two categories of sorting schemes arise. The first is based on *a priori* assumptions about how the system being simulated evolves and lead to very compact and efficient methods. Methods in the second category make no such assumptions, and are referred to as *exhaustive* schemes. This classification is much the same as that found in the literature on sorting algorithms. Each of the individual methods will have some quality that appeals to the developer, leading them to chose one over the other. The most important of these factors are examined with a view to their impact on the system developed in this work. An exhaustive scheme based on the standard sorting algorithm, heapsort, forms the spatial sorting strategy for the χmal system. The sorting algorithm and its implementation for spatial sorting is explained including an example of it's use.

Chapter 3 forms the core of the thesis. It describes the derivation of a new representation scheme used to describe the surface geometry of objects. The driving force behind this development was to reduce the computational cost of performing contact resolution (the detailed stage of contact detection). The scheme is referred to as the *discrete function representation* (DFR). Section 3.2 describes the first implementation of the DFR scheme, which has been used to represent 2D objects. A contact resolution algorithm based on the 2D DFR scheme is then developed. The performance of the 2D contact resolution algorithm is compared with that of a method frequently used for contact resolution in physical simulation. A dramatic speedup is achieved, particularly for complex object geomteries. The results of a set of timing tests are presented and the complexity of the algorithm discussed.

Section 3.3 extends the concepts developed in Section 3.2 to derive a DFR scheme for 3D surfaces. The 3D DFR scheme is more general than its 2D counterpart and is capable of representing a comprehensive range of surface geometries including objects with holes. The scheme is described in terms of a method to capture 3D surface descriptions and how they are stored in the 3D DFR data structure. Details of the 3D DFR contact resolution algorithm follows, along with a discussion of expected running time for the method.

In Chapter 4 the general set of geometric primitives in the χ*mal* system are introduced. The various couplings (through contact) that each object may have with other types of primitive are discussed with a view to efficiently managing contact resolution between them. When a pair of objects come into contact, a computational abstraction called a *contact object* is created. This entity contains information about the objects and details describing the geometry of the contact region. The management strategy underlies the contact detection process whereby transient *contact objects* are identified and tracked over time. Ultimately *contact objects* are reduced to defining the contact forces necessary to keep pairs of objects from interpenetrating. A contact model based on Coulombic friction is described with details of the implementation for the various object couplings that are possible.

Chapter 5 describes the remaining components necessary to implement the software system. These include detailing the coding strategy, the hierachy of software primitives, the time integration scheme, the vector-matrix transform library, the graphics subsystem, and the user interface. Integrating the various parts yields a version of χ*mal* suitable for use on a serial computer.

Chapter 6 provides a short but general introduction to parallel computation. It is intended to be a self-contained chapter much like a tutorial. Its purpose is to familiarise the reader with some of the concepts on which the distributed χ*mal* system is based. The various classes of parallel computers are described, along with architectural characteristics that impact on software development (such as communication networks). Programming models used with these machine are then reviewed with coded examples in many cases. The chapter closes with an examination of new industry standard for the parallel programming model known as *message passing.* A short introduction to a portable software library to support this programming methodology is given, again with coded examples.

Chapter 7 completes the development by incorporating parallelism in the DEM modeling environment. The chapter starts by examining results of timing tests for parallel sorting. A parallel DEM algorithm is then developed and the performance of the scheme evaulated in terms of timing tests for various sizes of problems performed on different platforms. The consequences of these tests are then discussed.

Finally in Chapter 8 a series of DEM applications are described. These examples both test the capabilities of the system and introduce several additional methods used to solve several pratical problems arising in computational materials and other domains. In the second half of the chapter the capabilities of the simulation system are summarized by way of concluding the thesis and to point to future research directions.

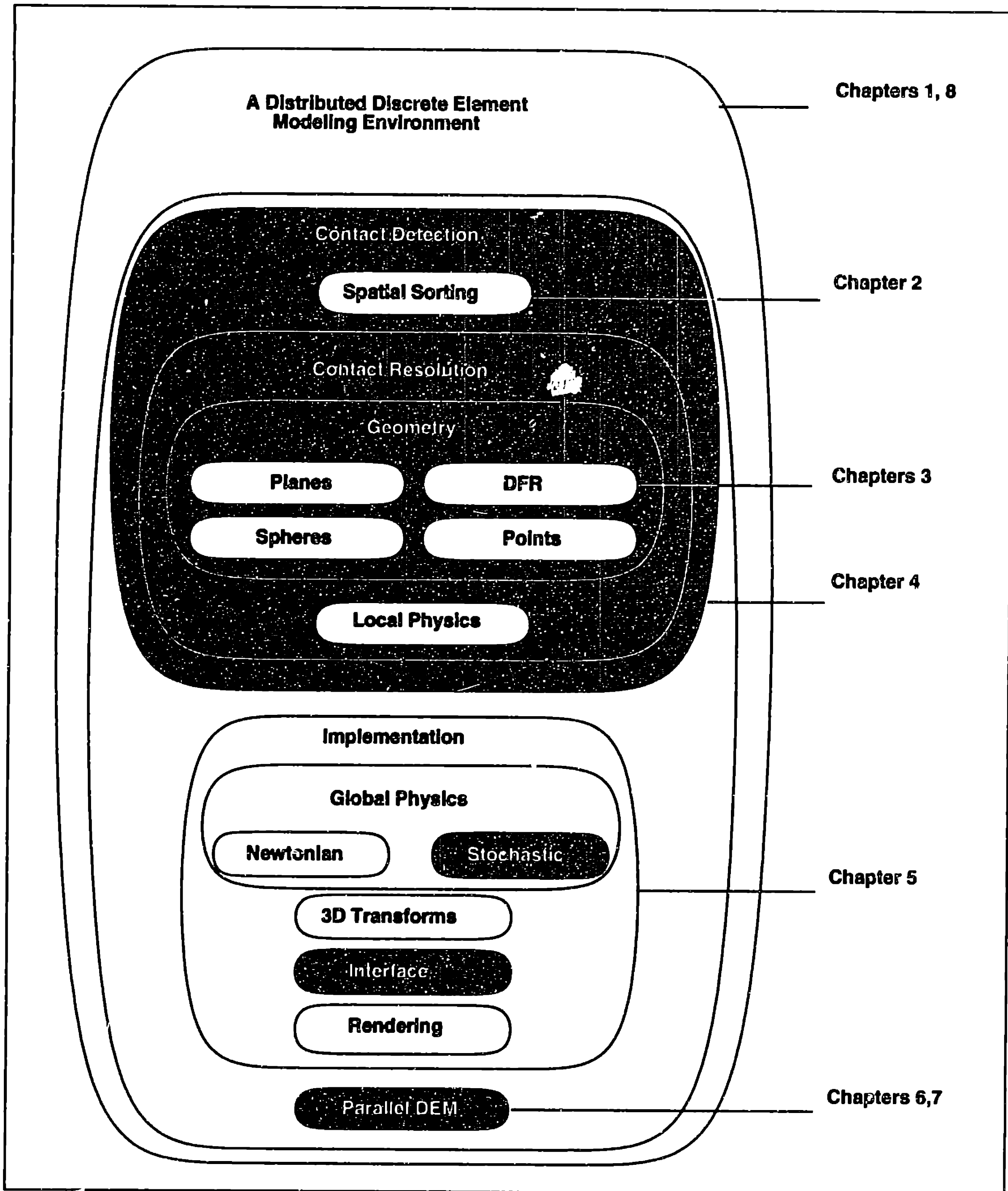

Figure 1.5: Chapter Hierarchy Plan-View

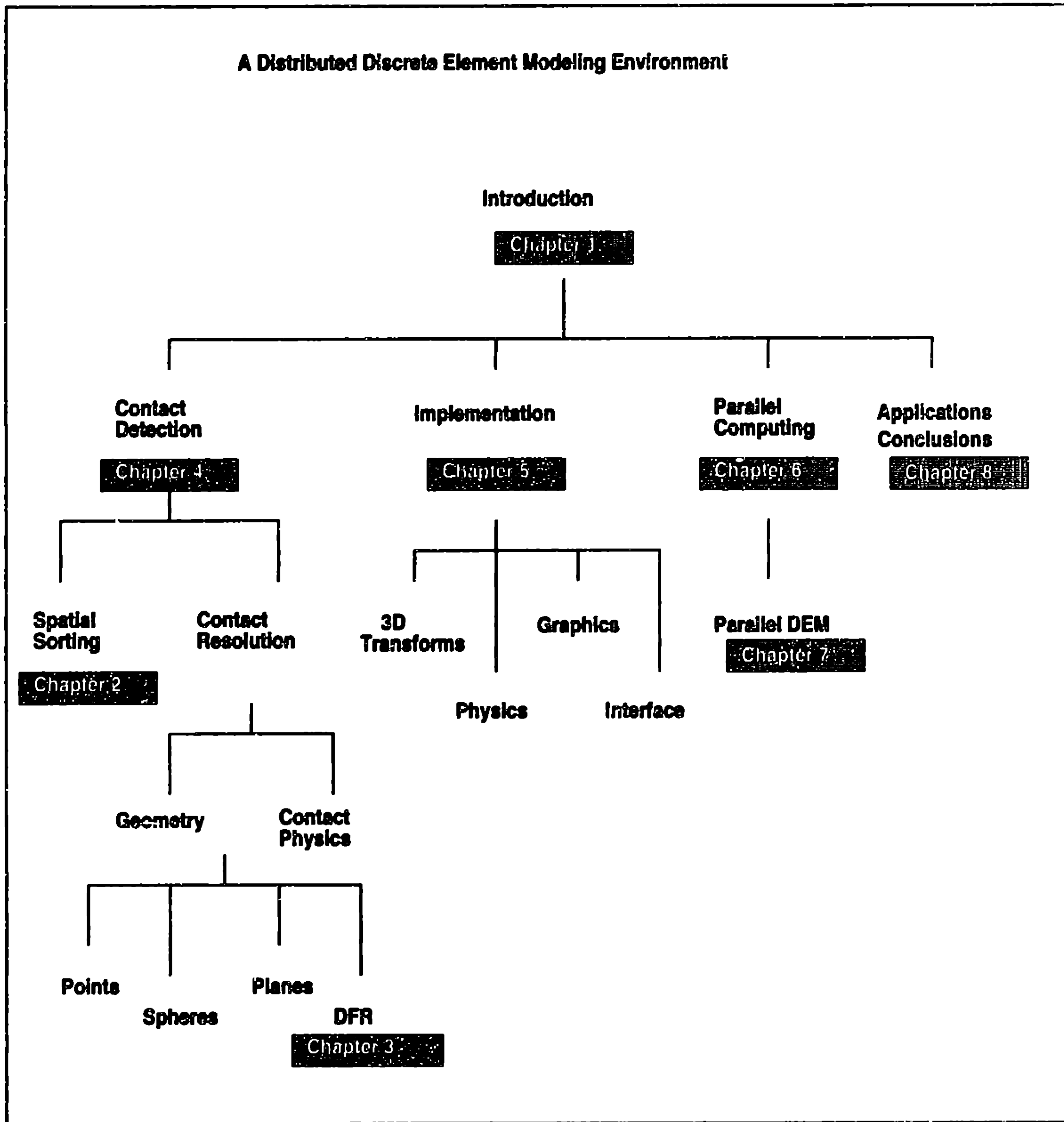

Figure 1.6: Chapter Hierachy Front-View

Chapter 2

Sorting for Contact Detection

To simulate the behavior of large multibody systems the principal computational cost lies in evaluation of the pairwise interaction of the objects. The interaction is due to the influence of *long range* potentials (Coulombic or gravitational), and/or the *short range* coupling of objects through boundary potentials or surface contact.

In the case of granular materials interaction principally occurs through surface *contact.* This is the primary mode of interaction considered in discrete element analysis. The process of identifying surface to surface contact is referred to as *contact detection.* It seeks to determine the conditions necessary for objects in the simulation to remain disjoint. The disjoint condition is simply the physical requirement that objects cannot overlap.

2.1 A Top Down Approach

Algorithmically *contact detection* is viewed as consisting of two distinct phases, *spatial sorting* and *contact resolution.*

Spatial sorting addresses the problem of deciding which pairs of objects should be considered for detailed contact resolution. It seeks to avoid conducting an exhaustive set of checks by spatially ordering the relative positions of the objects. The ordered set of objects is then *searched* for candidate pairs of objects on which to perform the second phase, contact resolution, a detailed check for contact.

Contact resolution determines if two object are in contact by calculating if their surface geometry intersects. It attempts to capture a geometric description of the

contact region and use this information to derive the contact forces necessary to maintain the objects in a physically realistic state, i.e. the *disjoint condition*. The treatment of contact resolution between objects with complex surface descriptions are discussed in Chapters 3. The implementation of the collective *contact detection* process and the evaluation of the contact forces at the interface are described in Chapter 4.

This chapter examines how *spatial sorting* for contact detection is performed. We start by reviewing several of the more common spatial sorting techniques that appear in the literature. A component common to all of these methods is that they perform some type of *sorting* operation. There are well tested techniques developed in computer science for sorting and searching data structures which have been adapted to the contact problem. Details of these methods can be found in any number of textbooks on algorithms, such as [66, 115, 107]. The spatial sorting methods are discussed in terms of the complexity of the sorting algorithms that underly them. This viewpoint is intended to provide a common basis to compare the relative merits of each method. The method used in this work is based on the sorting algorithm *heapsort*. This method is examined in some detail, with an example of its use.

2.2 Spatial Sorting Algorithms

Techniques to spatially organise sets of objects arise in a variety of contexts. These include multi-body dynamics, such as Discrete Element Methods (DEM), [127, 109, 118, 10], in particle systems used in celestial mechanics, fluid dynamics and molecular dynamics [126, 103, 55, 54, 9, 138, 40], geometric modeling [74], robotics [15, 76], computer graphics [100, 61, 7, 124, 39, 104, 105, 49] and more recently in graphical databases, such as Geographical Information Systems (GIS), [105]. In this section we review some of the more commonly used methods, several of which are depicted in Figure 2.1. The complexity terms specified in the sections below reflect the *average* running times for the algorithms. The nature of the specific application will nearly always introduce additional terms which will bias or dominate the expected running time. Where possible a *worst case* running time will also be noted.

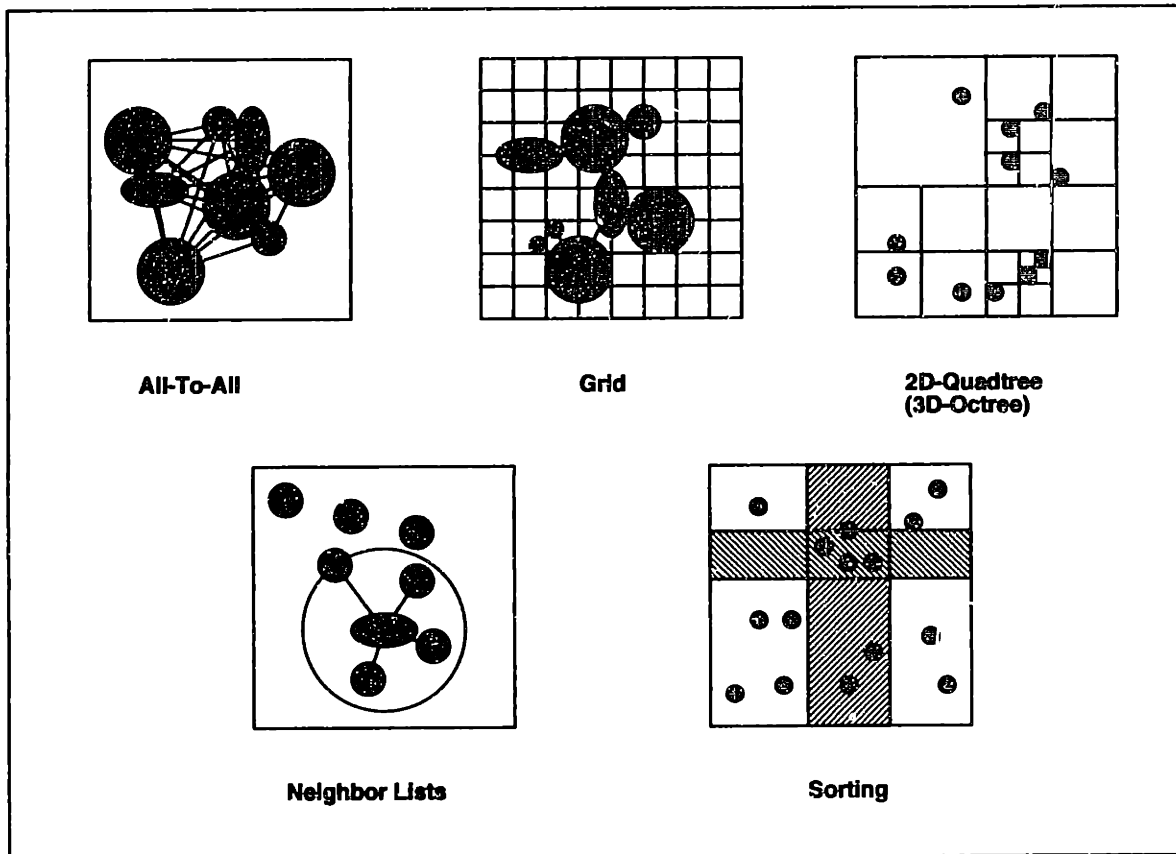

Figure 2.1: Various Sorting Strategies

Grid Subdivision

The grid subdivision method uniformly discretizes the simulation volume into rectilinear cells. Each object is superimposed on the grid and the cells which it overlaps are calculated. Each cell then maintains a list of the objects that fall within or cross its boundaries. Constructing the grid of cells in the form of an array, it can be treated as an adjacency matrix for direct access to neighboring objects. If the cell dimension is choosen to be greater than the largest object dimension, only 9 cells (itself and 8 neighbors) need to be examined for neighboring objects in 2D (27 cells in 3D). This method performs best in situations where the spatial distribution of objects can be assumed to be uniform and each one preferably of similar size. This ensures that the grid of cells is also uniformly populated. These assumptions are essentially those of the *bucket sort* algorithm which has an average running time of $O(M)$, where M is the number of objects being sorted.

When performing contact detection, each cell containing a list of objects is examined. For a given cell, a set of all-to-all contact checks are performed between objects in the same cell and all of those in the directly adjacent cells. For a grid of P cells and a uniform distribution of objects, the average number of objects in a cell list will be M/P. The average running time for contact checks in all P cells is then $O(M^2/P)$. Obviously, if the number of cells containing objects decreases, the worst case behavior of the algorithm will become $O(M^2)$. This has the additional effect of wasting storage space for $P-1$ cells. For instance, this would happen if the spatial distribution or the sizes of the objects vary significantly over the simulation volume. It then becomes a tradeoff between maximizing the cell sizes and minimising the the number of objects per cell. Unfortunately the assumptions made for uniform gridding rarely hold and it is more often the case that some objects will extend over several cell dimensions and/or the distribution of objects over the simulation volume is not uniform.

The grid method is used extensively for simulating systems that are said to be *quasi-static*, where the objects are initially evenly distributed and only undergo small displacements over the duration of the simulation. Improvements are possible over the course of a simulation, by re-sorting objects only when some object has move a distance greater than a prespecified distance (such as the dimension of a cell).

Grids are used in *particle in cell* (PIC) codes used to perform fluid dynamics simulations. These applications can contain several million particles, so even a small number of coarse cells can dramatically reduce the amount of computation necessary (compared with an $O(M^2)$ method). Spatial distribution is still important however, as particles will typically undergo large displacements and move through several cells over the course of a simulation.

Grid subdivision is particularly straightforward to implement as a parallel partitioning technique, where a processor is assigned to manage the contents of a cell or a contiguous group of cells. The effects of the spatial distribution of objects is then felt in the form of load balancing, where some processors may end up having to process much larger number of objects than others. This problem is addressed in later chapters.

Adaptive Gridding

Adaptive grid methods can be used to avoid some of the problems associated with spatial heterogeneity, albeit at the extra cost of having to fully sort the data set. In this technique the simulation volume is subdivided by cutting planes parallel to the principal Cartesian axes. This is done such that, for each subdivision of the simulation volume, there are approximately the same number of objects on both sides of the cutting planes. Since the positions of the partition (cell) boundaries now depend on the spatial distribution of the objects, the objects must first be sorted along the principal axes. Sorting the objects can be done using any of the standard algorithms, typically one of quicksort, heapsort or mergesort. These algorithms are known to have average running times of $O(M \log M)$ and generally characterize the complexity of the adaptive gridding method.

To search for pairs of objects that may be in contact, *all-to-all* checks are performed between the objects residing in each partition and, if necessary, those surrounding it. For a grid of P partitions, the average number of objects in each will be M/P. The average running time for contact checks in all P partitions is then $O(M^2/P)$. In contrast to the fixed grid scheme, this behavior will not deviate as much since the spatial distribution of objects is now matched by the distribution of partitions. However, if the object sizes span multiple partitions this will increase the number of checks that are necessary.

The scheme is probably least efficient where the state of the ensemble evolves from a heterogeneous to homogeneous distribution with respect to the simulation volume. In this case the work involved in maintaining a uniform distribution of cutting planes exceeds that of the fixed gridding approach. To avoid this problem it may be expedient to include a mechanism to track the motion of individual objects causes them to move into, or out of, a new partition. If they do not cross this threshold condition then the partitons do not need to be remapped and the average search time remains optimal.

Binary Space Partition Trees

Binary Space Partition Trees (BSP) are essentially a generalisation of the adaptive gridding technique just described. Instead of partitioning the simulation volume parallel to the principal Cartesian axes, the separating planes can have an arbitrary orientation. This provides a means to subdivide the set of objects more effectively, where more objects can be assigned to be wholly to one side of a given separation plane.

As in the adaptive gridding method, the objects are sorted in some manner before being partitioned. Again, the sorting method is characterized by an average running time of $O(M \log M)$. In this case there will be a distinct increase in the magnitude of this expression to account for the additional work required per object. This is due to having to perform additional tests on each object to determine where the best separating plane should reside. The method is considered to perform well if the set of objects does not move much and the partitions can be maintained over the simulation. This is much the same situation that occurs for adaptive grids. The same motion tracking heuristic can be applied to BSP ordering with increased benefit over having to remap objects. Because of the additional overhead creating the BSP trees this method is perhaps less suited to dynamic simulations where objects tend to move through large displacements.

k-ary Trees

Extending the ideas of both fixed gridding and, adaptive grids and BSP trees, k-ary trees attempt to capture some of the efficiencies of each method. Space is partitioned by recursively subdividing the simulation volume into cells that contain only small numbers of objects. In 2D, space is divided into quadrants and the tree is called a *quadtree.* In 3D it is called an *octree.* The method treats the simulation volume as consisting of rectilinear cells. Only those cells which contain objects are maintained in the tree structure.

Searching is heavily dependent on how the octree is first constructed. By minimizing the depth, hence, balancing the branches of the tree, the search can be minimized. The ubiquitous comment made by Knuth suggests that insertion into the tree should be done in a random fashion when using quicksort. This is to ensure a relatively uniform distribution of objects, i.e. a balanced tree. This occurs for much the same

reasons that the randomised strategy is recommended for quicksort when sorting pre-ordered data sets, [66]. The time to create the octree is of order $O(M \log M)$ and the time to search this tree is also of order $O(M \log M)$ (for a balanced tree and where cell size is small compared to the full space). As with BSP trees, the spatial subdivision method is better suited to partition the objects into disjoint groups or clusters, but will involve additional overhead when it has to maintain multiple entries for objects that span several cells.

Neighbor Lists

An alternative approach to subdividing the simulation volume is to base the center of the search cell about an object's centroid. Any objects lying within the cell are deemed to be candidates for contact checking and stored in a *neighbor list.* The neighborhood cells are usually circular or rectilinear. One of the earliest applications of this method was by Verlet, where it was applied to molecular dynamics problems.

To identify an object's neighbors the entire set of objects is first sorted. This can be done using one of the gridding schemes just described or by sorting along the coordinate axes. This is the method utilized in this work. The following section describes it in some detail, and includes an example of its use.

2.3 Spatial Sorting Using Heapsort

The method chosen to spatially manage objects in the simulation system is referred to here as *Spatial Heapsort.* The basic idea behind the method is to search an ordered set of objects clustered close to one another and pair off for contact resolution. Objects are first sorted by increasing ordinates along the principal axes of the simulation volume. Heapsort is one of several algorithms that can be used to sort the objects into an ordered list. The *heapsort* algorithm is explained below. In the case of spatial sorting the sort key is a bounding sphere extent parallel to the 3 Cartesion axes.

The method described here is based on that developed by J. Swegle of Sandia National Laboratories, [109]. The original report detailed the implementation as used in *smooth particle hydrodynamics* (SPH) applications. In these applications the objects are of a uniform size and compact extent, which allows for very efficient sorting and searching. For generality the algorithms have been adapted to account for

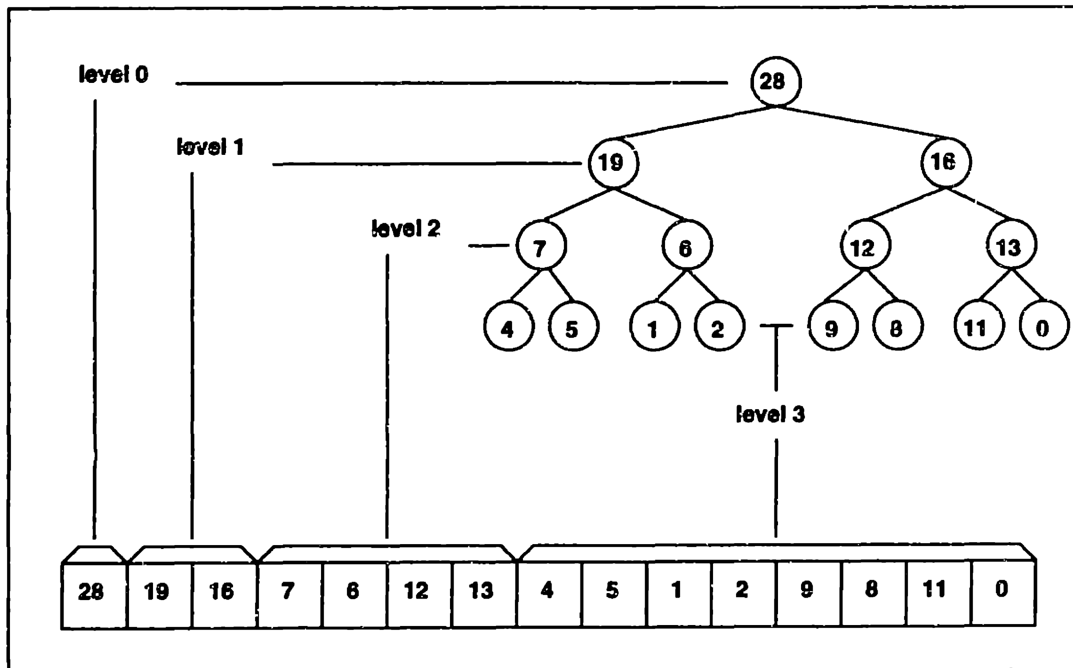

Figure 2.2: Binary Tree Representation of a Heap

non-uniform spatial distribution of objects of any size. These extensions increase the amount of computation required per object by a constant factor but do not change the running time complexity of the algorithm. The extensions are incorporated in an example discussed later in this section.

Heaps - Definition and Constructs

A *heap* is generally defined as a *k-ary* tree with the property that all of the nodes below a given (parent) node contain a key value that is less than that of the parent.

For the heapsort algorithm a binary heap construct is used, as shown in Figure 2.2. Each level of the binary heap can contain at most $\sum_{i=0}^{l} 2^i$ nodes, where l is the number of levels in the tree. There is no requirement that every level of the tree be entirely full.

For efficiency the tree is stored implicitly in an array of contiguous memory, as shown in the lower half of Figure 2.2. The array is filled by traversing the binary heap tree breadth first. Starting at the root of the tree, each node is visited level by level, from left to right.

2.3.1 Heapsort Algorithm

To sort an unordered set of M keys using heapsort, the most effective procedure is to store them in the form of an implicit binary tree, i.e. an array of size M. The array elements are assumed to be indexed from $0..M-1$.

Starting at the bottom level of the tree (array element $i = M - 1$), each node i is visited level by level, up to the root at location $i = 0$. At each node, i, a *heap* is created locally by examining the value of its key and the keys in its left and right children (if present). The children of node i are located at positions $2i$ and $2i + 1$ in the array. To create the local heap, the largest of the two children is first determined. If this key is also larger than the key in the parent node, it is exchanged. If an exchange takes place, this may cause nodes at levels below the child to no longer constitute a *heap*. A rearrangement of keys in the local heap must be propogated to the remainder of the tree that lies below the exchanged child node.

At the end of the first pass through the entire binary tree, the largest element will be located at the root node (position 0 of the array). Additionally, the binary tree is now in the form of *heap*. Having identified the largest value in the tree, the contents of the root node is exchanged with the rightmost leaf node of the tree. The size of the tree is then reduced by excluding the rightmost leaf node which now contains the largest value in the original tree. The exchange of the root with the rightmost leaf node has now placed an arbitrary value in the root node. As a consequence the tree can no longer be guaranteed to be a *heap*. This is the case if it fails the 'parent greater than both children' criterion.

A *bottom up* traversal is now performed on the reduced tree to identify the 2nd largest element and the tree size is once again reduced. This procedure applied repeatedly until the tree reduces to a single element which is implicitly a heap. The sequence of values placed in the array locations lying to the right of the tree as it was gradually shrunk, are now ordered by increasing value.

The algorithm was chosen for the following reasons:

- Heapsort is known to optimally sort an unordered collection of M objects $O(M \log M)$ operations, [66, 115, 107].

- Implementation is straightforward in software, with no special data structures necessary.

- The behavior of the algorithm is less sensitive[1] to the spatial distribution of the objects to be sorted (i.e. the relative clustering/grouping of the objects in the simulation volume is unimportant).

- Searching for spatially localized sub-sets of the objects can be performed efficiently in $O(M \log M)$ operations.

- The storage requirements are $O(M)$, the algorithm sorts in place.

2.4 2D Spatial Heapsort Example

A small example of the *spatial heapsort* procedure used to sort a collection of 2D objects is described to help in understanding the implementation. The structure of the example is based primarily on that of Swegle, [109], with extensions to account for non-uniform object size and shape.

A collection of 2D objects with various geometries are shown in Figure 2.3. The lower bound extents of the object geometries (or some bounding hull, such as a box or disc) are projected onto the X and Y axes as shown. To perform the *spatial heapsort*, two arrays of integer fields are required for each dimension of the problem domain. The first array of each pair is used to store the set of object indentifiers to be sorted and is referred to as the **id** array. The second array is used to *rank* the order of the objects when they are sorted and is referred to as the **rank** array. The collection of arrays are referred to as a *sorting table*.

2.4.1 Building the Sorting Table

For a problem containing M objects, each array in the sorting table has M fields. Objects identifiers are assumed to be numbered contiguously from $0..M-1$, corresponding to the index locations of the sort table arrays. The purpose of the **rank** array is to store the index location of each object in the **id** array. When the objects are sorted the **rank** array will contain the object *ranking* (1st, 2nd, 3rd etc). The one-to-one mapping of object id and array index can then be used to *lookup* the sort

[1] The Quicksort algorithm [66], while known to perform faster in many applications, degrades to a running time of $O(M^2)$ when applied to existing ordered sets.

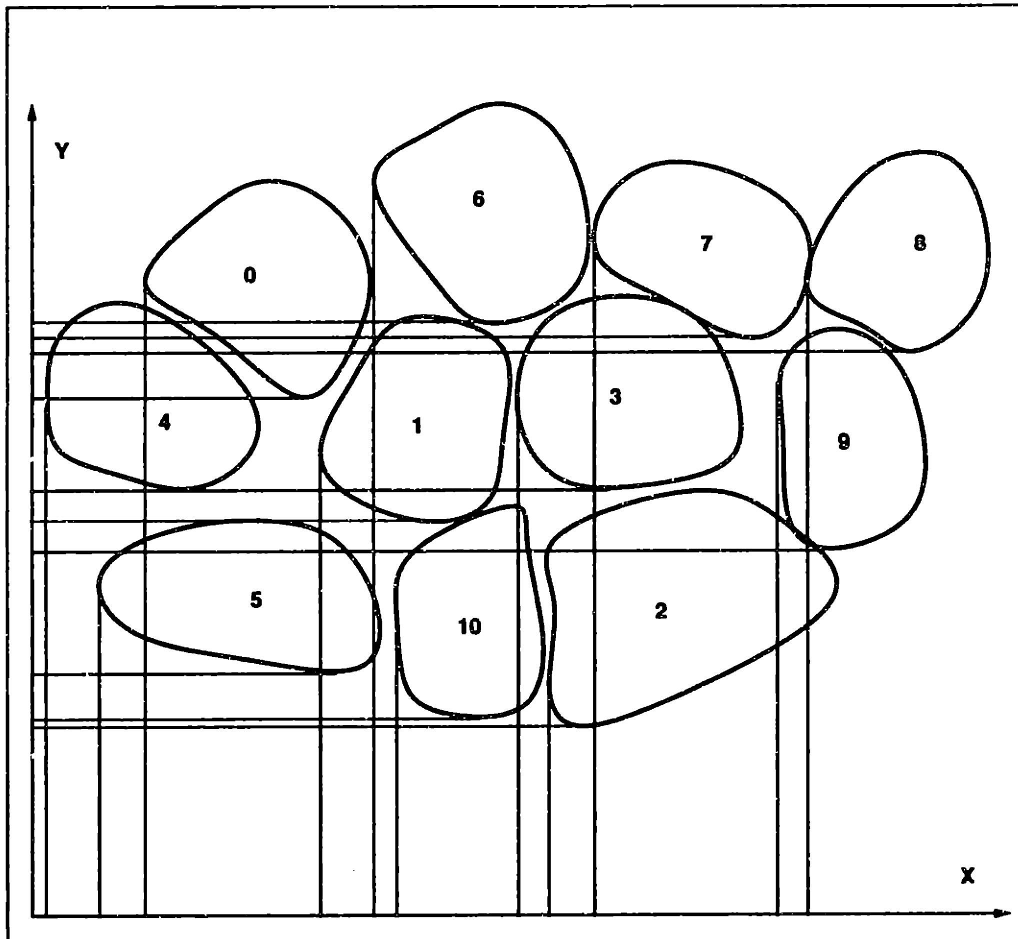

Figure 2.3: Heapsort Example

table for a specific objects location in the sorted lists. This procedure will become more evident as the example progresses.

The initial contents of the **id** and **rank** tables for thc example shown in Figure 2.3 are given in Table 2.1. Each of the **id** arrays is treated as an unsorted binary tree as described previously for the heapsort algorithm.

The heapsort routine is applied to the unsorted **id** array in each coordinate direction using the projection of the lower-bound ordinates of each object as the sort key. Entries in the **rank** array are calculated by making a single pass through the sorted **id** array. The object identifier stored in each element of the **id** array is used to index into the **rank** array and the index location of the **id** array is copied into it. After sorting in each direction the sorting table contains the entries shown in Table 2.2.

Index	0	1	2	3	4	5	6	7	8	9	10
X id	0	1	2	3	4	5	6	7	8	9	10
X rank	0	1	2	3	4	5	6	7	8	9	10
Y id	0	1	2	3	4	5	6	7	8	9	10
Y rank	0	1	2	3	4	5	6	7	8	9	10

Table 2.1: Sorting Table Before Heapsort

Index	0	1	2	3	4	5	6	7	8	9	10
X id	4	5	0	1	6	10	3	2	7	9	8
X rank	2	3	7	6	0	1	4	8	10	9	5
Y id	2	10	5	9	1	3	4	0	8	7	6
Y rank	7	4	0	5	6	2	10	9	8	3	1

Table 2.2: Sorting Table After Heapsort

2.4.2 Searching For Candidate Pairings

The next stage in the algorithm is to group together objects that are candidates for collision. An expedient way to identify candidate collision pairs is to process the sorted **id** arrays, object by object, along a designated axis. The object selected for testing is referred to as the *pivot* object. By processing the sorted list sequentially along the chosen axis, only those objects lying ahead of the *pivot* object need to be considered since contacts with objects occuring earlier in the list will have already been detected. Ultimately a set of upper and lower bound indices are obtained such that they delimit a range of candidate object **ids** within the sorted list for each coordinate direction. The following two steps detail the necessary steps:

1. Start at the index location of the *pivot*. This is the lowerbound search index. Traverse the sorted list using a binary search to identify overlapping objects in this direction. Objects overlap if largest ordinate (extent) of the *pivot* object parallel to the search axis lies beyond the smallest ordinate of objects lying *ahead* of it. Stop at the index location of the object that does not have a lowerbound extent less than the upperbound extent of the *pivot*. This is the upper-bound index.

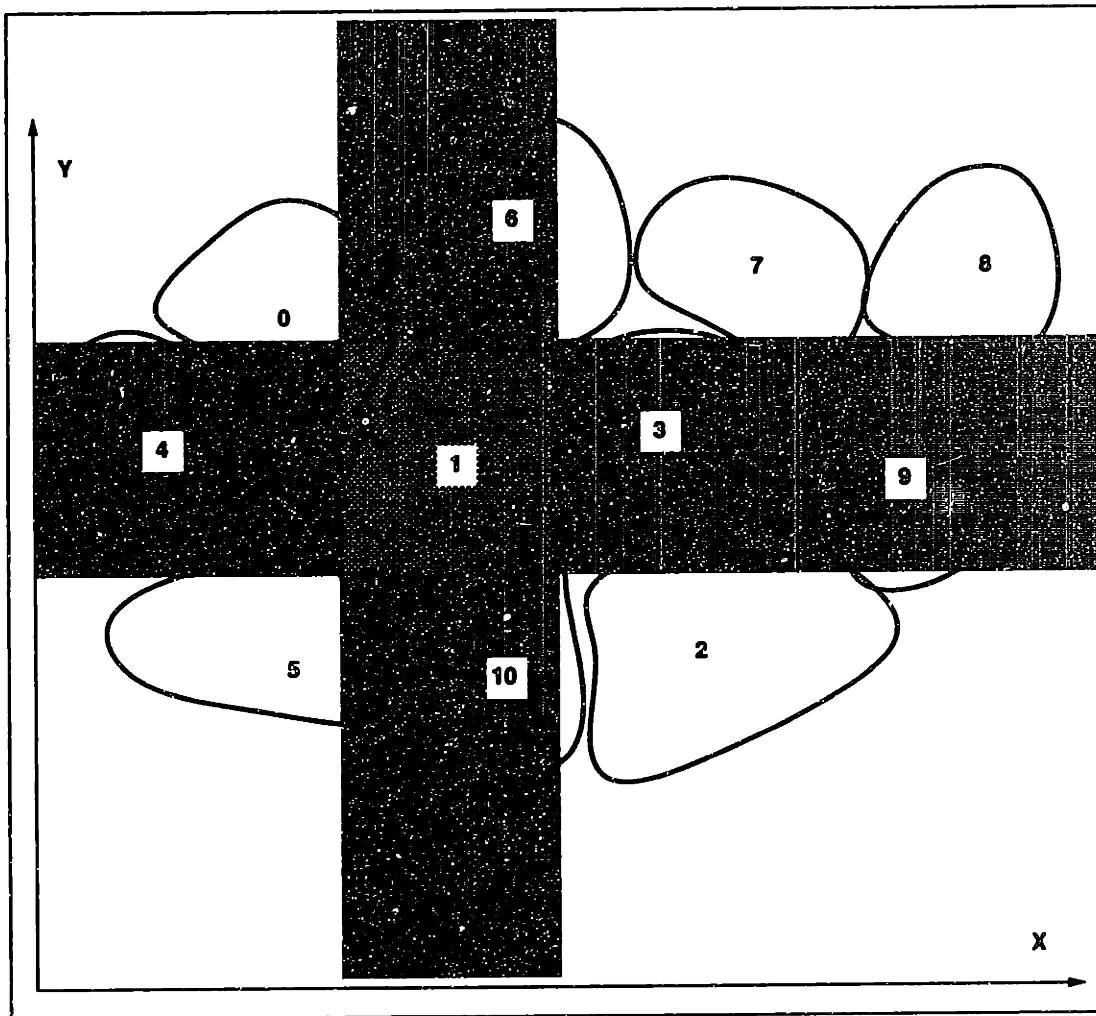

Figure 2.4: Index Intersection for the Heapsort Example

2. For the remaining axes, start at the index location of the *pivot* in these directions. Traverse the sorted list using binary search and identify *both* an upperbound and a lowerbound index.

These steps yield lower and upper bound indices for the **id** arrays in both coordinate directions. This corresponds to a set of objects possibly in contact with the *pivot* object. Contact resolution is then performed on pairs of objects drawn from the intersection of the **id** sets. The index set with the lowest number of elements is chosen to control the sequence of objects to be examined. For each index in this control list, the objects (canonical) index is found from the **rank** 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 direction(s). 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 that pair of objects.

