T-3634

TITLE: A MULTI-PROCESSOR IMPLEMENTATION OF A MONOTONIC LOGICAL GRID DATA STRUCTURE IN THREE DIMENSIONS

by

Gary D. Bond

T-3634

A thesis submitted to the Faculty and the Board of Trustees of the Colorado School of Mines in partial fulfillment of the requirements for the degree of Master of Science (Mathematics).

Golden, Colorado

Date: 4 Apr 89

Signed: Gary D. Bond

Gary D. Bond

Approved: Jean L Bell

Dr. J. Bell
Thesis Advisor

Golden, Colorado

Date: 4 Apr 89

Dr. Ardel Boes
Head Mathematics
Department

ABSTRACT

This thesis illustrates how the Monotonic Logical Grid (MLG) data structure can be used to take advantage of multi-processor architectures. The MLG was developed by J. P. Boris to solve the nearest neighbors problem in particle dynamics simulation programs. The MLG solves the nearest neighbors problem through preservation in the computer storage structure of the spatial relationships among simulated particles. The MLG's preservation of spatial relationships makes it particularly useful for multi-processing large data set problems.

The MLG's spatial preservation characteristic supports partitioning of simulated particles into independent subsets of related particles. The subsets of partitioned particles are allocated to separate processors where updating of the particles is performed concurrently. This thesis presents a Pascal implementation of the partitionable MLG data structure and the results of its performance. Performance results indicate the MLG data structure reduces execution times to near theoretical limits.

A mesh generation and reconfiguration algorithm, intended for use in simulating dynamic spatial regions, is also presented. The mesh generation algorithm defines a

collection of tetrahedral sub-elements which fill the simulated spatial region. As the spatial region moves, the tetrahedral sub-elements representing it deform. Left unchanged, these deformed sub-elements would result in numerical inaccuracies. The reconfiguration algorithm presented identifies deformed sub-elements and reconfigures the local areas where deformed sub-elements are located.

TABLE OF CONTENTS

ABSTRACT iii

LIST OF FIGURES vii

LIST OF TABLES x

ACKNOWLEDGEMENTS xi

INTRODUCTION 1

CHAPTER I -- THE MONOTONIC LOGICAL GRID DATA STRUCTURE 8

Initial Development 8
Monotonic Logical Ordering Maintenance 16
Spatial Representation 23

CHAPTER II -- A PARTITIONABLE THREE-DIMENSIONAL MONOTONIC LOGICAL GRID DATA STRUCTURE 27

Introduction 27
Particle Dynamics Model 29
A Three-Dimensional Monotonic Logical Grid Data Structure 34
Multi-Process Configuration 48
Monotonic Logical Grid Performance Evaluation 55

CHAPTER III -- MESH GENERATION AND RECONFIGURATION ALGORITHM FOR SPATIAL REPRESENTATION 69

Introduction 69
Derivation Of The Combined Mesh Generation Algorithm 72
An Overview Of The Combined Mesh Generation Algorithm 75
Combined Mesh Generation Algorithm Input And Data Structure 78
Boundary Vertex Triangulation 79
Non-Boundary Vertex Integration 101
Inter-Plane Tetrahedra Configuration 108
Inter-Plane Tetrahedra Refinement 110
Integration of New Vertexes Into The Tetrahedral Mesh 115

Reconfiguration of Ill-Configured Tetrahedra 116
Conclusions 119

CHAPTER IV -- CONCLUSIONS AND FUTURE DIRECTIONS 120

REFERENCES CITED 123

BIBLIOGRAPHY 124

APPENDIX A CODE FOR BORIS' VECTOR SORT 126

APPENDIX B CIRCLE AND SPHERE CALCULATIONS 129

LIST OF FIGURES

FIGURE		PAGE
1.1	Particle Mapping From Simulation Space To MLG Data Structure	12
1.2	MLG Lattice For Irregularly Spaced Particles	14
1.3	Non-unique MLG Lattices For The Same Collection Of Particles	15
1.4	MLG's Data Arrays With Address Offsets	18
1.5	MLG Data Structure In Boris' Full Sort	21
1.6	Vector Formation For Use In Boris' MLG Vector Sort	22
1.7	Triangular Sub-Element Characterization Points	24
2.1	Fifty Particle Orbits	30
2.2	Parameters Used In Determination Of Particle Orbits	33
2.3	Projection Of 1000 Particles Onto The XZ-Plane (Time=0.0)	41
2.4	Projection Of Particles With X-Index Values Of 1 Or 10 (Time=0.0)	42
2.5	Projection Of 1000 Particles Onto The XZ-Plane (Time=0.5)	44
2.6	Projection Of Particles With X-Index Values Of 1 Or 10 (Time=0.5)	45
2.7	Assignment Of Data Partitions To Separate Processors	47
2.8	Multi-Process Configuration	51

FIGURE PAGE

2.9 Timing Comparison For Three Process Configurations 58

2.10 Percent Total Real Time Per Iteration 59

2.11 Realized Reduction For Multiple Child Processes 60

2.12 Execution Times For Varying Iteration Levels 61

2.13 Percent Reduction For Varying Iteration Levels 62

2.14 Execution Times For Varying Particle Counts 63

2.15 Percent Reduction For Varying Particle Counts 64

2.16 Execution Times For Varying Particle Counts (Extended) 65

2.17 Execution Times For Multiple Trials 66

2.18 Real Time Per Iteration 67

3.1 Irregular Spatial Regions 80

3.2 Boundary Vertex Triangulation (Step 1) 83

3.3 Boundary Vertex Triangulation (Step 2) 85

3.4 Boundary Vertex Triangulation (Step 3) 87

3.5 Boundary Vertex Triangulation (Step 4) 92

3.6 Boundary Vertex Triangulation (Step 5) 94

3.7 Boundary Vertex Triangulation (Complete) 96

3.8 Feasible Vertex Test (Initial Convex Cone) 97

FIGURE PAGE

3.9 Feasible Vertex Test (Second Convex Cone) 99

3.10 Delaunay Triangulation 104

3.11 Insertion Polygon Identification 106

3.12 Insertion Polygon Reconfiguration 107

3.13 Final Tetrahedral Mesh 114

LIST OF TABLES

TABLE PAGE

1.1 Full Sort Groups 19

2.1 Theoretical Performance Of Multi-Processing 56

ACKNOWLEDGEMENTS

There are several people I would like to thank for their assistance in the preparation of this thesis. First, I thank Dr. Jean Bell, of the Colorado School of Mines, for her patience in reading and editing this manuscript, and her constant encouragement and enthusiasm in the project. I also thank Dr. Robert Underwood and Dr. Ardel Boes for their assistance. I also wish to thank Dr. Davis Oliver, of the Martin Marietta Aerospace Corporation, for his assistance in resolving several geometric problems and his willingness to proof read and verify their presentation. Finally, I thank my wife Angella, for her patience and willingness to endure the long hours spent at the office working on this problem.

INTRODUCTION

Large scale physical simulation models developed today are greater in magnitude and scope than their predecessors. Improvements in computational speeds of modern supercomputers have prompted simulators to develop simulations which deal with larger numbers of particles and which represent higher dimensional spaces than previously attempted. Unfortunately, the enhancements pursued by the simulators often require processing performance beyond the capabilities of available hardware. As is so often the case with computer models, the processing performance demanded is always greater than the performance available. Therefore, attempts must be made to take full advantage of the capabilities provided by modern supercomputers.

Technical problems and cost factors have forced computer designers to shift their emphasis from improving scalar processor speeds to other techniques which make computers perform better. In the past, designers of new computers were able to improve scalar processing speeds at considerable rates, but hardware design limitations, such as cooling and chip thickness, have slowed improvements in recent years. Some designers speculate that the rate of improvement in processing speeds will continue to drop, since they feel

current chip materials have a fixed upper bound on performance, which is rapidly being approached. As the rate of improvement in scalar processing speeds drops, and more and more time and effort is required, the cost of developing faster scalar processors increases. These influences have forced computer designers to pursue other techniques to improve computer performance.

In recent years, multi-processor architectures have become prevalent in supercomputers. Multi-processor architectures are a relatively cheap alternative to improving scalar processor speeds. Computer designers realized that combining several processors, built on existing technology, in a single cabinet can provide significant performance improvements without requiring significant development costs. Development costs for multi-processor architectures are lower since they do not require much additional hardware to be developed.

Taking advantage of multi-processor architectures requires the problem being analyzed to be divided in some manner. For computationally intensive problems, it is often possible to divide the problem into a set of sub-problems, which can be allocated to and solved by separate processors. For data intensive problems, there is a second approach called data partitioning, where the simulated data elements

are divided into mutually exclusive subsets. The data subsets are allocated to separate processors where each processor solves the identical problem, but each processor solves only the problem for data elements in its subset. Both these approaches to sharing computation among several processors can be used to reduce execution times.

This paper describes a data structure which is particularly useful for data partitioning. The data structure described is the Monotonic Logical Grid (MLG), which was developed by J. P. Boris for use in solving the nearest neighbors problem [Boris 1985], often encountered in particle dynamics models. Boris showed the MLG data structure reduced the time required to solve the nearest neighbors problem by keeping particles, which are near one another in the simulated space, near one another in its data structure. This spatial preservation is the characteristic which makes the MLG data structure useful in data partitioning.

The MLG's preservation of spatial relationships enables data partitioning to be achieved through indices into its data structure. The particles are mapped into the MLG data structure based on their locations in the simulated space. Once mapped into the MLG data structure, particles near one another in the simulated space are near one another in the

MLG data structure. A partitioning index is selected, from among the MLG's indices (assuming a three-dimensional simulation space there would be three possible indices), and contiguous values of the index are assigned to each subset. For example, creation of three data partitions when the partitioning index has a maximum value of 9 would be achieved by assigning index values 1, 2 and 3 to the first subset, 4, 5 and 6 to the second subset, and 7, 8, and 9 to the third subset.

In addition to the development and evaluation of the partitionable MLG data structure, this paper will also discuss the development of an automatic mesh generation and reconfiguration algorithm. Some physical simulation models, such as hydrodynamics models and finite element models, must define and maintain a collection of region filling sub-elements which represent spatial regions. Two algorithms are presented which support development of spatial representation models: a fully automated mesh generation algorithm which defines a collection of tetrahedral sub-elements for filling both regular and irregular spatial regions, and a mesh reconfiguration algorithm which identifies ill-shaped sub-elements and also reconfigures the local areas where they are located.

Many of the techniques currently available for generating a mesh for use in finite element models have been categorized into two groups by J. C. Cavendish [Cavendish, Field, and Frey 1985]. Cavendish defines these two groups as: interpolation mesh generators and automatic triangulators. The former require considerable effort on the part of the user in defining a partitioning of the region into sub-blocks. These sub-blocks are used by the interpolators to generate space curves which are blended together to form the solid. The latter produce space-filling assemblies of triangular (in three-dimensions, tetrahedral) elements. The technique developed in this research falls into the latter category.

The automatic mesh triangulator presented utilizes the tetrahedra as its basic sub-element. The ease of reconfiguring a tetrahedron provided the primary incentive for its use. A tetrahedron can be divided into several smaller tetrahedra without affecting neighboring tetrahedra, a property not found in polyhedra of higher order. Therefore, reconfiguration of local areas of the mesh can be achieved without affecting the entire mesh.

In addition to its reconfiguration properties, tetrahedral elements also reduce the number of cusps and irregularities arising in filling curved regions. Higher

order polyhedra require large numbers of small elements to represent a curved region, whereas tetrahedral elements can approximate a curved region without a significant increase in the number of elements.

The mesh reconfiguration algorithm presented was developed for use in simulating dynamic spatial regions. As the region moves the sub-elements which fill the region distort. The distorted sub-elements, left unchanged, can introduce numerical inaccuracies into the simulation. Factors contributing to distortions include: shear flows (where two fluids are flowing side by side at differing velocities), turbulent flows (a breaking wave), and colliding surfaces (the distortion of a bullet impacting an object). The numerical inaccuracies that arise may result in incorrect conclusions. Therefore, maintenance of a collection of well-shaped sub-elements is critical.

This paper will discuss in detail the development, implementation, and performance evaluation of the partitionable MLG data structure, in conjunction with a description of the development and implementation of the tetrahedral mesh generation and reconfiguration algorithms. Chapter I will discuss the theory of the MLG, and previous research relating to two-dimensional spatial representations using the MLG. Chapter II will discuss the implementation

and performance evaluation of the partitionable MLG. Chapter III will discuss the development and implementation of the three-dimensional mesh generation and reconfiguration algorithms. Chapter IV will discuss plans relating to further development of the partitionable MLG for multi-processing applications and further development of the mesh algorithms.

CHAPTER I

THE MONOTONIC LOGICAL GRID DATA STRUCTURE

Initial Development

The Monotonic Logical Grid (MLG) data structure was developed by J.P. Boris for use in solving the nearest neighbors problem [Boris 1985]. The problem has persisted in computational physics and computational geometry for several decades. It is defined as the problem of identifying which other particles have the greatest influence on any specific particle in the simulation. Once the particles which influence the reaction of a particular particle are identified, that particular particle's state may be updated. The particles which have the greatest influence are those which are nearest the particular particle. State updates usually take into account positions and velocities of the nearest neighbors.

The MLG data structure reduces the time required to solve the nearest neighbors problem by maintaining the particles in a data structure which preserves their spatial relationships. Preservation of spatial relationships implies that particles which are close to one another in the

simulated space are close to one another in the MLG data structure. Closeness in terms of the simulated space is determined by the concept of a Euclidean norm. Closeness in terms of the MLG data structure is determined by the difference in the particles' cells (as measured by the difference in their index values) within the data structure.

The MLG's preservation of spatial relationships within the data structure reduces the time required to identify a particle's nearest neighbors. A specific particle's nearest neighbors are easily identified by searching cells located close to the specific particle's cell. Those particles which have the greatest influence on the specific particle (its nearest neighbors) are located in cells close to the specific particle's cell, because they are the particles which are nearest the specific particle in the simulated space. Boris has shown that the nearest neighbors problem can be solved, using the MLG data structure, with a computational cost which scales as $O(n)$, rather than the $O(n^2)$ required by previous algorithms [Boris 1985].

The MLG data structure preserves spatial relationships through conditions relating to the mapping of particles from the simulation space to the data structure. These conditions, were defined by Boris in his original paper, and are repeated here [Boris 1985]. For N particles located in a

three dimensional space, three arrays containing a single component of each particles' position constitute an MLG if and only if:

```
X(i,j,k) <= X(i+1,j,k)  for 1<= i <=NX-1,
Y(i,j,k) <= Y(i,j+1,k)  for 1<= j <=NY-1, and
Z(i,j,k) <= Z(i,j,k+1)  for 1<= k <=NZ-1,
```

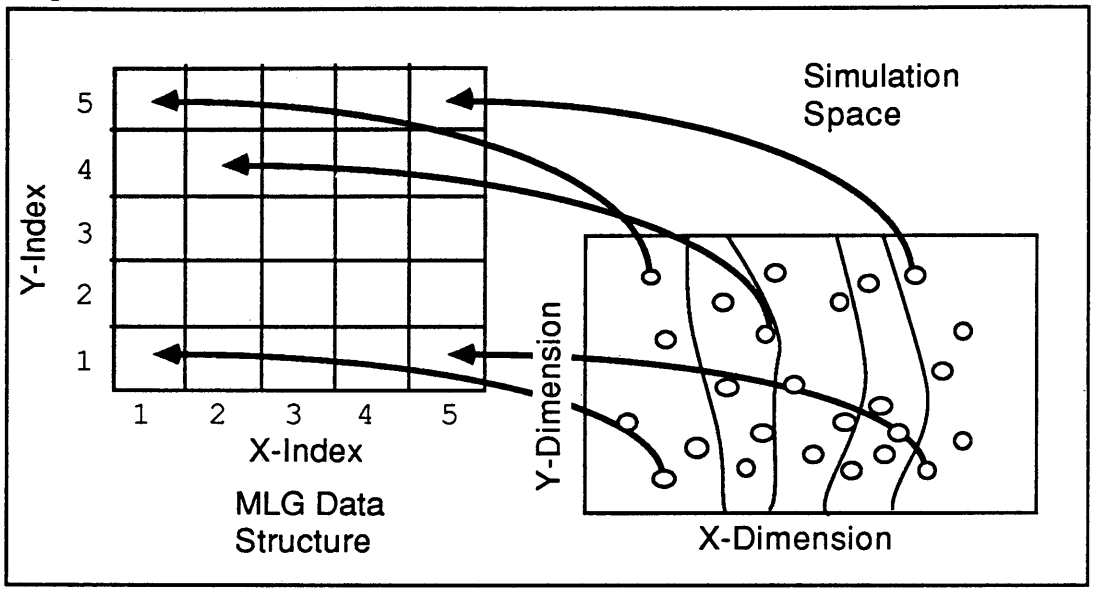
where X contains the X-component of the particles' positions, Y contains the Y-component, and Z contains the Z-component. The parameters NX, NY, and NZ are the maximum values assumable by the respective array indices, where the product of the parameters must equal N, the number of particles being simulated. These conditions imply, that for a constant Y-index value and a constant Z-index value, increasing the X-index value will identify particles whose position vector's X-component increase monotonically. Similarly, holding constant the Z-index value and the X-index value and increasing the Y-index value will identify particles whose position vector's Y-component increase monotonically. Thus, these conditions will be referred to as monotonicity conditions.

The MLG's monotonicity conditions are used to map particles into the cells of the data structure. Mapping of particles into the MLG data structure is performed by sorting the particles into monotonic logical order, based upon their

positions in the simulated space. Thus, for a two-dimensional problem where X and Y are the primary and secondary sort dimensions respectively, the particle mapped into cell 1-1 of the MLG will have the position with the smallest Y-component among the NY particles with the smallest X-components. The particle mapped into cell NX-NY will have the position with the greatest Y-component among the NY particles with the smallest X-components.

Figure 1.1 is an example of how particles are mapped from the simulation space into the MLG data structure. This figure depicts a collection of 25 particles in a two-dimensional simulation space which are being mapped into a five-by-five MLG data structure. The vertical lines drawn in the simulation space separate the particles into five groups, based on the X-component of their position. The group located on the lower end of the X-dimension consists of the particles with the smallest X-component values, and the group located on the upper end of the X-dimension consists of the particles with the greatest X-component values. The group in which a particle is placed determines the corresponding X-index value which it will be assigned.

FIGURE 1.1: Particle Mapping From Simulation Space To MLG Data Structure



The monotonicity conditions result in two important relationships between particle locations in the simulated space and within the MLG. First, the direction of movement between two particles in the simulated space will be approximately the same direction needed to move between the particles' cells in the MLG data structure. Second, particles whose positions in the simulated space are between two bounding particles will have cell locations which are between the bounding particles cell locations. These relationships are useful in determining areas of the MLG data structure to search when locating particles.

The MLG's monotone ordering maintains spatial relationships even though particle positions become

irregular. Figure 1.2 presents four snapshots of twenty-five particles in a two-dimensional simulation space. The connections between particles represent neighboring particles in the MLG data structure. Frame 1 of the figure presents the twenty-five particles in a configuration which could easily be represented by a regular two-dimensional array. Frames 2, 3 and 4 present the same particles as they move and spatial relationships become more irregular. These irregular relationships do not affect the ability to map into the MLG data structure, but they do make mapping into a regular two-dimensional array difficult.

The monotonicity conditions do not guarantee a unique mapping of particles. Multiple mappings can occur when two particles' positions have an identical component, thus enabling particle cells to be swapped and monotonicity conditions be met. Figure 1.3 presents an example where two mappings of the same collection of particles occurs; the two filled particles have positions with identical X-component values. The occurrence of multiple mappings of a given set of particles does not affect the results established in this research.

Figure 1.2: MLG Lattice For Irregularly Spaced Particles Copied from figure in [Boris 1985]

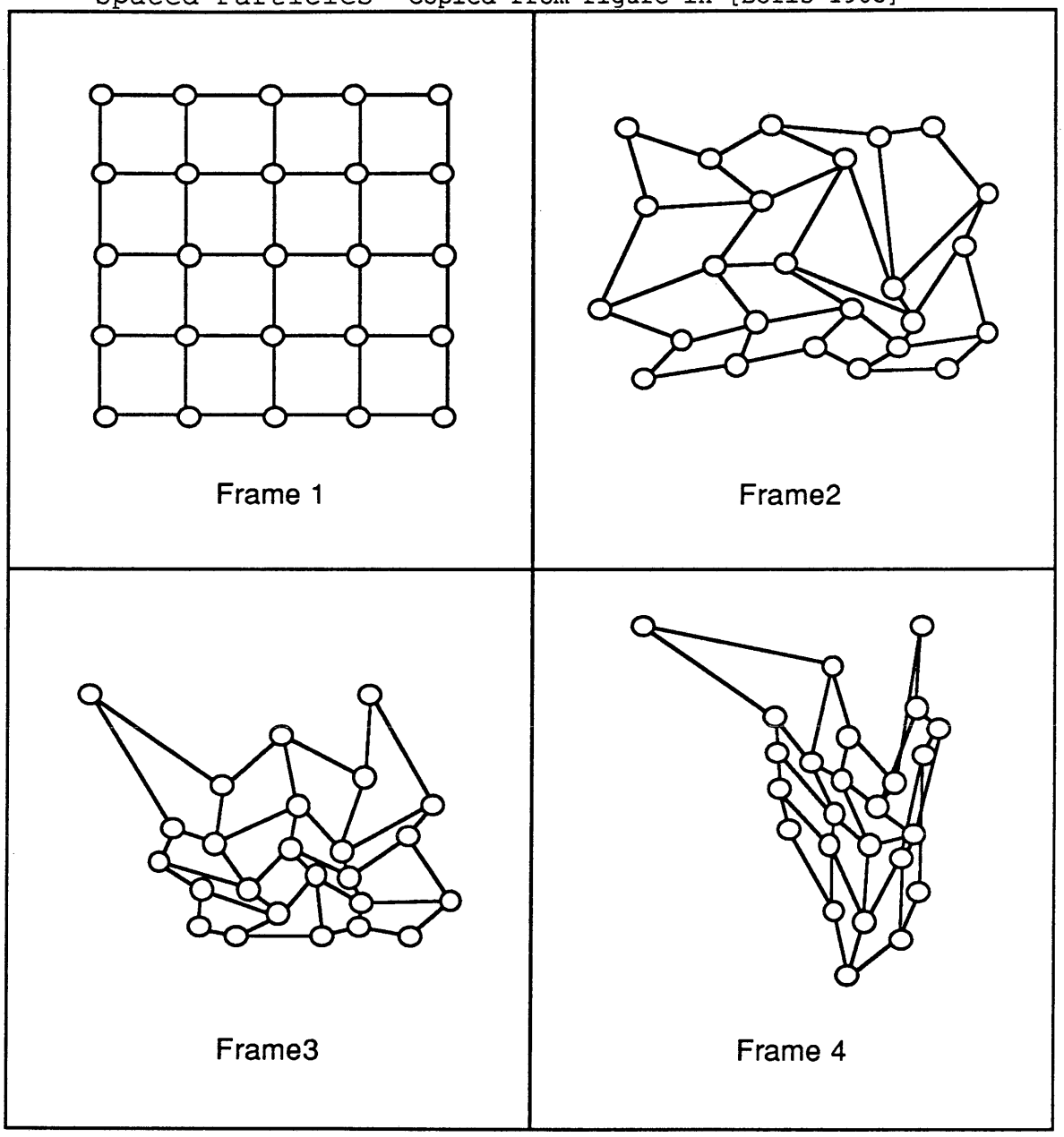
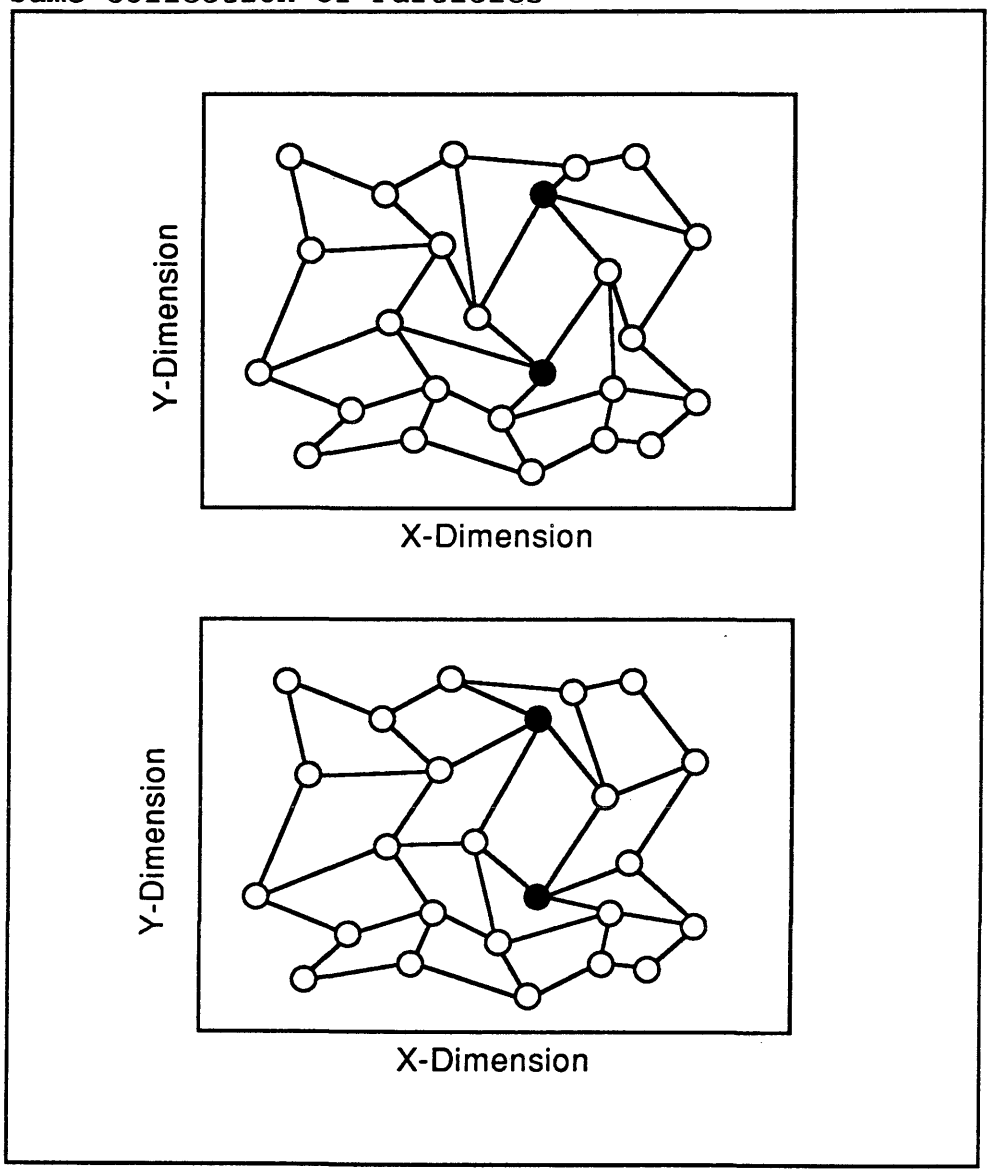