For example, selecting object 1 in Figure 2.3 and using the X axis as the primary search axis, the index bounds from the **id** array are 4 to 6. The X axis index bounds correspond to objects {6, 10, 3}. Along the Y axis both directions must be checked since X is the primary search axis. The index bounds here are 0 to 10. Full contact resolution for object 1 is then performed with the intersection of the index sets which yields the object identifiers {6, 10, 3}. The intersection is shown as the darkly shaded region in Figure 2.4.

2.5 Summary

In this chapter the *spatial sorting* component of *contact detection* was examined. Several of the more commonly used schemes in multibody systems were described in terms of their scope and running time performance. An exhaustive spatial sorting algorithm based on *heapsort* was discussed in detail. This is the scheme adopted in the χmal system. The choice was made on the basis of efficiency, generality and robustness. The scheme is be used in conjunction with *neighbor lists* which are developed in Chapter 4.

Chapter 3

The Discrete Function Representation

The *spatial sorting* methods described in Chapter 2 are used to identify pairs of objects that lie close to one another. This is the first phase in the *contact detection* process. The second phase, *contact resolution*, then identifies whether the two objects overlap in some geometric sense. This is determined through an analysis of the objects' surface geometries. If the surfaces are found to intersect then the objects are deemed to be in contact. The portions of the surface where the objects intersect are collectively referred to as the *contact region*. In DEM modeling the geometry of the contact region is used to calculate contact forces, applied to keep objects from further inter-penetrating.

The computational cost of performing contact resolution is intimately related to how the surface geometry is represented. For all but the simplest of objects this accounts for a significant proportion of the computation required in contact detection. It has been found that contact detection accounts for as much as 95% of the total computation in DEM applications[1] [23, 127, 52, 132]. For this reason a new object representation scheme is developed where the goal is to facilitate efficient contact detection for arbitrarily shaped objects. The scheme is called the *Discrete Function Representation* (DFR) .

[1]The contact detection problem can also be found under the guise of *clipping* in computer graphics [100, 80, 49, 39, 124], as *motion planning* in robotics [15, 106, 76], manufacturing [71], physically based modeling for animation [42, 81] and related fields such as virtual reality [94, 29]. The development of efficient techniques to identify overlapping objects are perennial research topics in these and many other fields of study.

The representation scheme is first described for 2D objects (2D-DFR) and then extended to a generalized 3D scheme (3D-DFR). Both versions are presented to document the chronological development of the DFR scheme.

The 3D-DFR follows directly from the 2D scheme, but extends the class of geometries representable. This is achieved through an improved data structure used to store the surface geometry. A 3D-DFR contact resolution algorithm is then developed for a general class of contact problems.

3.1 Object Representation

Two basic forms of object representation arise in practice, the boundary representation and the functional representation. Boundary representations discretize a surface by treating it as a collection of simple, interconnected, surfaces. Implicit (functional) representations describe the surface geometry as a mathematical function.

The cost of performing contact resolution with boundary representations is related to the number of facets used to approximate the surface geometry. In the functional representation the cost is related to the complexity of mathematically calculating the intersection of the surfaces. For a number of functional representations analytic solutions are available. It is more often the case that the intersections cannot be found analytically. Numerical schemes that sample the surfaces are then used to find the region of overlap.

Boundary Representation

The boundary representation or *b-rep* breaks the surface of each object into facets or patches. The geometry of the surface lying inside the patch is then interpolated. In the simplest case linear interpolation uses lines in 2D and triangles or planar polygons in 3D. For smooth surface approximation more accurate schemes such as *splines* are used.

For convex objects, intersection checks using unstructured boundary representations requires testing each patch from one object against each patch of the other object. In Chapter 1 this algorithm was shown to have a running time of $O(N^2)$, where N is the average number of patches per object. The basic idea of the algorithm is to identify if a *test facet* from one object intersects or lies on the interior of the

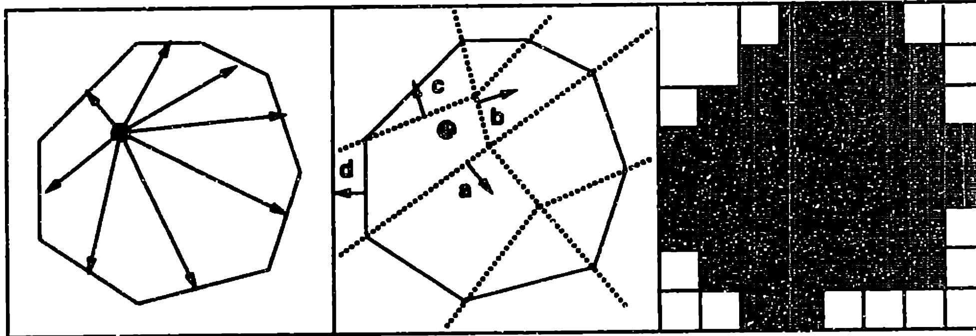

Figure 3.1: Point Classification For Various Boundary Representations

polygonal region representing the other object. The test facet must be compared against *all* of the boundary facets of the other object. Each boundary facet is visited in a fixed order, for instance a clockwise traversal. If there are N_b boundary facets, N_b checks are required. To test N_t facets, $N_t \times N_b$ checks are required in total. Assuming $N_t = N_b(= N)$ gives the $O(N^2)$ running time for algorithm. These representations are used in *computer aided design* (CAD) and *finite element methods* (FEM), and has been applied to both 2D and 3D discrete element systems [33, 127, 34].

Improved boundary representation schemes are possible through spatial partitioning methods such as *Binary Space Partition* trees (BSP trees)[63] or k-ary trees, such as *quadtrees* (2D) or *octrees* (3D) [104, 105]. Objects represented in these forms can be used to achieve $O(N \log N)$ running times for contact resolution. For k-ary trees space is partitioned parallel to a mutually perpendicular set of coordinate axes. In the case of BSP trees, space is divided by planes of arbitrary position and orientation. Often the separating planes are chosen to lie coincident with the boundary edges/faces of the object. For convex objects, the BSP tree may offer no advantage over the unstructured *b-rep* scheme for point containment tests, [63].

Implicit Representation

Implicit representations describe an entire surface geometry as a single function of the form $f(x, y, z) = 0$. For example, the surface of a sphere is represented by the equation $x^2 + y^2 + z^2 - R^2 = 0$, where x, y, z are points lying on the surface and R is the radius. Spheres and disks are popular in DEM analysis because of the speed with which intersections can be checked and the compactness of the representation. This is by far the most popular representation used in discrete element analysis, [26, 24, 123, 110]. The analyticity of discs and spheres has been extended to 2D and 3D ellipses by several researchers including Ting [118] and Ng [85].

Implicit representations have been generalized to a wider range of two and three dimensional shapes using superquadrics [6, 46, 95, 133, 134]. When using superquadrics the intersection check relies on the sampling of the surface at discrete points. Each point on the superquadric of one body is checked against the implicit function representing the other body. The order of the contact algorithm is then $O(N)$ where N is the number of surface points checked per body. Superquadrics are used to generate complex object geometries in Chapter 8.

Hybrid Representation

Although the order of the complexity gives information about how the algorithm scales, it does not take into account the number of calculations required for each check. In the case of patches, point containment and surface-surface intersection tests require numerical root-finding calculations require significant computation [4, 49]. This complexity tends to negate the value of using compact functional representations in large scale simulations such as DEM.

In order to support both boundary and implicit representations, a low level boundary representation called the *discrete function representation* (DFR) was developed. The scheme is both general, in its ability to handle a wide variety of shapes, and provides a structure with which contact resolution can be performed demonstrably faster than many techniques currently used.

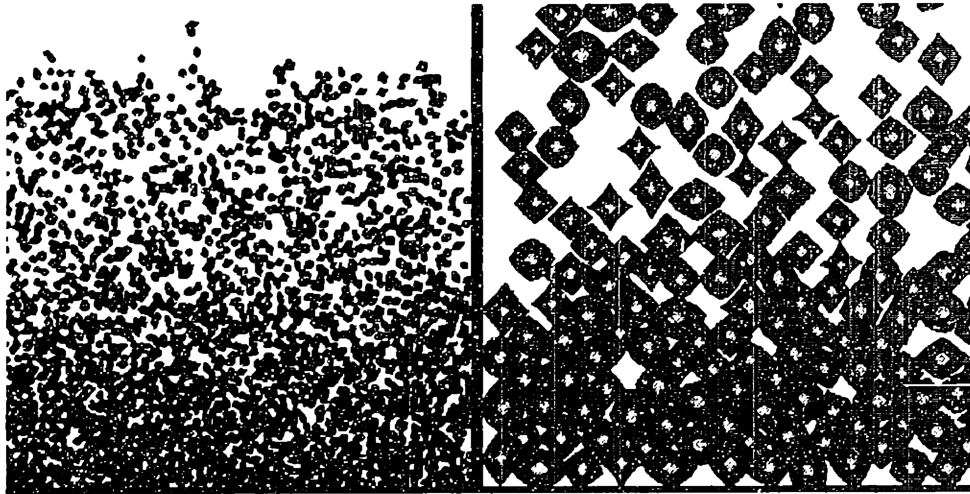

Figure 3.2: Deposition Simulation

3.2 The 2D-DFR Scheme

The first version of the DFR to be developed is used to represent the boundary geometry of 2D objects. It is referred to as the 2D-DFR scheme. The scheme is suitable for representing convex 2D regions and a restricted set of concave geometries. A structured storage format is used to encapsulate the boundary geometry in a way that can be efficiently retrieved when performing contact resolution. The storage scheme is described in the next section. A contact resolution algorithm based on the 2D-DFR scheme is then developed. The performance of the algorithm is evaluated empirically in a series of tests against another commonly used representation scheme. Examination of the 2D-DFR contact resolution algorithm establishes a computational complexity that is linear, $O(N)$, with respect to the number of points (N) used to define an object.

The 2D-DFR scheme treats the boundary of an object as being composed of a pair of single valued functions of one local parameter, for example $y = f(x)$. Closed regions are *cut* in two such that each sub-region is treated as a separate single valued function. This idea is shown in Figures 3.3.a and 3.3.b. The separating plane of a convex region readily yields a dividing axis about which the boundary can be treated in this manner. The function is then sampled at fixed intervals dx. This gives an

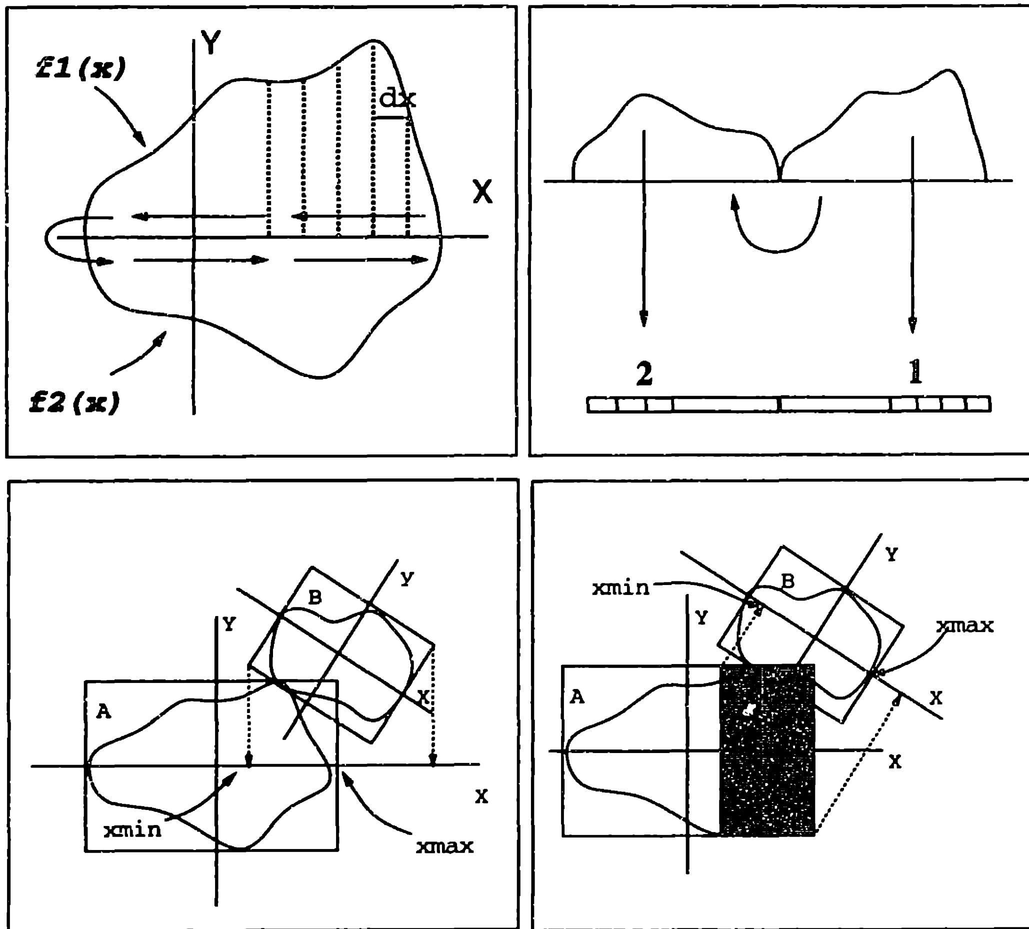

Figure 3.3: DFR Steps a) Traversal of Vertices b) Physical To Address Space Mapping c) Projection of B onto A d) Projection of Reduced A onto B

ordered set of points which can be searched extremely quickly[2]. To exploit the convex properties of the boundary geometry the local Cartesian coordinate, X, is used.

3.2.1 Storing the 2D-DFR Data

As previously stated, 2D objects are considered to be described as a closed region. The region is separated into two sub-regions each of which is treated as a single valued function.

The separating line is taken to be the local X axis of the object. This is shown

[2]A similar representation was developed by Hogue [52] in which she uses an angle as the local parameter. However it is not clear that this benefits the contact calculations siginificantly as the transformation will require evaulating several transcendental functions.

in Figure 3.3.a. Each of the half-regions are then quantized with a uniformly spaced sampling, dx, along the local X axis. A linear interpolation between boundary points is considered here but the algorithm can be extended to incorporate more sophisticated schemes, such as splines. The speed of the algorithm derives from the spatial ordering now imposed on the boundary points by the sampling. Once a search point is determined to lie between two of the quantized X coordinates, all other boundary points can be eliminated from the search.

A convention for the storage and traversal of the vertex list is shown in Figure 3.3.a. The mapping of vertices into memory starts at the maximum extent of the object along the X axis, in the 1st quadrant. The boundary points are traversed in an anti-clockwise fashion, and mapped into memory as in Figure 3.3.b. As the coordinate axis and computer memory are both linear, a constant scaling relation exists between the address space and the local physical space in this dimension. This scaling relation is used extensively in the contact resolution algorithm described in the next section[3].

3.2.2 2D-DFR Contact Resolution

The 2D-DFR contact resolution algorithm is now described. Two polygonal objects A and B are considered, each represnted with N vertices described in a local coordinate frame. A bounding box is also maintained for each object, also with respect to the local frame. The contact algorithm involves the following steps:

1. Transform bounding box of B into frame of A.

2. Project the upper and lower bound extents of the transformed bounding box onto the local X axis of frame of A, as in Figure 3.3.c. This delineates a subregion over a portion of the local X axis of object A.

3. Calculate the intersection of the subregion with the bounding box of object A. If they intersect go to step 4), otherwise exit.

[3]By making use of the implicit scaling relation, it is also possible to halve the storage required for the vertex list, by generating the x coordinates from the index of its storage location. This is at the additional cost of scaling address space ordinates at run time. For now this detail is not considered further.

4. Determine the maximum and minimum X coordinates of the projection along the X axis of frame A (as shown in Figure 3.3.c). Use the region bounded by the max-min coordinates to form a reduced bounding box in frame A and then transform it into frame B.

5. As in step 2), project extents of this reduced bounding box onto local X axis of object B, as in Figure 3.3.d.

6. From the bounding coordinates found in steps 1-5, calculate the bounding addresses for the storage of each object. The relationship between address and physical coordinate is formed using the scaling factors defined at the discretization stage.

7. For each point identified by its index, test for inclusion in the single trapezoidal strip that point maps to when transformed from its frame to the frame of the other object. Collect points found inside either of the objects boundaries, for further processing.

Details of the algorithm are now described. Transformations are expressed as 3x3 homogeneous matrices, [22, 67, 14]. The following notation is used:

- Matrices - bold face, upper case, e.g. $\mathbf{T}$
- Vectors - bold face, lower case, e.g. $\mathbf{v}$ or $\vec{v}$
- Scalars - single face, lower case, e.g. x, x_i

Superscripts denote the frame of reference. Subscripts identify the quantity or index being described; for example $\mathbf{T}_B^A$ is a matrix describing B with respect to the frame of A. $\mathbf{v}_i$ denotes the ith component of vector $\mathbf{v}$. $x_{B,l}^A$ denotes a scalar value expressed in frame A. The subscripts indicate that the quantity is related to object B and is a lower-bound. Matrices and arrays without a superscript are assumed to be described in the global reference frame. Indexing of ordinates and arrays are in the range $\{0 \ldots N-1\}$, where N is the number of elements in the set. The frame transformations from frame **A** to frame **B**, and vice-versa, are given by Eq. 3.1. $\mathbf{T}$ encapsulates the position and orientation of an object's frame with respect to a global frame.

$$\begin{aligned} \mathbf{T}_B^A &= \mathbf{T}_B \, \mathbf{T}_A{}^{-1} \\ \mathbf{T}_A^B &= (\mathbf{T}_B^A)^{-1} \end{aligned} \tag{3.1}$$

Contact Region Clip

Steps 1 through 4 (above) are referred to as the *contact region clip* stage. These steps bound a sub-region from each object that potentially intersect.

Projecting the transformed bounding box vertices of **B** onto the local X axis of **A**, as in Figure 3.3.c, the clipping bounds of the sub-region (the contact region) are calculated from:

$$\begin{aligned} x^A_{min} &= MAX(x^A_{lb}, x^A_{B,min}) \\ x^A_{max} &= MIN(x^A_{rb}, x^A_{B,max}) \end{aligned} \tag{3.2}$$

where,

x^A_{lb}, x^A_{rb}	-	left and right bounds of A
$x^A_{B,max}$, $x^A_{B,min}$	-	max/min transformed bounds
x^A_{max}, x^A_{min}	-	max/min sub-region bounds

If $x^A_{min} > x^A_{max}$ then no overlap has occurred and the examination is finished. Otherwise, the clipping bounds are used to calculate the *start* and *end* addresses of the panels from the *upper* half-space of object A that make up the contact region:

$$\begin{aligned} i_{us} &= \lfloor s^A_x(x^A_{rb} - x^A_{max}) \rfloor \\ i_{uf} &= \lfloor s^A_x(x^A_{rb} - x^A_{min}) \rfloor + 1 \end{aligned} \tag{3.3}$$

The *floor* operation, $\lfloor x \rfloor$, is a single floating point operation that returns the greatest integral value less than or equal to x, [96, 65]. The subscripts u, l, s, f stand for *upper*, *lower*, *start* and *finish* respectively. i_{us} and i_{uf} are then the start and finish indices of the sub-region lying in the upper half space of object A. s^A_x is the scaling factor between the physical space to address space for object **A**. This is simply the inverse of the discretization step size, dx.

The corresponding addresses in the lower region are calculated from the upper addresses under the simple relationship:

$$\begin{aligned} i_{ls} &= N - i_{uf} - 1 \\ i_{lf} &= N - i_{us} \end{aligned} \tag{3.4}$$

N is the total number of elements in the array storing the boundary points.

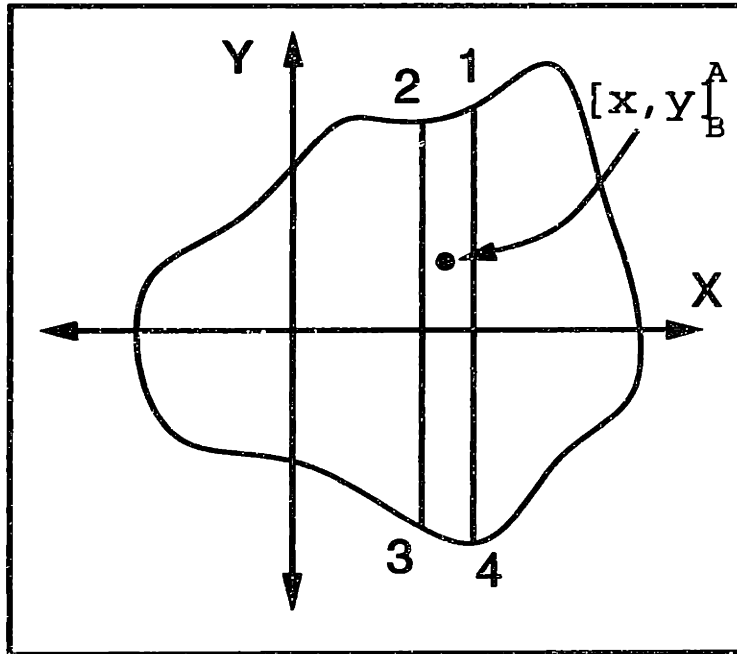

Figure 3.4: Simple Contact Bounds Test For Trapezoidal Region

These indices also mark off a sub-region from the bounding box of object A, which is referred to as a *reduced* bounding box for A. The preceding steps are then applied in projecting the *reduced* bounding box in **A** onto the local X axis of object B, as shown in Figure 3.3.d. The bounds of a sub-region found in B are calculated as before using Eqns 3.2 and 3.3, with B as the primary frame of reference.

Contact Test

The contact region clip steps yield two sets of indices (upper and lower) that demarcates boundary points from A and B. The points from each object are tested to see if they fall inside the *contact region* of the other object. For convex objects, contact is said to occur when at least one boundary point, from either of the objects, is found to be contained by the *contact region* of the other object.

Due to the implicit ordering of the objects boundary representation, the contact detection step is simply a one to one bounds test as follows:

1. Transform each point from its reference frame into the frame of the object to be tested against.

2. Project the point onto the local X axis and transform to the address space using the scaling factor as before in Equations 3.2 and 3.3.

3. Calculate the array index of the bounding edges of the partition in which the point potentially lies. Since the boundary representation is limited to a single-valued function, each partition corresponds *uniquely* to a single trapezoidal strip.

A transformed point is then tested for presence in a trapezoidal panel using a height test for the y ordinate to detemine containment. This is shown in Figures 3.4.

$$\begin{aligned} y_B^A &\leq y_1^A + \frac{y_2^A - y_1^A}{x_2^A - x_1^A}(x_2^A - x_B^A) \\ \textit{and} & \\ y_B^A &\geq y_3^A + \frac{y_4^A - y_3^A}{x_4^A - x_3^A}(x_4^A - x_B^A) \end{aligned} \tag{3.5}$$

Computational Requirements for the Contact Algorithm

Considering each object as being described by N boundary points or vertices, the *worst case* computational requirements of the algorithm are:

Step	Operation	Cost [flop]
Region clip	Eqn 3.3	$k, constant$
Panel lookup	Eqn 3.3	$O(N)$
Bounds test	Eqn 3.5	$O(N)$

Table 3.1: Operation Cost

The *worst case* refers to the situation in which all of the vertices from one object are contained by the other. The *floor* operation, $\lfloor\ \rfloor$, is considered a single floating point operation in the analysis, [96, 65]. Overall a computational complexity of $O(N)$ is found for the algorithm.

3.2.3 Timing Comparison with the Cyrus-Beck Method

To quantify the performance of DFR based contact detection a series of timing tests were performed. These times were then compared with a contact resolution method based on the Cyrus-Beck algorithm, [81]. The algorithm was originally developed for convex polygon clipping in computer graphics, [100] and has been used in physically based simulation by Hahn [42], Moore and Wilhelms [81] and in DEM applications [33].

Number of Vertices (N)	Discrete Function [msecs]	Cyrus-Beck [msecs]
50	0.5	25.8
100	1.3	97.1
200	2.7	381.4
300	3.9	849.6
400	5.2	1503.5
500	6.6	2341.5

Table 3.2: Timing Results

The series of tests were performed[4], for a worst case situation with complete overlap, between polygonal objects described with varying numbers of vertices.

The timing results shown in Table 3.2 and graphed in Figure 3.5, confirms empirically that the Cyrus-Beck has a quadratic time complexity, $O(N^2)$, and that the DFR is linear in time, $O(N)$. In Figure 3.5, the 2D-DFR algorithm results are almost coincident with the axis.

[4]Tests were performed on an IBM Power Station 320H, RS/6000 processor (20MHz).

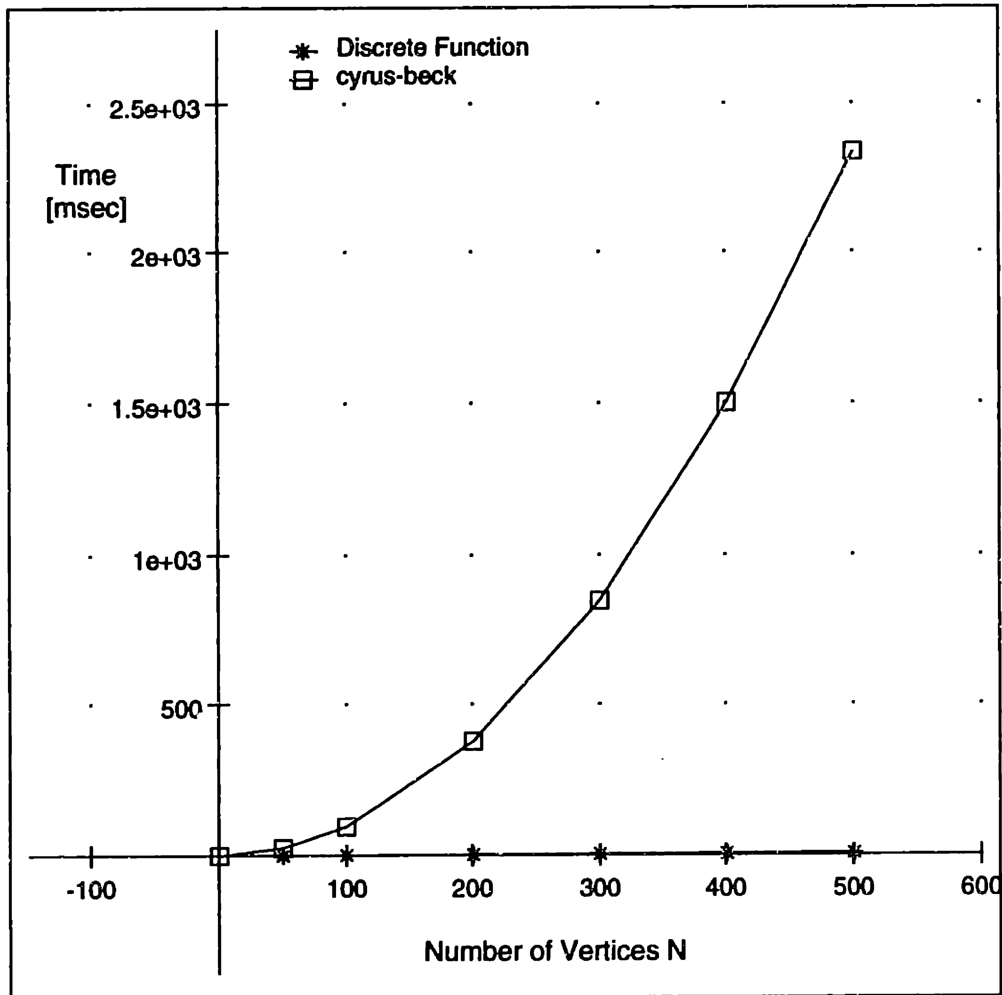

Figure 3.5: Plot of Time Vs. Point Sampling (N)

3.2.4 Summary

The 2D-DFR scheme provides a very efficient means to perform contact resolution in multibody simulation containing objects of different size and shape. The representation exhibits a dramatic speedup over a standard method[5], as shown in Table 3.3. The scheme has a drawback in that the sampling of the surface geometry is not uniform in both coordinate directions. This leads to some difficulties when trying to transform certain shapes into a 2D-DFR format.

[5]The scheme has been used in a production sized 2D simulation. This application is discussed further in Chapter 8.

Vertices	Speed Up
50	50
100	76
200	141
300	214

Table 3.3: Typical Speedup Factor

Adjusting to these limitations a more comprehensive representation scheme is developed for 3D objects. The underlying theme of storing the surface representation in the form of a lookup table is preserved, but the non-uniform sampling scheme in 2D is avoided by using a uniform sampling grid.

3.3 The 3D-DFR Scheme

The 3D-DFR scheme extends the structured storage format of the 2D scheme to be able to represent arbitrary 3D geometries, while retaining the characteristics necessary for efficient contact resolution. Contact resolution using the 3D-DFR scheme can be performed in times proportional to the volume of overlap, the *contact region.* The average computational cost is then *sublinear* with respect to the number of sample points used to represent the surface geometry.

The description starts with an overview of how the 3D-DFR scheme is constructed and one of the methods used to capture the surface geometry for this purpose. The second section details the 3D-DFR data structure which is used to store the geometric representation. The remaining sections of the chapter detail the steps necessary to perform contact resolution based on the 3D-DFR scheme.

3.3.1 Overview of DFR For Arbitrary 3D Geometries

A boundary representation scheme called the *discrete function representation* (3D-DFR) is developed to facilitate efficient contact resolution. The 3D-DFR algorithm achieves its speed by two means, namely 1) precalculating 3D-DFR lookup tables which order the space occupied by the object, and 2) using efficient algorithms to delimit the space to be searched.

The space occupied by a body is broken into an ordered set of uniform cubes (*voxels*). Cubes that intersect the object's surface are then used to approximate this surface by calculating 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 first phase of the 3D-DFR scheme uses a tessellation algorithm to generate a 3D isosurface from an input data set describing an object. The space occupied by the object is first diced into an ordered set of uniform cubes (*voxels*). Cubes that intersect the object's surface are then used to approximate the surface by calculating the location of the intersection points of the surface with the cube edges. The intersection points are then used to form the facets of the tessellated surface.

For our application we used Lorenson's *marching cubes* tesselation algorithm, [73]. Using this scheme, an isosurface can be derived from implicit functions, from raw scalar field data or inferred from existing unstructured polygonal surfaces created using other representation schemes. The surfaces we are interested in are considered

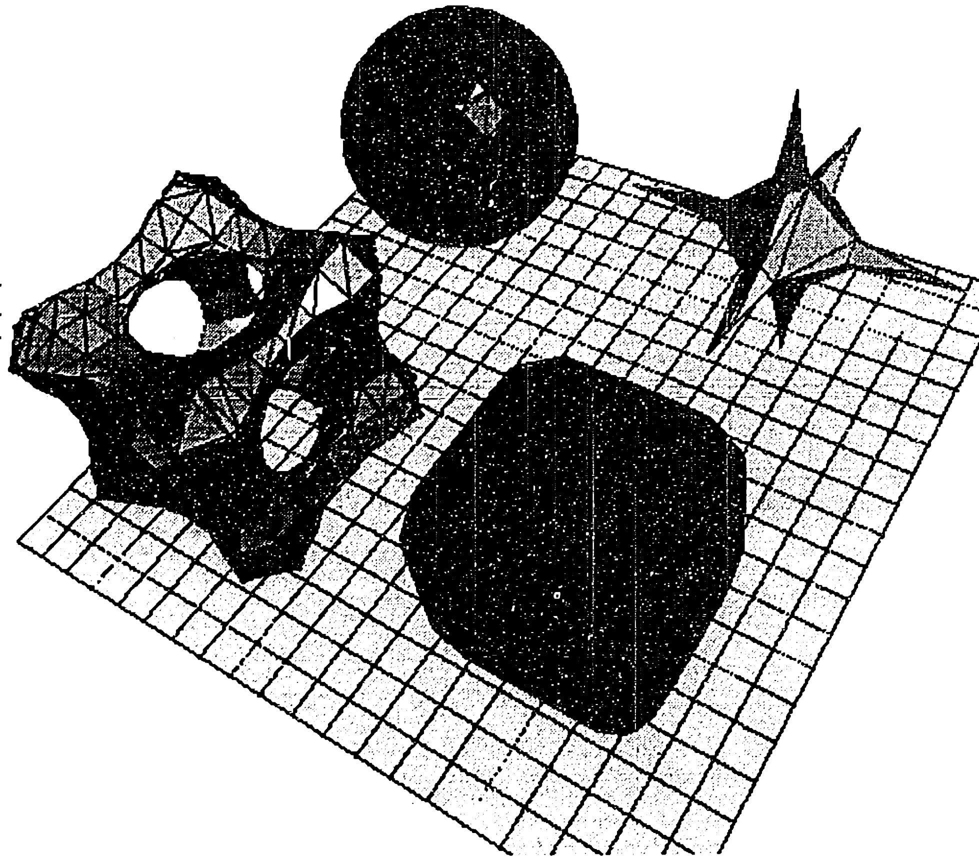

Figure 3.6: Sample Geometries

to be piecewise continuous functions in $\mathcal{R}^3$, even though the functions are never explicitly derived for the last two input formats. A small example of the range of geometries possible using this scheme are shown in Figure 3.6.

The Tesselation Process - Marching Cubes

In the simplest situation we assume the body is defined by an implicit function. (Other starting represention are discussed later in the paper.) As an example, we use the equation of a sphere. In this case the value of the function is the same at all points equidistant from it's origin. Choosing a specific *threshold* value identifies a

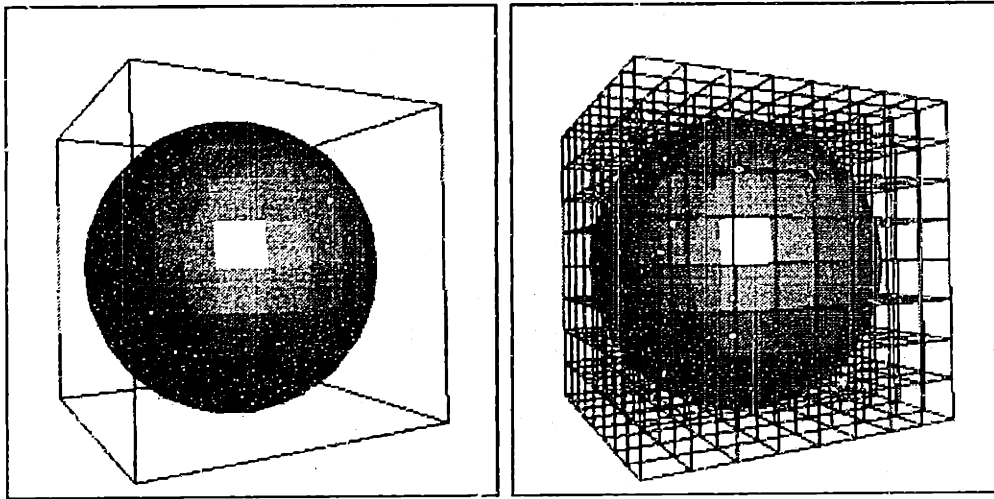

Figure 3.7: a. Object Geometry b.Sampling Grid

member of the family of spheres, as shown in Figure 3.7. To tessellate the surface, the value of the function is calculated at all cell vertices of a sampling grid such as that shown in Figure 3.7.

The *marching cubes* algorithm then compares the specified threshold with the value of the implicit function at the cell vertices. A cell whose vertices have values both less than and greater than the threshold indicates that the surface crosses a cell's boundary edges. Each edge that the surface crosses is then further examined to obtain a single crossing point by using an interpolation scheme. Joining adjacent edge crossing points describes a triangular facet. We show two examples of such a connectivity in Figure 3.8. In general, the portion of the surface cutting the cell can be decomposed into an enumerated set of triangles using surface continuity assumptions over the cell's dimensions.

Mapping Intersection to a Single Integer

We can enumerate all the ways in which a surface intersects a cube (subject to certain restrictions on the surface). By assigning a single bit to each vertex we can generate an identifier in the range 0-255 (i.e. 8 binary vertex states giving $2^8 = 256$ configu-

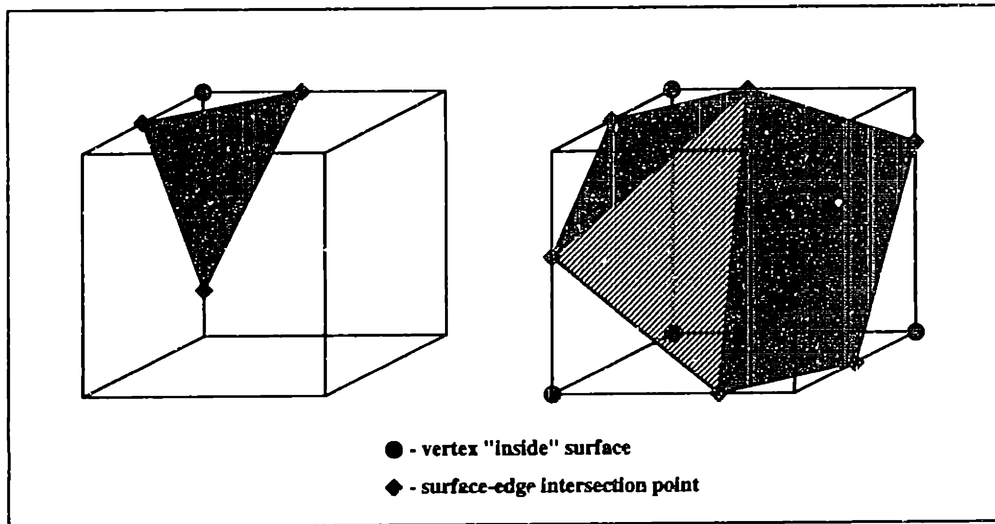

Figure 3.8: Example Cell Classifications: a) Single Vertex b) Multiple Vertices

rations). The enumerated set of triangles [6] that tessellate the surface in the cell is tabulated *a priori* for fast lookup access. This table contains topological information describing how the surface cuts the cell and the number of faces required to interpolate the surface through the cell. For example, in Figure 3.8 we see two of the possible surface tessellations. Each cell identified in this manner is stored separately in a *cell descriptor table* that can be used later for visualization and other purposes. This is shown graphically in Figures 3.9 and 3.9. The second of these, Figure 3.9, is simply a wire frame view of the same isosurface.

[6] We are aware that a number of ambiguities can arise using the original derivation of Lorenson's algorithm for certain surface topologies. However, we chose to use the original version as we found the algorithm ammenable to coding and more than satisfactory for our application.

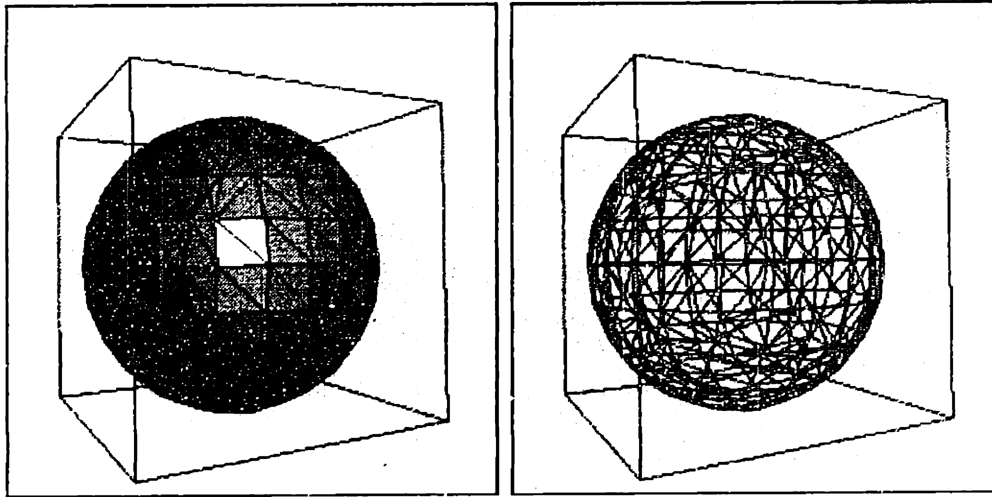

Figure 3.9: a) Tessellation b) Mesh Discretization

3.4 The 3D-DFR Data Structure

This section describes the structure of the 3D-DFR data set and how it is generated as part of the tesselation process. At the coarsest level the 3D-DFR corresponds to a cage of cells that completely enclose a 3D surface. The 3D-DFR data structure uses only the subset of the cells that intersect the surface. We refer to the cage of cells as the *discrete bounding hull* (DBH) of the data set. For each body we use the sampling grid as a local coordinate system with the cell side as the metric. We use the following nomenclature:

slice	-	a 2D section through the sampling space i.e. a plane of cells
prism	-	a 1D section of a **slice** ie. a line of cells.
cell	-	the fundamental building block in discrete space.

We call the axis perpendicular to the slice plane the **slice axis**, the axis perpendicular to the prism line the **prism axis**, and the remaining axis the **cell axis**. The DBH of a sphere represented in this manner is shown in Figure 3.10.

The 3D-DFR data structure is a ragged 3D array of cell indices called the **stencil** and a set of ragged arrays containing *offset/run-length* (ORL) pairs to describe the **stencil** layout. This structure is shown in Figure 3.11. The indices contained in the **stencil** refer to pointers to the data structure of the tessellated surface (not shown) which is stored separately.