Figure 1.3: Non-unique MLG Lattices For The Same Collection Of Particles



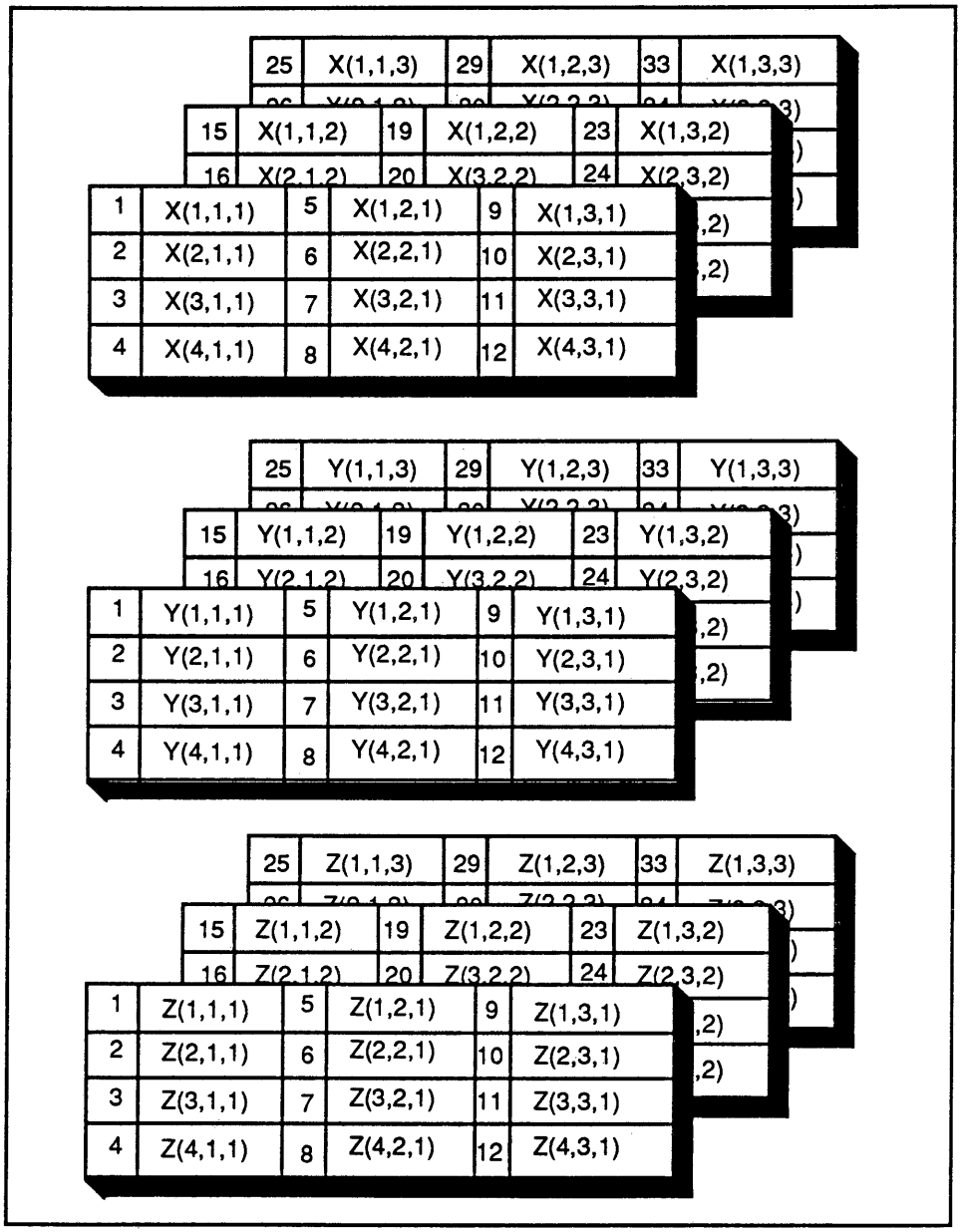
Monotonic Logical Ordering Maintenance

J. P. Boris has developed two algorithms for sorting particles, within the MLG data structure, into monotonic logical order [Boris 1985]. The first sort is an order (nlogn) sort, which will be referred to as the full sort. The second sort is an order (n) sort, which will be referred to as the vector sort. These sort algorithms are discussed in detail by Boris, and since they are not crucial to the analysis performed in this study they will only be discussed briefly in the following paragraphs.

Before the sorts are described, it is important to understand how particles are stored in the MLG data structure. The collection of all parameters relating to a specific particle is called a cell in the MLG data structure, although each different parameter is maintained in its own array. A triplet of index values is assigned to each particle. The triplet determines the location of the particle's cell. The triplet is used to calculate an address offset, relative to the start address of the MLG's parameter arrays. Thus, a particle's cell does not consist of a single location in memory, rather it is a collection of memory locations (one for each parameter) determined by a single triplet of index values.

Figure 1.4 depicts the relationship between address offsets and index values for the X-, Y-, and Z-arrays. The values in the single element columns are address offsets, which are determined by the index values (located within the parentheses) in the neighboring column. Thus, even though each of the three arrays have three indices they are stored in memory in a sequential (or linear) fashion, with the relationship between index values and address offsets determined by the programming language. This linear storage is made use of in Boris' full sort algorithm.

Figure 1.4: MLG Parameter Arrays With Address Offsets



Boris' full sort algorithm performs a bubble sort on consecutively smaller groups of particle cells. The first step in the full sort is to sort all particles based on their elements in the X-array. The second step is to divide the cells into NX groups of NY*NZ particles (where NX, NY and NZ are the maximum array index values for the three indices) and sort each group separately based on their elements in the Y-array. The third step is to divide the cells into NX*NY groups of NZ particles and sort each group based on their elements in the Z-array. These steps are summarized in Table 1.1.

Table 1.1: Full Sort Groups

Step	Number of Groups	Number of Cells Per Group	Sort Array
1	1	N=NX*NY*NZ	X
2	NX	NY*NZ	Y
3	NX*NY	NZ	Z

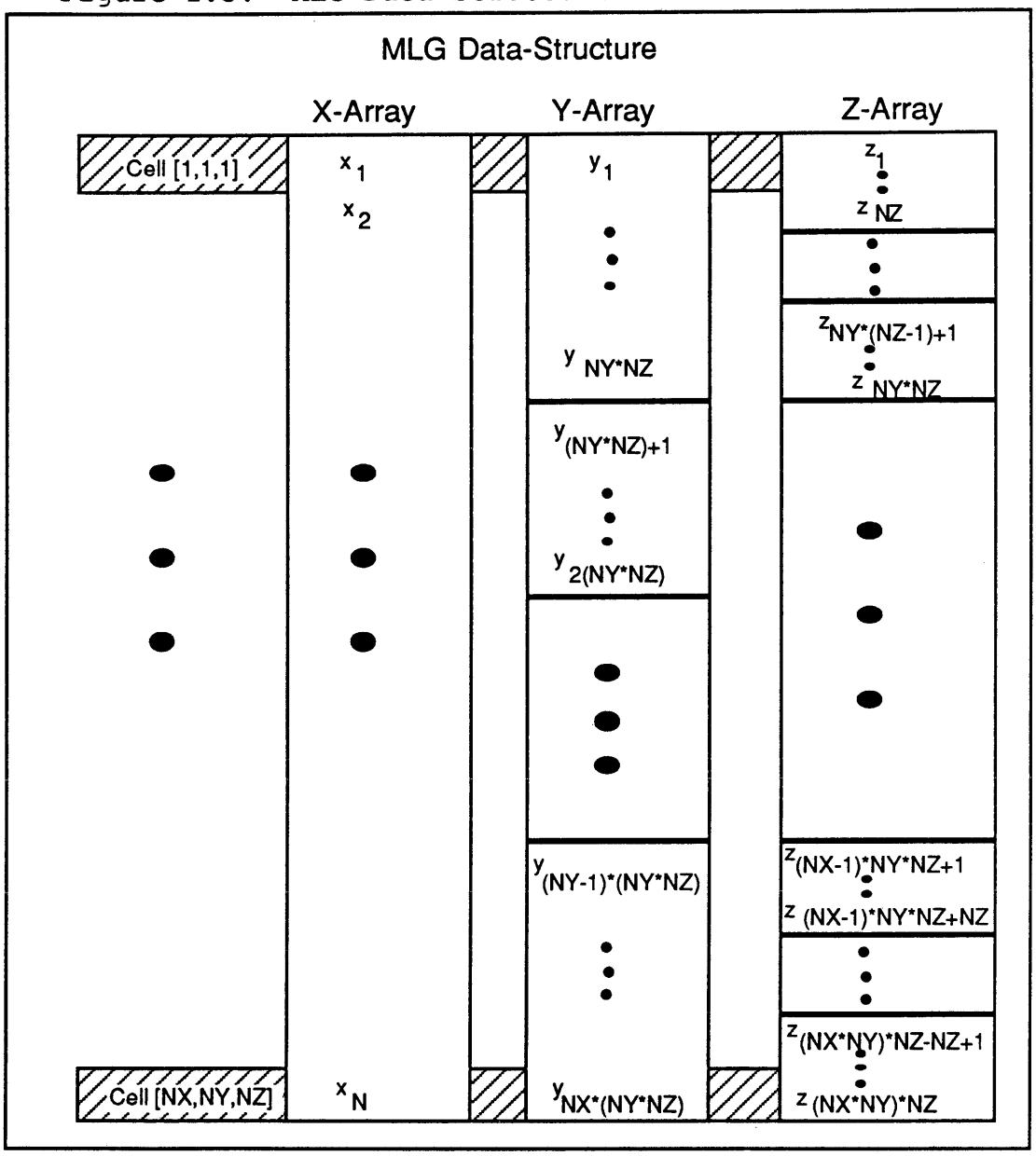
Figure 1.5 presents the MLG's data arrays in their linear form along with the cell groupings used in Boris' full sort. Each of the cell groups used in the three step full sort are identified based on the array used in the step. Since a particle's parameters reside in several arrays, whenever two cells are out of order then all of the data

arrays must be reordered. Completion of the above steps guarantees the monotonicity conditions can be met for any collection of particles.

Unfortunately, the computational cost of the algorithm precludes its use for most practical purposes. For practical purposes, Boris developed the vector sort algorithm which achieves the monotonicity conditions, and scales as order(n) rather than order (nlogn). Boris' vector sort achieves this performance by integrating the concept of dimensionality into the sort.

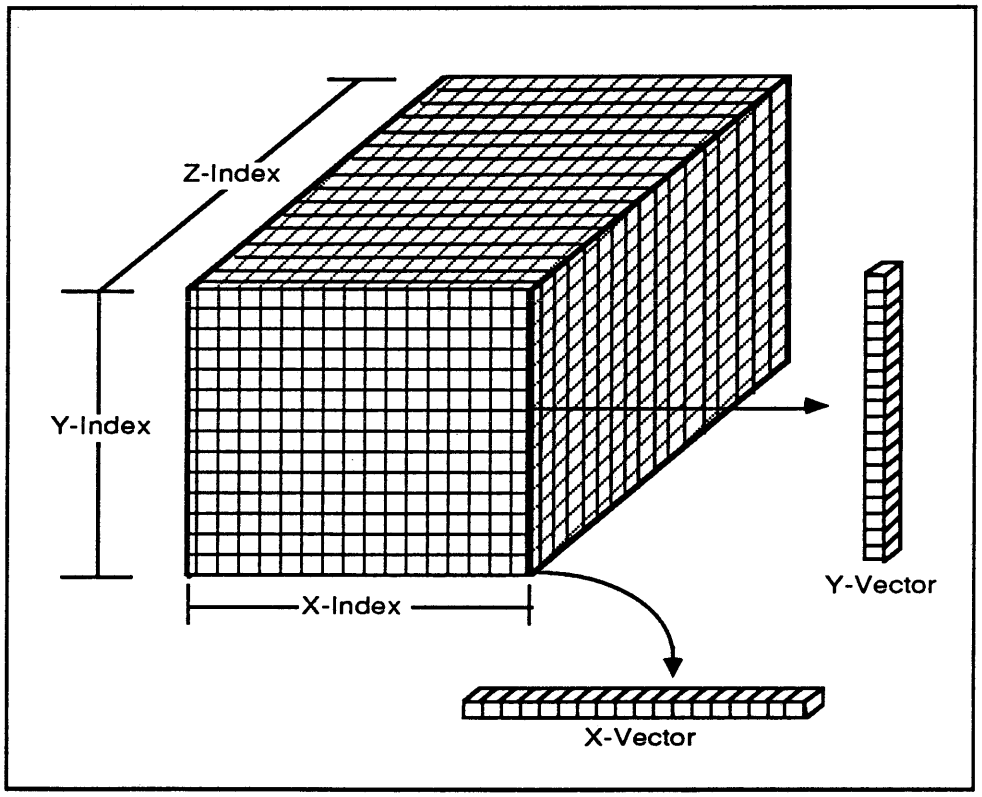
Dimensionality is integrated into Boris' vector sort through the maximum index values of the MLG's parameter arrays. Boris' vector sort decomposes the MLG's parameter arrays into a collection of vectors (see Figure 1.6). The number of vectors in each collection is determined by the MLG's maximum index values, where there are: NY*NZ X-vectors, NZ*NX Y-vectors, and NX*NY Z-vectors. The vectors are defined by freezing two of the indices of the MLG and varying the remaining one. For example, to form an X-vector, the Y- and Z-indices are held constant and X-index is varied from 1 to NX.