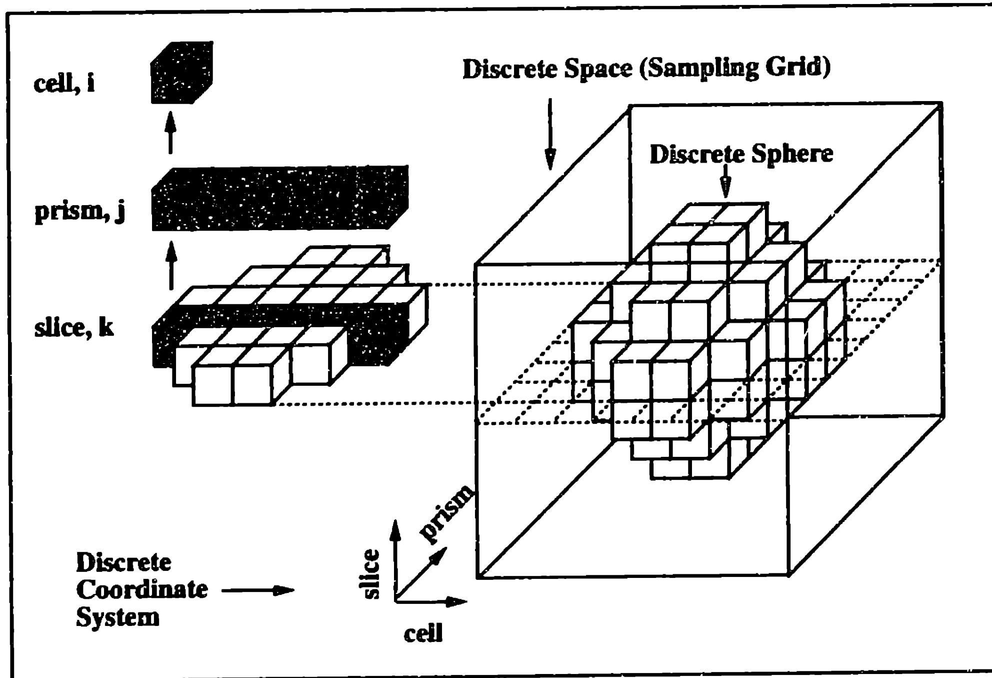

Figure 3.10: 3D-DFR Elements and Example Object (Discrete Sphere)

The ORL pair arrays are defined as follows:

1. **slice map** - a single ORL pair. The first element of the ORL pair is the offset in **cell** units to the first **slice** of the DBH from the origin of the discrete coordinate system. The second element is the run-length (the number of slices) that follow.

2. **prism map** - a 1D array of ORL pairs. The length of the array corresponds to the number of slices. The first element in each ORL pair denotes the offset to the first **prism** of the respective slice. The second element specifies the number of **prisms** in that slice.

3. **cell map** - a ragged 2D array of ORL pairs aligned with the slice and **prism** axes. Each element of the array is indexed by the offset in **cell** units to the current slice and **prism**. The first element of each ORL pair is the offset to the start **cell** of that **prism**. The second entry is the number of cells in that **prism**.

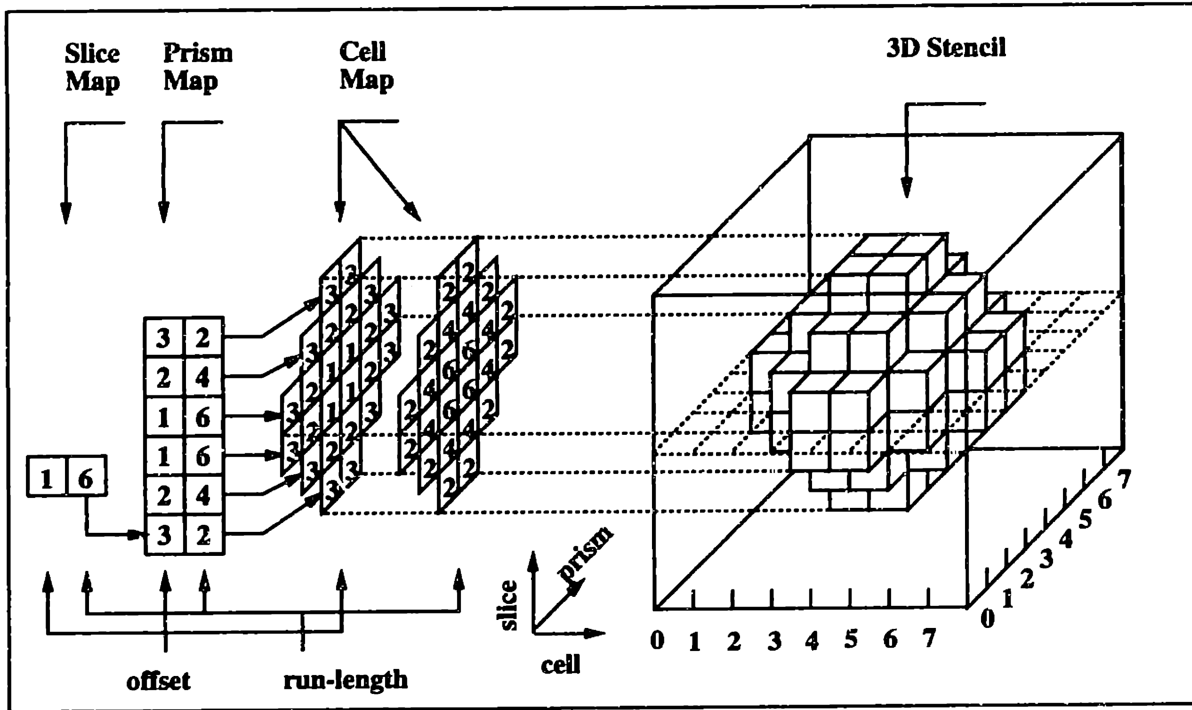

Figure 3.11: 3D Ragged Array Storage for Discrete Sphere

4. **stencil** - a ragged 3D array of *cell descriptor table* indices that demarcate the region contained by the DBH. The ragged array topology is configured to be equivalent to the aggregation of cells making up the DBH of the surface. Internally, the ragged array is described by the ORL maps described in Items 1, 2 and 3 above.

Having the memory layout for the 3D-DFR data structure parallel to the discrete coordinate system, there exists a one-to-one topological mapping between them. Each cell of the DBH in *discrete space* corresponds to a unique location in the 3D **stencil**.

To use the example discrete sphere of Figure 3.11, the ORL arrays in this case also describe the layout of the 3D **stencil** as being a discrete sphere in address space. This analogue allows the **stencil** to be mapped to the cells retained from the extended tessellation algorithm. The mapping process stores an index[7] in each field of the 3D

[7]Native language pointers or integer indices are most efficient but space considerations may dictate that single word or bitwise indices should be used. We use integer indices for simplicity

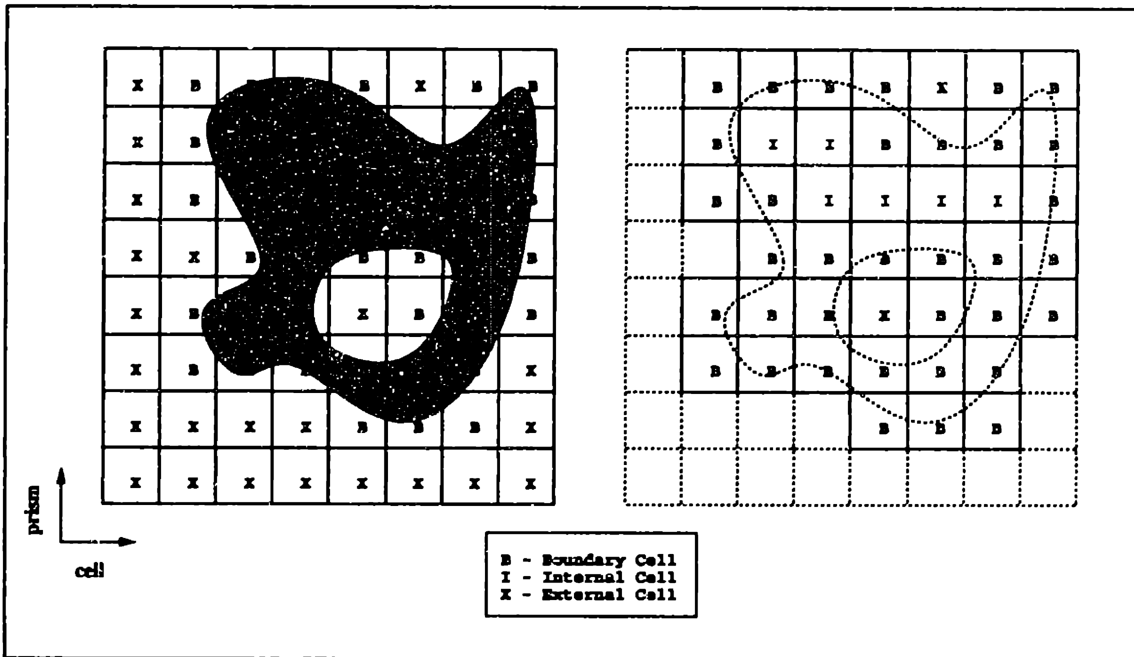

Figure 3.12: a) Cell Code Assignment, b) Discrete Bounding Hull (2D)

stencil, to the location of an entry in the *cell descriptor table.* An entry in the *cell descriptor table* describes the surface facets in the corresponding cell, as obtained from the tesselation algorithm.

Since the local and discrete coordinate systems are also parallel, mapping from one system to the other is achieved through a translation and a scaling along each axis. The scaling factors will be the ratios of the sampling grid dimensions and the resolution of a **cell**, measured along each axis of the local coordinate system. The translate terms are the offsets (if any) of the origin of the local coordinate system to the discrete coordinate system. This relationship allows 3D coordinates in the local coordinate system to be mapped to a **(slice, prism, cell)** coordinate triple in the discrete space. Once transformed, the discrete space coordinates can be treated directly as indices into the ORL arrays and in turn to a **stencil** entry.

To distinguish the cells referenced by the 3D **stencil** entries, we classify them into 3 groups: *boundary* (surface) cells, *internal* (completely enclosed) cells and *external* (empty) cells. The first grouping are cells that correspond to legitimate **stencil** index entries that point into the *cell descriptor table*. A legitimate **stencil** index refers to a **cell** with some portion of the surface geometry passing through it. We differentiate the second and third groups (internal, external cells) using reserved index values for their **stencil** values. Cells that these indices refer to, act as spatial placeholders without requiring a cell description to be maintained in the *cell descriptor table*. This is possible because a cell's dimensions are uniform over the sampling grid and therefore completely empty or completely enclosed cells will have the same geometric description. We will also see that many of the external (empty) cells that occur can be discarded as each **prism** is completely processed. We show this next in a cell classification example for a 2D region.

A 2D example with a detailed view of cell classification for a *discrete bounding hull* is shown in Figure 3.12. The shaded area represents the closed region which we wish to enclose in a *discrete bounding hull*. The hull is built incrementally as the grid of cells is processed in scan line order (i.e. cell by cell along a prism, prism by prism over the 2D section). Each cell is assigned a cell code (corresponding to an index) with the following meanings:

B - a cell that straddles the boundary of the region.
I - a cell completely internal to the region.
X - a cell lying completely external to the region.

Trailing external cells (**X**) must be retained for the complete traversal of a prism. At the end of each pass, trailing external cells can be trimmed from the list of cells retained[8]. We see this in Figure 3.12.b which shows the region superimposed on the grid after discarding external cells not required for the bounding hull. We note that the discrete bounding hull need not be strictly convex[9], rather that it encapsulates the entire region (including holes and concavities).

For 3D objects the complete DFR construction consists of the tessellation opera-

[8] *Leading* external cells (**X**) that occur before the first boundary cell is found, are simply ignored.

[9] Should the region be convex, the boundary cells describe a convex hull and no internal cells would actually need to be retained. For generality we chose to include the possibility of non-convex closed regions.

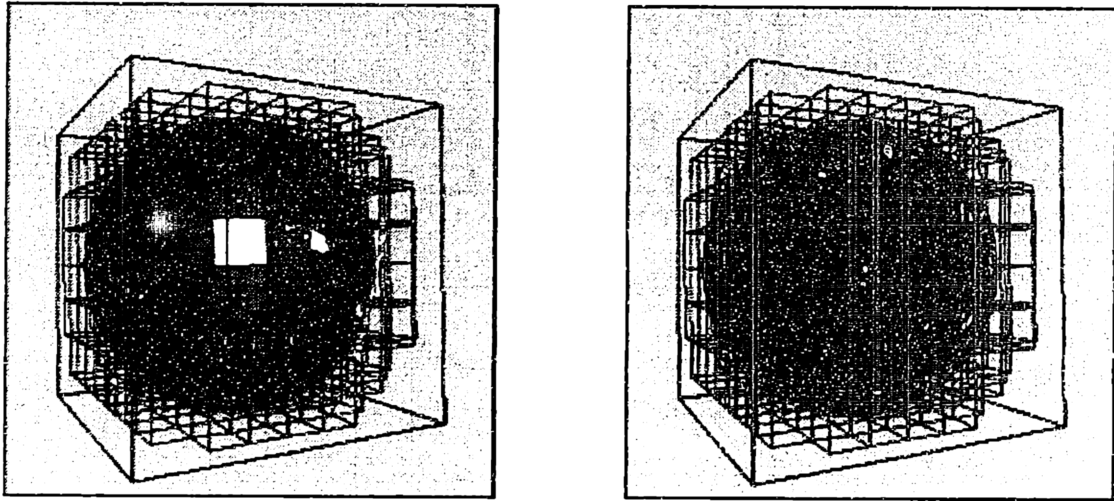

Figure 3.13: a) Discrete Bounding Hull; b) Internal Cells With Embedded Mesh

tion previously shown in Figures 3.7, 3.7, 3.9 & 3.9, and the simultaneous construction of the discrete bounding hull and internal DFR storage as shown in Figure 3.13. Figure 3.13.a shows the DBH of the spherical surface. As the surface is convex, all visible DBH cells will be boundary (B) cells. Figure 3.13.b shows the wireframe DBH with internal (I) cells filled, viewed through a wireframe outline of the isosurface.

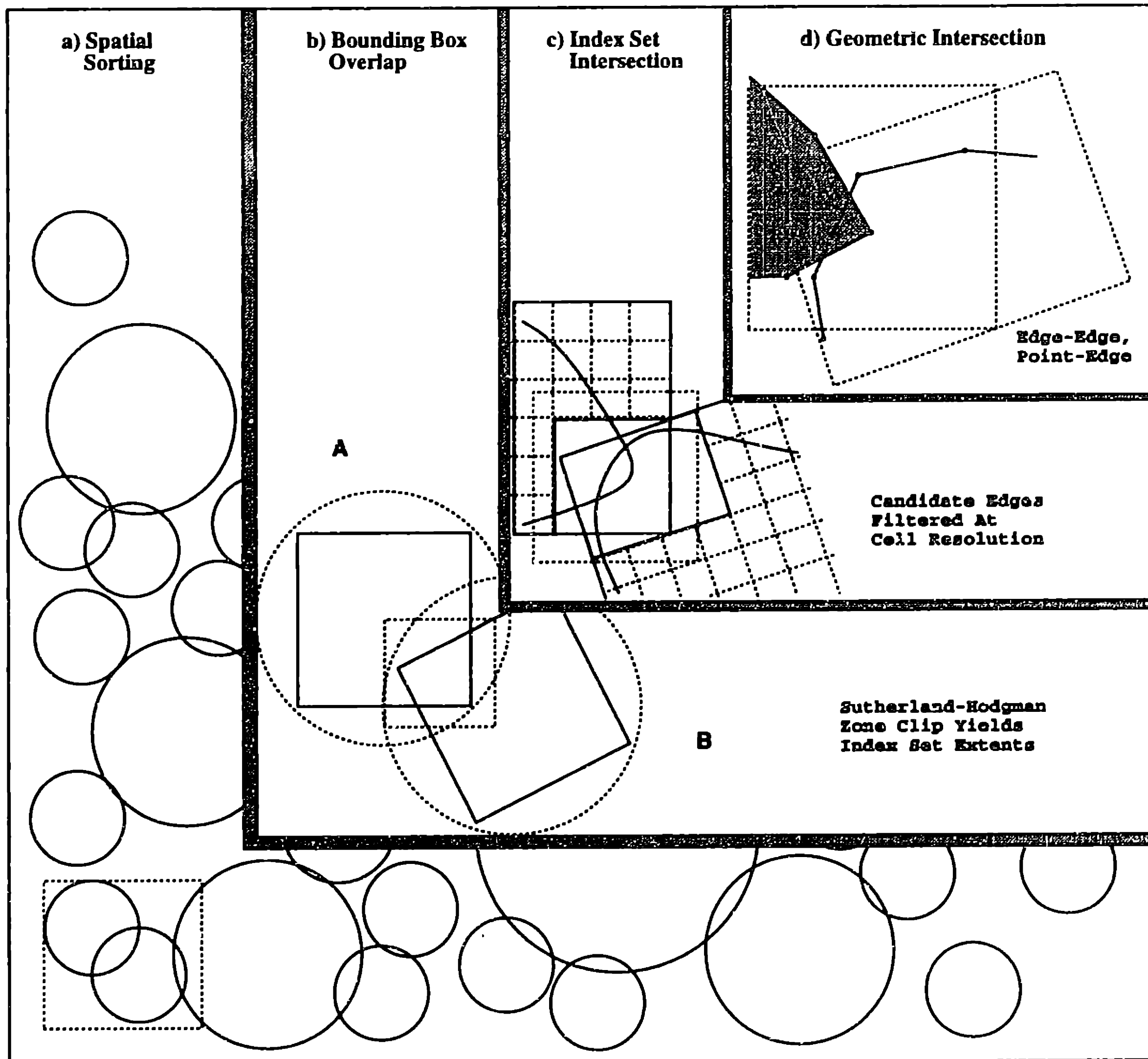

Figure 3.14: 3D-DFR Contact Detection as a Hierarchical Process

3.5 3D-DFR Based Contact Detection

A general algorithm for contact detection contains some or all of the following tasks. Initial steps seek to further cull nearby objects from consideration using a relatively crude but fast bounding box test. Intermediate steps attempt to reduce the amount of local geometric detail to be analyzed. The ultimate steps examine the higher resolution details of the intersection. In 3D-DFR based contact detection we follow this general outline and show the hierarchy of steps in Figure 3.14. First we start with an overview of the algorithm.

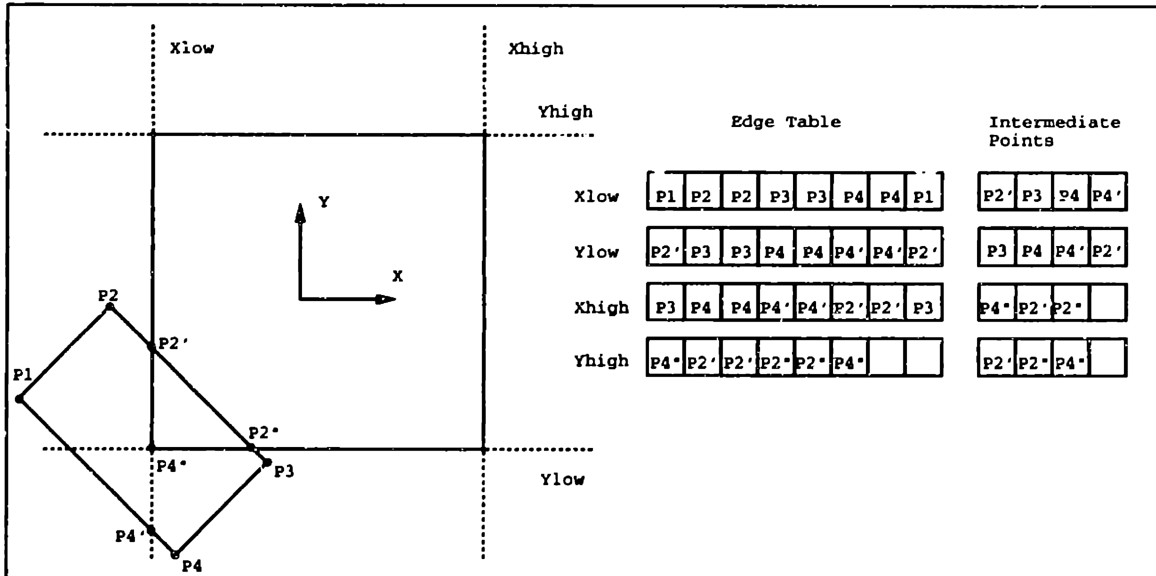

Figure 3.15: Zone from Sutherland-Hodgman Clipping

3.5.1 Algorithm Description

3D-DFR contact detection determines whether pairs of objects intersect and calculates a precise geometric description of the contact region. Each object maintains a reference frame describing the position and orientation of the object's local coordinate system. The local frame is in turn defined with respect to a fixed global reference frame. The geometric description of each object is expressed using the 3D-DFR scheme, in addition to a bounding box of the surface, both aligned with the local frame.

Zone Clipping

The first formal step in the contact detection process is to check for *bounding box* overlap. Since this operation occurs high up in the 3D-DFR contact detection hierarchy we expect it to be performed relatively frequently. For this reason we use an optimized *Sutherland-Hodgman* polygon clipping algorithm, [100], to determine the geometry of the overlap region between two bounding boxes. The optimization takes the form of hard coding the clipping operations to reflect that the clip volume is a rectangular parallelepiped, and that each polygon to be clipped to this region will

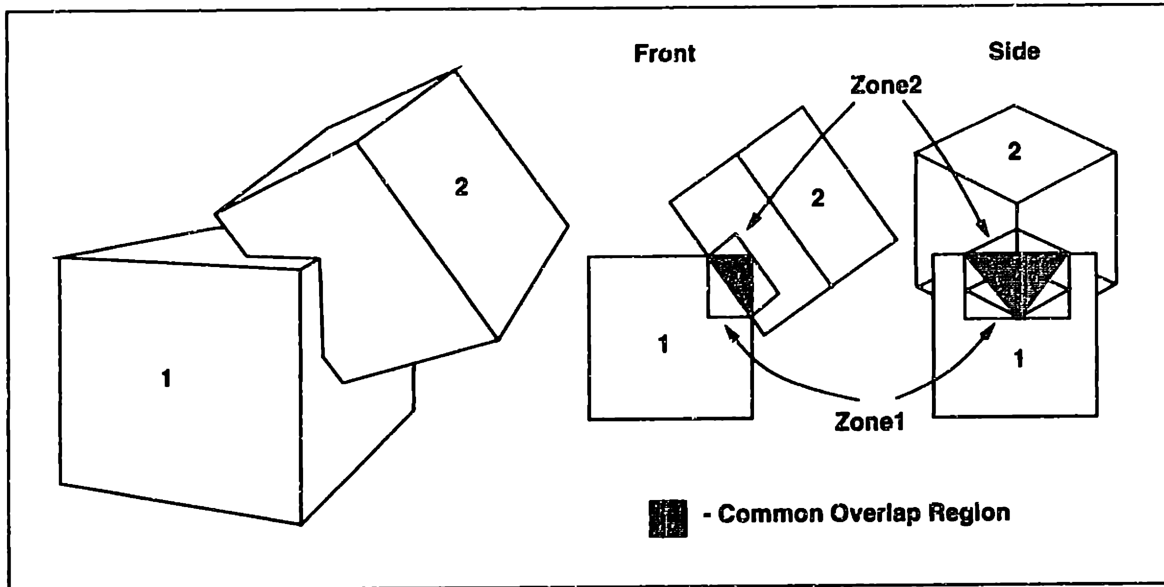

Figure 3.16: Zone From Bounding Box Overlap

be a rectangle. We show an example of the optimized algorithm for the 2D case in Figure 3.15.

The algorithm takes the polygon (P1, P2, P3, P4) and loads the end points into an *edge table* reflecting the connectivity of each edge of the input polygon. For the 2D example the input edges are (P1, P2), (P2, P3), (P3, P4), (P4, P1). The edges are then checked against the boundaries of the clip region (one boundary per iteration) by testing if an edge trivially crosses the extended boundary line and calculating the point of intersection when appropriate. Points that lie on the extended boundary or are potentially inside the clip region boundaries are cached in an intermediate buffer which is used to form the input *edge table* for the next iteration. For the first level of the input *edge table* the intermediate crossing points with the `Xlow` boundary are P2′, P3, P4, P4′. The points of intersection are buffered as intermediate points and then used as the input points to form the next level of the *edge table* which is (P2′, P3), (P3, P4), (P4, P4′), (P4′, P2′). We continue applying these steps with each subsequent *edge table* by comparing the edges with the remaining extended clip region boundaries. Finally a set (possibly empty) of points describing the edges of the overlap region are produced. For the example shown in Figure 3.15 the overlap region is described by the set of points P2′, P2″, P4″.

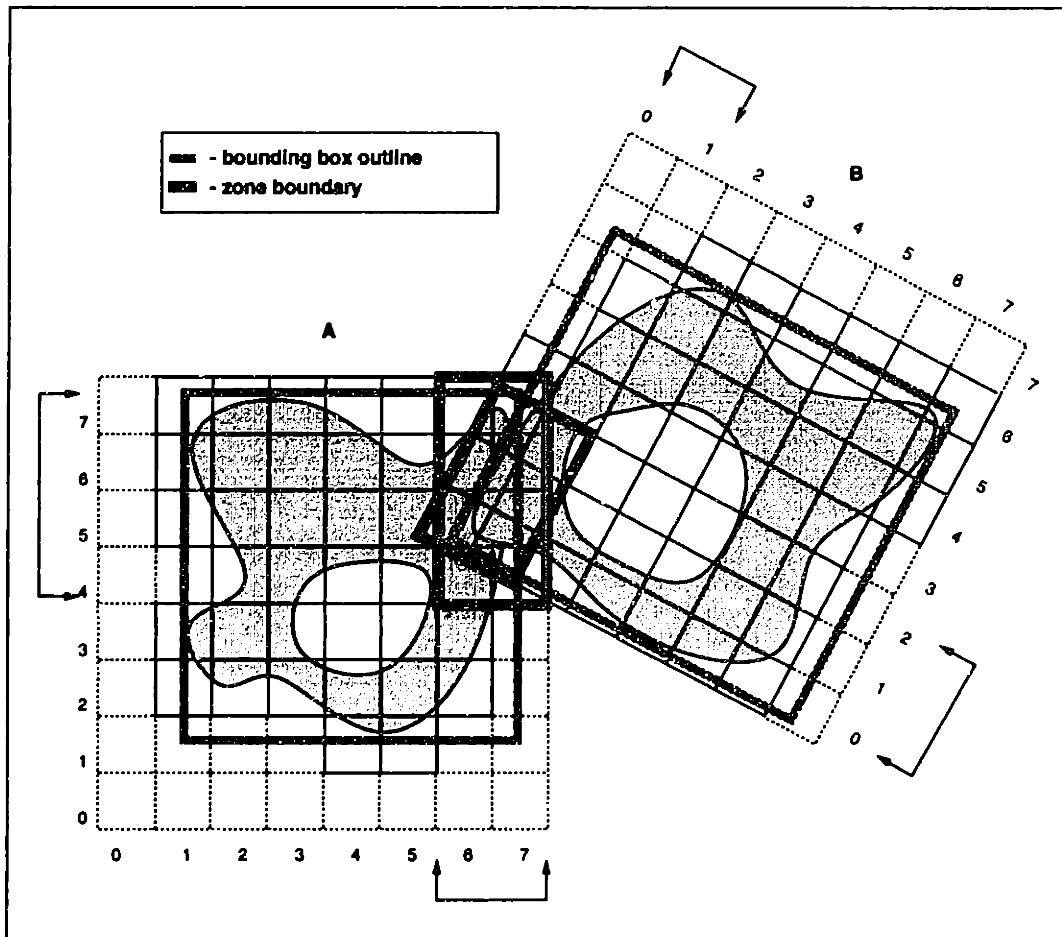

Figure 3.17: Zone Index Set From Clip Zone

Where bounding box overlap occurs, the upper and lower bound extents of the overlap region parallel to the local coordinate axes are calculated for each object and stored as a pair of 3D vectors, i.e. $(x, y, z)_{min}, (x, y, z)_{max}$. We refer to each vector pair as a *zone* of the object and show a possible *zone* pair from a bounding box overlap test in Figure 3.16. The *zones* now bound the region of each object that needs to be further considered in the contact detection process.

Zone Index Sets

Next the *zones* are transformed into discrete coordinates. To map to the discrete coordinate system each coordinate is first scaled by dividing by the unit cell dimension

in that direction and then taking the *floor* of the resulting value. In discrete space a coordinate triple corresponds to the location occupied by a finite cell, and so the discrete coordinates retain the property of enclosing the overlap region, but now at the discrete level. The transformed coordinates of the *zone* then delineate the extents of a set of index triples refered to as the *zone index set.* Geometrically the *zone index set* describes a localized region in discrete space which needs to be further checked for overlap.

The *zone index set* for a 2D overlap example is shown in Figure 3.17. The *zone boundaries* enclose the region of overlap between the two bounding boxes. The *zone index set* is then found by enumerating the cells that the *zone boundaries* contain. For this example the *zone index set* for object **A** would be: (6,4), (6,5), (6,6), (6,7), (7,4), (7,5), (7,6), (7,7), and for object **B**: (0,0), (0,1), (0,2), (1,0), (1,1), (1,2).

Zone Index Set Filtering

We can further reduce the scope of the overlap tests by taking the intersection of the *zone index set* with the ORL arrays describing the DBH of the object. The *zone index set* intersection yields a subset of index triples to be retained for detailed checks, and the complement can be simply ignored. The set intersection takes the form of an iteration over the *zone index set*, checking each index triple for a corresponding entry in the ORL arrays. A matching entry is an index triple that refers to an existing ORL entry in the 2D **cell map**, and in turn to a legitimate **stencil** entry that points into the *cell descriptor table.* For legitimate **stencil** entries, the surface data for this **cell** is then extracted from the *cell descriptor table.*

Cells that pass this discrete scale filtering are then examined at the highest resolution, the surface facets obtained from the tessellation process. The checks at this final level are the necessary *point-edge* and *edge-edge* containment tests, [127, 15, 10].

Surface Intersection

The data extracted from the *cell descriptor table* corresponds to a portion of the *source* object's surface that we need to check against the *target* object's surface. The *source* surface data (edges) are next transformed into the *target* object's frame.

While neither end-point of a candidate edge necessarily lies inside the discrete space, the *edge* may cross through the space containing cells with valid ORL array

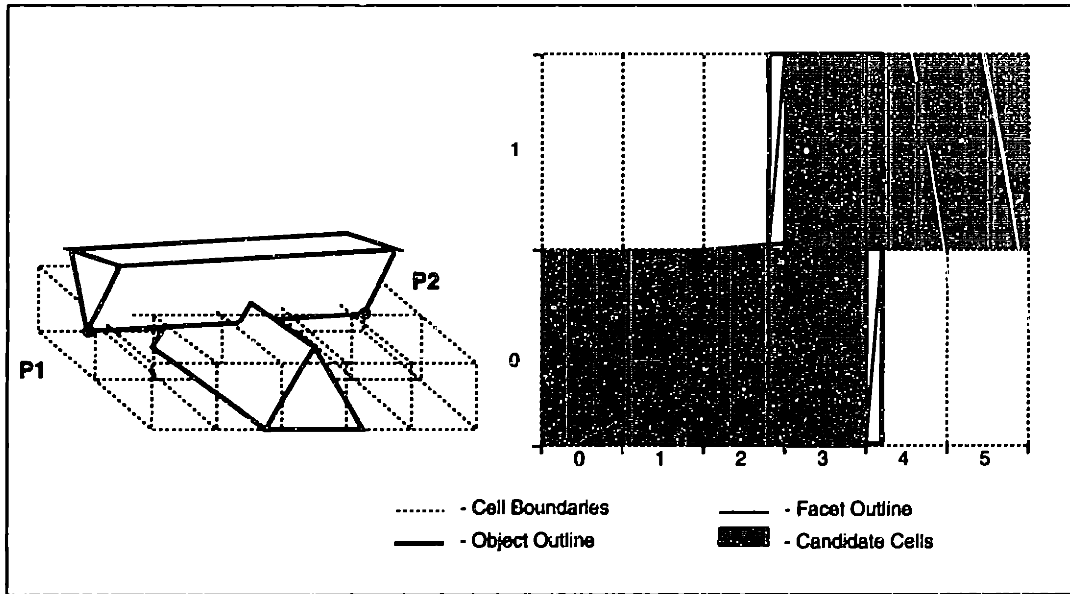

Figure 3.18: Discrete Edge - Edge Contact

entries. The general case of an edge passing through a portion of the *target* object DBH is shown in Figure 3.18. The edge (P1,P2) is to be checked against the *target* object i.e. the lower triangular prism. The *start* and *end* points of each candidate edge are clipped to the bounds of the *target* object, i.e. the bounding box of the target object's DBH. This classification is equivalent to clipping a line segment to a region (in this case a rectangular parallelpiped) to determine the visible portion therein. Having clipped the segment, we now have (possibly new) *start* and *end* points that lie on or inside the *target* object's discrete space and can be mapped to index triples in the *target* object's discrete coordinate system.

The index triples will either both refer to a single cell or individually to a pair of independent cells. Where a single cell is identified, it is first checked for a corresponding ORL array entry to verify some portion of the target object's surface exists there. This being the case, the edge can be directly tested for *edge-edge* intersection with the surface facets in that cell.

If the index triples map to different cells then we are required to check the *edge* against both of these and some number of the intermediate cells between them. For the example *edge-edge* contact situation in Figure 3.18 the edge (P1, P2) spans several cells. The problem here is to determine which cells the edge passes through before

attempting to perform the individual *edge-edge* tests. On the right of Figure 3.18 we show a plan view of the *edge-edge* contact with the tessellated geometry (triangular facets) outlined. The cells the line segment (P1, P2) passes through are shaded for clarity.

We choose one of the end-points as a starting point and map this point to a discrete space cell. Next clip the edge (P1, P2) to that cell's boundary using the *Sutherland-Cohen* algorithm, [100]. The clipping identifies an intermediate end-point for the edge that is also in the current cell. To complete the *edge-edge* test in the current cell, the intermediate line segment is checked against the portion of the *target* object's geometry contained in the cell.

The scan then continues on to the next cell that the original *edge* (P1, P2) spans. The next cell to visit is identified from the calculation of the previous intermediate end-point. This simultaneously identified which face the edge exits from the cell, and therefore the face of the neighboring cell it will next enter. Once determined, we can apply the previous two steps to all remaining cells spanned by the edge.

This completes the algorithm overview for 3D-DFR contact detection algorithm. We next describe the algorithm in terms of a set of operations more ammenable to implementation. Following this we detail some of the operations and transformations most easily expressed in a mathematical manner.

3.5.2 Algorithm Steps

The steps of the 3D-DFR contact algorithm are:

1. Transform a copy of the 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 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 *zone index set.* The local coordinates to discrete space coordinates transformation are calculated using:

$$w_u = \lfloor d_i + l_i/c_i \rfloor \tag{3.6}$$

These are the indices of the cells that bound a *zone* of candidate *source* cells to be checked against the *target* edges.

3. Perform cell index filtering by iterating over the *zone index set*. For each cell specified by the *index set* triples:

   (a) Map discrete space coordinates of cell (w_u) to ORL indices (ORL_u) using:

$$\begin{aligned} ORL_{slice} &= d_{slice} - o_{slice} \\ ORL_{prism} &= d_{prism} - (o_{prism})_{slice} \\ ORL_{cell} &= d_{cell} - ((o_{cell})_{prism})_{slice} \end{aligned} \tag{3.7}$$

i	-	local (real) axes x, y, z
u	-	discrete axes $cell, prism, slice$
w_u	-	discrete ordinate along discrete axis u.
d_i	-	offset to local origin from discrete space origin along local axis i
c_i	-	cell dimension parallel to axis i.
l_i	-	ordinates of point in local space along axis i.
$\lfloor\ \rfloor$	-	denotes the *floor* of a real value.
o_u	-	offset to start cell along discrete axis u

   (b) If the index triple maps to an ORL 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's geometry:

      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 coordinates of edge to cell coordinates using Eqn. 3.6.

         B. If the cell coordinate matches a valid ORL index triple then perform the geometric intersection operations, otherwise exit.

Number of Facets	$T_{Cyrus-Beck}$ [sec]	T_{DFR} [sec]	Speedup = $\frac{T_{Cyrus-Beck}}{T_{DFR}}$
8	0.000071	0.000027	2.6
16	0.000114	0.000028	4.1
28	0.000174	0.000028	6.2
44	0.000252	0.000028	9.0
104	0.000562	0.000025	22.5
188	0.000998	0.000025	39.9
296	0.001553	0.000025	62.1
380	0.002064	0.000025	82.6
536	0.003017	0.000025	120.7
716	0.004132	0.000025	165.3
824	0.004890	0.000025	195.6
1052	0.006359	0.000024	265.0
1304	0.007965	0.000026	306.3
1484	0.009161	0.000025	366.4
1736	0.010869	0.000025	434.8
2060	0.013071	0.000025	522.8
2312	0.014920	0.000025	596.8

Table 3.4: Point Classification Times and Speedup

3.5.3 Point Classification Timing Tests

A series of point classification tests are timed for both the DFR scheme and the Cyrus-Beck algorithm. The latter algorithm is the 3D extension of the 2D case discussed in Section 3.2. The tests check whether a point is contained inside a bounded spherical region represented using an unstructured list of polygonal facets (Cyrus-Beck), and then for the DFR scheme. The point classification test is the 3D equivalent of the 2D test that was described in Chapter 1.

Because the clock resolution is somewhat close to that of the operation being performed each test is applied for 10,000 iterations and the elapsed time averaged to give a mean time of execution. The timing results for polyhedra containing various numbers of triangular facets are shown in Table refpt-class-times and graphed in Figure 3.19. $T_{Cyrus-Beck}$ is the average time required to test a point using the Cyrus-Beck algorithm, similarly T_{DFR} is the average time required using the DFR representation.

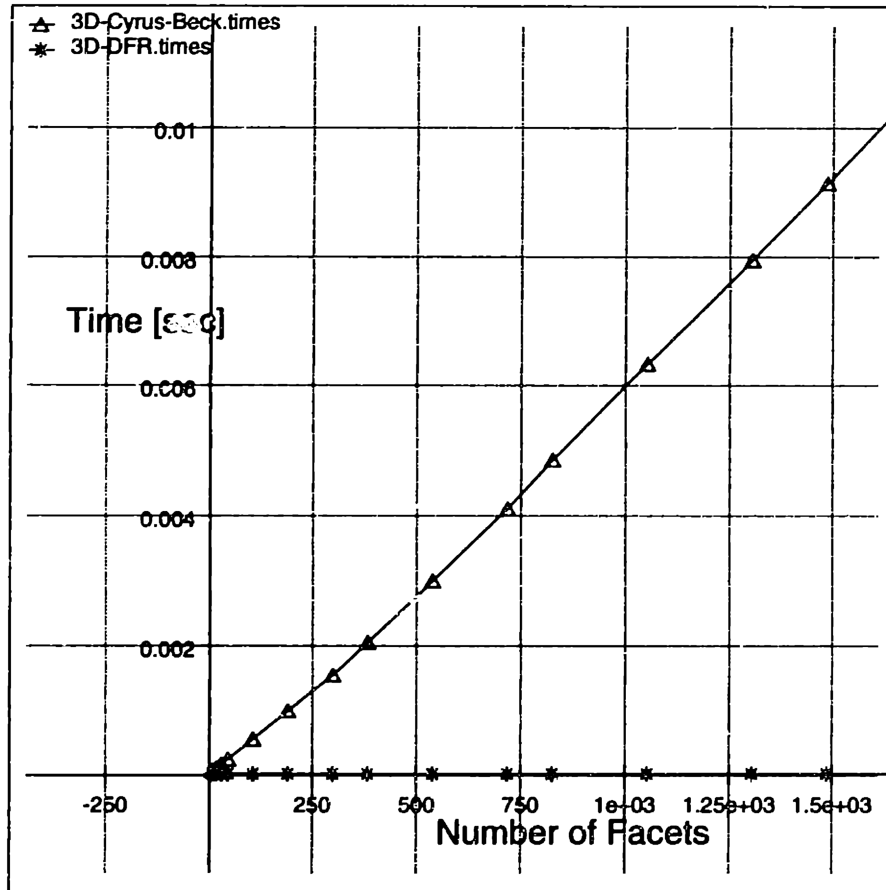

Figure 3.19: 3D Point Classification Test Times For DFR and Cyrus-Beck

The speedup using the DFR scheme over the Cyrus-Beck algorithm is shown in Figure 3.20. These results only relate to the simplest contact test (point containment), more comprehensive testing is ongoing. The results presented are therefore considered only to be a preliminary indication of the potential benefit of using the DFR scheme.

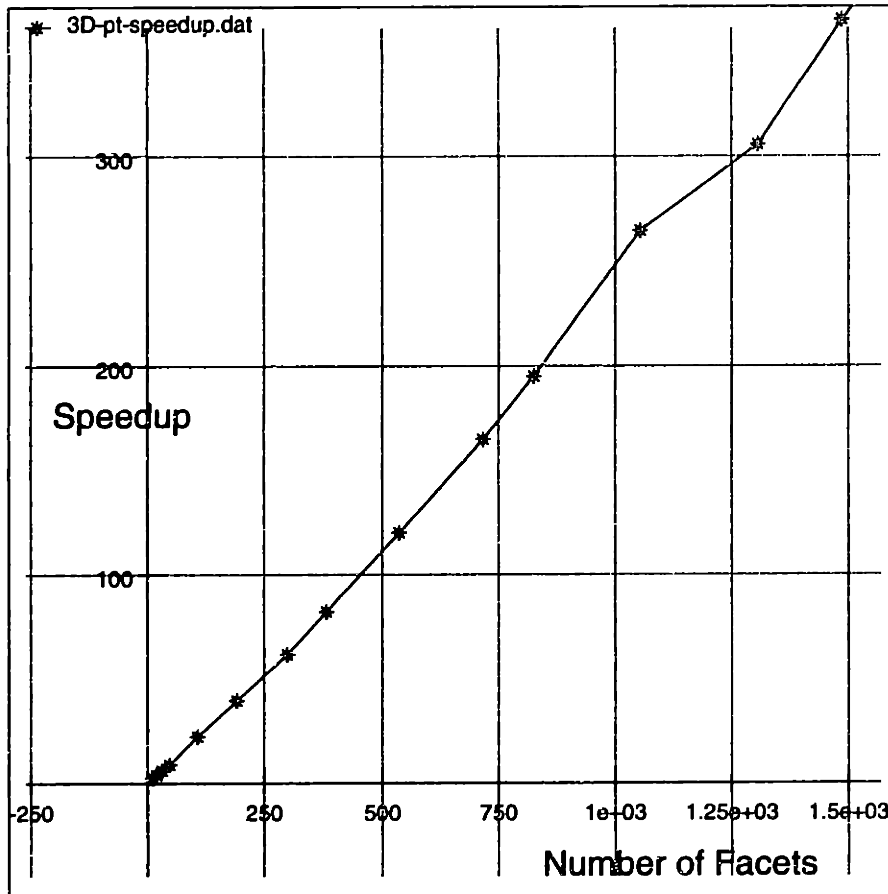

Figure 3.20: 3D Point Containment Speedup Using DFR Over Cyrus-Beck

3.6 Summary

In this chapter we have presented a general representation scheme for arbitrary 3D geometries. The scheme is designed to to facilitate efficient contact detection in multi-body simulation. The performance of the algorithm in basic timing tests indicates that it possesses significant benefits over certain traditional methods.

When used for *contact detection* a DFR based algorithm requires $O(N)$ operations, where N is the number of data points used to describe the surface geometry of the object. Each test of a point against the surface description of the other body can be treated as a constant time operation as evidenced by the timing results shown in Table 3.4. Each test involves only a transformation, a projection to map to a cell index, and a bounds test for inclusion against the surface represented in a single cell of the underlying structure.

In practice, the region of overlap for a pair of objects in contact depends on the resolution and scale of geometry describing the two objects, their locality to one another, and most importantly, the physical constraint of disjointedness, which dictates that objects may only overlap by an amount defined by the time scale of the numerical integration scheme being used. This time scale is in turn bound by the stiffness of the material components and the scale of the smallest object in the system.

The consequence of the numerical constraints can also be seen to introduce a variance in the number of points involved in the contact detection step over time and as such can only be measured experimentally. Intuitively we can expect the number of points involved will be significantly smaller than the total number describing the surface geometry and expect an average computational cost that is *independent* of the number of surface sample points (albeit for a relatively higher sampling density than most other schemes).

Chapter 4

Contact Detection

Contact detection is the algorithmic process used to automatically identify the occurence of contacting pairs of objects in physically based simulation. In the χ*mal* system the process starts with the spatial heapsort scheme presented in Chapter 2, which orders the objects along each of the principal Cartesian axes. Groups of objects that are neighbors (lying spatially close to one another) are then identified and paired-off for *contact resolution*. In Chapter 3 a contact resolution algorithm for objects described using the *discrete function representation* (DFR) scheme was presented. The result of DFR based contact resolution is a geometric representation of the contact region, a compact description of the surface elements that overlap. From this description a representative *face* and *vertex* are identified. The representative *vertex* is either part of the existing surface description of one of the objects or is constructed from the surface geometry describing the contact region. The culmination of the *contact detection* process is the derivation of the set of contact forces necessary to keep the objects in a physically realistic, disjoint state.

This chapter completes the description of the contact detection process. The first section discusses the treatment of contact between object not represented using the DFR scheme. In the following section contact forces between pairs of objects are calculated. Finally, an algorithmic procedure is described to manage pairs of objects that remain in contact over multiple time steps.

4.1 Theory and Practice

Up to this point I have described only one object representation scheme, namely the DFR scheme of Chapter 3. From a software engineering perspective the scheme is capable of describing objects using a single representation scheme. This avoids the need to handle multiple *special case* conditions which tend to clutter and often confuse the flow of the development process. Furthermore, the DFR provides this functionality without compromising the characteristics that lead to efficient contact detection.

However, a significant reduction in computational effort can still be obtained by exploiting the relative efficiencies associated with simple object types such as *points, spheres* and *planes.* In the case of spheres no discretization of object geometry is necessary (in contrast to the DFR scheme), and contact resolution is a set of simple operations that calculate the depth of overlap and the relative motions. This avoids calculating bounding box overlap, zone index sets or ORL indices as would be necessary if all objects were described using the DFR scheme. Similar arguments apply to the treatment of *planes.*

Given the simplicity and representational power of spheres (in addition to their wide spread use), it would be complacent to simply ignore their usefulness on the whim of a software engineering *rule of thumb.* For these and similar reasons the χ*mal* system currently supports 4 object types, namely *points*, *spheres*, *planes* and *DFR objects.* The various object types then give rise to an enumerable set of object-object pairings for which to perform contact resolution. In turn, each type of object requires a specific contact test. The treatment of a representative set of possible combinations are provided in the following sections.

4.1.1 Object Types

Of the four types of object used in the χ*mal* system, *spheres*, *planes* and *DFR* objects are each described with respect to an independent frame (a local coordinate system) associated with the individual object. *Points* are described with respect to the global frame of reference that is in effect.

Details specific to each object type can be summarized as follows:

1. *Points* are described as a vector of reals.

2. *Spheres* are described by a radius about the origin of a local frame.

3. *Planes* are represented as a closed quadrilateral region described by 4 vertices and a normal vector, all expressed in an local frame. Planes can be used to represent both bounded and extended regions depending on the context of the problem. For instance the quadrilateral may be used to represent a plane of infinite extent. In this case only a point lying on the interior of the quadrilateral and a unit normal to this plane are neccessary.

4. *DFR* objects are described by a local frame and a structured set of triangular facets which bound a *closed* 3D region. This representation was described in detail in Chapter 3.

4.2 Enumeration of the Contact Pairs

The set of object pairings is chosen so as to capture all of the possible combinations using the smallest number of methods necessary. For example, if a method is defined to calculate the intersection of a sphere with a plane, this method is equally applicable to finding the relevant geometric details of a plane intersecting a sphere. While the commutativity of these operations is obvious from a mathematical point of view, the implication in terms of software development is that only a single intersection test needs to be implemented between each unique pair of object types. Therefore we need only implement the intersection tests marked with an x in Table 4.1.

	Points	Spheres	Planes	DFR
Points	x	-	-	-
Spheres	x	x	-	-
Planes	x	x	x	-
DFR	x	x	x	x

Table 4.1: Cross Reference of Intersection Tests Performed

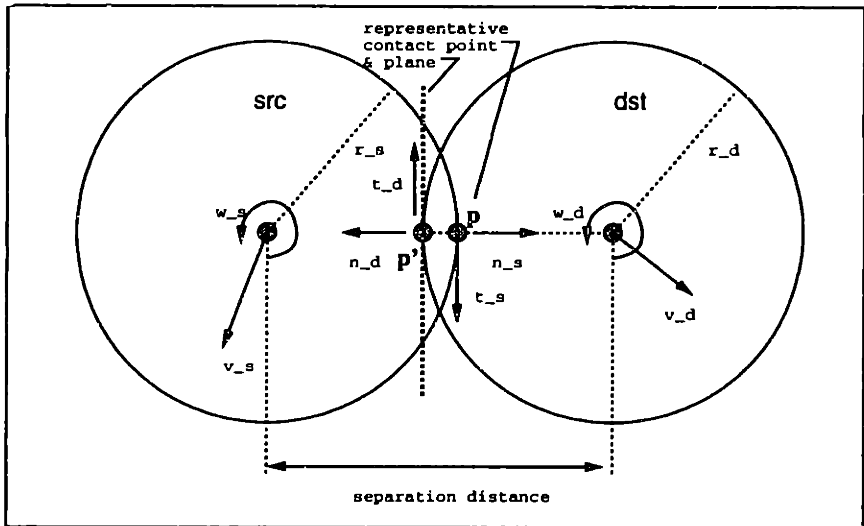

Figure 4.1: Sphere-Sphere Contact Details

The set of intersection tests is further reduced when degenerate cases are ignored. For instance, a *sphere-point* test is a degenerate case of a *sphere-sphere* test where one of the spheres is treated as a point. Similarly, the *plane-point* test can be considered a degenerate form of a *plane-sphere* test. Various ways to perform the *plane-plane* intersection tests were previously described in Chapter 3. The remaining tests are then the *sphere-sphere* and *plane-sphere* intersection calculations. These tests result in a representative description of the contact region, a contact point and a contact surface. The contact surface will be a polygonal facet in the case of planes and DFR objects, or the surface of a sphere object.

4.2.1 Spheres and Spheres

Contact resolution between a pair of spheres starts by designating one sphere to be the *source* object (**src**) and the other to be the *destination* (**dst**) object, as shown in Figure 4.1. Linear and angular velocities are denoted by $\mathbf{v}$ and $\mathbf{w}$ respectively. Subscripts s and $\mathbf{d}$ refer to quantities defined in the **src** and **dst** frames respectively. Each frame is expressed as a 4×4 homogeneous transformation matrix $\mathbf{T}$ which is composed of a translational component, $\mathbf{P}$, and a rotational component, $\mathbf{R}$ such that $\mathbf{T} = \mathbf{R} \cdot \mathbf{P}$.

The two spheres are said to be in contact if the sum of their radii is greater than the separation distance, d_{sep}, between their centers. The representative contact region then consists of a point on the surface of the **src** object and a tagent plane to the surface of the **dst** sphere. In the global frame, the relative linear and rotational velocities of the point $\mathbf{p}$ are given by Equations 4.1 and 4.2.The total relative velocity of the contact point in the **dst** frame is given by Eqn. 4.3.

$$\begin{aligned} \mathbf{vl}_{ds} &= \mathbf{v}_s - \mathbf{v}_d && (4.1)\\ \mathbf{vr}_{ds} &= (\mathbf{w}_s \wedge \mathbf{p}_s) \cdot \mathbf{R}_s - (\mathbf{w}_d \wedge \mathbf{p}_d) \cdot \mathbf{R}_d && (4.2)\\ \mathbf{v}_d &= (\mathbf{vl}_{ds} + \mathbf{vr}_{ds}) \cdot \mathbf{R}_d^T && (4.3) \end{aligned}$$

where $\mathbf{R}_d^T$ is the inverse of the rotational component of the frame tensor $\mathbf{T}_d$ describing the **dst** object's local coordinate system. Since $\mathbf{R}^T = \mathbf{R}^{-1}$ for a purely rotational transformation matrix, $\mathbf{R}_d^T$ orients quantities in the global frame into the local frame of the **dst** object. The representative contact face is taken to be a plane tangential to the surface of the **dst** sphere. The face normal is calculated at the point where a line joining the representative contact point $\mathbf{p}$ and the center of the **dst** object extends to intersect the sphere's boundary. The intersection point and the normal to the representative contact face are denoted as $\mathbf{n}_d$ and $\mathbf{p}'$ in Figure 4.1.

4.2.2 Planes and Spheres

For this combination of objects, the sphere is always selected as the **src** object, leaving the plane as the **dst** object. The top level intersection test for planes and spheres is to calculate the distance of separation between the sphere center and the extended plane in the global coordinate system. The distance of separation must be less than that sphere radius for intersection to have occured. This condition is necessary but not sufficient. When the condition is true, two further tests are required:

Test 1 Sample the quadrilateral vertices against the equation of the sphere. If any or all of the vertices lie inside the sphere, this is sufficient to indicate that the objects intersect. If intersection occured, retain the vertex that lies closest to the sphere center for further processing. If none of the vertices lie inside the sphere then the second test will exclusively decide whether the objects intersect.

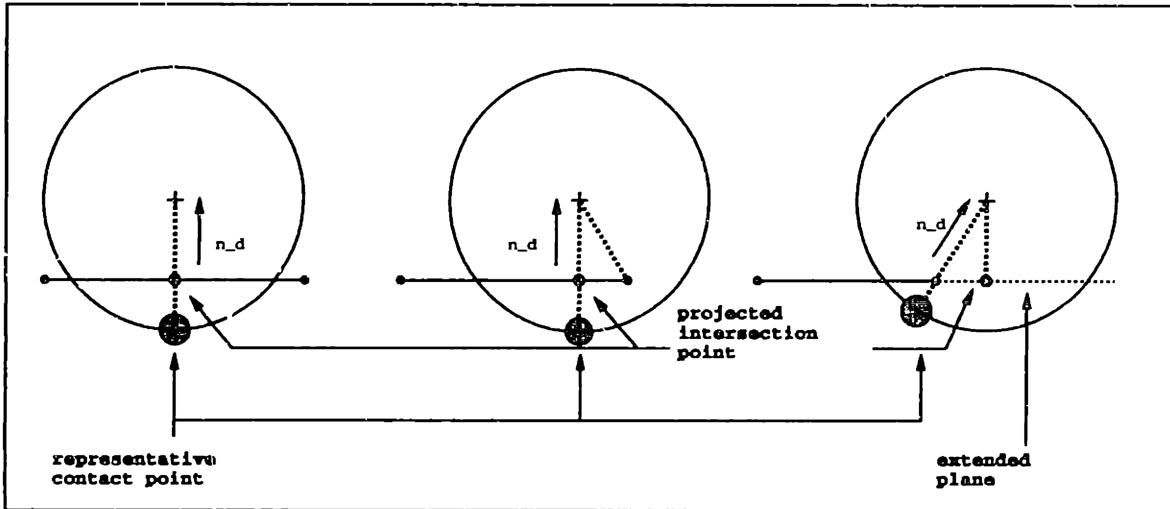

Figure 4.2: Construction of Sample Point for Sphere Plane Intersection Test

Test 2 Construct *the* point on the extended plane that is closest to the center of the sphere. This corresponds to the point of intersection of a line, parallel to the plane normal and passing through the sphere center, makes with the extended plane. The point is referred to as the *projected intersection point* shown in Figure 4.2. The *representative contact point* is then determined by two additional operations:

1. If the *projected intersection point* lies within the the quadrilateral, then this point is used as the *representative contact point.* This situation is shown in the first and second examples of Figure 4.2. The *representative contact point* is constructed as the point of intersection that a line segment emanating from the sphere center, and passing through the *projected intersection point*, makes with the sphere boundary.

2. If the *projected intersection point* does not lie within the boundaries of the quadrilateral but there are retained vertices from Test 1, then the *representative contact point* is constructed as the point of intersection on the sphere boundary that a line emanating from the sphere center and passing through the deepest penetrating of the vertices, makes with on the sphere boundary.

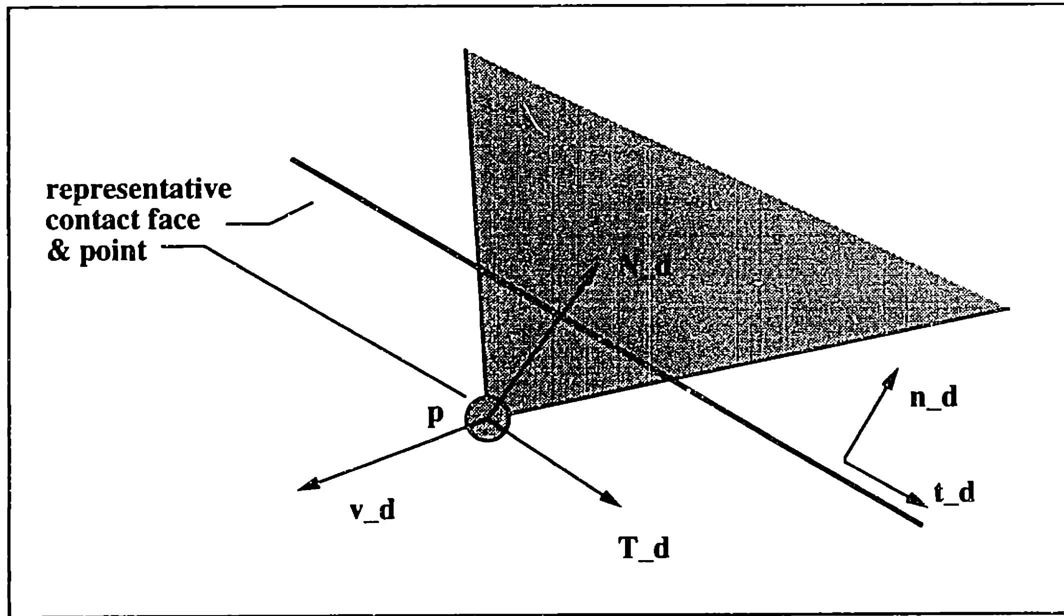

Figure 4.3: Contact Force Resolution

The *representative contact normal* is taken along the line joining *representative contact point* and the sphere center. The relative motion of the *representative contact point* with respect to the *representative contact face* is determined in the same manner described for *sphere-sphere* contact.

4.3 Force Resolution and State Update

Force resolution refers to the calculation of the normal and tangential forces arising at the interface of a pair of objects in contact. These forces (and torques) are required to maintain dynamic equilibrium of the particulate system being simulated. The contact force components are derived from a treatment of the intersecting portions of the two bodies (the *contact region*) and the linear and rotational motions of the object defined in the previous section. Once again, the *representative contact point* and *face* are described in the frame of the **dst** object. When objects penetrate one another, a penalty force is applied between the contact point and the contact face, so as to push the objects apart. This is simply modeled as a spring reaction at the interface. The most common penalty method attempts to remove the contact region in a single time step by applying an contact force that is sufficiently large. However, this will define a spring stiffness that is usually unrelated to the material stiffness (albeit the value for the bulk material). The use of this scheme can also introduce

large amounts of spurious energy to the system as a very small error in calculating the distance of penetration (the spring displacement) will be amplified.

To avoid this situation, and to utilize more appropriate measures for the material's stiffness, the χmal system incorporates an incrmental penalty force scheme that resolves the contact forces over a number of time steps by tracking the contact region. The idea is to apply a smoothly increasing and decreasing penalty force proportional to the rate of penetration of the contact point with respect to the contact face. Before describing the algorithms used to perform this task, the calculation of the incremental force components are first described.

Consider the contact region depicted in Figure 4.3. The contact force is calculated from the rate of penetration of the *representative contact point* ($\mathbf{p}$) with respect to the *representative contact face* defined by a unit normal, $\hat{\mathbf{n}}_d$. For one time step, dt, this corresponds to an increment in penetration of magnitude:

$$\delta_d = dt.|\mathbf{v}_d| \tag{4.4}$$

where $\mathbf{v}_d$ is the velocity of the contact point in the frame of the **dst** object. For a spring stiffness of k_d, the normal and tangential components of the incremental contact force are given by:

$$\mathbf{N}_d = \delta_d.k_d.\hat{\mathbf{n}} \tag{4.5}$$

$$\mathbf{T}_d = \delta_d.k_d.\hat{\mathbf{t}} \tag{4.6}$$

where $\hat{\mathbf{n}}$ is the unit normal to the contact face and $\hat{\mathbf{t}}$ unit tangent parallel to the contact face, oriented in a direction opposing the motion of the contact point.

To model sliding contacts a Coulombic friction model is applied. For a coefficient of friction denoted by μ, the shear resistance is given by Eqn 4.7. The limiting shear reaction, $\mathbf{T}_{limit}$, is calculated as the minimum of the tangential spring reaction, $\mathbf{T}_d$ and the Coulombic shear, $\mathbf{S}_d$, as in Eqn. 4.8.

$$\mathbf{S}_d = \mu\,|\mathbf{N}_d|\,\hat{\mathbf{t}}_d \tag{4.7}$$

$$\mathbf{T}_{limit} = MIN(\mathbf{S}_d,\ \mathbf{T}_d) \tag{4.8}$$

Finally, the torque at the interface due to the contact reaction is given by:

$$\mathbf{M}_d = (\mathbf{T}_{limit} + \mathbf{N}_d) \wedge \mathbf{p}_d \tag{4.9}$$

```
typedef struct contact
{
  struct
    particle * owner[2];     /* pointers to particle pair in contact    */
  int          srcidx;       /* index into owner vector of src object   */
  Matrix_t     f2f[2];       /* frame2frame transformation matrices     */
  int          orl[2][3];    /* ORL index triple of reference dst cell */
  int          dpi;          /* index into ORL cell triangles          */
  Vector_t     fni[2];       /* incremental global normal forces       */
  Vector_t     fti[2];       /* incremental global tangential force    */
  Vector_t     rcp;          /* representative contact point           */
  Vector_t     rcn;          /* representative contact normal          */
  float        dpmax;        /* max depth of penetration at contact    */
  int          cycle;        /* phase flag to mark stale contacts      */
  struct
    contact  * next, * prev; /* linked list pointers to contact objects */
} Contact_t, * Contact_pt;
```

Figure 4.4: Contact Object C Structure

4.4 Contact History Lists

The algorithm described in Section 4.2 yields a data abstraction referred to as a *contact object.* Each contact object contains the details necessary to resolve the normal and shear forces that occur due to a collision between two objects. The C language data structure describing a *contact object* is shown in Figure 4.4.

The previous section described how an increment of the penalty force and torque that results from the collision of two objects, is calculated. However, because of the magnitude of the time step used to calculate each increment, the instance of contact between two objects may not be removed immediately. In fact the two bodies may remain in a perpetual state of contact or their behavior would rather dictate that they move apart but in a more vigorous manner than that which would be possible from the *soft* penalty force previously described. This behavior is managed by maintaining a table of *contact objects* that contains a reference to all occurrences of contact between objects in the simulation. A 1D table of contact lists is used to keep track of the creation and consequent existence or removal of a contact between each unique pair of objects.

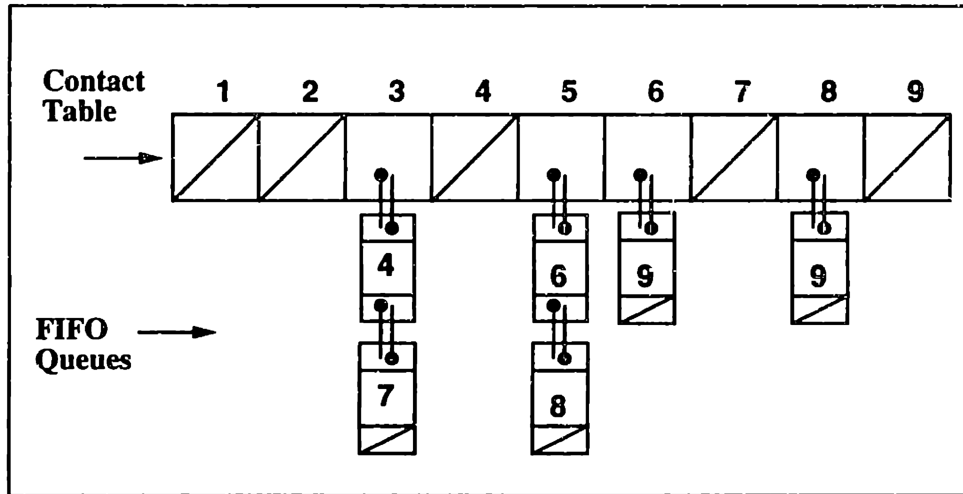

Figure 4.5: Contact Table Description

The operation of the contact table is described in the following steps:

1. Create a contact table of length N, where N is the number of objects in the simulation. Each table entry contains a pointer to doubly linked list of contact objects, and a counter to store 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 example, if object 3 is in contact with object 7, and object 9 in contact with object 6, then contact objects would be inserted into the contact table as entries 3 and 6, as shown in Figure 4.5. *Contact objects* are inserted at the tail of each list so that they can be processed in a FIFO (first in first out) order at each iteration or pass through the set of objects in the simulation. The FIFO ordering is intended to reflect temporal sequence of object-object contact instances.

At each iteration of the simulation, when a pair of objects are found to be in contact, the contact table is first consulted to see if there is a previous reference to this contact pair. This is 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 forces by appending the new increment of reaction forces and torques. Each time a contact object is added to the contact table, or if it is revisited, the `cycle` flag is set to a global value of `TRUE` or `FALSE`, corresponding to the phase of the current simulation cycle. This is used to differentiate between contact objects that have expired (the contact no longer exists) and those that we wish to continue to track. At the end of each simulation cycle, each list in the contact table is traversed to remove stale contact objects.

Chapter 5

Implementation

The purpose of this chapter is to document the structure of the complete software system and capture some of the decisions made over the course of this development. In Chapter 1 the development of a prototype simulation environment was presented as the theme of this thesis work. The objective is to provide a simulation system in which to model experiments performed on granular materials. The simulation system is viewed as an interactive software laboratory in which both experimental apparatus and the materials to be tested are modeled as physical objects that interact. The principal physical interaction that occurs is through surface contact which has been the technical focus of this thesis work.

The Modeling Environment

To tie the various components together an interpreter and *graphical user interface* based on the *Tcl/Tk* package are introduced. The functionality of these libraries are briefly described to facilitate an understanding of their expressive power and for later use in the development of χ*mal* applications discussed in Chapter 8. The Tcl interpeter becomes the principal channel through which a user interacts with the simulation environment. The mode of interaction will either be the graphical user interface, the command language, or more often, a combination of both.

To provide this interaction channel the library of built-in functions forming the command language understood by the Tcl interpreter is extended to include application specific functions. The added functions are high-level *hooks* into the dynamic simulation code which provides a mechanism to perform a DEM simulation interac-

tively via the interpreter. Complex simulations can be performed by writing application specific program scripts. Program scripts not only specify the initial state of the dynamic simulation but can be used to control the numerical experiment as it progresses. Within the interpretive environment the simulation system is represented as a set of abstract data types that encapsulate the primary components of the system, in addition to the application specific functions added to the command library.

Before detailing the implementation of the complete syste, the remaining components of the χ*mal* analysis pipeline need to be described, specifically the *physics* and *visualization* modules. This will complete the description of the four components making up the modeling system represented in Figure 1.4.

5.1 Dynamics of Motion

In this section a brief description of the numerical integration scheme used to update the simulation physics is presented. The governing equations for the linear and rotational motion of a discrete element are:

$$M\ddot{u}(t) + C\dot{u}(t) + Ku(t) = R(t) \tag{5.1}$$

$$I\ddot{\theta}(t) + C\dot{\theta}(t) + B\theta(t) = T(t) \tag{5.2}$$

M, K, I and B are $N \times N$ matrices corresponding to the mass, stiffness, moment of inertia and torsional stiffness of the body, respectively. C is the damping matrix. $R(t)$ and $T(t)$ are $N \times 1$ vectors corresponding to external forces and torques experienced by the body. The calculation of the external forces and torques, (F(t), T(t)), are calculated using the contact resolution algorithm explained in Chapter 4.

As with the calculation of object geometry, contact detection, rendering and volumetric properties, all solutions of the body motion are calculated with respect to the local coordinate system. The advantage of doing so in this case is that 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, [134, 95].

$$k = \frac{dt}{2} \tag{5.5}$$
$$A_n = k\, f(t_n,\, u_n,\, \dot{u}_n)r \tag{5.6}$$
$$\beta_n = k\,[\dot{u}_n + \frac{A_n}{2}] \tag{5.7}$$
$$B_n = k\, f(t_n + k,\, u_n + \beta_n,\, \dot{u}_n + A_n) \tag{5.8}$$
$$C_n = k\, f(t_n + k,\, u_n + \beta_n,\, \dot{u}_n + B_n) \tag{5.9}$$
$$\delta_n = dt\,[\dot{u}_n + C_n] \tag{5.10}$$
$$D_n = k\, f(t_n + dt,\, u_n + \delta_n,\, \dot{u}_n + 2C_n) \tag{5.11}$$
$$u_{n+1} = u_n + dt\,[\dot{u}_n + \frac{[A_n + B_n + C_n]}{3}] \tag{5.12}$$
$$\dot{u}_{n+1} = \dot{u}_n + \frac{[A_n + 2B_n + 2C_n + D_n]}{3} \tag{5.13}$$

Table 5.1: Runge-Kutta-Nystrom Numerical Integration Steps

5.1.1 Numerical Integration Scheme

The equations of motion are solved numerically using a forth order, *Runge-Kutta-Nystrom*, explicit time integration scheme. Equations (5.1, 5.2) are rewritten to formulate the explicit scheme as:

$$\ddot{u}(t) = M^{-1}\{R(t) - C\dot{u}(t) - Ku(t)\} \tag{5.3}$$
$$\ddot{\theta}(t) = I^{-1}\{T(t) - C\dot{\theta}(t) - B\theta(t)\} \tag{5.4}$$

Details of the Runge-Kutta-Nystrom numerical integration scheme are shown in Table 5.1, [67, 21]. The approximation of the position and velocity at time $t + dt$ are estimated from evaluating Equations 5.3, 5.4 at four stations as shown in Equations 5.6, 5.8, 5.9, 5.11. Equations (5.12 and 5.13 then yield the updated position and velocity of the body whose motion is being analyzed.

The value of the integration time step dt is determined from:

$$dt \le dt_{crit} = \frac{T_n}{\pi} \tag{5.14}$$

dt_{crit} is the critical timestep and T_n the smallest period of the system being analyzed. The critical time step is not usually known *a priori* and can require the somewhat circular approach of running the simulation with a configuration that is expected to require the most restrictive time step due to geometric considerations. Often the free-vibration frequency of a 1D oscillator is perhaps the simplest starting point, where $\omega_0 = \sqrt{\frac{k}{m}}$ (for a spring stiffness k and a mass m) corresponding to the discrete element requiring the smallest time step. For a uniform material stiffness the smallest discrete element needs to be examined. This value is then scaled by a nominal factor, for example $dt = T_n/10$ is often considered a satisfactory estimate [8].

5.2 Visualization Graphics

Two visualization channels are available in the current implementation of the χ*mal* system. The first and primary channel is through a software library interface to graphics hardware known as `gl` (which stands for *graphics library*)[1]. The second channel relies on a software library of `gl` emulation routines implemented using `Xlib` within the X Window System. Ultimately, the 3D rendering serves as a mechanism to analyse simulation results by using a variety of display formats and devices.

5.2.1 Visualization in Hardware

The principal use of graphics in χ*mal* is to render the time evolution of the physical system being simulated. This is performed either during the simulation or as a post-processing activity. In the first case the simulation size is considered sufficiently small enough that rendering does not adversely effect the overall running time and can often

[1]`gl` is a set of routines originally developed for use on Silicon Graphics workstations and a small number of custom graphics adaptors available on other vendors' machines. For this work the library interfaced with a *Gto* graphics adaptor connected to an IBM RS/6000 workstation. At the time of development `gl` was the only hardware interface library available for this work. A *de facto* hardware *independent* interface library known as *OpenGL* has since emerged as the preferred choice for development. The existing graphics code in χ*mal* is currently being ported to OpenGL but was not available before completion of this document.

be performed in real-time. The second method renders a sampled description of the simulation previously saved to disk. **gl** provides hardware support for drawing 3D points, lines, polygonal meshes and spline surfaces in addition to Z-buffer hidden surface removal and lighting. Graphics are displayed in **gl** windows using a virtual viewing device referred to as the ViewPOD (ViewPositionOrientationDevice). The ViewPOD is analogous to a movable camera that is positioned and oriented in 3D space.

5.2.2 Visualization in Software

Software emulation of the **gl** library was implemented to facilitate development of χ*mal* applications on platforms without graphics hardware support. The routines provide a minimal emulation layer for the essential graphics operations such as, creating a drawing buffer and viewport, a matrix stack for modeling and viewing transformations, and a set of drawing routines (points, lines, discs, polygons and triangle meshes). A benefit of equal importance is the ability to express, and/or use, the same sequence of **gl** rendering instructions between platforms that support only the X Window System and those additionally supporting **gl**. Many vendors have recognized that a software emulation library for OpenGL is attractive for developers who may not want to purchase dedicated rendering hardware for all of their machines. In the forseeable future much of dedicated hardware, usually in the form of Application Specific Integrated Circuits (ASICs), will be replaced with sets of *off the shelf* processors that perform the same tasks entirely in software.

The description is provided simply for completeness. As noted earlier, much of this effort will become redundant when OpenGL is more commonly available. The **gl** emulation library is called **glxem** and currently contains the routines listed in Table 5.2.

One of the more glaring shortcomings of the **gl** (and the OpenGL) library is the lack of an abstraction layer. Essentially all of the graphics hardware is accessed as a collection of global entities and requires all context and state switches to be managed through the user-level code. To provide the hardware emulation, a software stack and graphics window had to be created and made accessible *globally*. This deviates from the modularity and scoping of the overall software but was necessary to make the emulation layer compatible, and transparent to the user. Where possible

Function Name	Purpose
gconfig	initialize the emulation software, allocate buffers etc
winopen	create a rendering window
doublebuffer	allocate two drawing buffers for fast graphics display
zbuffer	allocate buffer for hidden surface removal
swapbuffers	copy contents of active drawing buffer to graphics window
clear	clear contents of active drawing buffer
zclear	clear contents of z-buffer
bgnpoint,endpoint	start—stop drawing points
bgnline,endline	start—stop drawing lines
bgnpolgon,endpolygon	start—stop drawing a polygon
bgntmesh,endtmesh	start—stop drawing a mesh of triangles
v2f,v3f	add a 2D—3D vertex to drawing list
n2f,n3f	specify a 2D—3D normal
loadmatrix	load a matrix at the top of graphics stack
pushmatrix	copy top matrix of existing stack and load on top
popmatrix	remove top matrix in graphics stack
multmatrix	pre-multiply top matrix of stack by another matrix
getmatrix	copy top matrix in graphics stack
translate	translation drawing coordinate system
rotate	rotate drawing coordinate system
scale	scale drawing coordinate system

Table 5.2: `gl` Emulation Routines

a secondary, private description of the graphics system is maintained to allow calls to the emulation library to be made from within a scope local to a given application routine. This mechanism is described in Section 5.5.

Function Name	Purpose
v_copy_3d	copy a vector
va_fill_3d	fill (mutate) vector with values
v_add_3d	add two vectors
v_sub_3d	subtract 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 tmatrix
v_orient_tm_3d	orient a vector by post-multiplying by a tmatrix

Table 5.3: Example Vector-Matrix Routines

5.3 Matrix Transform Library

The mathematics of multibody dynamics are effectively described in terms of homogeneous vector and matrix expressions, [22, 101]. To provide consistent descriptions of the various transformations a library of vector and matrix operations was implemented. These routines are based on the 2D and 3D χ*mal* vector and matrix primitives. A nomenclature[2] for the function names iscomposed of three fields as shown in the generic prototypes:

```
<return type> tm_<operation>_<dimension>(parameter list);
<return type> v[a]_<operation>_<dimension>(parameter list);
```

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

[2]This style is based loosely on the scheme used by Chen in the simulation system *3d*, [19].

suffix on the name field, `v[a]`, indicates that the contents of the source object in the parameter list is changed by the functions 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 dimensionalty of the operation, 2D or 3D. The principal routines contained in the library are listed in Table 5.3.

5.4 User Interface - Abstraction and Control

One of the primary goals of this work is to provide an environment that can be used by engineers to model and examine complex multi-body simulations. Specifically we wished to develop an environment that allowed researchers to interact with a simulation via channels at a high level of abstraction, using constructs and a language commensurate with the modeling process as opposed to the programming language that the system was developed in. The approach adopted here follows on the experience of the developers of *testbed* modeling systems such as *3d*, an interactive medical simulation system by Chen [19], real-time environments such as *bolio* [11] and *Thingworld* [94], and commercial systems such as, Autodesk's *AutoCAD* and Wolfram Research's *Mathematica*. The flexibility of these systems is considered to lie in the use of an embedded interpreter in which to execute user-defined applications in the form of scripts.

The interpreter used in the χ*mal* system is called Tcl (Tool command language), developed by Dr. John Ousterhout at U.C. Berkely. Tcl is currently a public domain resource in the form of a **C** language library. Tcl also has a powerful windowing extension called Tk (Tool kit) which runs under the X Window System. Use of these tools allows any application to be extended to include a command language interface through which applications are either scripted (interactively or as a text program) or through an interpreted graphical user interface layer. The behavior is much like that of a LISP or HyperCard interface with command extensions, or *hooks*, into compiled C, C++ functions provided by the application developer.

A key to the success of this resource is in the developers ability to keep application and interface coding independent of each other. Where needed, an interpreter shell is invoked inside the application code and can then be used to manipulate the flow control of the application sequence. By adding developer specified code to the

dictionary of interpreter commands, the language understood by the interpreter may be extended in any manner desirable.

An brief description of the Tcl and Tk libraries is provided to introduce the syntax of the language used to build the χ*mal* applications discussed in Chapter 8.

5.4.1 Tcl - Tool command language

Tcl is a C library of routines containing a parser, interpreter and a command language which supports variables, flow control constructs, 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 sequence 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 backslash (\) character denotes line continuation.

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

```
set attributes {red steel {yfield gravity} elastic}
```

which assigns (sets) the argument `attributes` to refer to the list:
`{red steel {yfield gravity} elastic}`.

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

```
set a 2
set b 3
set c [expr $a+$b]
```

sets the variable `c` to the result of the expression `$a+$b`, i.e. the string "5". Here the square bracket operators, [] `[` and `]`, force the interpreter to evaluate the expression

expr$a+$b before setting the value of the variable c. This explicit evaluation syntax can be hidden somewhat by defining a Tcl procedure such as:

```
proc plus {arg1 arg2} {
expr $arg1+$arg2
}
```

which, when called, returns a string corresponding to the value of the evaulated expression inside the procedure. The developer is free to extend and control the behavior of the Tcl environment to their own requirements through newly defined Tcl procedures providing an excellent medium in which to quickly prototype an applications behavior before casting it permanently in C.

5.4.2 Tk - Tool kit

Tk is an extension to the Tcl package that provides command level control of the windowing environment that the application is being developed in. The current release of the extension comes in the form of a library of C routines that build on top of the X Window System, specifically *Xlib*. Through this interface, the same protocol and *look and feel* as X and OSF Motif are available. The library contains a selection of predefined windowing primitives or *widgets* such as; *buttons, labels, entry widgets, drawing canvases*, etc. Again, all widgets are built on top of X windows, and can be accessed as such. Graphical user interfaces (GUIs) are constructed via user written scripts, automatic script generators such as XF, and predefined resources accompanying the package. New widgets, composite widgets and heterogeneous widgets can be created and managed uniformly within the same environment. For example, this feature provided a mechanism to encapsulate the non-standard windowing protocol of the gl graphics system described in the previous section.

5.4.3 Application Independence

A concerted effort has been made to maintain independence between the application and the interface. This characteristic is desirable from the point of view that the developers can manage the concurrent development of the environment in a platform independent 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 data transmission protocol 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. In the abstract sense this appears to be innocuous but from a software development leads to application code with interface code deeply embedded within it. Subsequently problems arise on both sides, when the application is required to be ported to some other interface platform, or simply for maintenance of the application code by new developers that only want to access details of the application side of the project. I have tried to maintain an implementation that adheres to the following guideline:

> The application specific code should not have, nor should it need, any knowledge of the interface component of the software system.

The is the obverse of the argument, the flipside however will require that the interface code knows something about the application. This can make the interface code just as complex as the application code was before. To simplify this problem a barrier layer of software, a *dispatch layer*, is constructed between the application code and the interface code. Data to be used by the application is obtained through the interface and then transferred through the dispatch layer. The *dispatch* functions are then convert the input data into a format digestible by the application routines. These functions are then registered with the `Tcl` interface read-eval-print main loop, corresponding to the *dispatch layer* in Figure 5.1.