Figure 1.5: MLG Data Structure In Boris' Full Sort



Each of the defined vectors is sorted as a separate group in Boris' vector sort. Whenever two elements in a vector violate the monotonicity conditions the related cells are reordered. The entire collection of vectors is sorted repetitively, until no swaps are performed on any vector. This sort was used extensively in the research, therefore an example of the code is provided in Appendix A.

Figure 1.6: Vector Formation For Use In Boris' MLG Vector Sort

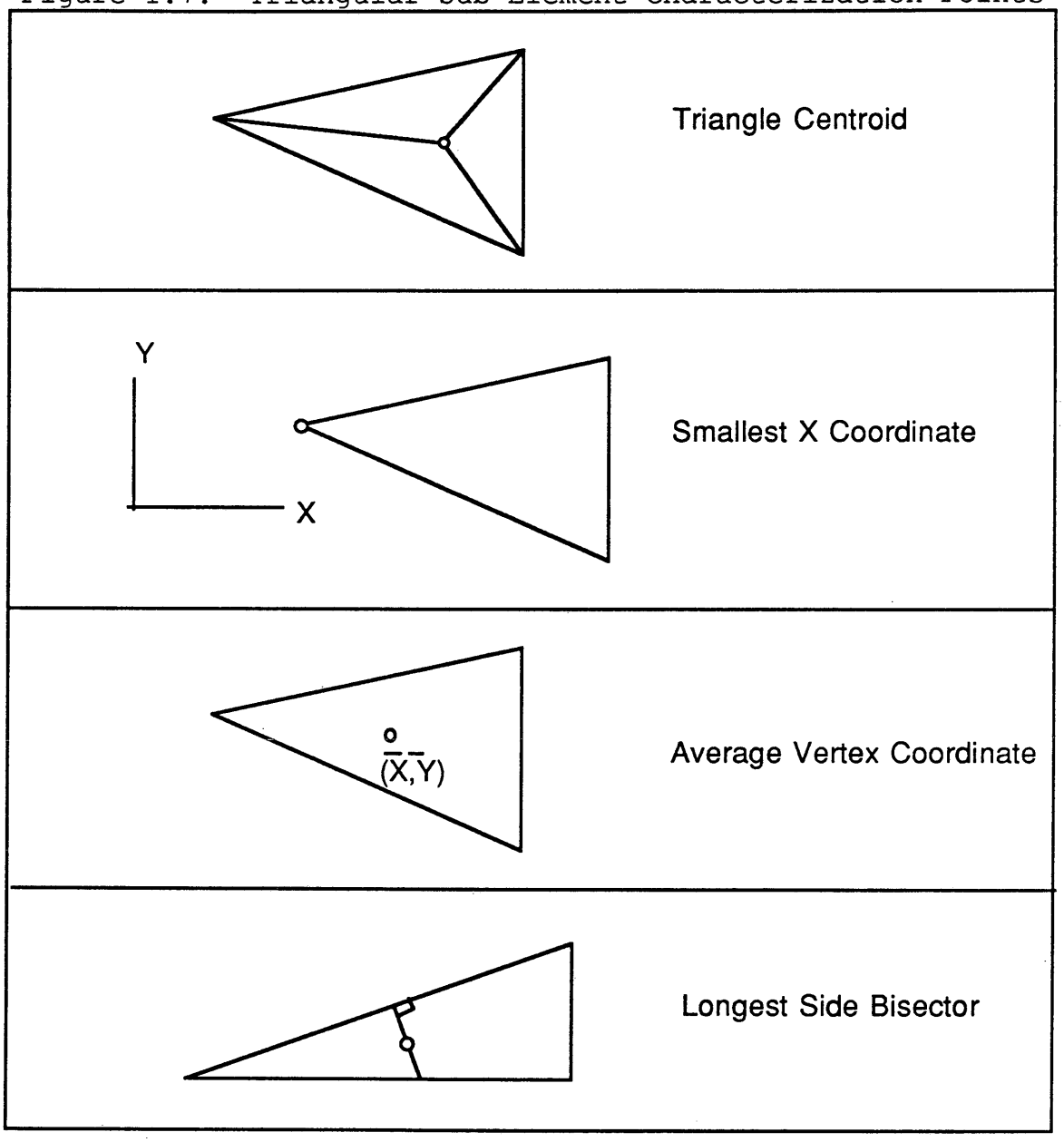


Spatial Representation

The MLG data structure has been extended to represent spatial regions as well as particles in space. M. Kelly extended the MLG data structure to represent two-dimensional triangular meshed regions [Kelly 1986]. His work investigated the efficiency of using an MLG in accessing information relevant to simulating spatial regions, including: triangles adjacent to a vertex, and triangles surrounding a triangle. Much of Kelly's work provided a basis for the work performed in this research.

Kelly developed the concept of a characterization point to represent the triangular sub-elements which fill the spatial regions [Kelly 1986]. He defined the characterization point as any single point in or around the sub-element which adequately represents the sub-element's location in the simulated space. Numerous points exist which can be selected to act as a characterization point, Kelly investigated four possible characterization points for triangular sub-elements (see Figure 1.7): the triangle centroid, the triangle vertex with the smallest X-coordinate, the average vertex coordinates, and the midpoint of the longest triangle side bisector. Kelly's research

Figure 1.7: Triangular Sub-Element Characterization Points



showed that the use of the triangle centroid, and the average vertex coordinates as characterization points provided superior performance in comparison to the others.

Kelly showed that the MLG can be used to maintain the spatial relationship among the triangles by retaining the triangle's characterization points in monotonic logical order. Therefore, triangles that neighbor each other in the spatial region are assigned locations which neighbor each other in the MLG data structure. The concept of a characterization point can be extended to include three-dimensional sub-elements, such as tetrahedra. Based on the spatial relationships of characterization points the sub-elements are mapped into the MLG data structure. Since the monotonicity conditions guarantee that the sub-elements near one another in the simulation space are near one another in the data structure, identification of neighboring sub-elements is determined in the same manner as for nearest neighbor particles. In addition to its utility in defining neighboring sub-elements the MLG can also be used to partition the sub-elements, thus allow updating to be performed concurrently.

This paper expands upon algorithms presented in this chapter. The expansion of these algorithms will help simulators take advantage of multi-processing architectures.

The use of Boris' MLG data structure for data partitioning is discussed in Chapter II. Partitioning of spatial regions, which was initially investigated by Kelly, is expanded to three-dimensions in Chapter III. These algorithms support the development of data partitioning in particle dynamics and spatial representation models. The partitionable MLG data structure relates directly to data partitioning, and the mesh algorithms relate indirectly. The mesh algorithms are needed by spatial representation models irrespective of the use of data partitioning.

CHAPTER II

A PARTITIONABLE THREE-DIMENSIONAL MONOTONIC LOGICAL GRID DATA STRUCTURE

Introduction

A partitionable three-dimensional data structure has been developed and implemented for testing in a Pascal program. The program was used to evaluate data partitioning in a particle dynamics framework. Original intentions included evaluation of data partitioning in a spatial representation framework, but due to time limitations, integration of the mesh generation and reconfiguration algorithms (presented in Chapter III) with the partitionable data structure was not completed. Even though planned goals were not achieved, numerous tests of the partitionable MLG have been performed using the particle dynamics framework. Derived conclusions relating to the partitionable MLG should not change as a result of the type of problem being simulated.

The Pascal program was developed on an ELXSI 6400 mini-supercomputer. The ELXSI was selected as the program host due to its accessibility and its multi-processor architecture. The ELXSI utilized for this research possessed

three Central Processing Units (CPUs), thus limiting the investigation of data partitioning to a maximum of three data partitions. Further partitioning of the data, and thus creation of more processes than processors, would result in contention for processors.

The existence of excess processes requires swapping of processes on and off the available processors. The swapping of processors requires additional activity by the ELXSI process scheduler, thus adding the ELXSI controller process to those already on the system. Processor contention for the available processors reduces the overall efficiency of the program thus increasing execution time. Even though the analysis performed only made use of three separate processors, it is believed there exists no factors which would imply the results of this study would not scale with the number of processors on the host computer.

The Pascal program developed consists of three major segments: the particle dynamics model (or particle mover), the MLG data structure and maintenance routines, and the multi-processing configuration. A thorough discussion of each of these segments is provided in the following sections of this chapter. The final section provides the performance evaluation results of the partitionable MLG data structure.

Particle Dynamics Model

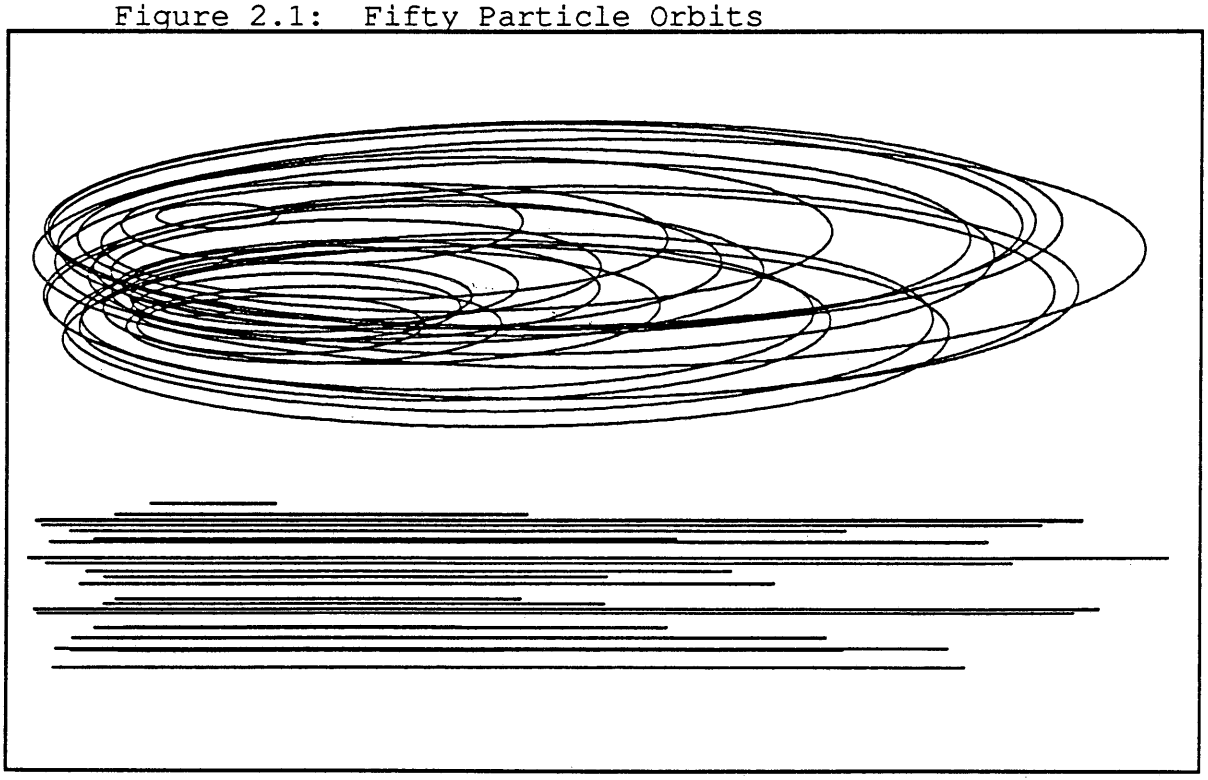
The program written to evaluate multi-processing with the MLG required a particle problem to simulate. Therefore, a relatively simplistic problem was designed with two basic requirements: enable particle updates to be calculated without requiring data on neighboring particles, and prevent particle paths from crossing one another. These simplifying design requirements were defined in order to make the particle updates as simple as possible.

The use of a more realistic or complex particle mover will have no effect on the results established by this study. A more complex particle mover increases the models total execution time but does not affect the time required to maintain the MLG. Since the MLG maintenance time remains constant and total execution time increases the percentage of total execution time consumed in performing MLG maintenance would decrease. The decreased percentage overhead required for MLG maintenance will result in improved performance.

The particle mover developed and implemented maintains the particles in a set of co-planar orbits about the Z-axis. The Z-axis defines a continuum of possible foci for the particle's orbits. Eccentricity defines whether or not the orbit is elliptical, circular, parabolic, or hyperbolic. In

this study all test runs were performed using elliptical orbits, and for any specific case the eccentricity of all orbits was the same. Elliptical orbits possess two focuses (or foci), which are defined as the two points where the sum of the distances between the two points and every point on the ellipse remains constant. The particle generator was designed to place one of these foci along the focal axis. Figure 2.1 provides two views of fifty elliptic orbits.

Figure 2.1: Fifty Particle Orbits



A particle's orbit is determined in the following manner: randomly generate the particle's initial position, project the initial position onto the Z-axis, calculate the particle's true anomaly, and finally calculate the particle's initial velocity. A particle's initial position, initial velocity, and elapsed time are the only parameters required to determine a particle's complete orbit.