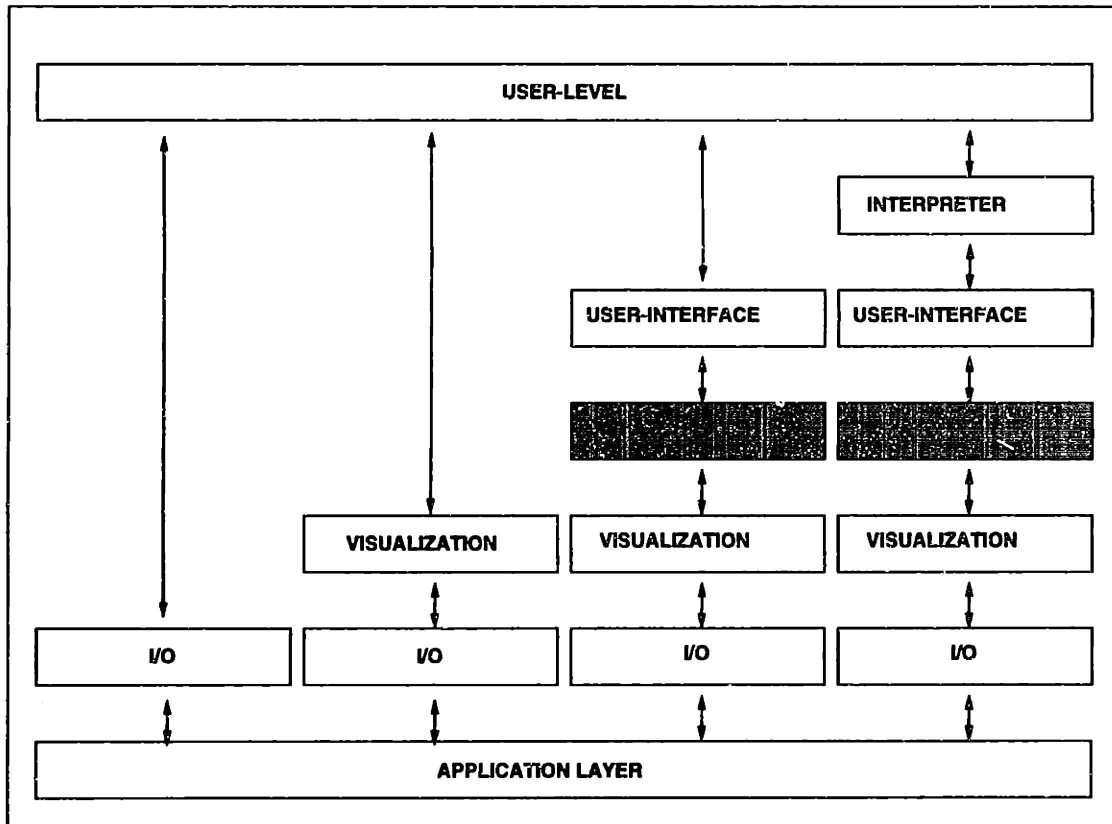

Figure 5.1: Application Independence

5.5 Abstract Data Types

The χmal environment is described in terms of a collection of objects (from a set of abstract data *types*[3] and a command language to operat objects. The top level data abstraction is a container of type `Xmal_t`. An ob this type is used to encapsulate an entire instance of the χmal system, in which other objects can be dynamically created, manipulated and destroyed over the course of a simulation. Reference to the container is sufficient to gain access to any component of the active simulation environment. This has a dual purpose. First, it provides a mechanism where all of the interface utilities related to the system need access to a **single** data channel. Secondly, it allows any instance or entity in the simulation to be explicitly referenced while avoiding the problem of common naming schemes in different contexts and/or the (re)declaration of each piece of data to have a global scope over the application and interface code modules.

[3]Not objects in the C++ sense, which would be inefficient and broken.

All dispatch functions then need only to have the a common (required) function prototype and template, internally accessing common global references to the χmal system and propagating it to the desired application routine. This also provides the data consistency checking necessary to ensure consistent communication between the two layers.

The primary abstract types in the χmal system are:

Xmal_t - A container to encapsulate the system as a hierachical set of references to all of the primary data abstractions in the system.

Simulation_t - A container for the simulation parameters describing the physics.

Particle_t - A common container to store a description of the χmal objects.

Stable_t A container for the sorting tables described in Chapter 2.

Matrix_t/Vector_t - Floating point primitives for numerical calculations.

Bbox_t - Floating point primitive for clipping calculations.

Contact_t - A container for the contact objects described in Chapter 4.

Co_table_t - A container for the contact tables described in Chapter 4.

UIParams_t - A container for the `Tcl/Tk` user interface.

GLXem_t - A container for parameters used in the `gl` emulation library.

Sball_t - A container for the virtual camera (ViewPOD).

The global container contains references to the primary χmal types as shown in Table 5.4. 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.

<table>
<tr><td rowspan="26">xmal</td><td rowspan="5">simulation</td><td>clock</td></tr>
<tr><td>clock time-step</td></tr>
<tr><td>force-fields</td></tr>
<tr><td>physical-constants</td></tr>
<tr><td>state-flags</td></tr>
<tr><td rowspan="2">particle-set</td><td>particle-table</td></tr>
<tr><td>number of particles</td></tr>
<tr><td rowspan="4">sorting-table</td><td>index-table</td></tr>
<tr><td>rank-table</td></tr>
<tr><td>flag-table</td></tr>
<tr><td>number of table entries</td></tr>
<tr><td rowspan="2">contact-table</td><td>contact-queues-table</td></tr>
<tr><td>number of contact-queues</td></tr>
<tr><td rowspan="4">user-interface</td><td>interpreter</td></tr>
<tr><td>tty/terminal interface</td></tr>
<tr><td>graphical interface window</td></tr>
<tr><td>I/O buffers</td></tr>
<tr><td rowspan="4">trace</td><td>trace list</td></tr>
<tr><td>list length</td></tr>
<tr><td>output stream</td></tr>
<tr><td>flags</td></tr>
<tr><td rowspan="3">graphics</td><td>window</td></tr>
<tr><td>buffer</td></tr>
<tr><td>context/state</td></tr>
</table>

Table 5.4: χ*mal* Objects

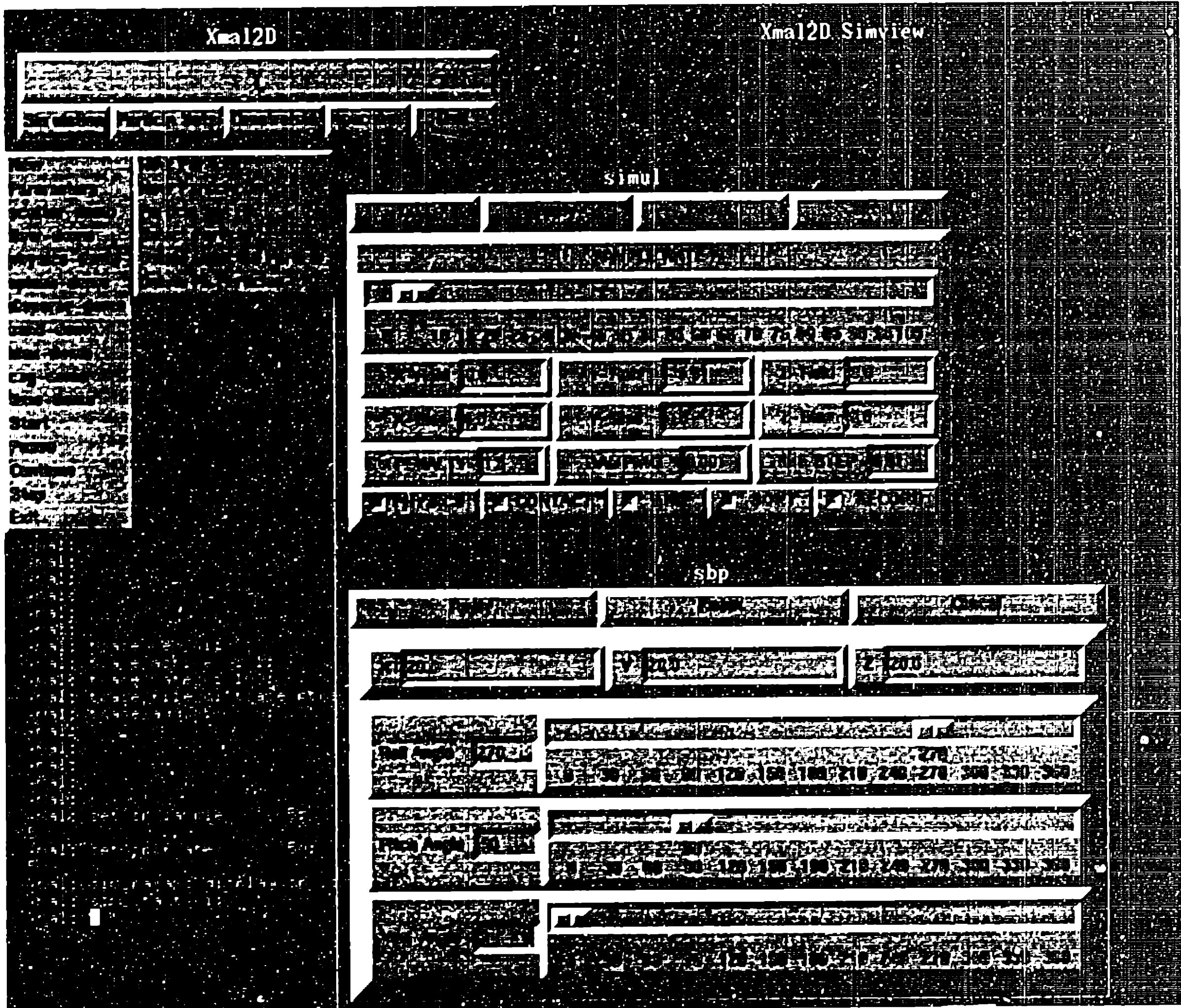

Figure 5.2: Example View of Graphical User Interface

5.5.1 A Template χ*mal* Application Script

When the χ*mal* system is first evoked, a shell is created with a command level interface to the application. Depending on the startup configuration a graphical user interface resembling that shown in Figure 5.2 may also be launched. Using either of these channels the basic ingredients of a χ*mal* simulation would contain the the `Tcl` code shown in Figure 5.3.

```
proc xmal-demo { < parameters> } {

# create new simulation object of dimension < args >;

sim_new      < args >;

# Initialise with some default simulation parameters < args >;

sim_params_init < args >;

# Create a description of physical objects that you want to simulate.
# For example, create two DFR objects described by < args >

p_dfr_init   < args >;
p_dfr_init   < args >;

# Create and initialise the sorting and contact tables

spatial_new;
co_table_new;

# start the simulation running until it is
# interupted by a shell command or completes

sim_start;
}
```

Figure 5.3: A Template Application Script

If this script was stored in a file called `demo-script.tcl`, it is loaded into the intepreter by typing `source demo-script.tcl` at the `xmal%` prompt and then executed by typing the name of the `Tcl` procedure `xmal-demo`, as shown below:

```
xmal% source demo-script.tcl;
xmal% xmal-demo;
```

The script can be embellished with additional graphical user interface commands and more complex `Tcl` syntax, but the basic structure is applicable to most χ*mal* simulation scripts. The code shown here forms the core of the scripts used to run many of the applications that are the subject of the next chapter.

Chapter 6

Parallel Computing

Over the past 10-15 years parallel computers[1] have become increasingly available to the research community, both in academia and industry. While this applied technology is relatively young, theoretical aspects of the field have been explored since the outset of modern computing in the late 1940's. In this chapter a review of the basic concepts involved in parallel computing is presented to serve as a background for the distributed simulation environment developed in this work. We first describe parallel machine models and then examine two of the more commonly used programming models. In particular we shall look at the *message passing* programming model for *distributed memory* parallel machines. The latter portions of the chapter describe an emerging programming standard used in the development of portable message passing applications. The standard is known as the *Message Passing Interface* (MPI) [31].

6.1 Parallel Computing Applications

Parallel computing applications arise in situations where the time required to process a data set on a serial computer is considered to be excessively long. This occurs where the problem to be solved is considered computationally complex and/or the size of the problem is very large.

Problems matching the situations described above are not hard to find in academic and industrial research, particularly in the area of numerical analysis and

[1] We shall use a working definition of *parallel computer* to mean a set of processors that are able to work cooperatively to solve computational problems, after Foster [32].

simulation. Some of the more challenging problems[2] include the simulation and forecasting of weather, modeling ocean currents, DNA sequencing and the development of new materials for the electronics industry. As the technology evolves and becomes more accessible, increasing industrial and commercial use of this technology has occured. Examples include tasks such as processing large sets of data (medical data, video images and bank transactions) and controlling complex systems (manufacturing, chemical and bio-engineering processes, and ironically in managing networks of computers).

Increasingly larger problems (in terms of data) can be expected to require more computation time. Memory is currently considered an economic constraint whereas practical considerations come to bear when attempting to perform computation over long periods of time. Two principal factors determine this limitation. The first is the speed of a processor measured in terms of the number of sequential operations it can perform per second. This has a hard upperbound in the speed of light. Secondly, material tolerances can dictate the time scale over which it is feasible to operate a single processor at a sustained (uninterrupted) rate, approaching capacity.

For these reasons, problems are divided (distributed) over a number of processors reducing the total sequential time required. The remainder of this chapter examines how this distribution is performed and the techniques used to manage it.

6.2 Parallel Computer Architectures

In this section we look at two contrasting parallel computer architectures. These are *distributed memory* parallel computers and *shared memory* parallel computers. As the terminology implies, a distributed memory machine is a set of processors each with its own local memory, interconnected through a communications network. A shared memory machine consists of a set of processors that *share* a common memory space. Access to the shared memory is provided through a single channel referred to as a *bus*. Processors then compete and cooperate reading and writing data. The shared memory architecture is also referred to as a *multiprocessor* machine. The two basic architectures are depicted in Figure 6.1.

[2]These problems, amongst several others, are referred to as *grand challenge* problems in science and engineering by the National Science Foundation (NSF) committee on High Performance Communications and Computing (HPCC).

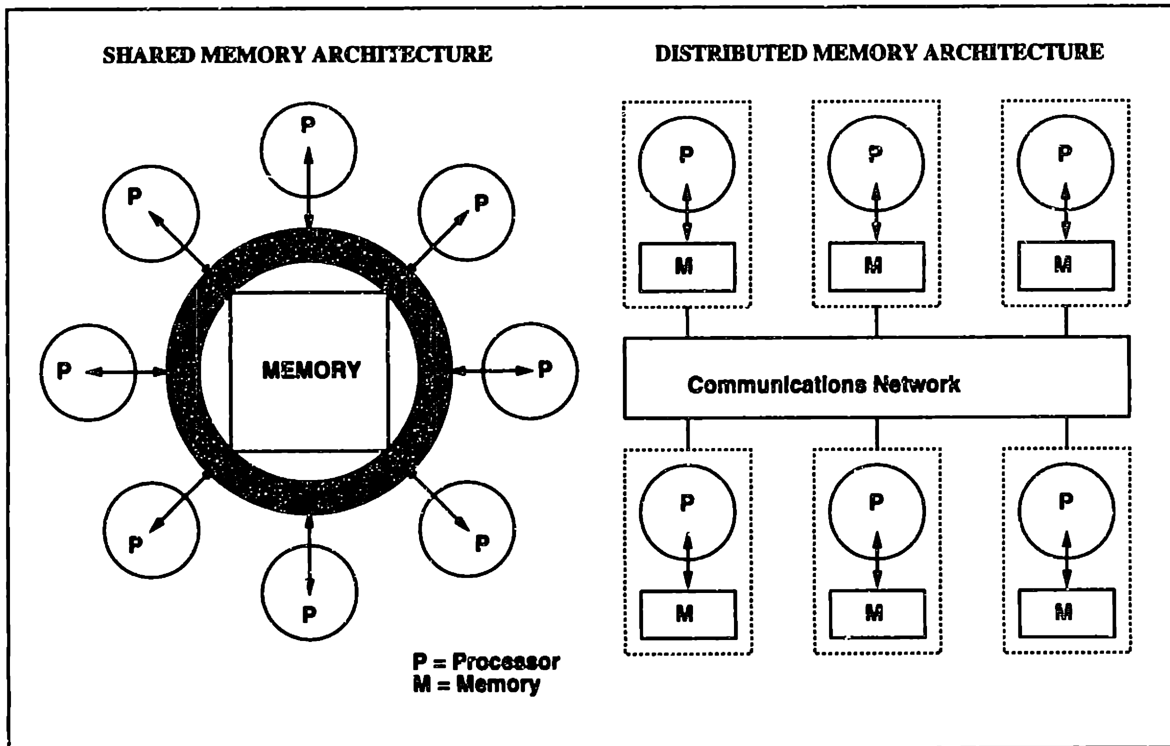

Figure 6.1: Simple Models of Parallel Computer Architectures

Both architectures are generally said to be multiple instruction, multiple data (MIMD) machines. This means that all processors in the machine is capable of executing a different program or instruction set, concurrently. A specialized (reductionist) form of MIMD machine known as a single instruction, multiple data (SIMD) machine has also been realized. SIMD machines operate by executing a single instruction set on *all* processors, synchronously. While this limits the scope of suitable problems, the relative simplicity of its hardware implementation makes it appealing for dedicated tasks such as image processing, free-text search, and the simulation of systems that can be easily modeled using grids, such as fluid dynamics.

At present distributed memory architectures appear to be more prevalent in industry than shared memory machines. For this reason a larger proportion of time will be spent considering the different aspects of distributed memory computers. We note however that algorithms developed for distributed memory machines are not excluded from implementation on shared memory architectures, and vice versa. In fact, many realizations of parallel machines blur the distinction between the models at various levels of hardware. Emulation of programming models devised for one form of architecture can be provided in the other, with only a small performance penalty.

6.2.1 Distributed Memory Machines

A distributed memory machine is a set of processors connected to each other though a communications network. Each processor has its own local memory which it can address directly. Through the connectivity provided by the communications network, the processors can also *indirectly* access the data stored in the local memory of other processors. Distributed computing thus introduces the concept of communication to computation. Data residing in the memory of one processor may be required by some other remote processor. This requires the transfer of data over the communication network. Sets of data transferred between processors are commonly referred to as *messages*. Performance evaluation on uniprocessor machines can be measured in terms of the number of *read/write* operations executed. Parallel computing performance now includes the number of communication operations such as sending and receiving messages[3].

Before describing the programming models used to execute this communication, we briefly examine how processor connectivity is provided through a communications network. The topology of the network is one of the principal factors requiring hardware and software developers to expend much effort in providing efficient implementations of the programming models used for with distributed memory machines.

6.2.2 Communication Networks

In an idealized distributed memory machine the communications network is assumed to consist of dedicated channels connecting each processor to every other processor in the system. A second assumption is that the cost of sending a message between two processors is independent of *all* other communication activities occurring on the network at the time.

The reality is that each processor has only a limited number of *ports* to support such connections. This reduces the number of dedicated channels that can be assumed. A more conservative assumption is to consider each processor as having

[3]Performance measures in terms of these operations are obviously a simplified view of reality. In numerical computation higher level metrics are chosen, such as the number floating point operations. Caching, buffering and pipelining can also have a significant effect on performance beyond that of the perceived number of read/write operations. Similarly in communications performance the measure of send/receive operations is subject to variation. This will depend on factors such as the startup time for a communication connection between two processors, or the size of the messages involved.

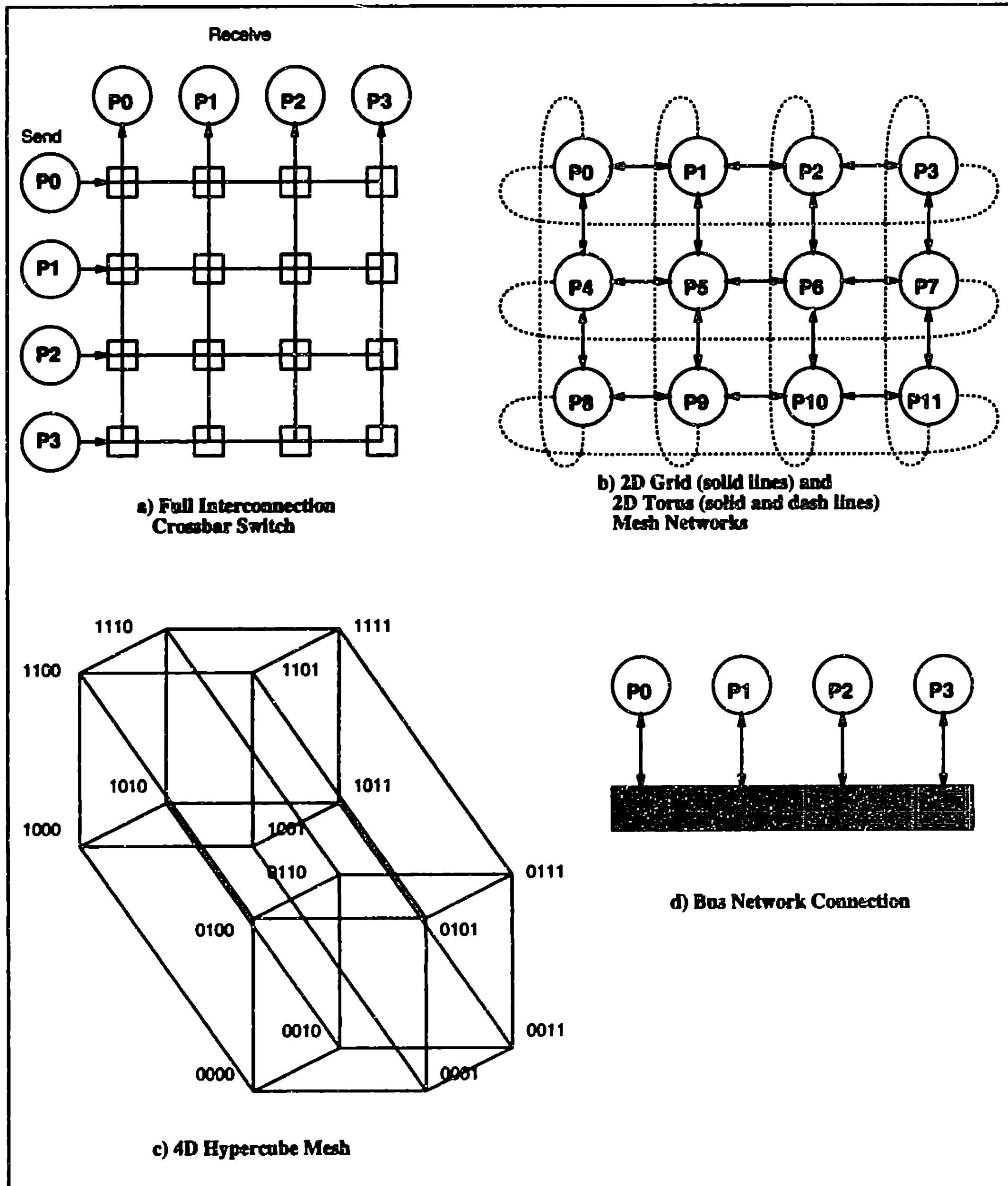

Figure 6.2: Communication Networks

only a single communications port. This implies that when two processors try to communicate with a third processor at the same time, they will be in contention for the communications channel.

A tradeoff in communications overhead versus the number of ports a processor maintains is achieved through using a variety of highspeed switching networks, dedicated to route messages in predefined patterns. The idea is to relay messages through intermediate processors to reach their destination. The design of the network then becomes a tradeoff between the number of wires and the rate at which data can be transferred between processors.

The most effective communication networks using this model is one where communication between one pair of processors is independent of all other processors. This form of connectivity is implemented through an all-to-all *crossbar switch* as shown in Figure 6.2.a. In this network all processors are connected to one another through input and output lines from the switch. This is indicated in the figure by showing each processor twice. This form of network connectivity does not scale well as the number of input/output lines are predefined. They are also expensive to build.

A more common communications network is a mesh that connects nearest neighbors on a grid of processors. An example of a 2D network of this form is shown in Figure 6.2.b. To account for boundary processors, wraparound connections are often provided yielding a torus topology. 3D grids and 3D toroidal meshes are also implemented in this manner. Networks of this form have the advantage that they are relatively easy to build in terms of wiring connectivity. Communication takes place by transferring messages through intermediate processors.

Another common mesh topology network is that of the 4D hypercube as shown in Figure 6.2.c. As in the 2D and 3D meshes, each processor is connected to nearest neighbors, in this case to 4 other processors. Message transmission is again performed via intermediate processors on the grid.

Lastly, the *bus* based network connection is shown in Figure 6.2.d. This form of communication channel is used with a *network of workstations* (NOW) . Examples of *bus* based communications are Ethernet and more recently *asynchronous transfer mode* (ATM) switches. Communication on these networks is always contentious as a bus can only transmit a single message at a time. Transmission bandwidth coupled with the constraint of the single communication channel detract from the performance of NOWs utilized as a parallel machines.

The choice of network topology depends on factors such as the problem domain that the machine will most often be applied to. For instance, a machine dedicated to image processing might interconnect the processors using a 2D grid and numerical analysis in structural mechanics applications would be best served using a 3D grid. Signal processing techniques using Fourier analysis would in turn benefit from a hypercube configuration. Since communications operations are typically much slower than computational operations such as a floating-point calculations, much attention is given to selecting the most appropriate communications network to optimize the computation. Unfortunately this has a downside in that committing to a dedicated network topology may reduce the generality of the machine in terms of efficiency. In recent years a network topology known as a *FAT-tree* has attracted much attention, [70]. This architecture is considered to yield the best general qualities of any of the networks discussed above, while remaining highly scalable.

We have seen that the topology of the communications network that the hardware controls may differ across platforms. We also note that most currently available systems employ some form of specialized communications hardware. As a consequence of these design issues, the user level machine can look very different from platform to platform. This becomes apparent in the way the vendors implement the software interface to the communications network. As one can imagine, there are many ways to provide this functionality on an individual multiprocessor machine. As a consequence most vendors differ in their implementations. This becomes problematic when trying to develop *portable* applications, applications that can run on a variety of machines without changes. This issue is of primary concern to many developers[4] We will return to this issue it in subsequent sections. First we look at some of the more generalized programming models that incorporate communication in computation.

[4]In the past year alone industry has seen the demise of several of the major parallel computer vendors. These include Thinking Machines Corp (TMC), the makers of Connection Machines; Kendall Square Research (KSR) one of the better known shared memory architecture vendors, and CRAY, perhaps the best known supercomputer manufacturer in the world.

6.3 Parallel Programming Models

We next describe two different programming models that arise when using distributed memory machines for parallel processing. As we have seen, many protocols for communication between processors are possible with the variety of communications architectures available. However it is possible to generalize these to a small number of abstract programming models for interprocessor communications[5]. We shall look at two of these general models here, the message passing model and the data parallel model.

6.3.1 The Data Parallel Programming Model

The data parallel approach to programming attempts to divide the problem into data elements, each of which is associated with its own processor. Each processor then performs the same computational operation on each element by receiving instructions from a central control program. There is only one thread of control, executed concurrently on every processor. This model is analogous to the SIMD (single instruction multiple data) machine model described earlier. Indeed, the data parallel model is perhaps the most effective manner in which to program SIMD machines.

Data parallel languages typically make the transfer of data from remote processers transparent to the user at the software level by providing a general syntax to address the combined memory of all processors involved in the computation. This abstraction provides the user with a view of a virtual machine composed of all of the processors' memory accessible through a flat address space. A detailed knowledge of the underlying machine architecture is then deemed unnecessary. Although in reality this is a great simplification and serves only to contrast against more *explicit* communication programming models such as message passing, which will be described shortly.

While the largest of the so called *massively* parallel computers contain only hundreds or thousands of processors, it is easy to imagine a data set that contains a larger number of elements. To accommodate this situation, data parallel programming languages provide an abstraction known as the *virtual processor*. This construct provides a way to associate multiple data elements with a single *physical* processor,

[5]In this section the collective term *processes* is used to denote a set of programs executing on either a single *processor* or as a set of individual programs running on independent processors.

while the user level machine retains the view of one processor per data element. The model also supports the concept of *contexts* where different sets of virtual processors coexist and interact under user control. A general syntax to address the combined memory of all processors involved in the computation makes the transfer of data from processor to processor transparent at the software level.

Data Parallel Matrix Transpose Example

As an example of a data parallel program we show how one might perform a matrix transpose operation. We use the C* programming language[6] for the implementation and assume that the reader has some familiarity with the C programming language, [65].

To set the scene, we first show a serial implementation of a 2D matrix transpose routine written in ANSI C, Figure 6.3. For simplicity we assume that the matrix is square and the dimension is known *a priori*.

```
#define   MSIZE   1024

void serial_transpose (int (* a_in)[MSIZE], int (* a_out)[MSIZE])
{
  int i, j;

  for (i = 0; i < MSIZE; i++)
   {
     for (j = 0; j < MSIZE; j++)
      {
        a_out[i][j] = a_in[j][i];
      }
   }
}
```

Figure 6.3: Serial Code for Matrix Transpose

Using C* the matrix transpose operation can be implemented as shown in Figure 6.4. In this example we assume that a 2D matrix is distributed over the set of

[6] C* (pronounced *C-star*) is a data parallel extension of ANSI C. It was developed by, and is a trademark of Thinking Machines Corporation, Cambridge, Massachusetts, [113, 114].

```
#define MSIZE    1024
#define ROW         0
#define COL         1

Shape   matrix[MSIZE][MSIZE];

void dataparallel_transpose (matrix:int a_in, matrix:int a_out)
{
   [pcoord(COL)][pcoord(ROW)]a_out = [pcoord(ROW)][pcoord(COL)]a_in;
}
```

Figure 6.4: Data Parallel Code for Matrix Transpose (Using C*)

processors also configured in the shape of a 2D grid of `MSIZE` rows, each containing `MSIZE` elements. In C* the layout of the processors is defined in terms of an intrinsic data structure called a `Shape`. Using this form of descriptor each *virtual* processor is considered to contain a single element of the matrix. To transpose the matrix, elements are indexed by their processor location using a semantic referred to as *left indexing*. This mechanism behaves in much the same way as indexing multidimensional arrays in C, except that the square bracket operator, `[]`, occurs to the left of the array name. For instance, if `foo` is a 2D array stored on a 2D grid of processors of the same dimensions, an expression such as `[1][3]foo` would access the contents of the processor at row 1 and column 3. The C* intrinsic function `pcoord()` is used to calculate the rank of a processor along the axis specified in the argument to the function. For example `pcoord(0)` gives the rank (position) of a processor along the 0th axis of the processor layout defined by its `shape`.

6.3.2 The Message Passing Programming Model

The most general and straightforward programming model for communications on a distributed memory architecture provides a consistent interface for individual processors to *explicitly* transfer data to and from other processors. At the user level, *message passing* defines an interface to transfer data from one processor to another, and how this data is represented. As we have noted, this is an *explicit* communications method where the programmer must specify what data is to be transferred and the destination and source of this data. Message passing programs are said to be either *single program, multiple data* (SPMD) or *multiple program, multiple data* (MPMD). In the SPMD model, an instance of a single program is executed on multiple processors concurrently. Control flow in each program is defined using the value of data it operates on. In other words, the program may branch depending on values derived from data on that processor or data received from other processors executing the same program (but with potentially different data). The main differences with this and the SIMD data parallel model are:

1. The processors do not need to be synchronized at each cycle/instruction of the programs execution.

2. The concept of a virtual processor is not part of the message passing abstraction (although data parallel code is often translated to SPMD code for compilation on machines with small numbers of relatively large processors, e.g. CM-5).

The MPMD model consists of different programs executing on multiple processors. This would arise in the case where an application might want to communicate or delegate work it cannot perform internally. Obviously this is one of the most general models whereby programs on different processors cooperate to perform larger scale tasks.

In both SPMD and MPMD models, processors typically communicate using vendor supplied libraries that make use of the underlying communications network hardware. These libraries provide some or all of the following functionalities (shown graphically in Figure 6.5):

1. Message Creation - Methods to encapsulate data into message packets using existing or derived data types.

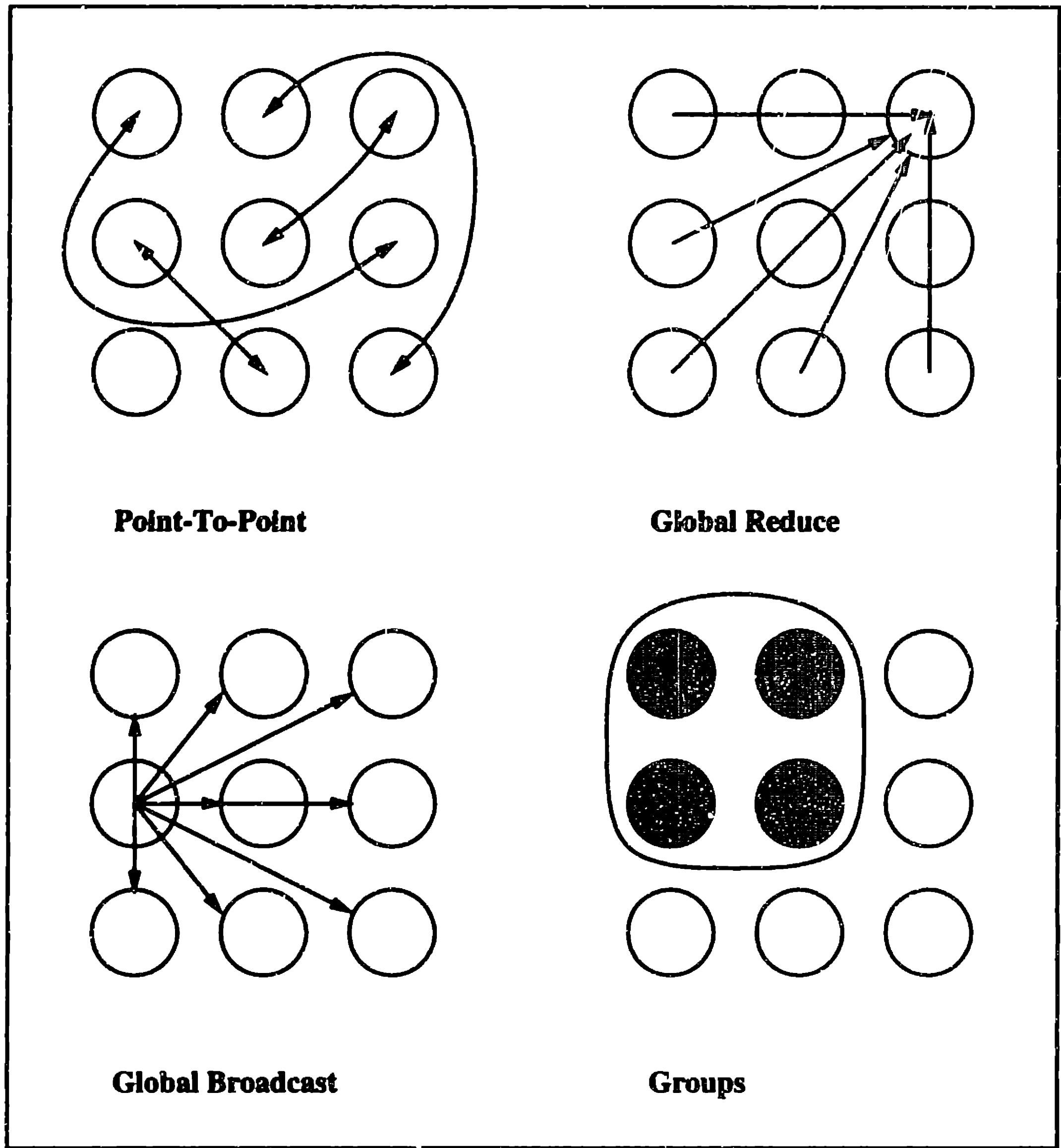

Figure 6.5: Message Passing Communication Operations

2. Point-to-Point communication - Routines that allow a pair of processers to communicate independently.

3. Broadcasting - Routines to allow one process to distribute messages to one or more processers in a group.

4. Set reduction - operations to collect data from all processers in a group and reduce it to a single global description.

5. Synchronization - Routines to synchronize processers at agreed states or locations in a programs execution.

6. Groups - Mechanisms to define subgroups of processers that interact independent of some other group of processers.

7. Topologies - Methods for specifying communication network topologies in software to facilitate problem specific processor addressing schemes and to utilize hardware support for communication operations when available.

8. I/O - Routines to allow the user to interact with the processers and extract results from the computation.

We can see that these general descriptions of basic message passing functionality are open to interpretation, both in the semantics and syntax of their implementation. As a result, there exists a large number of incompatibilities amongst vendor supplied message passing libraries. This makes the development of portable code using any single vendor library effectively impossible. In response to this issue of portability, a group of developers from research institutions, industry and academia has developed a standard specification for this form of interprocessor communication. The standard is known as the *Message Passing Interface* (MPI) [31]. In the next section we will describe some details of the specification and look at several commonly used communication primitives defined therein.

6.4 The Message Passing Interface Standard

As we have seen, the message passing paradigm for programming distributed memory machines is both a flexible and general methodology. Underlying the various message passing libraries that are available, there exists a common basis in the description of the processor communications. MPI is an attempt to standardize an interface to this basis in the form of a platform independent library of message passing routines and utilities. The specification of the standard is the result of several years of research by groups of interested parties from research institutions, academia and industry. The principal influences in the design of the standard arise from previous efforts at writing general purpose interprocessor communications libraries, including p4[7], PVM (parallel virtual machine) [35], Intel's NX/2[8], and work performed at the IBM T.J. Watson Research Center[9].

[7] p4 was developed at the Argonne National Laboratory.

[8] NX is the operating system used on some Intel parallel machines.

[9] IBM has developed a proprietary MPI library for it's SP2 parallel computers, [58]

In the following sections we will examine some of the more important components of the standard. We start with an overview of the basic currency of message passing systems, the messages themselves. We then look at some of the syntax and semantics of *point-to-point* communication. Following this we will examine some of the higher level constructs available in MPI, such as support for *collective communications*, *process groups* and *communication topologies*. The reader is referred to the MPI standard, [31] for the definitive description of these topics, our intention here is only to provide an introduction. We finish this description with an implementation of the *matrix transpose* example, by way of a working example using an MPI library.

6.4.1 Message = Data + Envelope

The MPI standard uses a *letter and envelope* metaphor to define the structure of a message. Each message is said to consist of data (the letter) and instructions for its delivery (the envelope).

Message Data Message data consists of any predefined language datatype such as characters, integers, floating point representations, or formed out of derived datatypes such as *structures* in C. All predefined language datatypes have an associated integer identifier in MPI. For example, the C language datatype **int** has a corresponding MPI datatype descriptor, **MPI_INT**. These identifiers are used to ensure data integrity during message transmission. They indicate to the message recipient the datatype being received and the storage size required for it. Messages can be of arbitrary length (within system limits) but must exist as a block of contiguous memory when transmitted.

Message Envelope The transmission instructions associated with the message *envelope* specify the following items:

1. *Source* - The address[10] of the message sender.
2. *Destination* - The address of the message recipient.
3. *Tag* - An integer descriptor to distinguish a message from others sent by the same *source* to the same *destination*.

[10] The logical address of a processor is a system designated integer identifier or *rank*, starting at 0 and numbered contiguously.

4. *Communicator* - A descriptor that indicates the currently active communications protocol and its scope. At a minimum, a *communicator* consists of the following information:
   (a) *group* - A description of the processers that can participate in the current communication activity. This is stored in the form of an ordered set of process identifiers. Each process in a group is associated with an unique integer, the *rank* of the process, which start at 0 and numbered contiguously.
   (b) *context* - A tag associated with each process in a communicator group that defines whether or not the process is active in the current communications activity.

6.4.2 Point To Point Communication

The most basic communication activity that occurs between two processers is referred to as *point-to-point* message passing. The activity is defined in terms of a *source* process, the message sender, and a *destination* process, the message recipient. MPI defines several methods to achieve this form of communication. We will look at the most common form, **send** and **receive**. The syntax for each operation is shown below.

```
int MPI_Send (void * buff, int count, MPI_Datatype dtype,
              int dst, int tag, MPI_Comm com);

int MPI_Recv (void * buff, int count, MPI_Datatype dtype,
              int src, int tag, MPI_Comm com, MPI_Status * stat);
```

The parameters common to both functions are:

- `buff` - pointer to (address of) start of message buffer.
- `count` - number of elements of type MPI_Datatype in message buffer.
- `dtype` - MPI handle denoting type of data in message buffer.
- `tag` - integer descriptor used to differentiate messages.
- `com` - handle to MPI object describing communications protocol in effect.

and the operation specific parameters are:

- `src` - integer descriptor denoting **rank** of process sending message.
- `dst` - integer descriptor denoting **rank** of process receiving message.
- `stat` - pointer to MPI object containing message transmission information.

The behavior of `MPI_Send` and `MPI_Recv` is said to be **blocking** in nature. The semantics of these operations dictates that they must wait until either a matching operation is posted or intermediate buffering for the message is provided by the system. For example, if process **A** sends a message to process **B** using a *blocking send*, then process **A** will not return (complete) until either process **A** copies the message to an intermediate system buffer or a matching *receive* is posted by process **B** and the message is successfully transferred. This behavior is necessary to guarantee that attempting to overwrite the message buffer following the *send* operation does not corrupt the contents of the message.

The *send-receive* operations available in MPI can be performed in the following communication modes:

1. **non-blocking** mode - where intermediate system buffering is guaranteed;
2. **synchronous** mode - where the message recipient is guaranteed to have posted a matching *receive* and has accepted the incoming message.
3. **ready** mode - where a *send* operation can only start if a matching *receive* has already been posted.