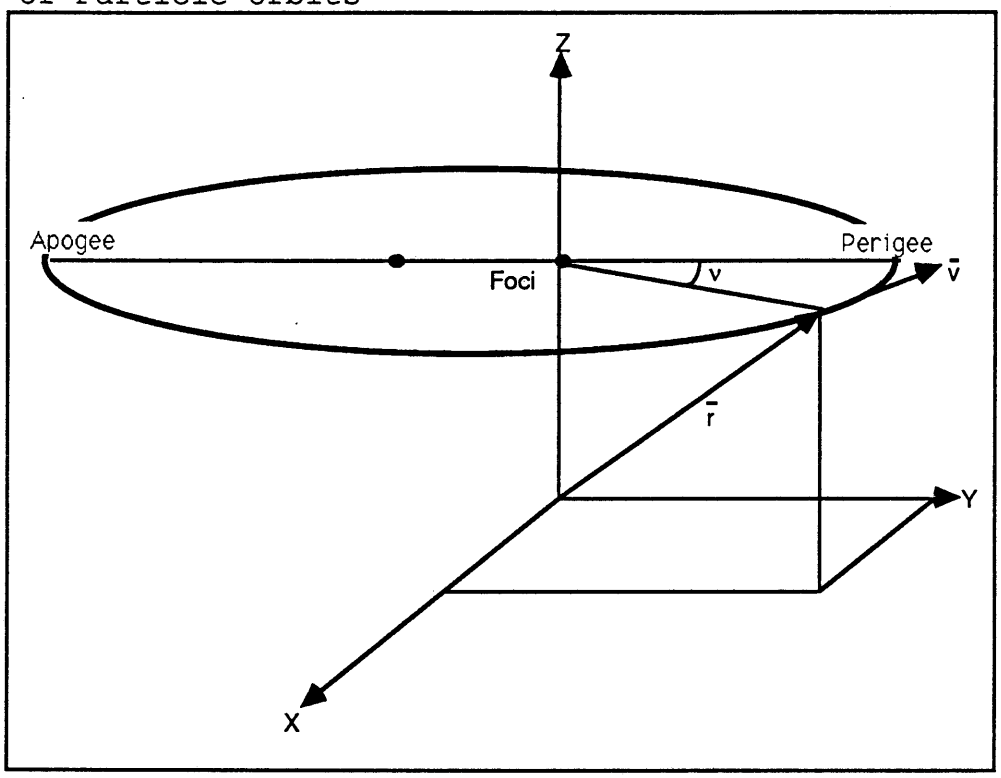
Each particle's initial position is randomly generated by the particle generator. The user is required to define a three-dimensional region which must contain the initial position of all particles. Each particle's initial position is determined based on random values generated from the standard uniform distribution. For each particle, three random values are drawn, one for each axis. The actual components of a particle's initial position are determined by scaling the random values between the boundary limits of the appropriate axis.

The elliptical orbit's focus is the projection of the particle's initial position onto the Z-axis. The Z-axis is coincident with the focal-axis. The orbit's focus is then used, along with the initial position, to define the true anomaly of the initial position. The true anomaly is defined as the angle between a point in the orbit and the orbits perigee (the orbits minimum range from the primary focus).

The true anomaly is derived based on the vector dot product rule, $\underline{a} \cdot \underline{b} = abcos\theta$, where $\underline{a}$ and $\underline{b}$ are vectors and θ is the smallest angle between the vectors.

The initial position, eccentricity, and true anomaly are required to determine the particle's initial velocity. The initial velocity is defined in a manner which places the plane of its orbit parallel to the XY-plane. The major axis of the particle's orbit is defined to lie parallel with the Y-axis and perpendicular to the Z-axis. Figure 2.2 displays the physical relationships among the parameters used in defining a particle's orbit.

Figure 2.2: Parameters Used In Determination Of Particle Orbits



The representation of particle dynamics using orbital calculations enables the use of well developed algorithms to calculate state vector updates. The universal variable formulation of Gauss' problem was used to determine a particle's position. Gauss' problem can be stated as: given an initial position, an initial velocity, and a time of flight calculate the current position and the current velocity. Several sources provide descriptions of this problem and its solution [Bate, Mueller and White 1971].

A Three-Dimensional Monotonic Logical Grid Data Structure

The MLG data structure as defined in Chapter I assumes a particular design implementation. Advantages and disadvantages of this design and two alternate designs are discussed in this section, in the hopes of preventing the inefficient implementation of the MLG.

The design, presented by Boris, assumed the existence of three three-dimensional real arrays, X, Y and Z. In FORTRAN these arrays would be defined as follows:

```
DIMENSION X(MAXXDIM,MAXYDIM,MAXZDIM)
DIMENSION Y(MAXXDIM,MAXYDIM,MAXZDIM)
DIMENSION Z(MAXXDIM,MAXYDIM,MAXZDIM)
```

where the X-, Y- and Z-arrays contain the X-, Y-, and Z-position components of the particles. The structure is rather simple but it is also inefficient. It requires each of the three separate component arrays to be reordered whenever any two particles are not in monotone order. Two out of order cells in the X-array require reordering of the X-, Y- and Z-arrays, thus requiring three separate swaps to be performed. If additional parameters, such as particle velocities and accelerations, were stored in similar arrays, they also would need to be reordered. Although a few swaps

per reordering may seem trivial, performing millions of particle swaps in this manner is considerably inefficient.

A modification of the design eliminates the need to reorder multiple arrays. Let the MLG data structure be defined as follows (also in FORTRAN):

```
DIMENSION MLG(MAXXDIM,MAXYDIM,MAXZDIM)
```

where MLG is an array of integer values which relate to particle numbers. The values in the MLG array map into the particle array, defined as follows:

```
DIMENSION PART(MAXPART,MAXPARTPARAM)
```

where the first index identifies the particle's number, and the second index is used to define particle parameters. This design is a simpler and more efficient design for sorting two cells which are out of monotone order. This design requires only the two cells of the MLG-array to be swapped when particles are out of order. In this example, the parameter defining the maximum index value, MAXPARTPARAM, determines the maximum number of parameters per particle.

Assuming MAXPARTPARAM to have a value of 12, a representative set of parameters would be:

Locations 1-3	Initial Position
Locations 4-6	Initial Velocity
Locations 7-9	Current Position
Locations 10-12	Current Velocity.

This design simplifies the task of reordering particles in the MLG, but it still has one minor deficiency. The handling of particle parameters via the second index of the PART-array lacks clarity and flexibility. To reference a particle parameter the developer must know which value of the PART-array's second index identifies the parameter. For the programmer who developed the original code, remembering the appropriate index values and the parameters they define is not difficult, but to programmers unfamiliar with the code, potential for mis-referencing parameters increases. The increased potential for mistakes will reduce reliability in the model. A more troublesome problem arises when additional parameters are introduced into the PART-array. If the new parameters are assigned index values which place them somewhere other than at the end of the current parameters, a thorough check of the code must be performed to insure that all parameter references are updated to reflect the shift in index value assignments. Again, this problem may seem

trivial, but failure to consider it can result in time consuming and costly maintenance.

The design of the MLG data structure used in this research has taken advantage of capabilities provided by the Pascal programming language. The MLG data structure has been defined as follows:

```
MLG:
   array[1..MaxXDim,1..MaxYDim,1..MaxZDim] of
   ParticlePointer;
```

where MaxXDim, MaxYDim, and MaxZDim are the maximum index values of the MLG-array, and ParticlePointer is an integer value which indexes into the particle array. The particle array is not an array of real variables but an array of data records, defined as follows:

```
Vector = array[1..3] of real;

ParticleRecord =
record
   InitialPosition   : Vector;
   InitialVelocity   : Vector;
   CurrentPosition : Vector;
   CurrentVelocity : Vector;
end;

ParticleArray :
 array[1..MaxParticles] of ParticleRecord;
```

This design is similar to the previous design with the exception that the PART-array discussed above is replaced by the ParticleArray. Therefore, whenever two cells of the MLG are out of monotone order only the MLG-array must be reordered. The ParticleArray defined above provides clear definition of particle parameters by having a unique name for each parameter even though a parameter may be non-scalar.

Pascal provides the capability of defining data types. Data types enable the developer to create non-scalar variable types, used in the same manner as the familiar integer and real scalar types. The vector data type used in the ParticleRecord below is a user defined data type, which is defined as a three-dimensional real array. The vector data type would be used to represent parameters such as position, velocity and acceleration, which require direction and magnitude.

The ability to define complex data types enables a library of standard functions which operate on the data type to be created. To achieve the same capability, using the PART-array described above, is not trivial. Vector parameters which are required for input must be extracted from the PART-array, and vector parameters which are output must be replaced in the PART-array, every time a vector routine is called. The creation of a vector library provides

three benefits: it enables the developer to deal with calculations at a higher level of abstraction; it reduces the potential for encoding errors; and it reduces the amount of testing which must be performed, because there is no need to check the computation every time a vector operation is performed.

Each of the MLG designs discussed provide the capabilities needed to support data partitioning. Therefore, the remainder of this section will discuss data partitioning with respect to the final design, which was used in the Pascal implementation.

Partitioning of the simulated particles is determined by the indices of the MLG-array. A partitioning index is selected and contiguous values of the index are allocated to each subset. Therefore, particles are assigned to a subset based on their partition index value. The subsets of particles are then allocated to separate processors where the processors are updated.

Frame 1 of Figure 2.3 provides an example of how the MLG preserves the spatial relationship among particles in a three-dimensional simulation space. Frame 1 shows a projection, onto the XZ-plane, of the positions of 1000 particles immediately after generation (time zero) by the particle generator. The different symbols represent the

different X-index values assigned to the particles. The random spread of symbols throughout the plane result from the particle generator's random placement of particles into the MLG-array.

Frame 2 of Figure 2.3 shows the same set of particles after they have been sorted into monotonic logical order, using Boris' full sort. In this frame, like symbols form a set of ten columns. The formation of columns indicates the spatial preservation characteristic of the MLG. Once reordered to meet monotonicity conditions the particles which are near one another in the simulation space are near one another in the MLG data structure, which implies they have small deviations in index values. Figure 2.4 is the same as Figure 2.3 except it only shows those particles which have X-index values equal to 1 (dots) or 10 (asterisks), which makes it is easy to see the spatial preservation provided by the MLG's monotonicity conditions.

Figure 2.3: Projection Of 1000 Particles Onto XZ-Plane (Time=0.0)

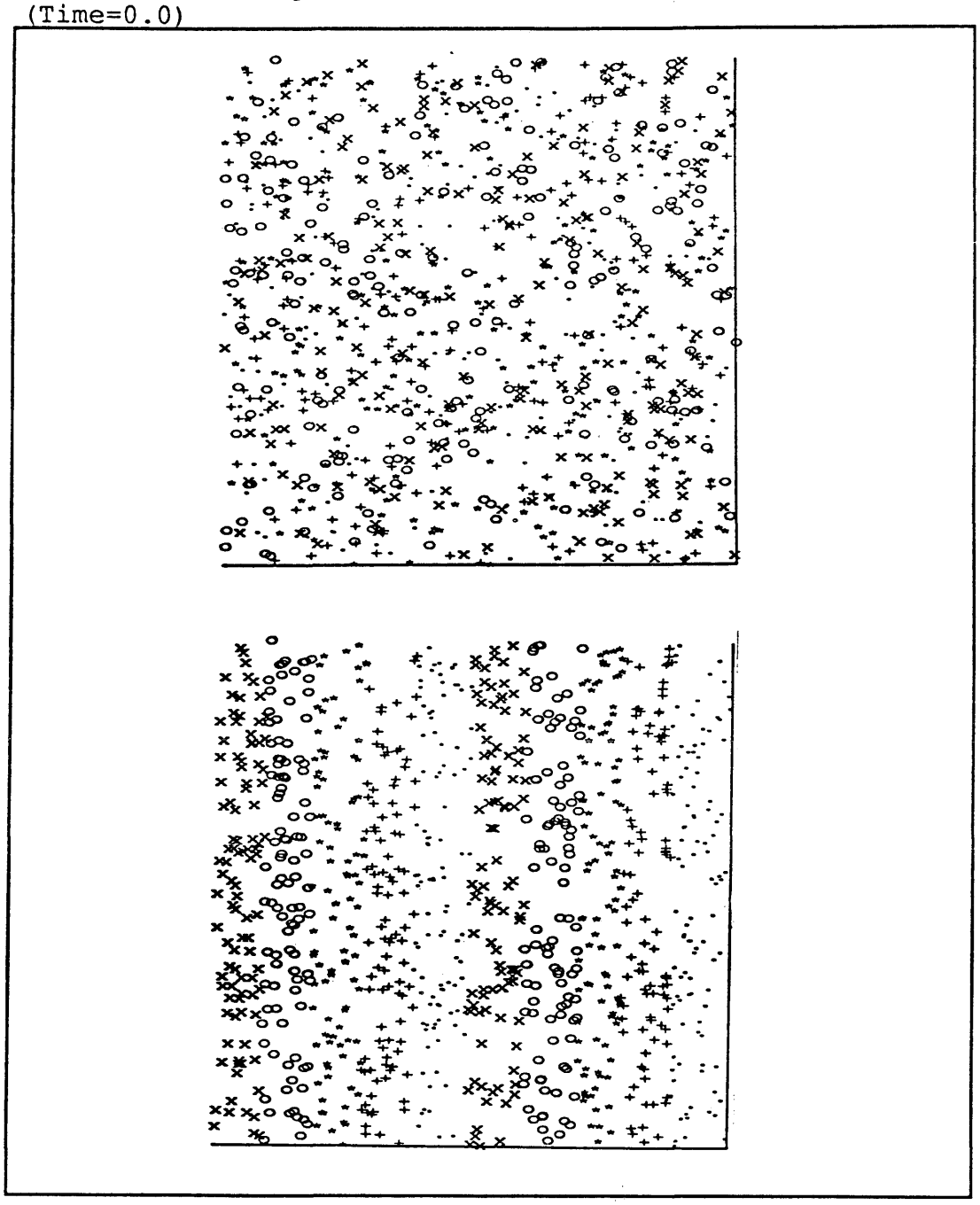
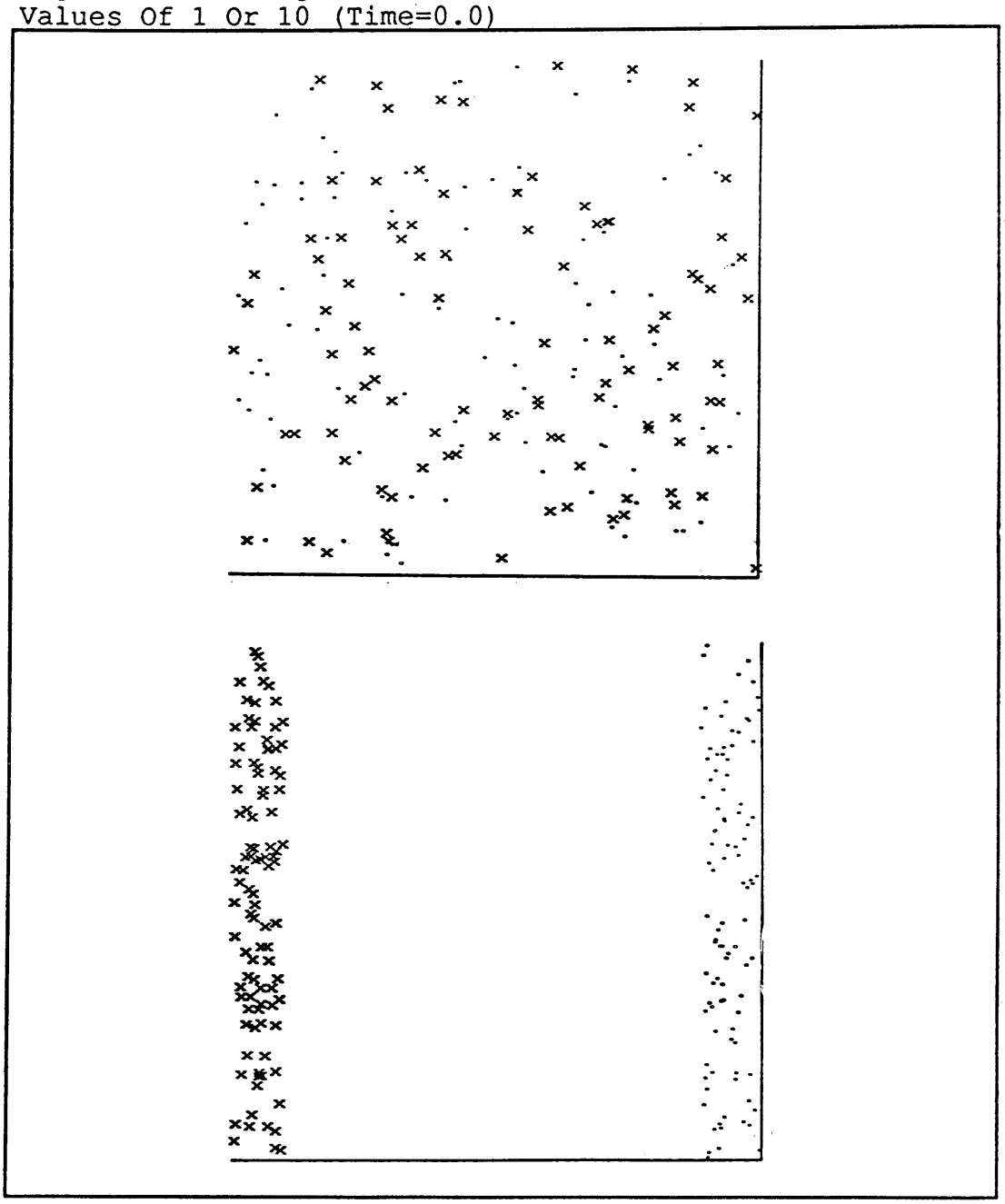


Figure 2.4: Projection Of Particles With X-Index Values Of 1 Or 10 (Time=0.0)



Frame 1 of Figure 2.5 shows the same 1000 particles after an elapsed time of 0.5 seconds, but before the particles have been sorted. As a result of the particles differing orbits, some have moved beyond others which they were previously behind. Frame 2 shows the particles after they have been sorted using Boris' vector sort. By careful comparison of Frames 1 and 2 some symbol changes can be detected, indicating changes in the X-index values of several particles.

Figure 2.6 provides a clearer picture of the reordering performed. It is identical to Figure 2.5 except for the removal of particles assigned X-index values between 2 and 9. The reordering of particles in this figure is detected by relating the spread of like symbols between the two frames. The symbols represent the particles with X-index values equal to 1 (dots) or 10 (asterisks). In the first frame the particles have not been sorted since their position update, in the second frame the particles have been sorted using Boris' vector sort. When compared, a narrowing in the spread of like symbols can be detected. This narrowing is a result of particles having their X-index values changed. The change

Figure 2.5: Projection Of 1000 Particles Onto XZ-Plane (Time=0.5)

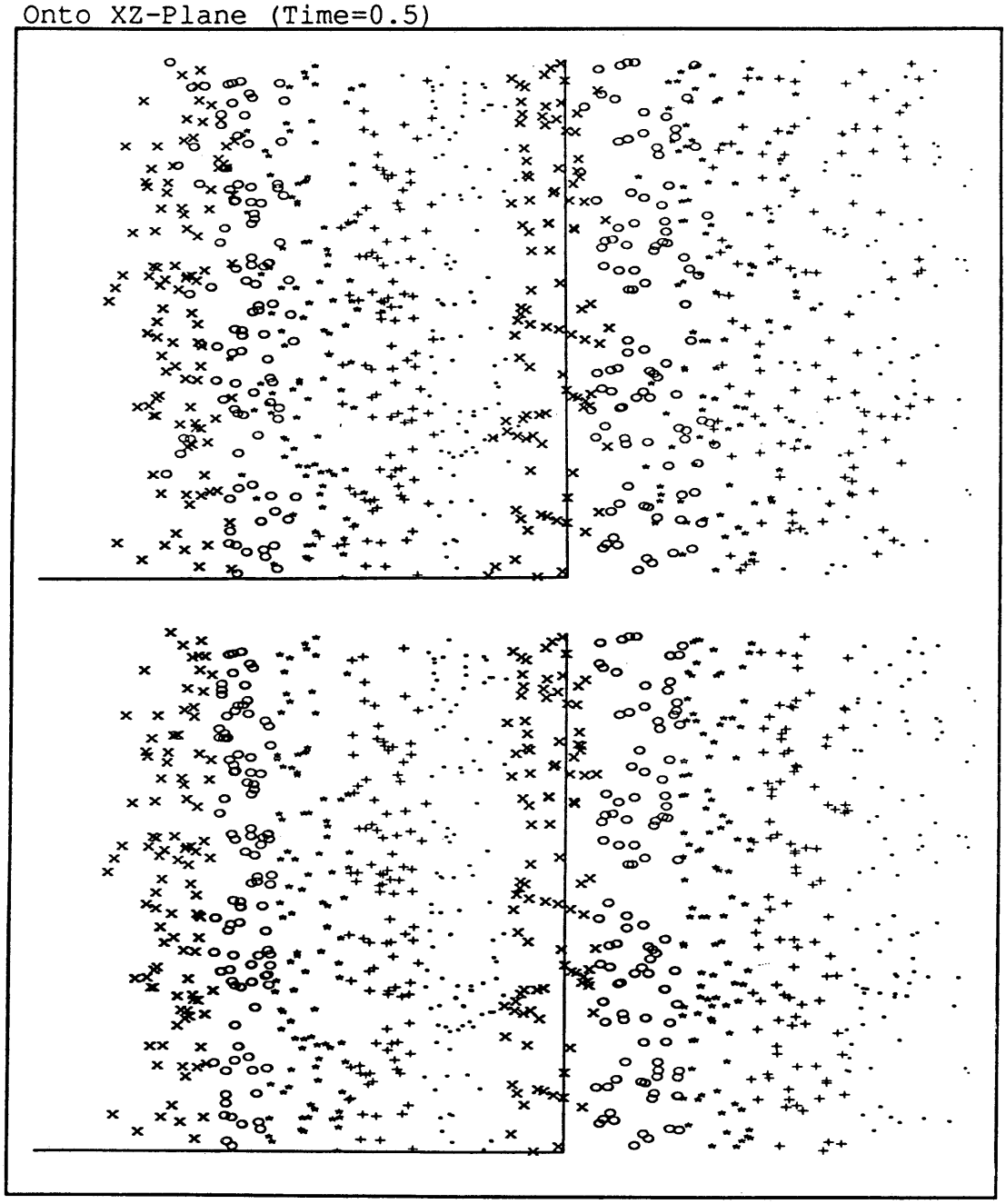
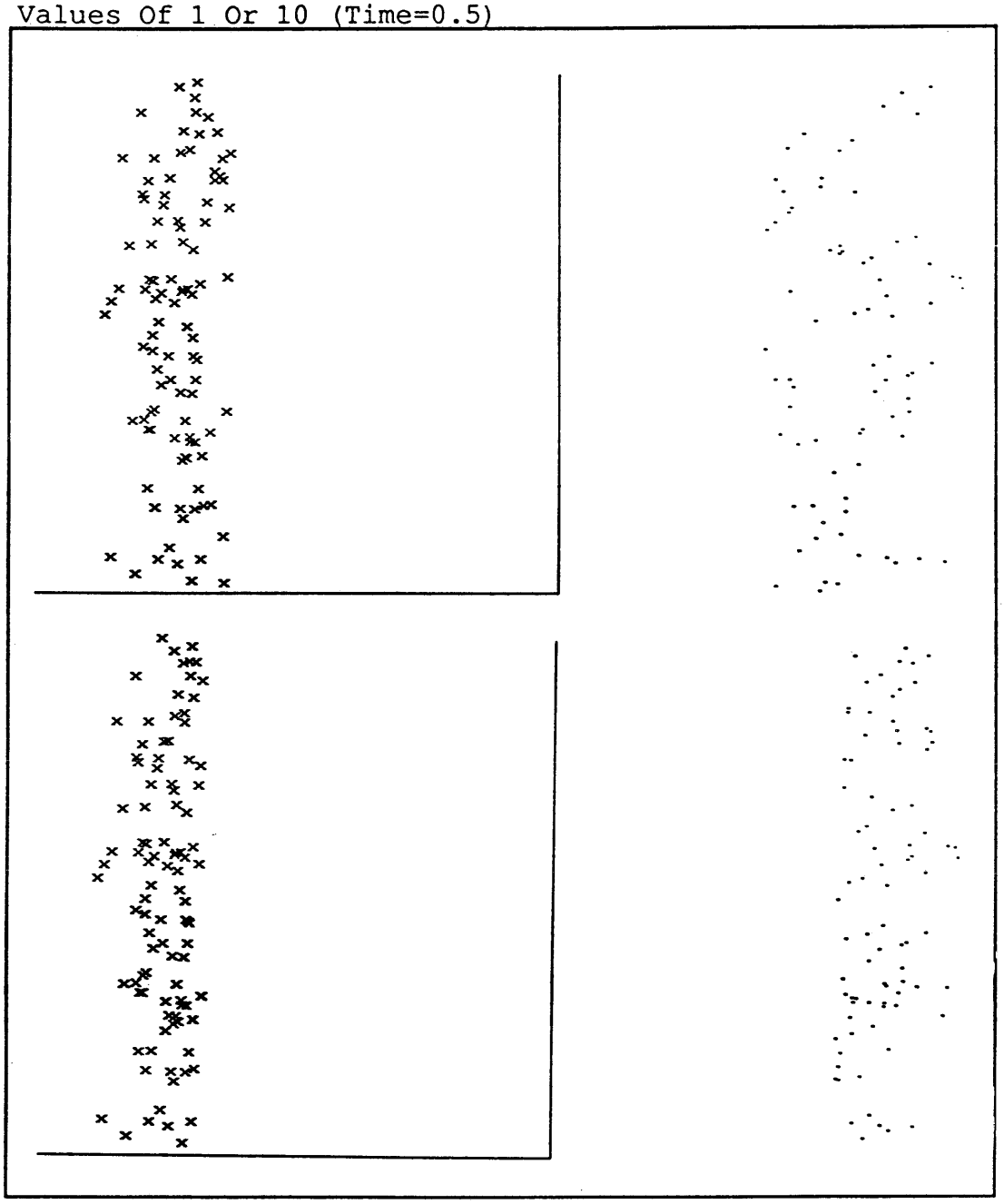


Figure 2.6: Projection Of Particles With X-Index Values Of 1 Or 10 (Time=0.5)



in particle X-index values results from changes in particle relationships in the simulated space.