The reader is referred to Chapter 3 of the MPI standard [31] for further details relating to point-to-point communication and the treatment of messages containing *derived* datatypes.

6.4.3 Collective Communications

Broadcast, *reduction* and *synchronization* all fall under the general description of *collective* communications. In these operations some or all of the processors communicate with each other to achieve some form of global status. Examples of collective communications are synchronizing the execution of specific tasks (global *all-to-all* operation) or distributing data residing in one process to all other processers (a *scatter* operation). The principal collective communication operations defined in the MPI standard are:

1. Barrier Synchronization
2. Broadcast

3. Gather
4. Scatter
5. All-to-All Scatter/Gather
6. Global Reduction
7. Scan

The reader is referred to Chapter 4 of the MPI standard [31] for further details on collective communication.

6.4.4 Matrix Transpose Example Using MPI

We return to the matrix transpose example to highlight some of the differences between the message passing model and the data parallel model considered earlier. Two algorithms of increasing sophistication are described, along with implementations using routines defined in the MPI standard. These algorithms also serve to highlight the scheduling and synchronization issues that arise, in all parallel programming models. Ignoring these issues can lead to deadlock (where the program will simply stop) or processor contention which can result in serialization (essentially causing the program to execute sequentially and *very* inefficiently). As these examples will show, issues such as these account for a large proportion of the work involved designing parallel algorithms.

All-To-All Transpose

The first implementation uses a so called *all-to-all* algorithm. We consider a square matrix $\mathcal{M}$ of dimension N, stored as one row per processor. Thus for N processors, each row consists of N elements. Each element of the matrix can be indexed using the relationship $\mathcal{M}[i,j] : 0 \leq i,j \leq N$, where i corresponds to a row of the matrix (and consequently denotes a processor) and j denotes any element of a row. Each processor then exchanges N-1 elements with the processors holding the appropriate row of $\mathcal{M}^T$ under the mapping $\mathcal{M}^T(p,q) = \mathcal{M}(q,p)$ where $0 \leq p,q \leq N$. The set of exchanges are take place over N communication cycles or phases. To avoid *deadlock* we must sequence and synchronize the sending and receiving of the messages.

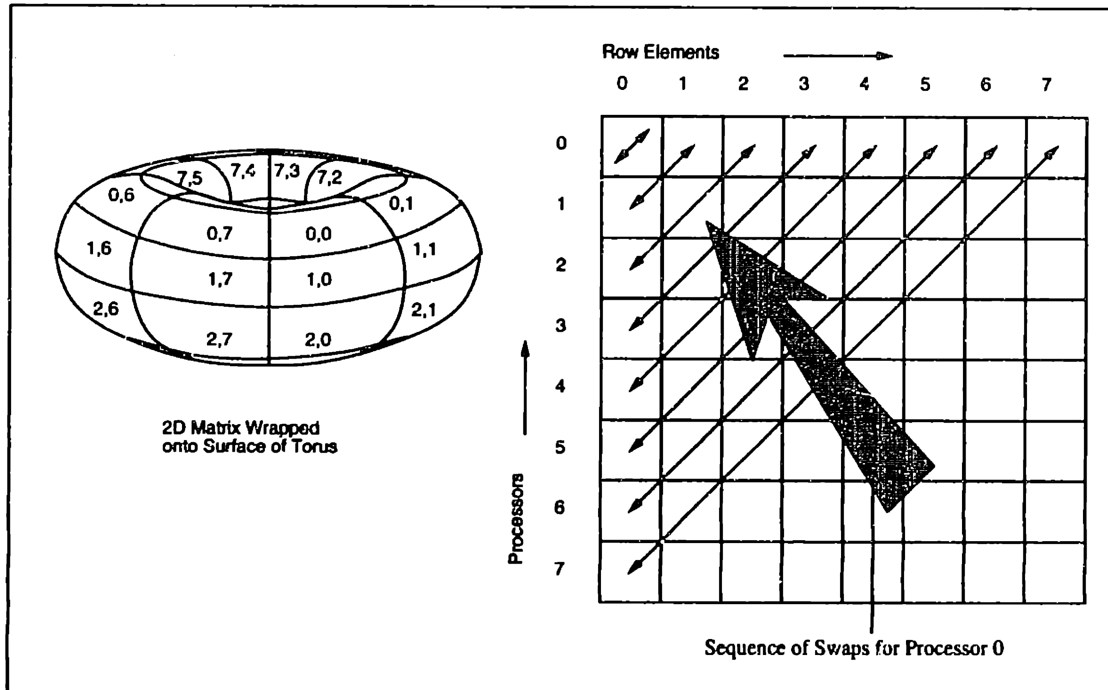

Figure 6.6: All-To-All Transpose Pairings For Row 0

For example, deadlock would occur if we were to send a message to a processor that was also trying to send a message back at the same time. This can be handled in several ways. We describe just one possibility where processors are uniquely paired off to *exchange* row elements. The uniqueness of the pairing also guarantees that no two processors will attempt to send a message to a third processor at the same time avoiding processor contention.

The address or rank of the processor with which to exchange elements is calculated using the following equation:

$$far_addr(N, P, C) = (N - 1 - (P + C)\%N)\%N; \tag{6.1}$$

where N is the number of processors, P is the rank of a given processor and C corresponds to the current cycle (pass) of the algorithm. The % symbol denotes the modulus operator. This is the synchronization step of the algorithm. The address pairings can be thought of as connecting processors that are diagonally separated by N processors on the 2D matrix if it were wrapped onto a torus, then cyclically shifted at each loop. Processor pairings for row 0 of an 8×8 matrix using this algorithm are shown in Figure 6.6. The pairings for other rows are listed in Table 6.1.

proc \ loop	0	1	2	3	4	5	6	7
0	7	6	5	4	3	2	1	0
1	6	5	4	3	2	1	0	7
2	5	4	3	2	1	0	7	6
3	4	3	2	1	0	7	6	5
4	3	2	1	0	7	6	5	4
5	2	1	0	7	6	5	4	3
6	1	0	7	6	5	4	3	2
7	0	7	6	5	4	3	2	1

Table 6.1: Processor Pairings for 8×8 Matrix Transpose

We still need to sequence each exchange to avoid deadlock. This is done by adopting the commonly used protocol of *even processors send then receive, odd processors first receive then send.* Finally we also need to buffer elements being sent after new elements are received to avoid overwriting. The algorithm progresses in N cycles corresponding to the N elements that need to be transferred. Pseudo-code for the algorithm is shown in Figure 6.7.

An implementation of this algorithm using MPI routines is shown in Figure 6.8. The parameters to the function `mpi_all2all_transpose` have the following values/meanings. `row` is a pointer to a vector of `n` integers corresponding to the row of the matrix being transposed. `p` denotes the rank of the processor that the function is called on. We assume that this is determined elsewhere in the program, rather than being repeatedly calculated at each invocation of the transpose function. The local variables are used as follows: `even` is a flag to indicate if the processor rank is odd or even; variable `c` is used to iterate through the `n` element exchanges each row will participate in; `far` stores the processor address (rank) that processor `p` is paired with in iteration `c`, and `tmp` is used to store the matrix element to be exchanged when the processor must receive a message before sending. Local variable `status` is an MPI object which is used in `MPI_Receive` as a container for information relating to the incoming message (e.g. the senders address, the length of the message etc). The sequence of communications follow those specified in the algorithm pseudo code discussed earlier. The row vector is used directly as the message buffer for both the incoming and outgoing message on processors with even addresses.

```
MESSAGE_PASSING_ALL2ALL_TRANSPOSE
{
   n <- number of processors;
   p <- processor rank;

   FOR c = 0 -> n cycles
    {
      far <- FAR_ADDR(n, p, c);

      IF p is even
       {
         SEND    M(p, far) TO   far;
         RECEIVE M(dst, p) FROM far;
        }
        ELSE
        {
         tmp <- M(far, p);

         RECEIVE  M(far, p) FROM far;
         SEND     tmp       TO   far;
        } FI

    } END FOR
}
```

Figure 6.7: An All-To-All Transpose Algorithm

At the end of each iteration or communication phase the MPI routine `MPI_Barrier` is called to synchronize the processers explicitly. This guarantees that each processor is aligned at the same point in the codes execution and that all underlying system calls involved in the communication and themessage buffers have been processed.

Scatter-Gather Transpose

To reduce the $O(N)$ number of communication operations necessary in the *all-to-all* transpose above, an algorithm was derived to perform the same task in $O(log_2(N))$ communications cycles. The algorithm achieves a speed up by lowering the number of messages each processor must send/receive. This depends on the hierarchical symmetry of the transpose operation for square matrices. This property is shown in

```
void mpi_all2all_transpose (int * row, int n, int p)
{
  int        even, c, far, tmp;
  MPI_Status status;

  even = (p%2 == 0);

  for (c = 0; c < n; c++)
   {
     far = (n - 1 - (p + c)%n)%n;

     if (even)
       {
        MPI_Send (row+far, 1, MPI_INT, far, p, MPI_COMM_WORLD);
        MPI_Recv (row+far, 1, MPI_INT, far, far, MPI_COMM_WORLD, &status);
       }
     else
       {
        tmp = *(row+far);
        MPI_Recv (row+far, 1, MPI_INT, far, far, MPI_COMM_WORLD, &status);
        MPI_Send (&tmp,    1, MPI_INT, far, p, MPI_COMM_WORLD);
       }
   }
}
```

Figure 6.8: MPI Code for All-To-All Transpose Operation

Figure 6.9, where a matrix of size $N \times N$ (N = number of processors), is transposed in $log_2(N)$ element exchange steps. The process can be thought of as a recursive operation. At the top level the matrix is bronken into 4 macro elements that are transposed as in Figure 6.9 for $L = 0$. At the next level of the recursion each of the macro cells are each divided into 4, and each of these sets transposed. This is shown in Figure 6.9 for $L = 1$. The transpose continues in this manner until the macro cells equal the size of a single matrix element as shown in Figure 6.9 for $L = 3$.

The algorithm works by incrementally moving elements towards their transposed positions, storing them on intermediate processors en route. At the end of each cycle some of the elements will have been moved to their correctly transposed position while the remainder can be thought of as lying *closer* (using some distance metric) to their final positions.

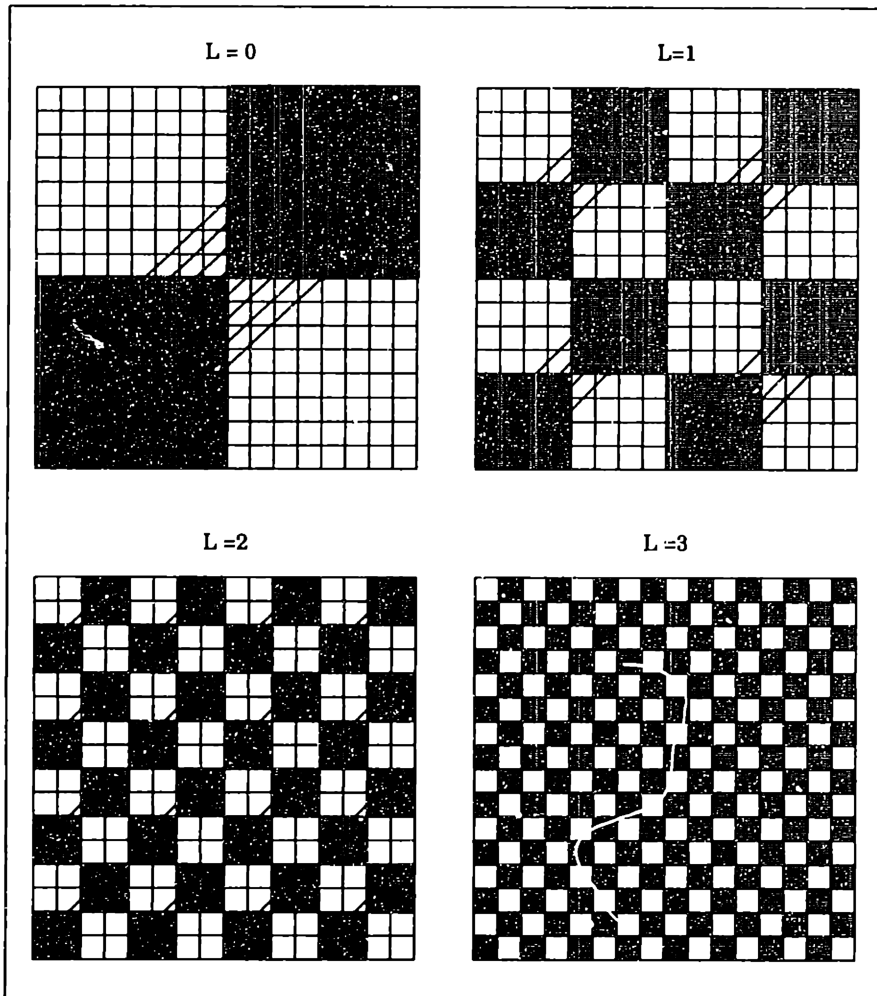

Figure 6.9: Scatter Gather Swap Pattern, L = pass

For example, in the first communication cycle of the algorithm, the elements in the second half of row 0 is exchanged with the first half of row $N/2$. Similarly row 1 exchanges elements with row $N/1 + 1$ and so on for all other rows in parallel. This is shown in Figure 6.9 for $L = 0$.

In the second communication cycle the second and forth quarters of row 0 are exchanged with the first and third quarters of row $N/4$. To maintain symmetry, the second and forth quarters of row $N/2$ are exchanged with the first and third quarters of row $3N/4$. Once again this operation occurs on all other processors in parallel. This is shown in Figure 6.9 for $L = 1$.

The exchanges are based on consolidating, or *gathering*, segments of the matrix on each row into bundles that have a common destination processor for that communication cycle. By permitting elements to reside on intermediate rows (processors)

en route, more work on forming the transpose can be performed locally rather than explicit exchanges involving communication (as was the case with the *all-to-all* algorithm considered previously). At each intermediate processor, the contents of the message is unpacked and *scattered* to their corresponding positions in the matrix.

This process is generalized as follows. Each processor stores one row of the matrix (of dimension $N \times N$, where N is the number of processors). The initial segment size is $N/2$ elements.

For each communication cycle, processors exchange messages with those at an address offset equal to the segment size for that cycle (this is a consequence of the symmetry). The size of the segment is subdivided at each pass of the algorithm until the segment size becomes 0. This will take $log_2(N)$ passes. The pattern describing the swaps is then, for $L = \{0, .., log_2(N)\}$ communication cycles, processors $\{0, .., N/(2^{L+1})\}$ swap $S = 2^L$ segments of size $N/(2^{L+1})$ with processors $N/(2^{L+1}) + 1, .., N - 1$.

Processor contention is again suppressed by applying an odd-even swap convention that guarantees all swaps are between independent processor pairs. Examining the algorithm, $O(log_2 N)$ communication cycles are required to transpose the matrix.

The algorithm for matrix transpose implemented using MPI routines is shown below. The structure of the code follows that of the algorithm just described. The most important points to note in the example code are the sequence of the *send-receive* calls and the *barrier synchronization* at the end of each communication cycle. A set of results comparing the *all-to-all* algorithm versus the *scatter-gather* algorithm is shown in Figure 6.10. These times were obtained on a Connection Machine CM-5 using 128 nodes without vector units (VU).

```
void mpi_sg_transpose (void * v, void * sbuff, void * rbuff,
                       int np, size_t bsz, int    this)
{
  int        nseg = 1, segsz = np >> 1, hi_addr = np*bsz - 1;
  int        i, stride, d_addr, bw, flag, b_addr, nb;
  MPI_Status status;

  while (segsz > 0)
   {
     nb     = bsz*segsz;       /* number of bytes per segment              */
     stride = nb << 1;         /* skip even segments (assumes mem aligned) */
     bw     = (this/segsz)%2;  /* black = 1, white = 0                     */

     if (bw)
      d_addr = this - segsz;
     else
      d_addr = this + segsz;

     for (i = 0, b_addr = !(bw)*nb; i < nseg; b_addr += stride, i++)
      memcpy (sbuff + i*nb, v + b_addr, nb);

     if (bw)
       {
        MPI_Send (sbuff, nb*nseg, MPI_BYTE, d_addr, this, MPI_COMM_WORLD);
        MPI_Recv (rbuff, nb*nseg, MPI_BYTE, d_addr,
                  d_addr, MPI_COMM_WORLD, &status);
       }
     else
       {
        MPI_Iprobe (d_addr, MPI_ANY_TAG, MPI_COMM_WORLD, &flag, &status);

        while(!flag)  /* idle wildly */
          MPI_Iprobe (d_addr, MPI_ANY_TAG, MPI_COMM_WORLD, &flag, &status);

        MPI_Recv (rbuff, nb*nseg, MPI_BYTE, d_addr,
                     d_addr, MPI_COMM_WORLD, &status);
        MPI_Send (sbuff, nb*nseg, MPI_BYTE, d_addr, this, MPI_COMM_WORLD);
       }

     for (i = 0, b_addr = !(bw)*nb; i < nseg; b_addr += stride, i++)
      memcpy (v + b_addr, rbuff + i*nb, nb);

     d_addr   = this + (d_addr >> 1);
     nseg   <<= 1;     /* segment                          */
     segsz  >>= 1;     /* decrease segment size accordingly */
     MPI_Barrier (MPI_COMM_WORLD); /* all procs must flush to update state */
   }
}
```

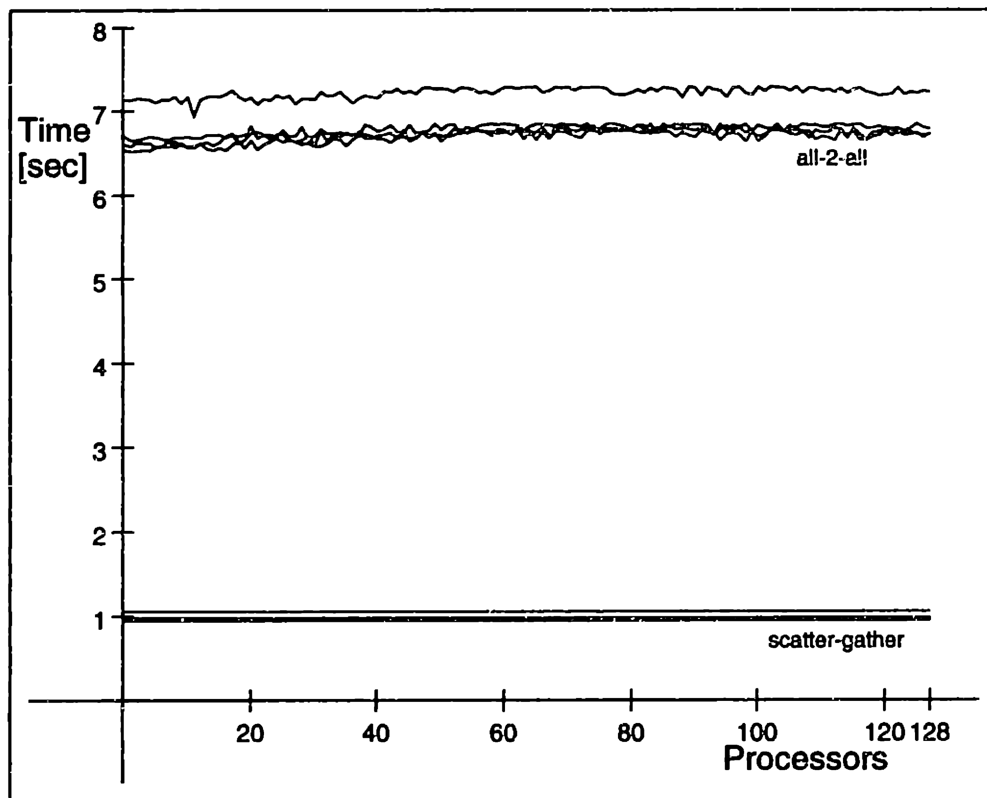

Figure 6.10: All-to-All Vs. Scatter Gather Transpose Timings

6.4.5 Summary

In this section we have introduced the basic functionality defined by the MPI standard. Even with the small number of routines described here, powerful, and portable, message passing applications can be developed. MPI actually contains over 100 library routines in its current form. Further extensions are anticipated to provide formal support of dynamic process creation and standardized parallel I/O.

The final example of the matrix transpose problem also introduced a powerful algorithm for reducing the number of communications operations in message passing programs while simultaneously ensuring that no processor contention occurs. Using this technique can be extremely advantageous on machines with very high communications overhead, particularly networks of workstations. We will see this algorithm in another form when we implement the message passing component of the distributed simulation system.

Chapter 7

Distributed Discrete Element Simulation

This chapter extends the χ*mal* system by incorporating parallelism into the simulation environment. All of the software extensions to the system are developed using the MPI message passing library describedin the previous chapter. The discussion starts with an examination of a general distributed (parallel) sorting algorithm. The algorithm is implemented and benchmarked on several distributed platforms. The performance of the algorithm is then compared against a sequential algorithm applied to problems of the same size. We will see that parallel sorting does not necessarily provide a great advantage in terms of the computation time over the range of problem sizes that can be feasibly approached when using a relatively small numbers of processors. This has practical implications in terms of the availabilty of large scale parallel computer resources at the present time.

The second half of the chapter describes the distributed implementation of the χ*mal* system. Parallelism is first incorporated into a 2D application which is once again timed on a several distributed architectures. A series of 3D simulations containing spheres are then timed to provide a reference benchmark. Since spheres are perhaps the simplest primitives that can be used in DEM analysis, all other schemes are expected to require larger computation times. The effectiveness of more complex object representations can then be compared against an insensitive base case.

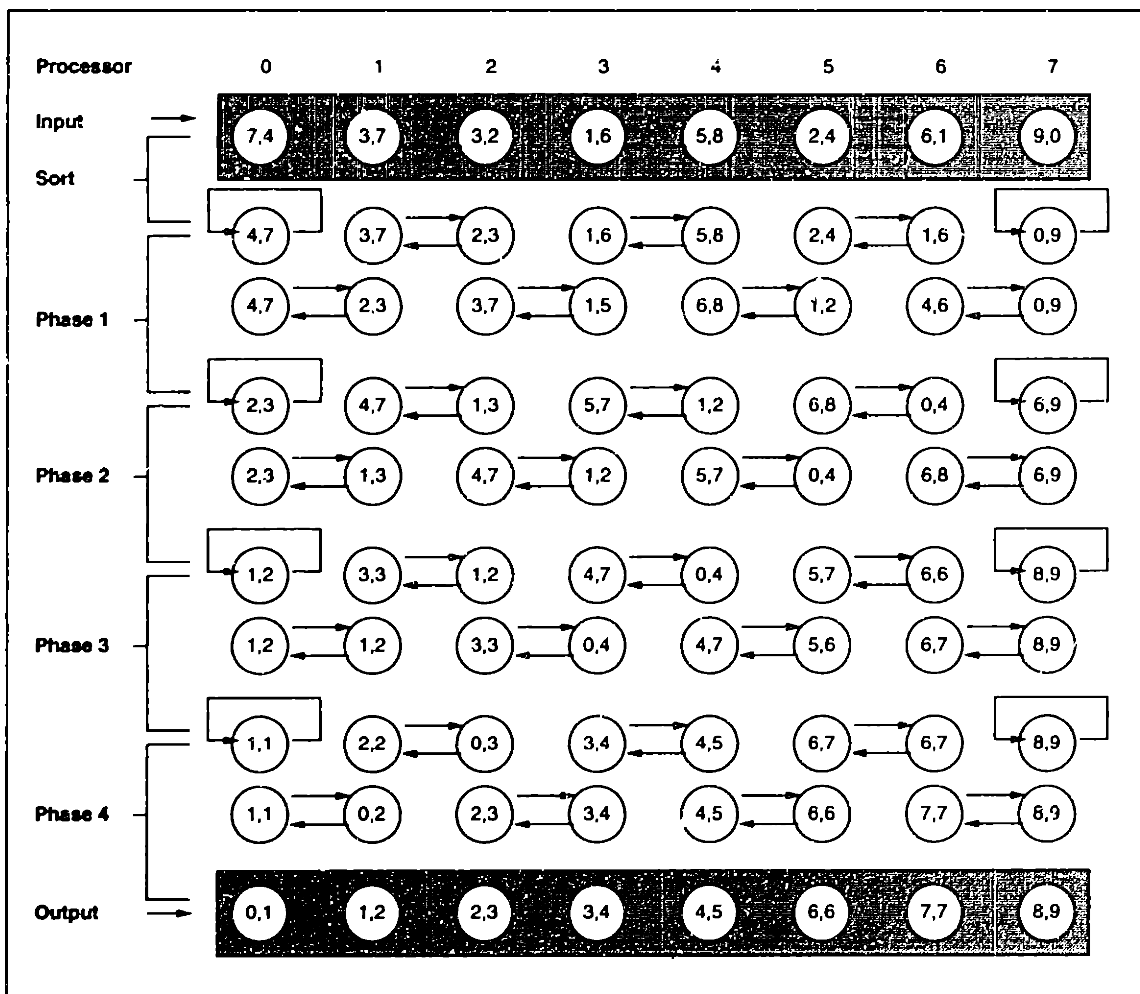

Figure 7.1: Simple Parallel Mergesort on 8 Processors

7.1 Parallel Sorting

Up to this point we have discussed issues pertaining to spatial sorting on a single processor machine, using sequential algorithms. Common to all of these methods was some form of sorting algorithm used to order the data set (the objects). In this section the examination of spatial sorting is extended to include algorithms suitable for distributed memory machines.

7.1.1 A Simple Parallel Sorting Algorithm

A parallel sorting algorithm known as *merge-splitting* is first examined. The algorithm is based on that given in Akl, [1]. The method described in the text was designed to perform cost-optimal sorting on machines using a perfect shuffle interconnection. Some modification was made to run it on general set of processors (i.e. a MIMD machine) but the underlying SIMD structure of the algorithm remains.

The algorithm assumes that N keys are to be sorted using P processors, where both N and P are powers of 2 and $P < N$. Each processor is allocated N/P keys to start. Each processor must also be large enough to perform the following operations:

1. Sort a sequence of keys of length $2N/P$ using a sequential sorting algorithm.
2. Merge two sorted lists, each of length N/P, into a single list of length $2N/P$ using a sequential merging algorithm.

The steps of the modified algorithm are as follows:

1. Sort the sequence of N/P keys on each processor using the sequential sorting algorithm.
2. Repeat for $P/2$ parallel phases:
   (a) Odd processors, P_{2i+1}, send sorted sequence to even processors, P_{2i+2}.
   (b) Even processors, P_{2i}, send sorted sequence to odd processors, P_{2i-1}.
   (c) Odd processors, P_{2i+1}, merge received sequence with local and extract lower N/P keys. Even processors, P_{2i}, merge received sequence with local and save upper N/P keys.
   (d) Odd processors, P_{2i-1}, send sorted sequence to processors, P_{2i}.
   (e) Even processors, P_{2i}, send sorted sequence to odd processors, P_{2i+1}.
   (f) Odd processors, P_{2i-1}, merge received sequence with local and save upper N/P keys. Even processors, P_{2i}, merge received sequence with local and save lower N/P keys.

A small example using these steps on 8 processors is shown in Figure 7.1. Each processor is first assigned an array containing an arbitrary sequence of 2 numbers, the **input**. Each processor then sorts the contents of the array using the prescribed sequential algorithm. The sorted arrays then become the input for the 4 parallel merge phases.

It is evident from the example that the final sorted sequence remains distributed over the processors. Depending on the requirements of the task for which sorting is performed, it may be necessary to gather the sorted components so that a copy of the collective sequence resides on all processors. There will be an additional communications overhead associated with the extra step. For *small* input sets this will have an impact on the overall performance. This point is considered further in later sections when the performance of the parallel sorting scheme is evaulated.

7.1.2 Parallel Sorting Tests

The *merge-splitting* sorting algorithm was implemented on two parallel platforms and timed for a range of data sets of various sizes. The parallel sorting times are benchmarked against the sequential sorting routine, *heapsort*, developed previously in this chapter. The main purpose of these tests is to examine the effectiveness of distributing the spatial sorting component of contact detection over multiple processors.

Resource Considerations

Two parallel and two serial platforms are considered in the tests. They are:

1. A single Sun SPARCstation 4.
2. An IBM RS/6000-590 (a single SP2 node).
3. A network of workstations (NOW), 4 Sun4s.
4. An IBM SP2 using 4 nodes.

These environments were chosen because of their relative accessibility as opposed to their *apparent peak performance.* The results reported should not be used to benchmark other machines, rather they are an attempt to provide a control measurement for applications developed specifically on these machines at a later stage. With this in mind we wished to evaluate the average performance of these machines when run for moderately long periods (hours-days) without having to be specially configured or dedicated to particular tasks.

The parallel sorting algorithm was implemented using a portable implementation of the emerging MPI standard, called MPICH [12], which was developed at Argonne National Laboratory and Mississippi State University.

Machine	Sun4	Sun4	RS/6000	RS/6000	Network	Network	SP-2	SP-2
Size (N)	wall	cpu	wall	cpu	wall	cpu	wall	cpu
8	0.0010	0.0000	0.0000	0.0000	0.1237	0.1333	0.0394	0.0100
16	0.0010	0.0000	0.0000	0.0000	0.1025	0.0999	0.0238	0.0200
32	0.0010	0.0000	0.0000	0.0000	0.1083	0.0999	0.0251	0.0200
64	0.0010	0.0000	0.0001	0.0000	0.1136	0.1166	0.0239	0.0100
128	0.0020	0.0000	0.0002	0.0000	0.1072	0.0999	0.0237	0.0200
256	0.0050	0.0000	0.0006	0.0000	0.1901	0.1166	0.0231	0.0200
512	0.0100	0.0000	0.0013	0.0000	0.1168	0.1166	0.0273	0.0200
1024	0.0210	0.0000	0.0031	0.0000	0.1273	0.1166	0.0263	0.0200
2048	0.0480	0.0333	0.0069	0.0000	0.1591	0.1499	0.0323	0.0200
4096	0.1050	0.0999	0.0156	0.0200	0.2422	0.2333	0.0382	0.0300
8192	0.2300	0.2333	0.0336	0.0400	0.4478	0.4333	0.0488	0.0500
16384	0.5010	0.4999	0.0730	0.0700	0.7515	0.7333	0.0774	0.0800
32768	1.0830	1.0832	0.1583	0.1500	1.4469	1.3999	0.1478	0.1400
65536	2.3510	2.3499	0.3399	0.3400	3.4505	3.0332	0.2753	0.2600
131072	5.1290	5.0997	0.7328	0.7300	5.8118	5.6497	0.5393	0.5200
262144	10.9640	10.9495	1.5887	1.5700	11.7247	11.5162	1.0965	1.0600
524288	23.6159	23.4990	3.4191	3.3900	25.1065	24.9490	2.2851	2.1200
1048576	50.2389	50.0979	7.3410	7.2500	53.7314	52.2479	4.7583	4.5600

Table 7.1: Sorting Time for N Keys

Timing Results

The results from sorting tests for various numbers of keys are shown in Table 7.1 and graphed in Figures 7.2, 7.3, 7.4. The times recorded for small sets of keys (less than 256) are very noisy because they execute in times close to the clock resolution. They should not be considered accurate. Of more interest are the times at which the various platforms either overlap or diverge. From these points we can identify the size of data set for which it is more effective to apply parallel sorting.

Observations

We see that the 4 node SP2 outperforms the single SUN-4 workstation for relatively small numbers of keys. The crossover occurs in the region of 1,000 to 2,000 keys. This is considered a surprisingly small number of keys for the overhead of communications to become negligible even though each SP2 processor is significantly faster than that

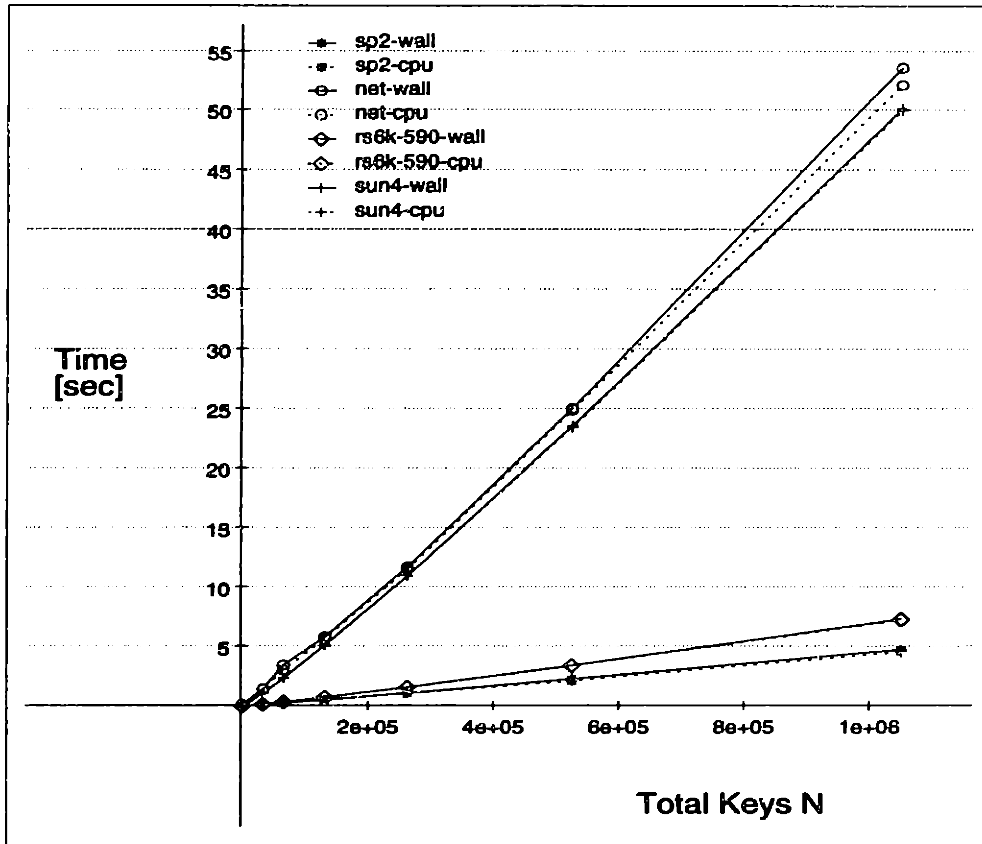

Figure 7.2: Graph of Sorting Time for N = 8.. 1,000,000 Keys

of the SUN-4. This reflects positively on the SP2 communications network[1].

Sorting on the network of workstations (NOW) performed worse than a single workstation from the same NOW configuration. This is shown over the entire range of input data sets where the trend for larger sets was diverging. This is due in part to the structure of the sorting algorithm but is primarily due to the serial communications on a NOW connected via Ethernet.

From the results we see that data sets containing $\approx 10^6$ keys are sorted sequentially on a single SP2 node in a time slightly greater than 4 nodes in parallel. The parallel efficiency is only 38% for 10^6 keys. This rating does include the fact that the sorted data set remains distributed over the processors. To obtain a complete set of sorted data on each node further communication steps are necessary. This reduces the time difference between parallel and sequential sorting for the size of data set considered.

[1]These tests were performed using the default IP communications switch, as opposed to the high speed switch which is said to perform communications operations 2-3 times faster.

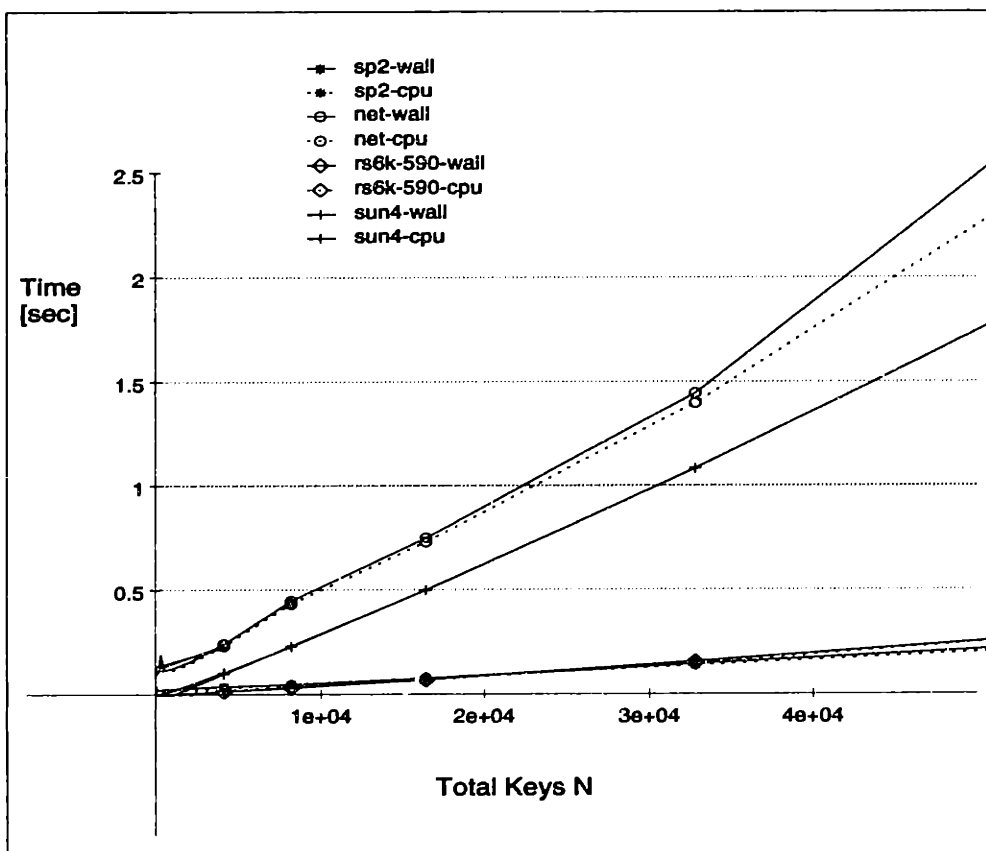

Figure 7.3: Closeup of Sorting Times for N = 10,000.. 40,000 Keys

When 8 nodes of the SP2 are utilized the parallel efficiency for 8 processors drops to 31%. However, the times do scale well for increasingly larger numbers of keys as shown in Table 7.2.

The size (numbers of processors) of the parallel platforms are small by most standards. As such, the scaling behavior of the parallel algorithm was not observed. As already stated, these results were obtained to evaluate the platforms locally available. With this mind, it was concluded that no benefit is obtained by distributing the spatial sorting on either the NOW or the small SP2 configuration described. If larger parallel machines become available *and* larger problems are considered, then the performance measurements should be re-evaluated appropriately.

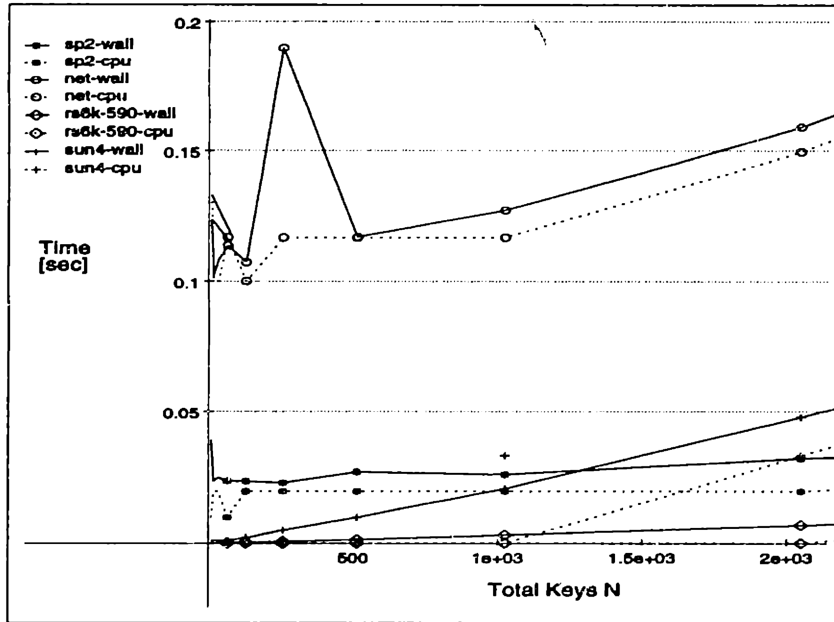

Figure 7.4: Closeup of Sorting Times for N = 100.. 2000 Keys

Number of Keys	Wall [sec]	CPU [sec]
1048576	2.956998	2.960000
2097152	6.061453	6.060000
4194304	12.543865	12.510000
8388608	26.202503	26.120000

Table 7.2: Sorting Time for N Keys on 8 SP2 nodes

7.1.3 Evaluation of Parallel Sorting

With a view to distributing the spatial sorting task in the χ*mal* system a parallel sorting algorithm (*merge-splitting*) was examined. The algorithm was implemented using the MPICH communications library [12] and timed for a range of data sets with up to one million keys. The timing tests were performed on two platforms containing a small number of processors. The parallel running times were compared with the performance of the sequential *heapsort* algorithm for the same range of problem sizes. The test results indictate that parallel sorting is *not* sigificantly faster when applied to relatively small numbers of keys ($O(10^6)$). The parallel efficiency in this range is considered poor (30%-40%). For the size of problem and processing platforms used in the tests, parallel spatial sorting is not considered useful[2].