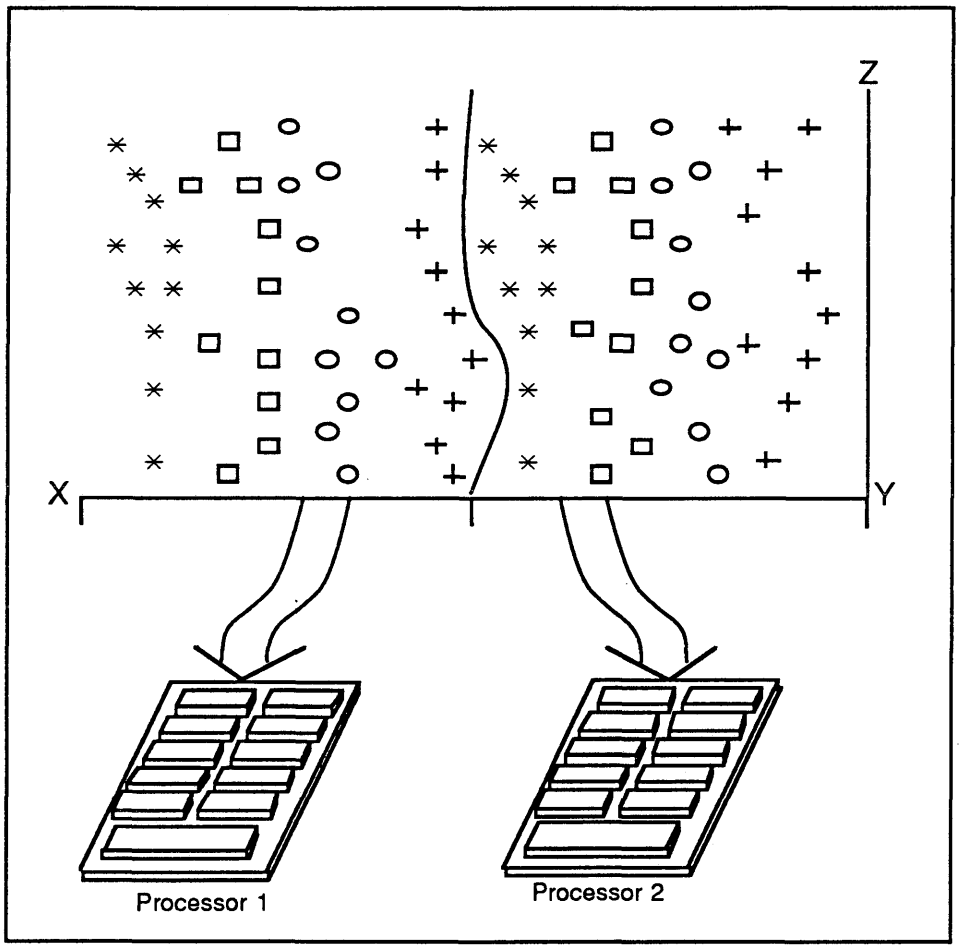
The columns of like symbols in the figures discussed above are referred to as quasi-planes (quasi since the planes have width). The like symbols represent particles with the same X-index values. The location of the symbols is determined by a parallel projection of the position of the particles onto the XZ-plane. The projection eliminates all information relating to the positions' Y-component, but it does provide information on the range of X- and Z-component values. A column of like symbols indicates the position of these symbols range over an interval in the X-dimension and Z-dimension of the simulation space. The same holds for the Y-dimension. Since like symbols range over intervals along each of the simulation space's dimensions a volume is generated.

Quasi-planes are created by holding constant one of the MLG-arrays index values and varying the remaining two. Thus, quasi-planes can be defined based on any desired index. In the figures discussed above the quasi-planes are generated by holding constant the X-index value.

Data partitioning is achieved by assigning consecutive quasi-planes to a partition set. The partition set is then passed to a processor where updating of particles in the

quasi-planes is performed. Figure 2.7 depicts this process of assigning sets of quasi-planes to separate processors. The creation and controlling of the processes which update the particles is discussed in the following section.

Figure 2.7: Assignment of Data Partitions To Separate Processors



Multi-Process Configuration

Utilizing the ELXSI's multi-processor architecture required definition of a multi-process configuration. Considerations relating to the definition of the multi-process configuration include: what functions need to be performed and which are to be concurrently processed, how will the concurrent processing be controlled, and how will the large amounts of data be made available to the separate processes.

Two basic functions which needed to be performed were identified: initialization and control, and MLG updating. The initialization and control function was implemented in a single parent process, since it needed to be performed only once. The MLG updating function was implemented in a child process, which could be spawned, or instantiated, multiple times. The parent process performs the functions of initializing data, spawning child processes, and controlling the processing of the child processes. The child processes perform particle position updates and MLG sorting functions.

Control of child processing was performed through ELXSI's inter-process message system. The parent and child programs create a set of inter-process message links, using ELXSI utility procedures, which enable the parent process to

send processing initiation messages to the child processes, and the child processes to send processing complete messages to the parent. Through inter-process communications the parent maintains control over the child-processes.

The final consideration in defining the multi-process configuration, was how to provide the child processes access to the particle data. The ELXSI provides the capability to share data among multiple processes. Processes are allowed to offer some or all of their global memory to be accessed by other processes. The processes desiring access to another process' data must receive a memory link from the offering process and then use the memory link to bid on the offered memory. Through shared memory the child processes access the parent processes global memory where all particle data is stored. The child processes require access to the data on the particles in order to update positions and sort the particles.

The research performed on the MLG used a parent multi-child configuration, where the child processes were permitted by the parent to access a portion of the parent's global memory (see Figure 2.8). The child processes are separate (but identical) programs which are different from the parent. The child processes are initiated by the parent through a call to an ELXSI intrinsic procedure. The ELXSI intrinsic

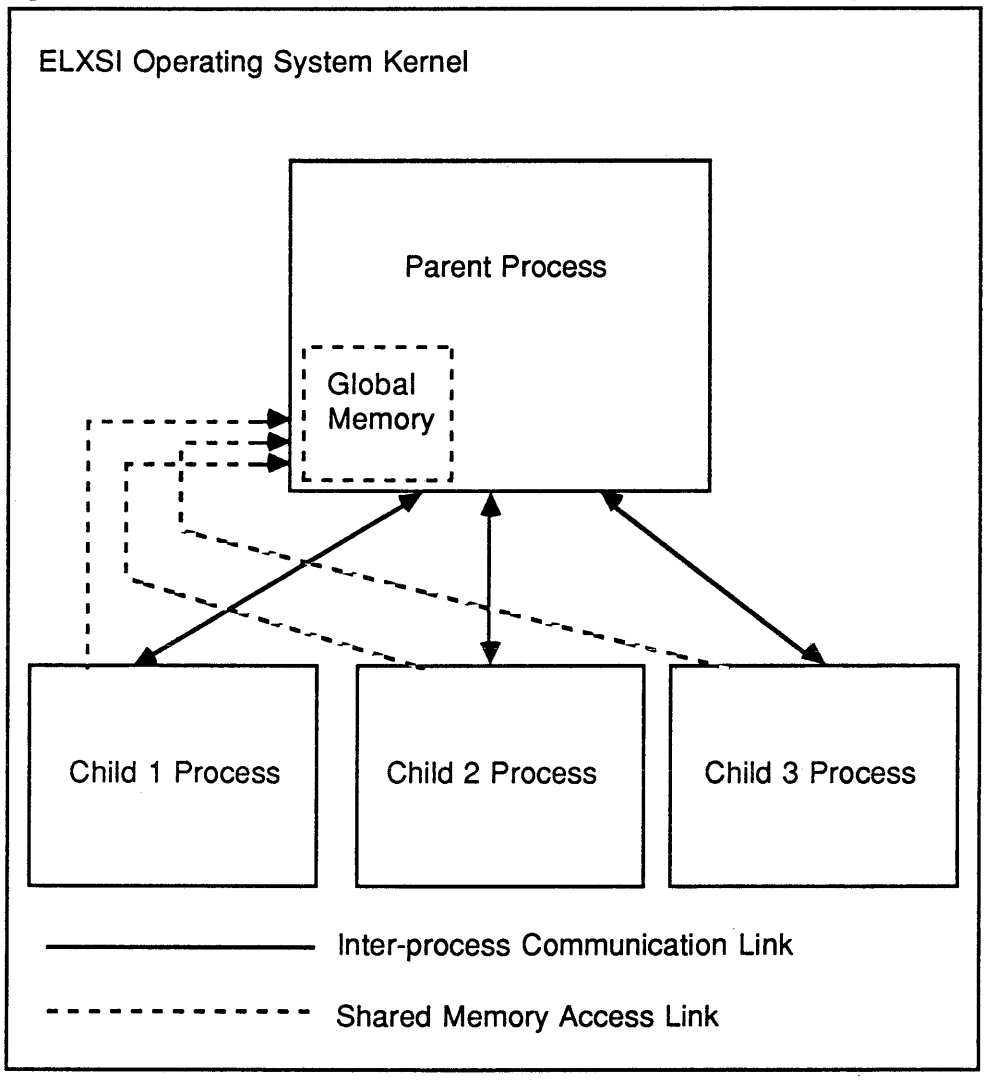
invokes the ELXSI operating system kernel, which then spawns (initiates) the child processes. After the kernel spawns the child processes the parent resumes processing. While spawning of the child processes is being performed by the kernel the parent process is in a quiescent state.

After the child processes are spawned, the parent creates a funnel (a message receipt queue) into itself and creates a communications link into the funnel. The parent then passes the communications link to its child processes, through a default communications link created by the kernel during spawning of the child processes. The link passed to the child processes enables them to pass messages to the parent. As soon as a child process receives the link a message is sent to the parent confirming receipt of the link.

After all child processes have confirmed receipt of their parent link, the parent offers a portion of its global memory for shared access. In order to offer memory for shared access the parent must create a link into the memory and pass the link to its child processes. The child processes then use the memory link to identify to the kernel that they have access to the specified area of the parent's global memory. This activity is called bidding for shared memory. The MLG array and the particle array are the

portions of the parent's global memory which are shared with the child processes.

Figure 2.8: Multi-Process Parent-Child Configuration



The communications links between the parent and the child processes are used to control processing. Through inter-process communications messages the parent notifies its child processes of which sections of the MLG they are responsible for updating, the current simulation time, and when to stop all processing (or to die). The inter-process communication links enable the child processes to notify the parent when they have completed processing for the current iteration.

Through the inter-process communications links, processing is controlled by the parent process in the following manner. After all child processes are spawned, inter-process communications links are established and confirmed, and memory is shared, the parent passes each child process a process initiation message. The message identifies to each child process, the current simulation time and the indexes of the MLG for which it is responsible. For those particles in its assigned portion of the MLG, the child process updates positions and velocities based on the current simulation time (using the universal variable formulation to solve Gauss' problem). After updating the position of its assigned particles the child process sorts them into monotonic logical order. After the child process has

completed updating and sorting its assigned portion of the MLG, a processing complete message is sent to the parent.

During child processing the parent is in a quiescent state. After the parent sent the processing initiation messages to its child processes its processing halts. The quiescent state is initiated by a call to an ELXSI utility, the receive message utility. The receive message utility causes the kernel to stop any further processing, for the process making the call, until a message arrives in the receive message queue. This allows the processor on which the process was active to be utilized by other processes. The parent remains in its quiescent state until a message arrives in its receive message queue. The processing complete messages sent by the child processes resume parent processing. When the parent receives a processing complete message the parent records the completion of the child's processing, based on a child identification number on the message record, and determines whether or not all child processes have completed their processing. If any child process has not completed its processing the parent returns to its quiescent state.

Once all child processes have completed their processing and notified the parent a final sort of the MLG is performed. The parent process performs the final sort to reorder any

particles which are not in monotonic logical order along the boundaries between the MLG's partitions. After the parent completes the final sort of the MLG the next iteration is begun, starting with the updating of the simulation time, and followed by the parent notifying each of its child processes to start processing for the new simulation time.

Termination of child processes is controlled through the parent process. Once the simulation time has reached its final time step and the child processes have completed their processing the parent sends each child process a termination message. The parent then waits to be notified, through another ELXSI utility, that each of its child processes have been terminated. After all child processes have terminated and the parent has been notified the parent process terminates itself (i.e. stops execution).

Monotonic Logical Grid Performance Evaluation

Evaluation of the partitionable MLG data structure focused on establishing the time savings achievable through its use in data partitioning. Theoretically, if the particles can be divided into two mutually exclusive subsets and the subsets can be updated by two concurrently executing processors, the time required to update the particles can be completed in half the time required by a single process. If the particles can be divided into three mutually exclusive subsets and the subsets can be updated by three concurrently executing processors, the time required to update the particles can be completed in one-third the time required by a single process. As the number of particle subsets increases and if an equal number of processors are available the time required to update the particles decreases, with the reduction in execution time growing smaller and smaller. Theoretical performance through data partitioning is provided in Table 2.1. Column 1 is the number of data partitions generated. Column 2 is the theoretical execution times required for various numbers of partitions. Column 3 is the marginal return per additional processor. Marginal return is the reduction in execution time achieved by adding an additional data partition (or process). The reduction in

marginal return implies that beyond a relatively small number of partitions, addition of more partitions will provide little improvement. The reduction in savings does not imply pursuit of data partitioning is futile, it implies that the most significant improvement will be provided in the first few partitions, and thus computer architectures require only a few processors to make data partitioning worthwhile.

Table 2.1: Theoretical Performance Of Multi-Processing

(1) Number of Data Partitions	(2) Total Real Time	(3) Marginal Return Per Processor
1	T	
2	$\frac{1}{2}T$	$\frac{1}{2}T$
3	$\frac{1}{3}T$	$\frac{1}{6}T$
4	$\frac{1}{4}T$	$\frac{1}{12}T$
n	$\frac{1}{n}T$	$\frac{1}{n(n-1)}T$
∞	$\frac{1}{\infty}T=0$	$\frac{1}{\infty\,(\infty-1)}T=0$

Results indicate, in most real applications, the MLG can achieve near theoretical performance in relation to partitioning and updating particles. For most large-scale physical simulation models large numbers of entities are simulated over many iterations. Time considerations limited the tests reported in this section to cases of relatively small numbers of particles and small numbers of iterations, but tests indicate that an increase in either area results in improved performance by the partitionable MLG data structure.