[2]For larger datasets or numbers of processors these results should be re-evaulated.

7.2 Distributed Discrete Element Simulation

In a first attempt at parallelizing the χ*mal* DEM environment we utilise the message passing style of implementation and make the following assumptions:

1. We have shown that sorting entire data set on a single node is almost as fast as doing the sorting in parallel (albeit without redistributing the sorting tables to all other processors).
2. The space requirements is sufficiently small that each node can quite feasibly hold a copy of the entire data set, giving a space redundancy while dramatically improving the computational speed up for this type of system.

We will see later that the consideration of maintaining a copy of the entire data set on each processor is valid insofar as the current CPU requirements for relatively small numbers of objects far exceeds the available processing power. This implies that if we had a data set that could not fit onto a single processors memory, it would most likely take too long to simulate anyway. Very large datasets are considered by other researchers but the often report times for one simulation cycle, rather than results from large production runs. Presumably this is a reflection of the still infeasible runtimes associated with large datasets.

7.2.1 Implementation Details

The implementation entailed the following constructs;

1. Each node maintained a copy of the entire simulation data set.
2. The complete data set was sorted on each node concurrently.
3. Each node processes N/P objects for contact searching and force resolution components of contact detection.
4. The resulting contact forces are applied by accumulating the contact force components from the contact table over all nodes using a *folding* reduction algorithm, in $log(P)$ phases.

Timing results are presented for sample runs of the system containing up to 10,000 2D bodies on an IBM SP2, a Thinking Machines CM-5 and a small network of Sun

workstations. We also timed the application on serial machines to maintain a reference point.

Four processing environments were considered for these tests:

1. Relatively fast single processor workstations with sufficient local memory.
2. A network of 6 single processor workstations (Sun4s) configured as a virtual multiprocessor.
3. 32 nodes of a Connection Machine, CM-5.
4. 8 IBM RS/6000-590 processor nodes of an IBM POWERparallel, SP2.

The tests were evaluated by running a DEM application with various numbers of 2D objects over reasonably small simulation times. The averaged times are shown in Tables 7.3, 7.4, and graphed in Figures 7.5, 7.6 and 7.7.

Number of 2D Objects	Sun Sparc4 CPU [sec]	IBM RS6K 320 CPU [sec]	IBM RS6K 590 CPU [sec]
100	0.38	0.18	0.05
400	1.88	0.80	0.30
900	4.23	2.35	0.95
1600	9.99	5.76	2.05
2500	20.40	12.77	4.42
3600	38.71	25.23	8.31
4900	69.98	45.82	14.46
6400	115.31	87.25	23.87
8100	180.79	129.95	38.28
10000	270.83	186.05	70.02

Table 7.3: CPU Times on IBM RS/6000 320H and 590, and Sun4 Workstations

Number of 2D Objects	Sun4 Network 6 machines	CM-5 32 nodes (w/out VUs)	SP-2, 8 nodes (IP mode)
100	0.28	0.28	0.03
400	0.98	0.43	0.19
900	1.83	0.82	0.37
1600	3.44	1.43	0.65
2500	7.11	2.24	0.87
3600	11.88	3.45	1.57
4900	-	5.17	2.58
6400	-	7.82	4.01
8100	-	11.64	6.17
9025	-	12.98	7.40
10000	-	-	9.19

Table 7.4: CPU Times for SP2 and CM-5 and network of Sun Workstations

7.2.2 Parallel DEM - Scaling and Limit Considerations

The objective of this section is to identify some working limits on the scale of problem that can be tackled using DEM simulation techniques. The limits take the form of computational resources available in conjunction with numerical and empirical bounds on the size and duration of problems that can be feasibly considered. They are intended to demarcate the spatio-temporal scales necessary to capture phenomenological characteristics of a physical system (the lower bounds), and the limits defined by the complexity of the computation (the upper bounds). We will define these working limits in terms of a reference system comprised of spheres.

Lower Bounds

The lower bounds on computation arise in the form of the spatial and temporal discretization applied to the problem. They can be summarized by the following parameters:

1. Courant condition and numerical stability.

2. Discretization and scaling effects.

The temporal discretization will dictate the accuracy of the system's time evolution (both numerically and more importantly, *the causal sequence*). Its effect as a con-

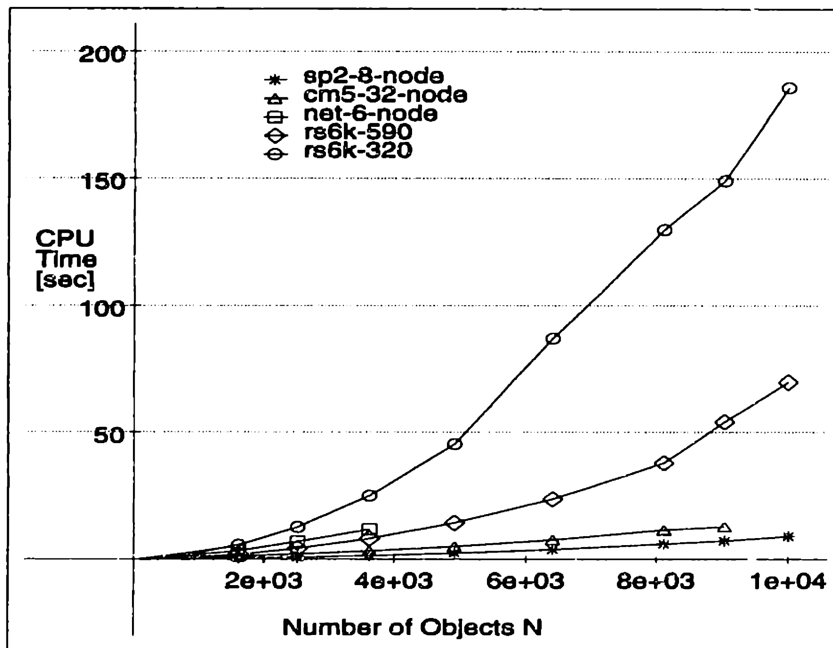

Figure 7.5: CPU Time [sec] Vs. Number of 2D Objects, N

straint is to define the number of iterations necessary to simulate a nominal amount of time. The number of objects used in the model must be able to capture the geometric characteristics of the system to some order of accuracy. Similarly, the number of objects used in the model must be able to capture the geometric characteristics of the system to some order of accuracy. For efficiency we wish to minimise both the number of iterations and the number of objects used in the model.

Upper Bounds

The upper bounds of the computation are defined in terms of the following parameters:

1. Space - memory.
2. Time - processing rate.

The upper bounds occur as practical constraints, which is to say, they are artificial but binding. The first bound arises in the limit of space (memory) available to repre-

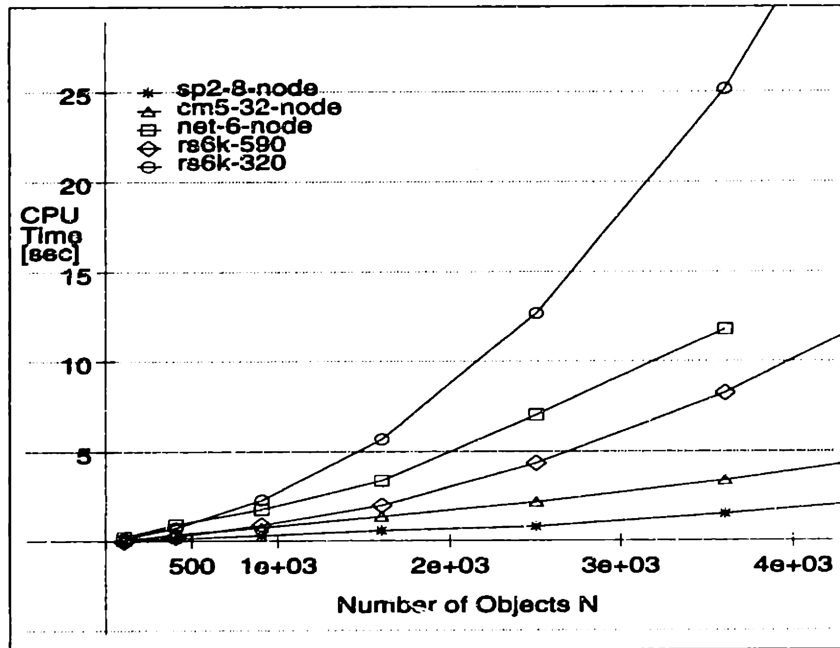

Figure 7.6: Detail (for network) of CPU Time [sec] Vs. Number of 2D Objects, N

sent the problem. The second bound is determined by the speed at which operations can be performed and the longest feasible run time over which these operations can be expected to be performed. The upper bounds are first defined in terms of single processor machines and then, by extension, using parallel computer architectures.

These factors are of great practical importance since the nature of this form of analysis requires that many experiments need to be performed so that the observed results are meaningful in a *statistical* sense.

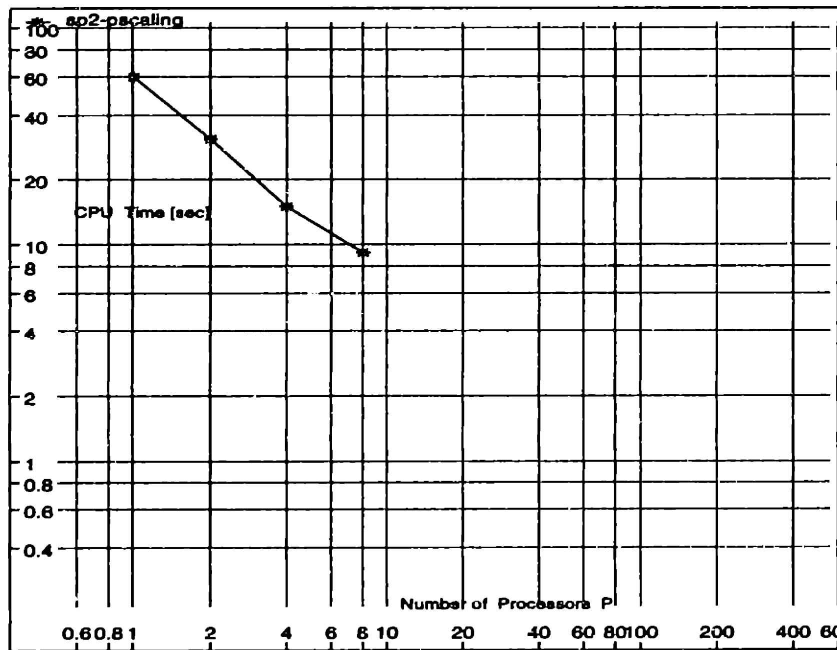

Figure 7.7: Scaling of CPU Time Vs. Number of SP2 Processors (10,000 2D objects)

7.2.3 Base Case: Spheres - Space/Time Bounds

Spheres are considered the base case for DEM, in that they are perhaps the simplest computational primitive that practitioners utilize. Spheres (and discs in 2d) exhibit minimal characteristics in terms of both space and time. The geometry of a sphere can be compactly described in terms of frame (a centroid and possibly orientation), and a radius. The temporal characteristic which we measure is defined by the complexity of checking contact between two spheres. This is minimally achieved by comparing the distance between the sphere centers and the sum of their radii.

We use this as a base case from which to reference more complex geometric primitives such as polyhedra and mathematical surfaces (compactly described as parameterized functions). We make the tacit assumption that each complex primitive will require both additional memory to store the geometric description, and require a commensurate increase in the amount of computation to calculate contact between two objects. The times for a series of tests with different numbers of spheres are shown in Table 7.5 and graphed in Figure 7.8.

Number of Spheres	Single IBM RS/6000-320	Two Networked RS/6000-320s	SP-2, 8 nodes (IP mode)	SP-2, 12 nodes (IP mode)
125	0.06	0.07	0.02	0.08
512	0.31	0.34	0.07	0.12
1000	0.71	0.75	0.14	0.20
4000	5.10	4.43	0.59	0.65
8000	27.02	15.21	1.83	1.73
12000	208.29	106.71	8.63	6.60
16000	463.79	236.92	18.20	13.11
20000	790.98	401.07	32.55	23.39
24000	1192.32	603.48	45.01	31.46
28000	-	-	64.70	45.37
32000	-	-	81.83	57.02
64000	-	-	323.43	223.73
100000	-	-	957.44	666.03

Table 7.5: CPU Times 3D Spheres Test

Summary

Times for large numbers of objects quickly become prohibitive and we see that 1 second of simulation time for a system of 10000 spheres and a nominal time step of 10^{-3} seconds would require approximately 5 days of dedicated CPU time on a single RS/6000 320H processor. This factor is of great practical importance since the nature of this form of analysis requires that many experiments need to be performed so that the observed results are *statistically* meaningful.

The crossover point for the pair of networked machines versus a single machine occurs at approximately 1000 objects. Not only is this surprisingly low, but the speed up for data sets of 10,000 are observed to benefit from near optimal linear speedup (although 2 machines is hardly a generalizable scenario, tests with more machines are ongoing).

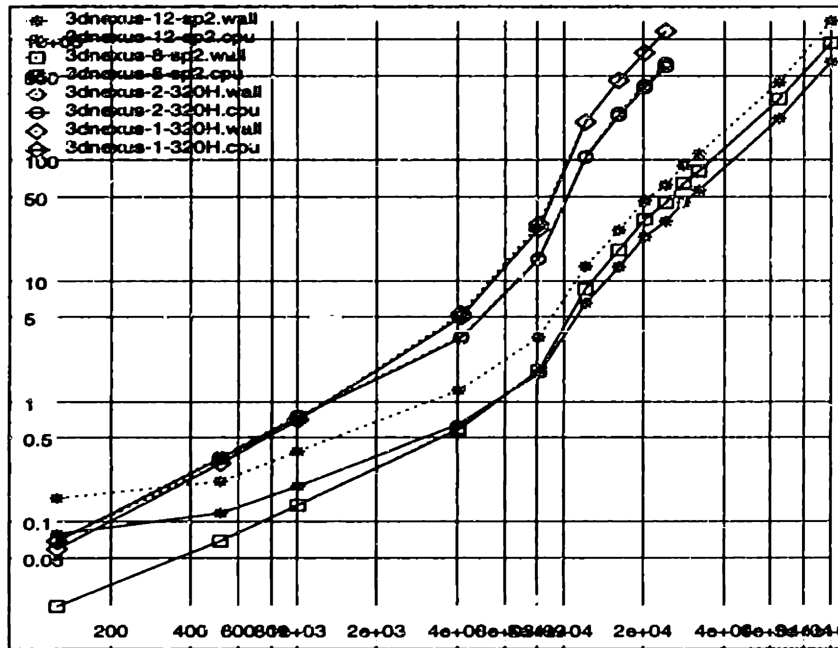

Figure 7.8: CPU Time [sec] Vs Number of Objects on Different Machines

Chapter 8

Applications

This chapter describes several applications of the χ*mal* system. The following examples are presented:

1. 2D Sandglass
2. Deposition - Compaction
3. 3D Geometric Models of a Permeable Solids
4. 3D Retaining Wall Collapse
5. 3D Shock Wave Propagation

8.1 Sandglass - A 2D Hopper-Flow Simulation

The first example in this chapter is a 2D hopper flow timing benchmark. The problem description originally appeared in Williams *et al.* [132]. The test is intended to evaluate the efficiency of 2D contact detection algorithms for arbitrary shaped objects. In particular it is designed to exercise algorithms belonging to the class that make no *a priori* assumptions about the system's evolution. The test exercises both the spatial sorting and the contact resolution algorithms. As such it is expected that 2D simulations using only discs will perform faster.

The test has the following attributes: 1) There are large numbers of contacts at each time step, 2) there is large motion so that contacts cannot be predicted a priori, 3) particles are of arbitrary polygonal shape, 4) the test is insensitive to the detailed shape of each particle and to the shape of the container. The test requires that the dynamic equilibrium equations are solved for the collision of rigid bodies. The χmal system was run with no special tuning for this problem and exhaustive contact testing was executed at each time step.

The parameters to be measured are 1) average CPU time per time step, 2) time spent in sorting and searching, 3) time spent in contact resolution. These will scale with 1) total number of particles, 2) average number of points defining a particle's geometry.

The flow simulation consists of 1000 objects of varying shape and size. The surface geometry of the objects is described using an average of 20 sample points. Shown in Figure 8.1 objects were poured into the upper chamber of a pinched container resembling an *sandglass*. The valve (neck) of the container is initially closed while the objects are deposited. Once the objects have consolidated in the upper chamber of the container, the valve is opened and the objects flow freely through the throat into the lower chamber which is sealed and re-entrant at the base. The closure of the base imposes the condition that all objects must be included in the simulation for the duration of the test.

The simulation was run for 1000 time steps so that average times could be measured for states with little coherence between initial and final configurations. Tests were performed on an IBM Power Station 320H with RISC System 6000 processor (20MHz). All calculations are assumed to take place in core memory. The call sequence for the principal functions the simulation code is shown in Table 8.1.

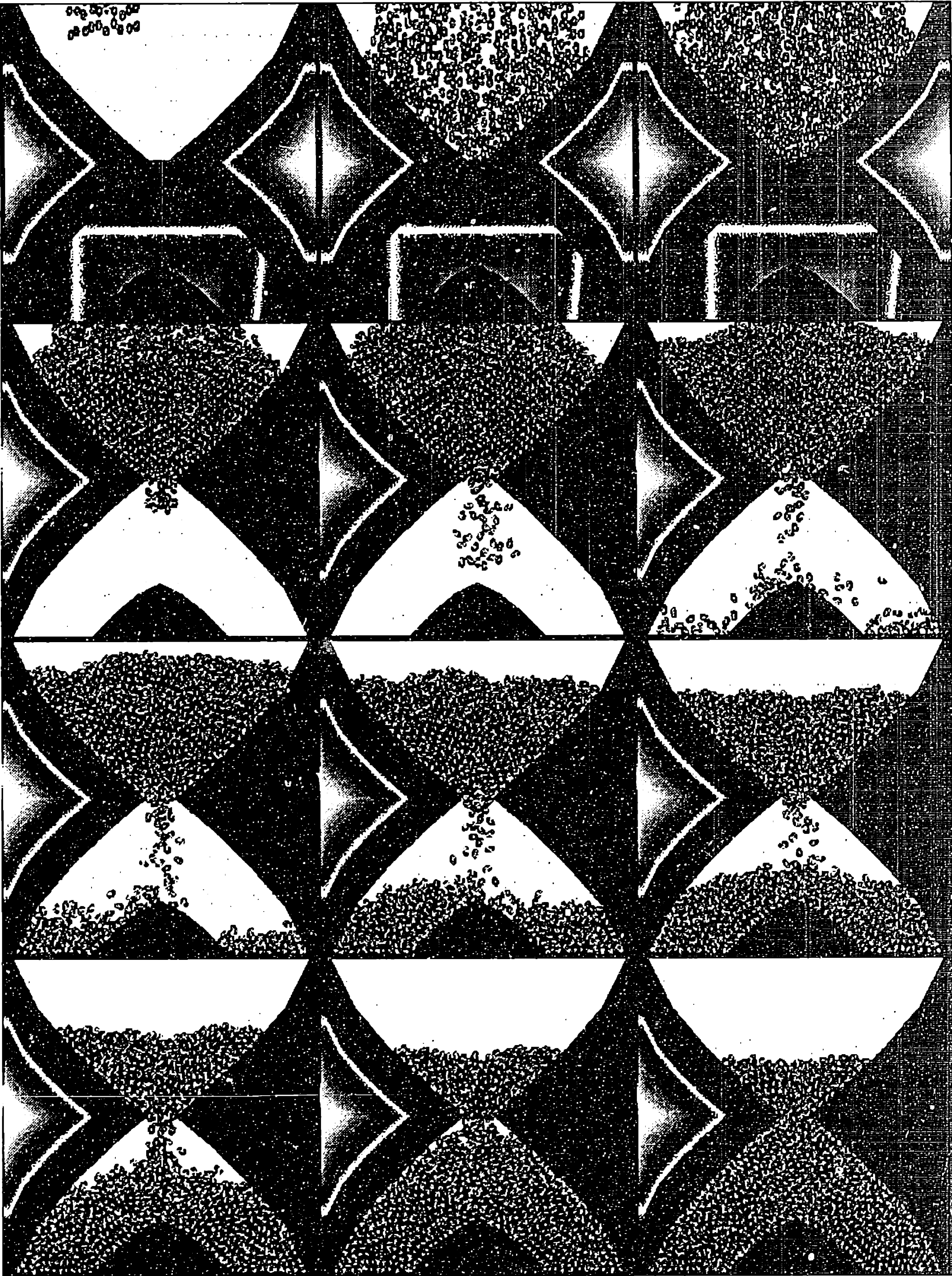

Figure 8.1: Simulation of Particulate Flow
Through *Sandglass*

Parent		Descendant
simulate	→	spatial sort spatial search collide detect
collide detect	→	contact resolve

Table 8.1: Function Call Sequence

Function Name	Time in Self [msec]	Time in Descendents	Total Calls
simulate	77.20	2102.19	1000
spatial sort	0.0	108.89	1000
spatial search	1197.89	0.00	2056000
collide detect	422.90	255.70	8482677
contact resolve	71.33	0.00	981745

Table 8.2: Simulation Profile

The average CPU time per iteration step was 2.2 seconds. The percentage times spent in various phases were determined by running the standard Unix profiler *prof* and are summarized in Table 8.2. It is noted that 90% of the CPU time devoted to contact detection with the remaining 10% associated with solving for the physics of rigid body motion.

Breaking down the contact detection into its two stages, we note the spatial sorting and searching phase accounts for 60% of the CPU time and the contact resolution (DFR) 30%. The force generation using penalty functions was included in the DFR timing but accounted for only 3.0% of the total CPU time for each simulation iteration.

8.2 2D Deposition and Compaction

The problem here is how do we set up the objects at the start of a simulation? We wish to fill a container with a particulate material so that experiments can be performed. However, the *filling* process is extremely time consuming, often requiring more effort than the DEM experiment itself. This application describes an efficient algorithm to pack irregularly shaped objects into a bounded region. The algorithm relies soley on satisfying a geometric constraint that particles do not overlap, essentially a mass conservation law.

8.2.1 Particle Layout And Compaction

The particle deposition phase of the simulation makes use of a fast packing algorithm[1]. The technique is based on parallel updating schemes used in cellular automata methods [121, 27], particle deposition and segregation [2, 30, 28, 64], percolation theory [108], packaging [16, 17], and other applications [71, 89]. The main extension is that the method is applied *off* the lattice, avoiding the so called *lattice artifacts* of the related methods. The scheme has been implemented both *serially* and in *parallel* with substantial reductions in execution time, speeding compaction by at least one order of magnitude over existing techniques. The algorithm has two principal components:

- An incremental state update using a *biased diffusion* model as the underlying mechanism.
- A *parallel configuration space* from which new particle positions and orientations are selected.

Biased Diffusion Configuration Update

The update law is modeled as a simple stochastic process, specifically a random walk with a bias to a preferred direction of motion. The general case of an unbiased random walk can be shown to have a valid physical counterpart in the *diffusion* equation, [47, 72, 64, 13]:

$$\frac{\partial \rho}{\partial t} = D\nabla^2 \rho \tag{8.1}$$

[1]This scheme was originally developed for a data parallel application on a Connection Machine, CM-5[90].

where $\rho(x, y, z)$ is the density of particles over $\mathcal{R}^3$ and D is the coefficient of diffusion.

We use this basic mechanism to describe the motion of the particles over time, with the addition of *scaling* and *bias* terms in each of the coordinate directions. These terms are introduced to control the direction and flux of the particles, typically in a *nominal* flow direction. The scaling terms have the macroscopic effect of making the particle density distribution function spatially anisotropic. The *bias* term will cause the net distribution of particles to migrate in a chosen direction. Rewriting Eq. 8.1 to include these terms we have:

$$\frac{\partial \rho}{\partial t} = D_x\{\frac{\partial^2 \rho}{\partial x^2} + B_x\} + D_y\{\frac{\partial^2 \rho}{\partial y^2} + B_y\} + D_z\{\frac{\partial^2 \rho}{\partial z^2} + B_z\} \quad (8.2)$$

where D_i and B_i are the coefficients of diffusion and spatial biases along axis i respectively.

Parallel Configuration Space

This section describes a method to perform a parallel update of a particle configuration. The update consists of moving as many particles as possible into new positions and orientations in which they remain disjoint from all others. We use the *biased diffusion* equation, Eq. 8.2, to update the particle state (position and orientation) as the principal mechanism of depositing and compacting particles into a container.

Figure 8.2.a shows a set of 2D objects with increments in their state between times $\mathbf{t}$ (shaded) and $\mathbf{t} + \tau$ (unshaded). At time $\mathbf{t}$, the objects make up the $site_0$ configuration. Each object's state vector is then buffered and the copy incremented to states at time $\mathbf{t} + \tau$ using the *biased diffusion* update rule. The *second* set of objects then forms the $site_1$ configuration.

To achieve a viable physical configuration we must ensure that all objects in $site_1$ are disjoint from all others in the same state. For each pair of objects that are not disjoint, the *first* of the pair is deemed to revert to the $site_0$ state. This is tagged by deleting its $site_1$ entry. Moving only the first object of a pair is a consequence of the spatial ordering.

Next, objects that still reside in $site_1$ are checked for contact with the $site_0$ objects. Collectively, a viable configuration is then composed of the $site_1$ state at time $\mathbf{t} + \tau$ **and** the states previously captured in the $site_0$ description. For a change in an object's state to be valid, it must therefore be disjoint in both sites simultane-

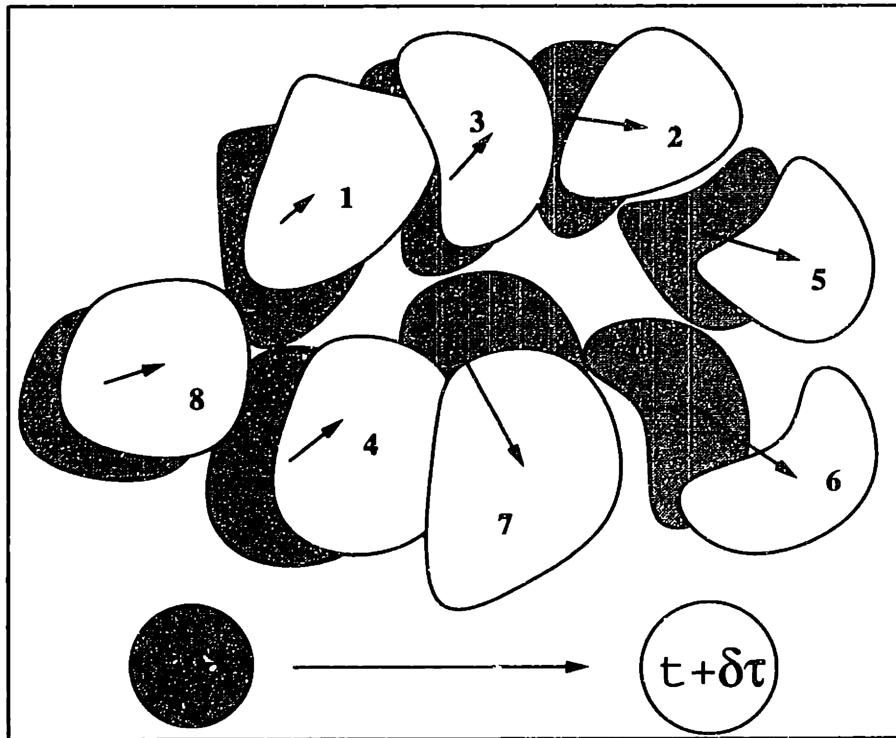

Figure 8.2: Parallel Update Configuration Space

ously. The composite of the two sites, $site_0$ and $site_1$, defines what we term the *parallel configuration space.* By using the $site_0$ configuration, we can guarantee that back propagation is bounded (artificially) to two state changes and prevents further unraveling to previous update steps.

For the configuration shown in Figure 8.2.a, the following state changes occur:

1. First pass through $site_1$ configuration:
   Objects 1 and 4 revert to $site_0$ as they are not disjoint with 3 and 7. Objects 3 and 7 are retained as they occur as the second element of the conflicting pairs. All other objects are retained in the $site_1$ state.

2. Second pass of $site_1$ entries against the entire $site_0$ set:
   Objects 2, 3, 5, 6, 7, 8 were retained in $site_1$ and are now compared with the $site_0$ configuration. Objects 3 and 8 are not disjoint with the $site_0$ configuration and revert to their $site_0$ states. Objects 2, 5, 6 and 7 are disjoint over the entire parallel configuration space and their $site_1$ description is retained.

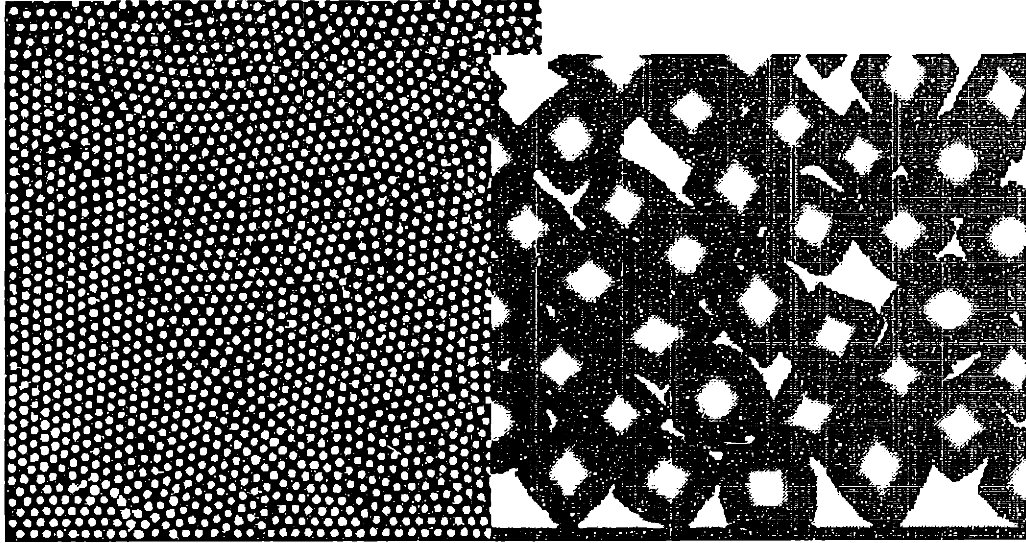

Figure 8.3: Particulate Material Composed of a) Discs b) Superquadric Elements

8.3 3D Geometric Models of Permeable Solids

Modeling[2] materials such as permeable soil or rock requires the creation of a geometric representation of the solid matrix enclosing the interstices. For the model to be permeable, the interstitial spaces must form networks that cross the boundaries of the bulk material. In naturally occurring porous media, these paths are often convoluted and consequently complex to describe mathematically. In this example we present an application to automatically generate models with complex interstitial geometries. The geometric models are intended for use as input to numerical experiments such as fluid flow simulations.

The models are created by taking a solid *cast* of the pore spaces in a densely packed assemblage of particles or grains. One can visualize the numerical process as injecting wax into the pore spaces of the assemblage and dissolving the solid particles in some manner. The wax which remains is a *cast* of the original pore space. Due to the connectivity of the initial particle assemblage, the pore spaces of the wax cast can be guaranteed to connect across the boundaries, ensuring a permeable model.

In order to obtain valid void geometries, the modeling process must first generate a matrix that is in mechanical equilibrium. Only then can we take a wax cast of

[2]Portions of this description appear in a paper presented at the ASME Joint Applied Mechanics and Materials Summer Conference held at La Jolla, CA, 1995, [91]

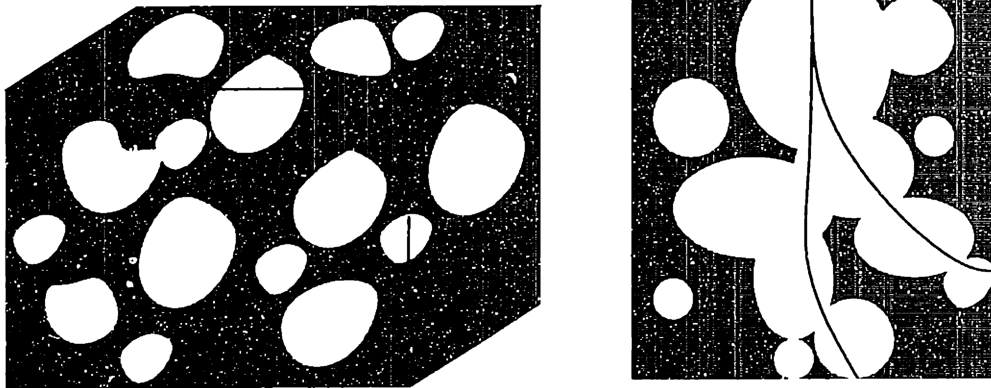

Figure 8.4: Porous Monolith and 2D Section

a void space that is geometrically valid *and* exists as a stable mechanical system. The numerical model must therefore support multibody mechanics, and in particular *contact detection.*

An example of a mechanically viable system can be seen in Figure 8.3.a where we show a 2D simulation of a material described as a collection of disc shaped particles. The particles in this example form contact networks that support the material elements through normal and tangential (frictional) contact forces. To model more complex media, the representation techniques of Chapter 3 are used to describe the particle geometry and the mechanisms by which they interact. For example, in Figure 8.3.b, a 2D system made up from particles of varying shape and size is shown.

Permeable Media

We distinguish between two similar, but distinct, forms of permeable material. The first form is that of particulate materials such as sands and granular soils. Here the permeability of the material is a consequence of the void space between the individual particles.

The second general form, occurs as a *continuous* matrix of material where interstices are formed through a variety of chemical or physical processes. These processes can take place during the formation of the material, or are due to environmental interaction at a later time. For example, in the chemical erosion of limestone by rainwater, channels and cavernous pore spaces (possibly isolated) are formed. The

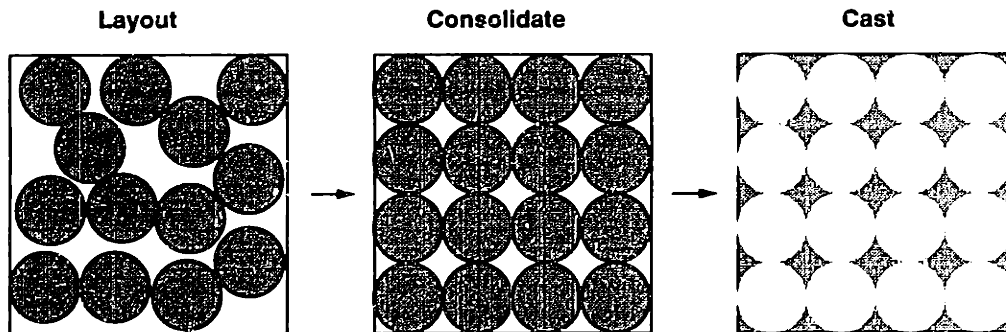

Figure 8.5: Porous Model Generation

variation in how the material is attacked is primarily due to the composition of the material, in this case being made up from sedimented organic remains such as shells. It is this second form of material that we refer to as *monolithic* permeable media.

8.3.1 Overview of Method

In general we can only talk about 3D porous monoliths as a 2D region *cannot* be simultaneously contiguous (monolithic) *and* at the same time contain pores connecting differing points on the boundary of the bulk material. To help visualize the form of the model we are attempting to build, we could think of a porous monolith as something like a block of Swiss cheese, as shown in Figure 8.4. The block will be permeable where some set of the cavities in the bulk material interconnect such that they form a channel through the region. We indicate two such channels on the 2D section in Figure 8.4. It is a model of this form that we wish to generate.

To describe the creation of these model we use the following physical analogies. We first consider a collection of solid bodies or particles placed in a container. The particles are packed into the container using some mechanism mimicking deposition. The structure of the packed material will arise through inter-body contact. For the material to stable, objects lying on top must be supported underneath by a network of contacts reaching down to the bottom objects.

Finally, we treat the entire volume of the container to be a solid material and then remove the particles. This is simply a mathematical inversion of the space, interchanging solid for void and vice-versa. The inversion of the geometric description

is perhaps initially the least obvious. A simple analogy would be to think of filling *all* of the pore spaces with wax and by removing all of the particles, obtaining a solid cast of the pore space. These steps are shown in Figure 8.5. We separate the model generation process into three distinct stages:

1. Object Representation - Generate the geometric description of the particles with support for contact detection.
2. Compaction - Creation of a dense assembly with contact networks to support the structure.
3. Void Space Casting - Calculation of the solid complement of the void space.

8.3.2 Object Representation

In Chapter 3 the DFR scheme was developed to describe the geometry of 3D objects. Support for contact detection comes directly from the same structure. We saw that the scheme incorporates the ability to transform functional descriptions of surfaces into the DFR format. The surfaces are first converted to an implicit form and then evaluated on the sampling grid. In this application we utilize a compact mathematical representation for the geometry of the particles that also allows us to perform the *casting* process. To generate relatively complex object geometries we use a family of implicit functions called superquadrics, [6].

Superquadrics

Superquadrics (superquads) are a generalisation of mathematical functions known as quadric surfaces. The extension comes about by raising the exponents of the variable terms to values other than 2. From the family of possible superquadric functions perhaps the best known is the superellipsoid.

$$\mathcal{I}(x,y,z) = \left\{ \left| \frac{x-x_0}{a_1} \right|^{\frac{2}{\xi}} + \left| \frac{y-y_0}{a_2} \right|^{\frac{2}{\xi}} \right\}^{\frac{\xi}{\eta}} + \left| \frac{z-z_0}{a_3} \right|^{\frac{2}{\eta}} \tag{8.3}$$

where (x_0, y_0, z_0) is origin of the function; a_1, a_2, a_3 are the dimensions of the superquadric semi-major axes extents; and ξ and η are the roundness-squareness pa-

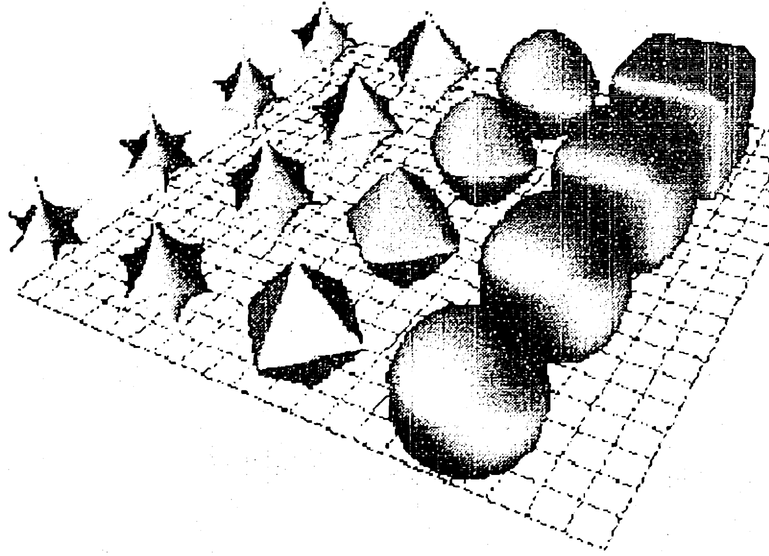

Figure 8.6: Superquadric Ellipsoids with varying exponents, ξ, η

rameters of function in the east-west and north-south directions respectively. We show a range of possible shapes with this function in Figure 8.6.a. In the case where ξ and η both equal 1, the equation reverts to the more familiar ellipsoid.

To manipulate the surface definition we additionally apply control parameters, α and β, to fine tune the locality and intensity of the function:

$$\mathcal{F}(x,y,z) \ = \ \beta\, e^{-\alpha \mathcal{I}(x,y,z)} \tag{8.4}$$

where the parameters α and β define the strength and the degree of decay of the isopotential field $\mathcal{F}_i$ surrounding the potential source $\mathcal{I}$.

To generate even more complex objects we can couple several localized isopotential functions as:

$$\mathcal{M}(x,y,z) \ = \ \sum_{i=1}^{n} \mathcal{F}_i \tag{8.5}$$

which results in aggregate geometries, $\mathcal{M}$.

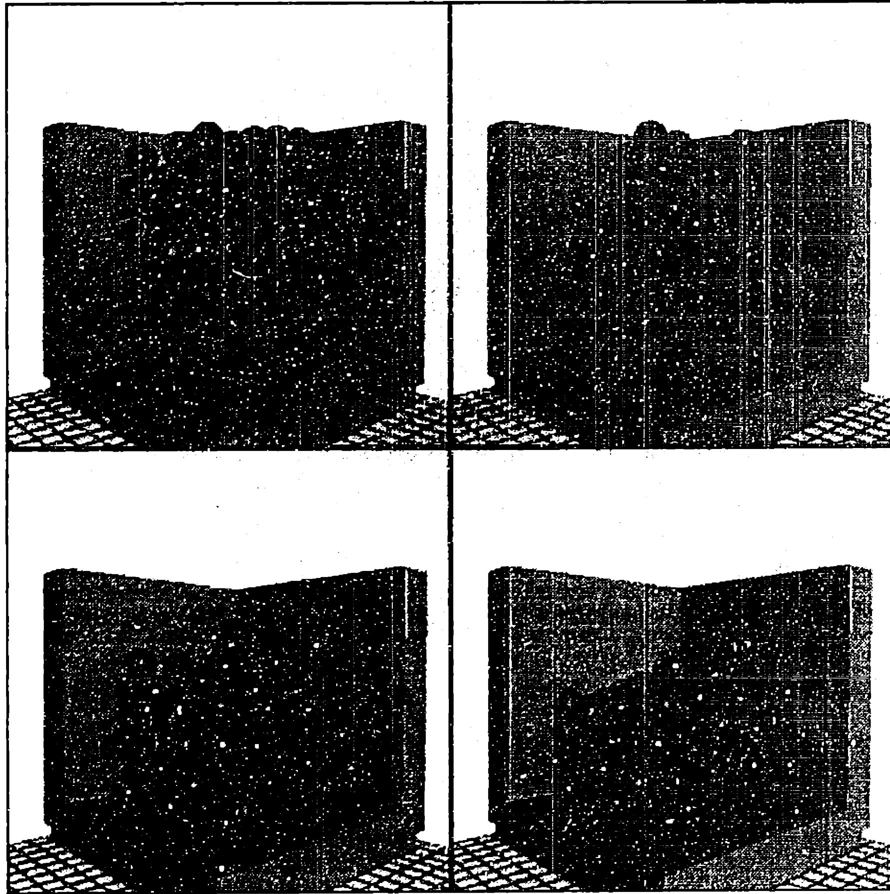

Figure 8.7: 3D Spheres Packing

8.3.3 Particle Layout And Compaction

To generate particle configurations for DEM problems, automatic and semi-automatic layout can be used to accelerate the activity, [118, 62, 71]. Generalized compaction is however a computationally intractable problem and most techniques reported rely on some form of heuristic to achieve satisfactory results, yet still requiring considerable computational effort.

Automatic methods described in the literature rely on human interaction through visual channels. For systems containing large numbers of objects this approach becomes less attractive as it still requires considerable manual effort to remove, or adjust, overlapping or freely suspended objects from the configuration. In 3D models, except for handfuls of objects with very simple geometries, the problem becomes increasingly difficult as interaction is non-trivial as a manual process. Furthermore, visualizing such an activity requires substantial software and hardware support.

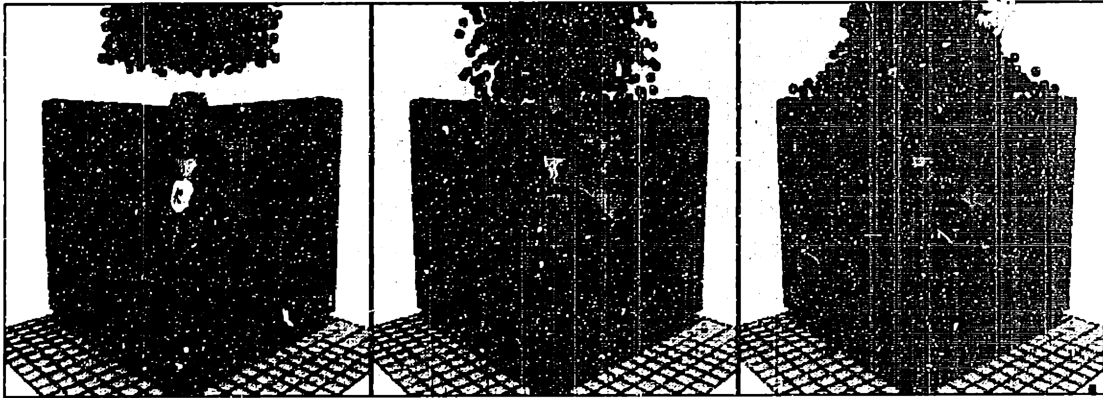

Figure 8.8: 3D Particulate Packing (10,000 grains)

The algorithmic component of the compaction process was described in the deposition application in Section 8.2. Here we show several frames from a 3D packing simulation in Figures 8.7 and 8.8. The flow of particles is directed downwards into a box shaped container. To visualize the packing we have removed the two front facing walls from the rendering.

8.3.4 Void Space Casting - Isosurface Casting

In this section we describe the mathematical operation of taking the complement of the void space to obtain a geometric description of the porous monolith.

So far we have shown the various components necessary to model loose particulate materials. We also reviewed an algorithm to densely pack the particles into a bounded region. These steps allow us to generate a model of a porous granular media. The porosity of a model like the packed spheres in Figure 8.2.b is simply due to the void spaces in between the objects. In this section we further seek to extract a complementary geometric description of the pore spaces. Our analogy is to think of filling *all* of the pore spaces with plaster and then removing all of the particles, obtaining a solid cast of the pore space. We previously showed this simple idea in Figure 8.5.

The trick to obtaining a *cast* of the pore space is to model each of the particles *and* the container using the implicit functions in Eqs. 8.3, 8.4 and 8.5. Using the compact

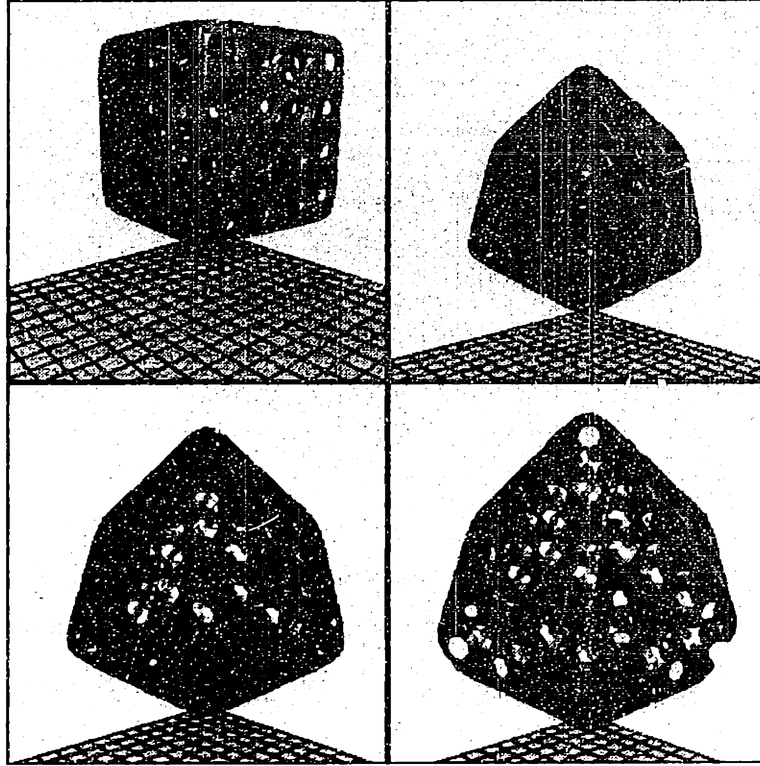

Figure 8.9: Various Views of a Porous Material

mathematical representation of each objects geometry we can describe a collection of particles in the simple algebraic form of Eq. 8.5, i.e. solely in terms of localized potential functions. For a collection of n particles in the form of Eq. 8.4, we can represent the aggregation in the form of Eq. 8.5, such that:

$$\mathcal{A} = \sum_{i=1}^{n} \mathcal{F}_i \tag{8.6}$$

Similarly we model the container as a single object using an implicit function representation, calling this function $\mathcal{C}$. To capture the geometry of the pore space we then literally subtract the aggregate expression for the particles, $\mathcal{A}$, from the expression for the container, $\mathcal{C}$:

$$\mathcal{P} = \mathcal{C} - \mathcal{A} \tag{8.7}$$

By taking the *isosurface* of the resulting potential field $\mathcal{P}$ in Eq. 8.7, the particles are effectively removed and we are left with a surface description of the void space topology. The *isosurface* representation can then be mapped into a DFR structure.

We can now treat the void space cast as a solid body and apply very efficient contact detection tests against it when used in numerical simulations.

8.3.5 Summary

In this example we have described an application of several novel DEM techniques to generate useful geometric models of porous media. We find that starting the modeling process with a general mathematical representation for object geometry and a comprehensive set of DEM algorithms provides a rich set of tools to model complex systems. A simple example of the *Void Space Casting* technique is shown in Figures 8.9. A cubic container was filled with a set of spherical particles and compacted. The resulting isosurface of the void space cast describes the complex monolithic matrix shown. This model can then be used in numerical experiments, such as transport mechanisms in filter design or two phase flow.

8.4 3D Embankment Collapse

In this example a wall of 2600 equi-size spheres was constructed by placed the spheres between two vertical plates and closing the cavity at each end. The spheres are initially hexagonal close packed, Gravity is first applied for 1000 iterations at a time step of 10^{-3} seconds. At $t = 1$ second the left boundary wall was released and the ensuing collapse is shown as a collage of video frames in Figure 8.10. The simulation was run for a total of 3000 iterations taking approximately 2 PCPU hours on 8 nodes of an IBM SP-2 POWER parallel system.

8.5 3D Shock Wave Propagation

As in the previous example, a wall of 2600 equi-size spheres is constructed by placing the spheres between two vertical plates and closing the cavity at each end and from below. Once again the initial packing of the spheres was hexagonal. Gravity was first applied for 1000 iterations at a time step of 10^{-3} seconds. A stream of slightly denser particles of the same size are dropped under gravity from above the wall. On impact the stream creates a series of stress waves which can be seen in Figure 8.11. Later in the simulation the wavefront has been reflected off the boundary walls and interferes with the waves still originating at the source of the disturbance. The simulation was run for 4000 took approximately 3 PCPU hours on 8 nodes of an IBM SP-2 POWER parallel system.

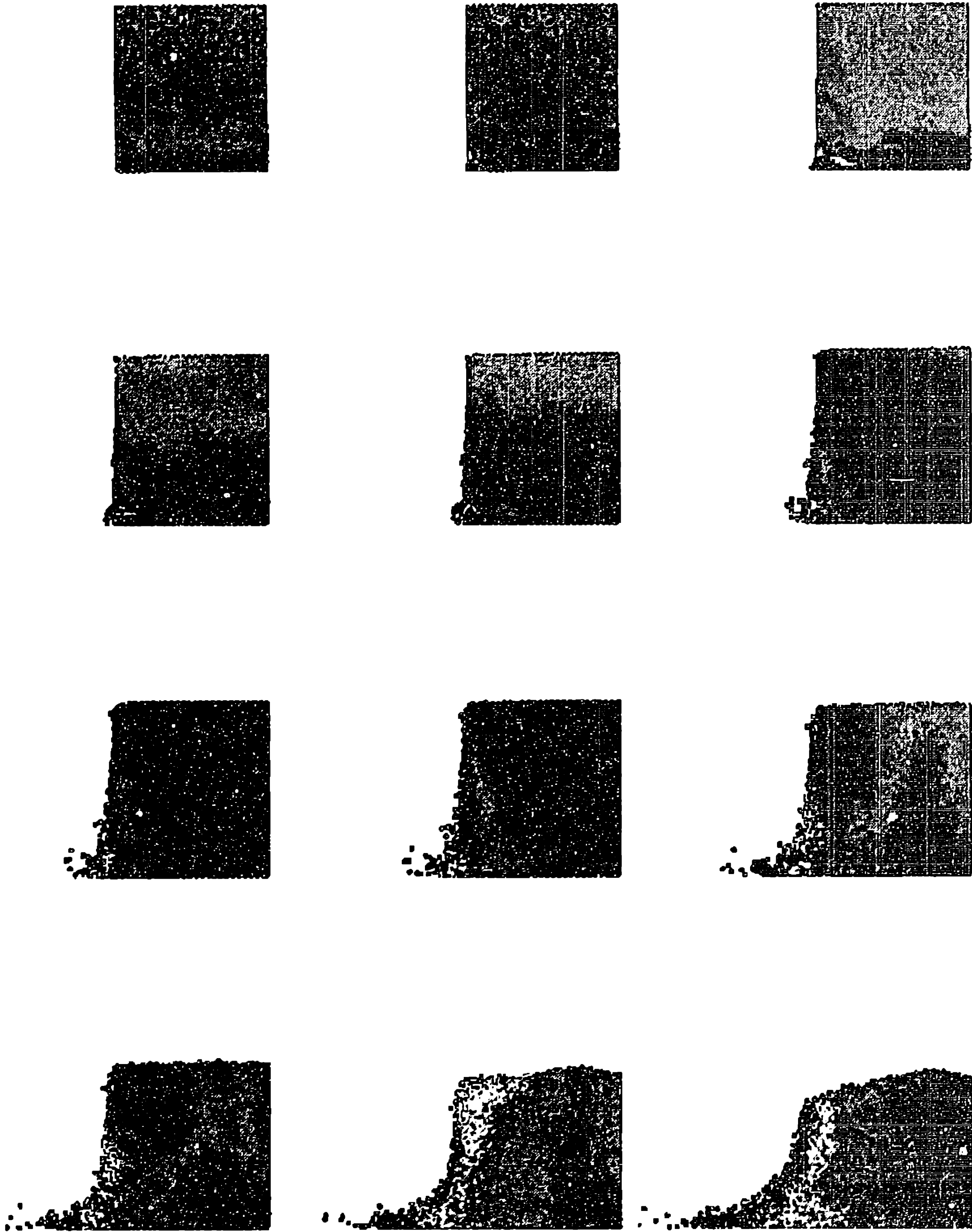

Figure 8.10: Failure of Cut Embankment

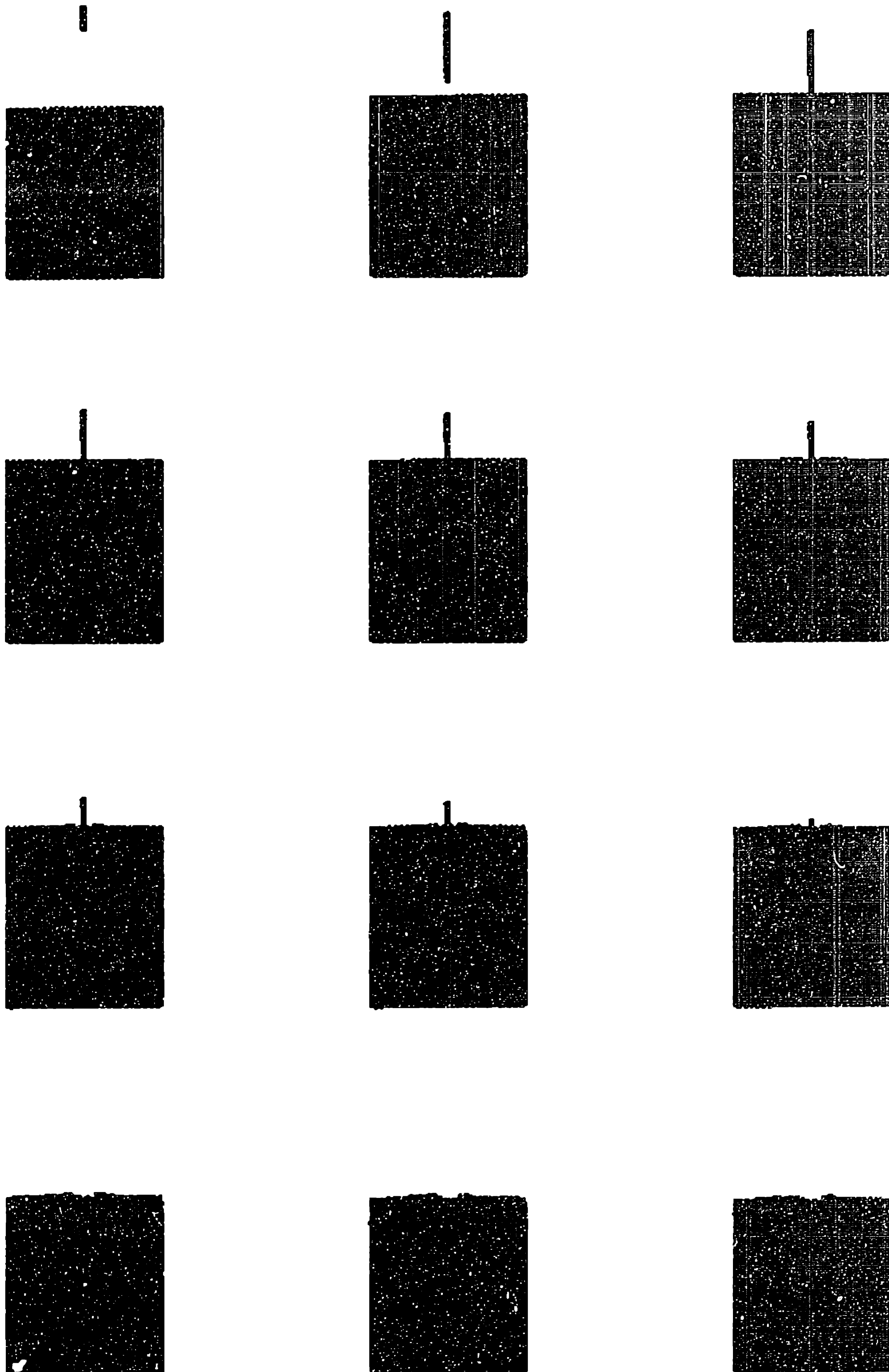

Figure 8.11: Shock Wave Propagation in 3D Particulate Fluid

8.6 Contributions and Future Directions

In this thesis the following contributions have been presented:

1. A new representation scheme for efficient contact analysis, Chapter 3.
2. A large software platform to support parallel DEM modeling that is both portable and modular, Chapter 5.
3. A efficient compaction algorithm, Chapter 8.
4. A technique to generate non-trivial geometric models of porous monoliths, Chapter 8.

Benefits Arising From This Work

The DEM community has in general shunned the use of polygonal representations because of the cost of performing contact detection. The DFR scheme addresses this problem and provides a significant improvement in the treatment of polygonal surface representations. Through the use of parallel computer architectures, and parallel algorithms, the scale of DEM simulation it is feasible to model has been dramatically increased.

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* test models for the physics without having to wait large amounts of time to verify the proposed model numerically.

Secondly, it facilitates the simulation of systems containing numbers 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.

8.6.1 Related Applications

It was noted in Chapter 3 that contact detection arises in a variety of differing forms across a broad spectrum of computer modeling applications. A short list of potential industrial applications includes:

- Engineering Materials - aggregate storage, bulk transport, granular flow, filter design.
- Industrial - Feeder and conveyer belt design.
- Geometric Modeling - CAD interference tests
- Manufacturing - Packaging optimization.
- Physically based simulation - real-time simulated environments and teleoperation (remote operation).
- Robotics - Motion planning (collision avoidance).
- Computer Graphics - rendering
- Engineering Design - prototyping.

8.6.2 Future Work

Three principal extensions are proposed for future development of the χmal system, namely, experimental validation, fluid interaction and scalability.

Experimental Validation As an analysis program the χmal system still requires significant validation through invasive and non-invasive laboratory experimentation coupled with computer imaging techniques.

Fluid Interaction Sand alone is only half a story, fluid, specifically water, will form the counterpoise. This companion volume is glaring in its absence. Perhaps the most important future extension to this work resides in the development of methods to capture the physics of sands and water. At present two directions are being considered, namely, lattice gas models and smooth particle models. The latter approach is a direct extension to the particle based methods already

developed in this work. Lattice gas models have the great appeal of computational elegance in their relative simplicity, while possessing remarkable ability to capture microscale hydrodynamic phenomena. Lying somewhere between the two are lattice Boltzmann methods which may provide the most flexible approach, [18].

Scalability The parallel algorithm developed in Chapter 7 is extremely efficient for relatively large numbers of objects. However, in the future machines with very large numbers of processors will become more widely available. To benefit from the additional computational power it is important that the problem size can scale with the increased size of the machines. This is an ongoing research interest and will be developed in the coming year.

Bibliography

[1] Selim G. Akl. *Parallel Sorting Algorithms.* Academic Press, HBJ Publishers, New York, 1985.

[2] Francis J. Alexander and Joel L. Lebowitz. Driven diffusive systems with a moving obstacle; a variation on the brazil nuts problem. *Journal of Physics A*, 23:375,381, 1990.

[3] Per Bak and Michael Creutz. Dynamics of sand. *Materials Research Society Bulletin*, pages 17,21, 1991.

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

[5] Ricardo Barbosa and Jamshid Ghaboussi. Discrete finite element method. In *Proceedings of the 1st U.S. Conference on Discrete Element Methods (DEM)*, 1989. see [84].

[6] Alan Barr. Super quadrics and angle preserving transformations. *IEEE, Computer Graphics and Applications Vol 1, No.1*, 1981.

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

[8] Klaus-Jurgen Bathe. *Finite Element Procedures in Engineering Analysis.* Prentice Hall Inc., Englewood Cliffs, New Jersey, USA, 1982.

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

[10] 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.

[11] Peter Brett, Steven D. Pieper, and David Zeltzer. Putting it all together: An integrated package for viewing and editing 3d microworlds. In *Proceedings, 4th Usenix Computer Graphics Workshop*, 1987.

[12] 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`.

[13] L. Brieger and E. Bonomi. A stochastic cellular automaton simulation of the non-linear diffusion equation. In Gary D. Doolen, editor, *Lattice Gas Methods - Theory, Applications and Hardware.* MIT Press/North Holland, 1991.

[14] Richard Stevens Burington. *Handbook of Mathematical Tables and Formulas.* McGraw Hill, 4th edition, 1965.

[15] John Canny. *The Complexity of Robot Motion Planning, ACM Doctoral Dissertation Award 1987.* MIT Press, Cambridge, MA, USA, 1988. ACM Distinguished Dissertation Series.

[16] Andrea Casotto, Fabio Romeo, and Alberto Sangiovanni-Vincentelli. A parallel simulated annealing algorithm for the palacement of macro cells. *IEEE Computer*, CAD-6(5):838,847, 1987.

[17] Andrea Casotto and Alberto Sangiovanni-Vincentelli. Placement of standard cells using simulated annealing on the connection machine. *IEEE Transactions on Computer-Hardware*, 7:350,353, 1987.

[18] Carlo Cercignani. *The Boltzmann Equation and Its Applications.* Springer-Verlag, 1988. Applied Mathematical Sciences, 67.

[19] David T. Chen. *Pump It Up: Computer Animation of a Biomechanically Based Model of Muscle using the Finite Element Method.* PhD thesis, Media Lab, MassachusettsInstitute of Technology, 1992.

[20] Shiyi Chen, Karen Diemer, Gary D. Doolen, Kenneth Eggert, Castor Fu, Semion Gutman, and Bryan J. Travis. Lattice gas automata for flow through porous media. In Gary Doolen, editor, *Lattice Gas Methods, Theory Applications and Hardware*, pages 72,84, USA, 1991. MIT Press/North-Holland.

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

[22] John J. Craig. *Introduction to Robotics: Mechanics and Control.* Addison-Wesley Pub. Co., 1986.

[23] P.A. Cundall. Ball - a program to model granular media using the distinct element method. Technical Report TN-LN-13, Dames and Moore, Advanced Technology Group, London, 1978.

[24] P.A. Cundall and O.D.L. Strack. A distinct element model for granular assemblies. *Geotechnique*, 29:47,65, 1979.

[25] Peter A. Cundall and Roger D. Hart. Numerical modeling of discontinua. In *Proceedings of the 1st U.S. Conference on Discrete Element Methods (DEM), see [84]*, 1989.

[26] R. Dobry and Tang-Tat Ng. Discrete modeling of stress-strain behavior of granular media at samll and large strains. In Graham G. Mustoe, M. Henriksen, and H-P Huttelmaier, editors, *Proceedings of the 1st U.S. Conference on Discrete Element Methods (DEM)*, Colorado School of Mines, Golden, CO, 1989.

[27] Gary Doolen, editor. *Lattice Gas Methods, Theory Applications and Hardware*, USA, 1991. MIT Press/North-Holland.

[28] G.D. Duffy and M.I.Darby. Simulated structure of particulate deposition from a model of turbulent gas flow. *Journal of Physics D*, 24:1665,1672, 1991.

[29] Paul Dworkin. Effcient collision detection for real-time simulated environments. Master's thesis, Media Lab, Massachusetts Institute of Technology, 1994.

[30] G.C. Barker et al. Comment on 'three-dimensional model for particle-size segregation by shaking' (with reply). *Physical Review Letters*, 70:2194,2195, 1993.

[31] 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`.

[32] Ian Foster. *Designing and Building Parallel Prgrams.* Addison Wesley Pub., USA, 1995.

[33] G.G.W. Mustoe G. Hocking and J. R. Williams. Validation of cice discrete element code for ride-up and ice ridge/cone interaction. In *Proceedings ARTIC '85 San Francisco.* ASCE, New York, 1985.

[34] J.R. Williams G. Hocking and G.G.W. Mustoe. Dynamics analysis for three dimensional contact and fracturing of multiple bodies. In *Proceedings of NU-META 1987, Numerical Methods in Engineering, Theory and Applications*, Rotterdam, 1987. Balkema.

[35] A. Geist, A. Beguelin, J. Dongarra, Jiang, and R. Manchek. *PVM3 Users Guide and Reference Manual.* Oak Ridge National Laboratory, Tennessee, 1993.

[36] Jamshid Ghaboussi, Milind M. Basole, and S. Ranjithan. Three dimensional discrete element analysis on massively parallel computers. In *Proceedings of the 2nd International Conference on Discrete Element Methods (DEM)*, pages 95,105, 1993. see [131].

[37] Suresh Goyal, Elliot N. Pinson, and Frank W. Sinden. Simulation of dynamics of interacting rigid bodies including friction 1: General problem and contact model. *Engineering With Computers*, 10:162,174, 1994.

[38] Suresh Goyal, Elliot N. Pinson, and Frank W. Sinden. Simulation of dynamics of interacting rigid bodies including friction 2: Software system design and implementation. *Engineering With Computers*, 10:175–195, 1994.

[39] Stuart Green. *Parallel Processing for Computer Graphics.* MIT Press, USA, 1991. Research Monographs in Parallel and Distributed Computing Series.

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

[41] Howard Gutowitz, editor. *Cellular Automata, Theory and Experiment.* MIT Press/North-Holland, USA, 1991.

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

[43] Motohiko Hakuno, Kazuyoshi Iwashita, and Yoshihiko Uchida. A dem of cliff collapse and debris flow. In *Proceedings of the 1st U.S. Conference on Discrete Element Methods (DEM), see [84]*, 1989.

[44] Motohiko Hakuno, Yuji Tarumi, and Kimiro Meguro. A dem simulation of concrete fracture, fault rupture and sand liquefaction. In *Proceedings of the 1st U.S. Conference on Discrete Element Methods (DEM), see [84]*, 1989.

[45] Motohiko Hakuno, Daisuke Uemura, and Yamamoto. A dem simulation of underground structures. In *Proceedings of the 1st U.S. Conference on Discrete Element Methods (DEM), see [84]*, 1989.

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

[47] Milton E. Harr. *Mechanics of Particulate Media - A Probabilistic Approach.* McGraw Hill, 1977.

[48] Brosl Hasslacher. Discrete fluids. *Los Alamos Science*, 1987. Special Edition.

[49] Brian Von Herzen, Alan H. Barr, and Harold R. Zatz. Geometric collisions for time dependent parametric surfaces. In *Proceedings of ACM SIGGRAPH 90*, volume 24, pages 39–48, 1990.

[50] W. Daniel Hillis. The connection machine: A computer architecture based on cellular automata. *Physica D Nonlinear Phenomena*, 10(1&2):213,228, 1984. Proceedings of Interdisciplinary Workshop. Los Alamos, New Mexico. Edited by Doyne Farmer and Tommaso Toffoli and Stephen Wolfram.

[51] W. Daniel Hillis. *The Connection Machine.* MIT Press, USA, 1985. ACM Distinguished Dissertation Series.

[52] Caroline Hogue and David Newland. Efficient computer simulation of moving granular particles. *Powder Technology*, 78(1):51,66, 1994.

[53] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison Wesley Pub. Co., USA, 1979.

[54] 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.

[55] 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.

[56] Gen hua Shi. Discontinuous deformation analysis - a new numerical model fo the statics and dynamics of deformable block structures. In *Proceedings of the 1st U.S. Conference on Discrete Element Methods (DEM), see [84]*, 1989.

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

[58] International Business Machines. *IBM AIX Parallel Environment, Parallel Programming Subroutine Reference*, 1994.

[59] Isao Ishibashi, Tarun Agarwal, and Syed A. Ashraf. Anisotropic behaviors of glass spheres by a discrete element model and laboratory experiment. In *Proceedings of the 1st U.S. Conference on Discrete Element Methods (DEM), see [84]*, 1989.

[60] Jay A. Issa and Richrad B. Nelson. Numerical analysis of micromechanical behavior of granular materials. In *Proceedings of the 1st U.S. Conference on Discrete Element Methods (DEM), see [84]*, 1989.

[61] Steven Feiner James Foley, Andries van Dam and John F. Hughes. *Fundamentals of Interactive Computer Graphics.* Addison Wesley, USA, 2nd edition, 1990.

[62] X. Zhuang J.D. Goddard and A.K. Didwan. Microcell methods and the adjacency matrix in the simulation of the quasi-static mechanics of granular media. In *Proceedings of the 2nd International Conference on Discrete Element Methods (DEM)*, pages 3,14, 1993. see [131].

[63] George Vanecek Jr. Towards automatic grid generation usnig binary space partition trees. Technical report, Department of Computer Science, Purdue University, West Layfayette, IN 47907, 1994.

[64] Remi Jullien, Paul Meakin, and Andre Pavlovitch. Three-dimensional model for particle-size segregation by shaking. *Physical Review Letters*, 69:640,643, 1992.

[65] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, 2nd edition, 1988.

[66] D.E. Knuth. *The Art of Computer Programming, Volume 3, Sorting and Searching.* Addison Wesley, Reading, Mass, USA, 1973.

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

[68] Matthew R. Kuhn and James K. Mitchell. The modeling of soil creep with the discrete element method. In *Proceedings of the 1st U.S. Conference on Discrete Element Methods (DEM), see [84]*, 1989.

[69] Jonathan Leech and Russell Taylor. Interactive modeling using particle systems. In *Proceedings of the 2nd International Conference on Discrete Element Methods (DEM)*, pages 105,117, 1993. see [131].

[70] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffery V. Hill, W. Daniel Hillis, Bradley C. Kuszamaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak. The network architecture of the connection machine cm-5.

Technical report, Thinking Machines Corp., 1994. Also available by ftp from the Laboratory for Computer Science, Massachusetts Institute of Technology.

[71] Zhenyu Li. *Compaction Algorithms for Non-Convex Polygons and Their Applications.* PhD thesis, Division of Applied Sciences, Harvard University, Cambridge, Massachusetts, 1994.

[72] J. Litwiniszyn. Stochastic methods in mechanics of granular bodies. Presented at International Centre for Mechanical Sciences, Department of General Mechanics, Udine, Poland in 1972. Courses and Lectures No. 93.

[73] 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.

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

[75] Norman Margolus. Physics-like models of computation. *Physica D Nonlinear Phenomena*, 10(1&2):81,95, 1984. Proceedings of Interdisciplinary Workshop. Los Alamos, New Mexico. Edited by Doyne Farmer and Tommaso Toffoli and Stephen Wolfram.

[76] Tomas Lozano-Perez Joeseph L. Jones Emmanuel Mazer and Patrick A. O'Donnell. *HANDEY, A Robot Task Planner.* MIT Press, Cambridge, Massachusetts, USA, 1992.

[77] N.J. Meegoda and D.W. Washington. Massively parallel computers for microscopic modeling of soils. In H.J. Siriwardane and M.M. Zaman, editors, *Proceedings of the 8th International Conference on Computer Methods and Advances in Geomechanics*, volume 1, pages 617,622, 1994.

[78] Dimitri Metaxas. Fast dynamic point-to-point constraint algorithm for deformable bodies. In *Proceedings of the 2nd International Conference on Discrete Element Methods (DEM)*, pages 27,38, 1993. see [131].

[79] Kim Molvig, P. Donis, and Christopher Teixeira. Lattice gas aerodynamics. *Physics of Fluids*, 1992. Submitted for Review.

[80] Claudio Montani and Michele Re. Vector and raster hidden-surface removal using parallel connected stripes. *IEEE Computer Graphics and Animation*, 7(7):14,23, 1987.

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

[82] Ante Munjiza and D.R.J. Owen. Discrete elements in rock blasting. In *Proceedings of the 2nd International Conference on Discrete Element Methods (DEM)*, pages 287,300, 1993. see [131].

[83] Ante Munjiza, D.R.J. Owen, and J.R. Williams. On a rational approach to rock blasting. In H.J. Siriwardane and M.M. Zaman, editors, *Proceedings of the 8th International Conference on Computer Methods and Advances in Geomechanics*, volume 1, pages 857,862, 1994.

[84] G.G. Mustoe, M. Henriksen, and H-P Huttelmaier, editors. *Proceedings of the 1st U.S. Conference on Discrete Element Methods (DEM)*, Colorado School of Mines, Golden, CO, 1989.

[85] Tang-Tat Ng and H. Eliot Fang. Cyclic behavior or arrays of ellipsoids with different particle shapes. In *Proceedings of Joint ASME Applied Mechanics and Materials Summer Conference, Mechanics of Materials with Discontinuities and Heterogeneities Symposium*, volume AMD-Vol. 201, pages 59,70, UCLA, Los Angeles, 1995.

[86] Tang-Tat Ng and H. Eliot Fang. Cyclic behavior or arrays of ellipsoids with different particle shapes. In *Proceedings of Joint ASME Applied Mechanics and Materials Summer Conference, Mechanics of Materials with Discontinuities and Heterogeneities Symposium*, volume AMD-Vol. 201, pages 59,70, UCLA, Los Angeles, 1995.

[87] Tang-Tat Ng and Xiaoshan Lin. Numerical simulations of naturally deposited granular soil with ellipsoidal elements. In *Proceedings of 2nd International Conference on Discrete Element Methods (DEM)*, pages 557,567, 1993. See [131].

[88] Gregoire Nicolis and Ilya Prigogine. *Exploring Complexity - An Introduction.* W.H. Freeman and Company, New York, 1989.

[89] F.D. Nobre, A.M. Mariz, and E.S.Sousa. Spreading of damage: An unexpected disagreement between the sequential and parallel updatings in monte carlo simulations. *Physical Review Letters*, 69(1):13,16, 1992.

[90] Ruaidhrí O'Connor. Packing of granular materials using, vectorized collision detection. In *Student Workshop on Scalable Computing*, Brewster, MA, 1994. MIT Laboratory for Computer Science.

[91] Ruaidhrí O'Connor and John R. Williams. Geometric modeling of porous media using the discrete element method. In Anil Misra and Ching S. Chang, editors, *Symposium on the Mechanics of Materials with Discontinuities and Heterogeneities*, UCLA, La Jolla, CA, USA, 1995. ASME Joint Applied Mechanics and Materials Summer Conference.

[92] John F. Olson and Daniel H. Rothman. Three-dimensional immiscible lattice gas: application to sheared phase separation. *J. Stat. Physics*, 1995. Submitted for Review.

[93] G.N. Pande, G. Beer, and J.R. Williams. *Numerical Methods in Rock Mechanics.* Wiley, 1992.

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

[95] Alex P. Pentland and John R. Williams. Good vibrations: Modal dynamics for graphics and animation. *ACM Computer Graphics*, 23(3), 1989.

[96] P.J. Plauger. *The Standard C Library.* Prentice Hall, 1st edition, 1992.

[97] Dale S. Preece. A numerical study of bench blast row delay timing and its influence on percent-cast. In H.J. Siriwardane and M.M. Zaman, editors, *Proceedings of the 8th International Conference on Computer Methods and Advances in Geomechanics*, volume 1, pages 863,870, 1994.

[98] Dale S. Preece and Steven L. Burchell. Variation of spherical element packing angle and its influence on computer simulations of blasting induced rock motion. In *Proceedings of the 2nd International Conference on Discrete Element Methods (DEM)*, pages 255,264, 1993. see [131].

[99] Dale S. Preece, Steven L. Burchell, and D. Scott Scovira. Coupled explosive gas flow and rock motion modeling with comparison to bench blast field data. In Hans-Peter Rossmanith, editor, *Proceedings of the 4th International Symposium on Rock Fragmentation By Blasting*, pages 239,245, Vienna, 1993.

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

[101] David F. Rogers and J. Alan Adams. *Mathematical Elements for Computer Graphics.* McGraw Hill, 1976.

[102] Daniel H. Rothman. Cellular-automaton fluids: A model for flow in porous media. *Geophysics*, 53:509,518, 1988.

[103] 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.

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

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

[106] 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.

[107] Robert Sedgewick. *Algorithms in C.* Addison Wesley, 1990.

[108] Dietrich Stauffer. *Introduction To Percolation Theory.* Taylor and Francis Inc., 1987.

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

[110] L.M. Taylor and Dale S. Preece. Simulation of blasting induced rock motion using spherical element models. *Engineering Computations*, 9(2), 1992. See also [84].

[111] Christopher M. Teixeira. *Continuum Limit of Lattice Gas Fluid Dynamics.* PhD thesis, Dept. of Nuclear Engineering, Massachusetts Institute of Technology, 1992.

[112] Charles Terzaghi. Modern conceptions concerning foundation engineering. *Journal of the Boston Society of Civil Engineers*, 12(10), 1925.

[113] Thinking Machines Corp., Cambridge, Massachusetts. *The Connection Machine System, Introduction to Programming in C/Paris*, 1989.

[114] Thinking Machines Corporation, Cambridge, Massachusetts. *The Connection Machine System, C* Programming Guide*, May 1993.

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

[116] Colin Thornton. Applications of dem to process engineering problems. In *Proceedings of the 1st U.S. Conference on Discrete Element Methods (DEM), see [84]*, 1989.

[117] Colin Thornton, Guoping Lian, and M.J. Adams. Modelling of liquid bridges between particles in dem simulations of particle systems. In *Proceedings of the 2nd International Conference on Discrete Element Methods (DEM), see [131]*, 1989.

[118] 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.

[119] 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.

[120] John M. Ting and Larry Meachum. Effect of bedding plane orientation on the behavior of granular systems. In Anil Misra and Ching S. Chang, editors, *Symposium on the Mechanics of Materials with Discontinuities and Heterogeneities*, pages 43,58, UCLA, La Jolla, CA, USA, 1995. ASME Joint Applied Mechanics and Materials Summer Conference.

[121] T. Toffoli and N. Margolus. *Cellular Automata Machines.* MIT Press, USA, 1987.

[122] Tommaso Toffoli. Cellular automata as an alternative to (rather than an approximation of) differential equations in modeling physics. *Physica D Nonlinear Phenomena*, 10(1&2):117 127, 1984. Proceedings of Interdisciplinary Workshop. Los Alamos, New Mexico. Edited by Doyne Farmer and Tommaso Toffoli and Stephen Wolfram.

[123] B.C. Trent and L.G. Margolin. A numerical laboratory for granular solids. *Engineering Computations*, 9:191–197, 1992.

[124] Bala R. Vatti. A generic solution to polygon clipping. *Communications of the ACM*, 35(7):56,63, July 1992.

[125] Otis R. Walton. Numerical simulation of inclined chute flows of monodisperse inelastic, frictional spheres. *Mechanics of Materials*, 1992. Draft Manuscript.

[126] 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.

[127] 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.

[128] John R. Williams and Kevin Amaratunga. Wavelet representation of geometry for analysis. In *Proceedings of the 2nd International Conference on Discrete Element Methods (DEM)*, pages 65,80, 1993. see [131].

[129] John R. Williams, Andrie Chen, and Dennis Petrie. Ice island interaction with conical production structures. In *Proceedings of the 1st U.S. Conference on Discrete Element Methods (DEM)*, 1989. see [84].

[130] 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.

[131] John R. Williams and Graham G.W. Mustoe, editors. *Proceedings of the 2nd International Conference on Discrete Element Methods (DEM)*, Dept. of Civil & Environmental Engineering, Massachusetts Institute of Technology, 1993. IESL Publications.

[132] John R. Williams and Ruaidhrí O'Connor. A linear complexity intersection algorithm for discrete element simulation of arbitrary geometries. *International Journal of Computer Aided Engineering - Engineering Computations*, 12(2), 1995. Special Edition on Discrete Element Methods.

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

[134] 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.

[135] Stephen Wolfram. Universality and complexity in cellular automata. *Physica D, Nonlinear Phenomena*, 10(1&2):1,35, 1984. Proceedings of Interdisciplinary Workshop. Los Alamos, New Mexico. Edited by Doyne Farmer and Tommaso Toffoli and Stephen Wolfram.

[136] Stephen Wolfram, editor. *Thermodynamics and Hydrodynamics of Cellular Automata*. World Scientific, 1986.

[137] Karen Worgan and Graham G.W. Mustoe. Application of the discrete element method to modeling subsurface penetration of a uniform ice cover. In *Proceed-*

ings of the 1st U.S. Conference on Discrete Element Methods (DEM), 1989. see [84].

[138] F. Zhao and S.L. Johnsson. The parallel multipole method in the connection machine. *SIAM, Journal of Scientific Statistical Computating*, 12:1420,1437, 1991.