Figure 2.9 provides a performance comparison of three different process configurations. For each configuration, three measures of real time (real time will be defined as elapsed wall clock time, in seconds) are provided. Measure 1 is total real time, defined as the time from parent initiation to parent termination. Measure 2 is total real time less initialization time, defined as the time required to initialize data and spawn the child processes. Measure 3 is total real time less initialization time and first iteration time. The initial MLG sort requires substantially more processing time than any of the subsequent iterations due to the random loading of particles (see Figure 2.10 which provides the percent of total real time required to perform each iteration). Measures 2 and 3 are presented to provide MLG performance exclusive of process overhead. Process

overhead is the time required to spawn child processes (which is machine dependent) and the time required to perform the initial MLG sort (which is dependent upon how the particles were loaded into the MLG).

Figure 2.9: Timing Comparison For Three Process Configurations

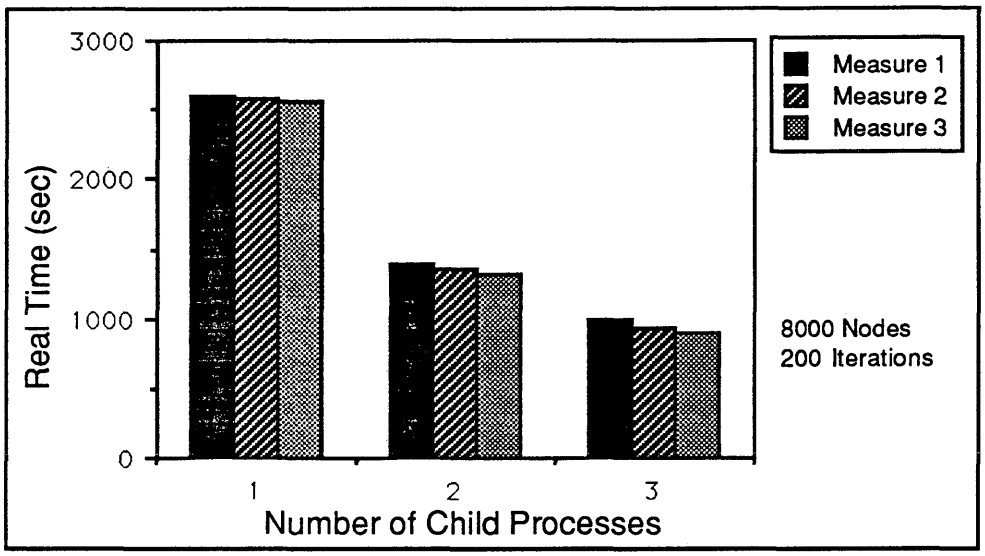
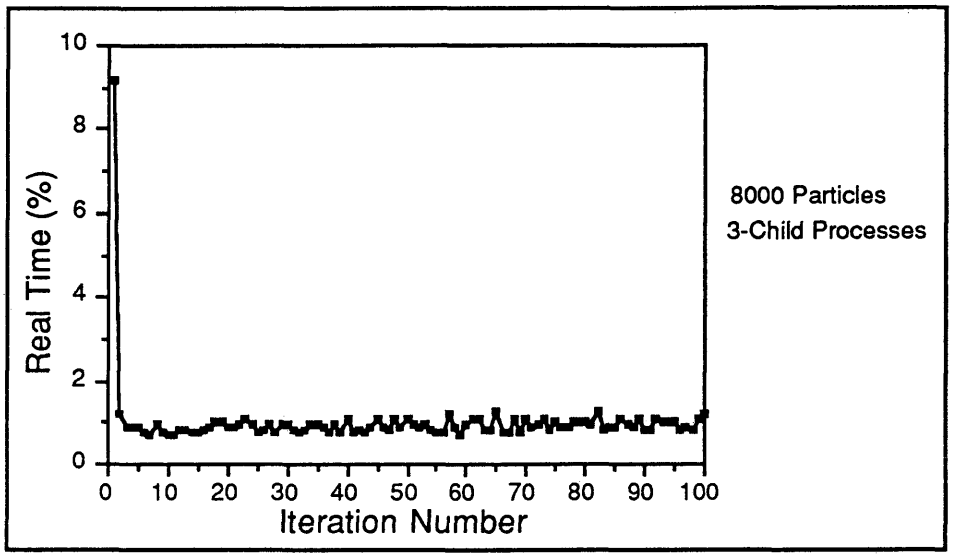


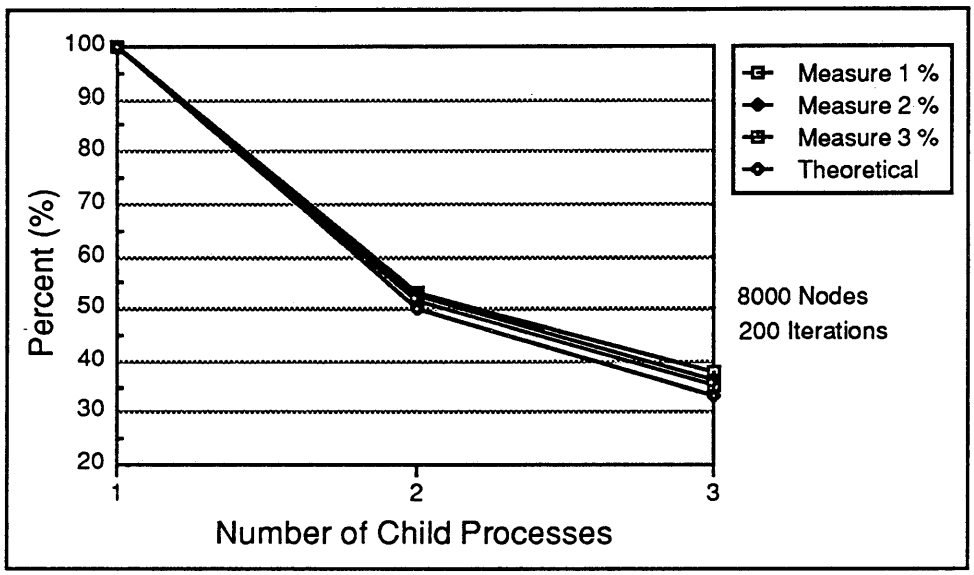
Figure 2.10: Percent Total Real Time Per Iteration



The execution times realized, as the number of data partitions (and number of child processes) increased from one, to two, to three, were near theoretical limits. Figure 2.11 provides a comparison of the realized reduction achieved by the two multi-process configurations, where realized reduction is the percent of the single-process configuration's execution time required by the multi-process configurations. The bottom curve is the theoretical limits achievable, the other three curves are the realized percent reductions for each of the three measures discussed above. The third measure, total real time less time required for initialization and iteration one, shows performance slightly above the theoretical limits. This slight amount of inefficiency is attributable to the overhead required by the

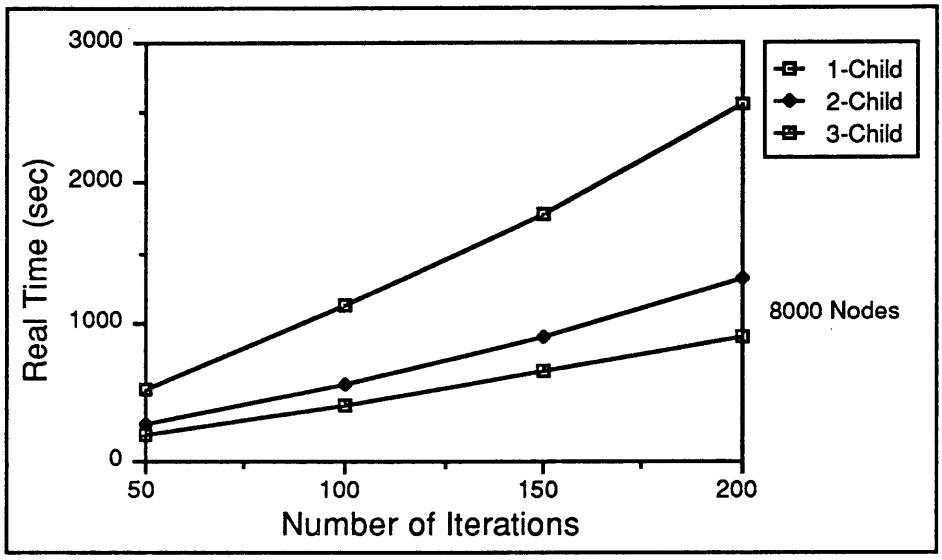
ELXSI to perform the multi-processing and also maintain the operating system kernel.

Figure 2.11: Realized Reduction For Multiple Child-Processes



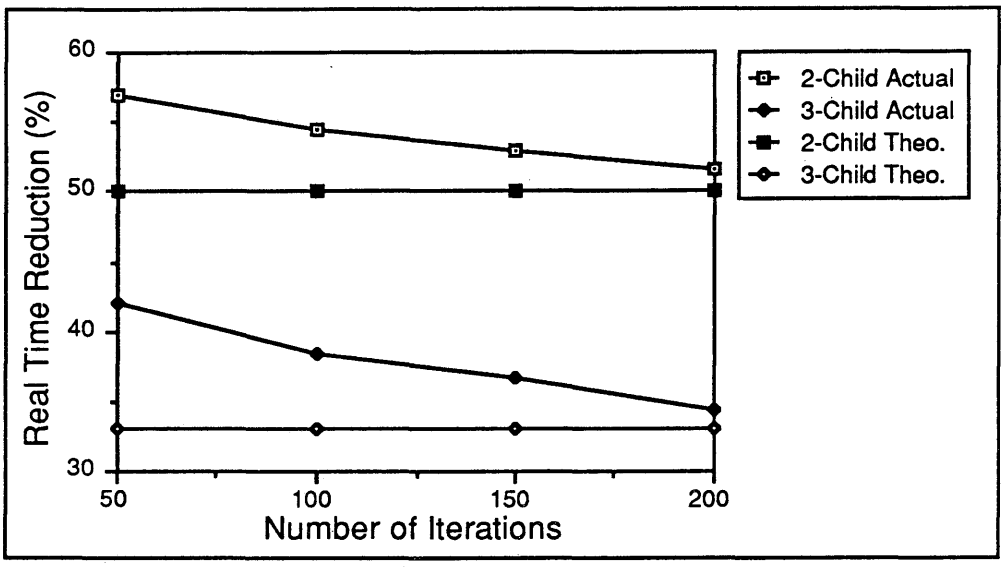
The second test performed established the effect of increased iterations on the MLG's performance. Figure 2.12 shows the execution time (measure 1) required by the 1-, 2-, and 3-child configurations, for 8000 particles simulated for varying numbers of iterations. Figure 2.13 shows the percent reduction, in terms of the 1-child configuration's execution time, for the 2- and 3-child configurations. The two straight lines in the figure are the theoretical limits, which are constant irrespective of the number of iterations. The other two curves are the realized percent reductions.

Figure 2.12: Execution Times For Varying Iteration Levels



The shape of these curves indicate, as the number of iterations increase the percent reduction in real time approaches the theoretical limit. Thus, an increase in the number of iterations will result in a reduction in required processing time.

Figure 2.13: Percent Reduction For Varying Iteration Levels



The third test performed established the effect of an increase in the number of particles on the MLG's performance. Figure 2.14 presents the execution time (measure 1) for the 1-, 2-, and 3-child configurations, for varying numbers of particles. Figure 2.16 presents the percent reduction in the 1-child configurations execution time for the 2- and 3-child configurations. Again the straight lines are the theoretical performance and the other curves are the realized percent reductions. The shape of the curves indicate that as the number of simulated particles increase the percent of single-process execution time decreases. Thus indicating, any increase in the number of particles simulated will increase the benefit of using multi-processing.

The two tests indicate that data partitioning through the MLG data structure will become more efficient with increases in the size of the problem. The primary reason for the improved efficiency is the decrease, as a percentage of total execution time, of initialization and the first iteration.

Figure 2.14: Execution Times For Varying Particle Counts

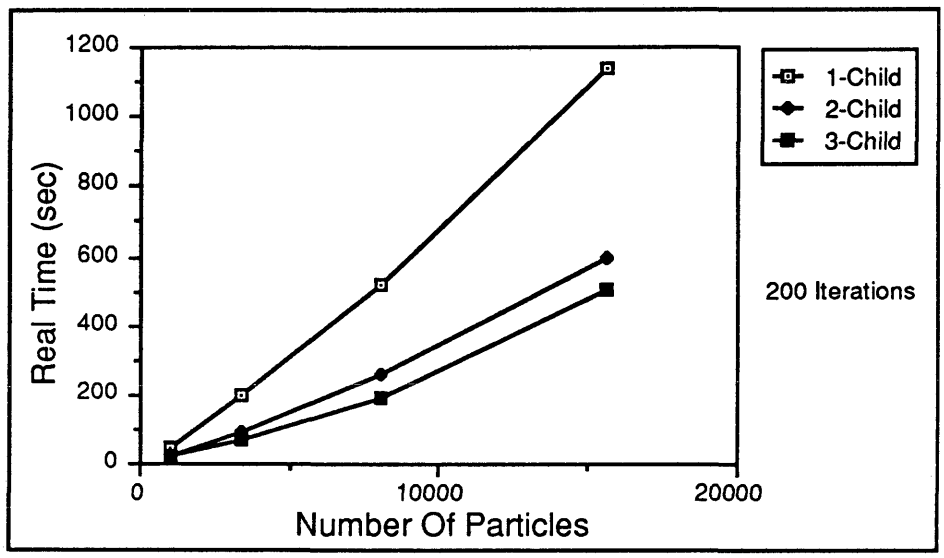
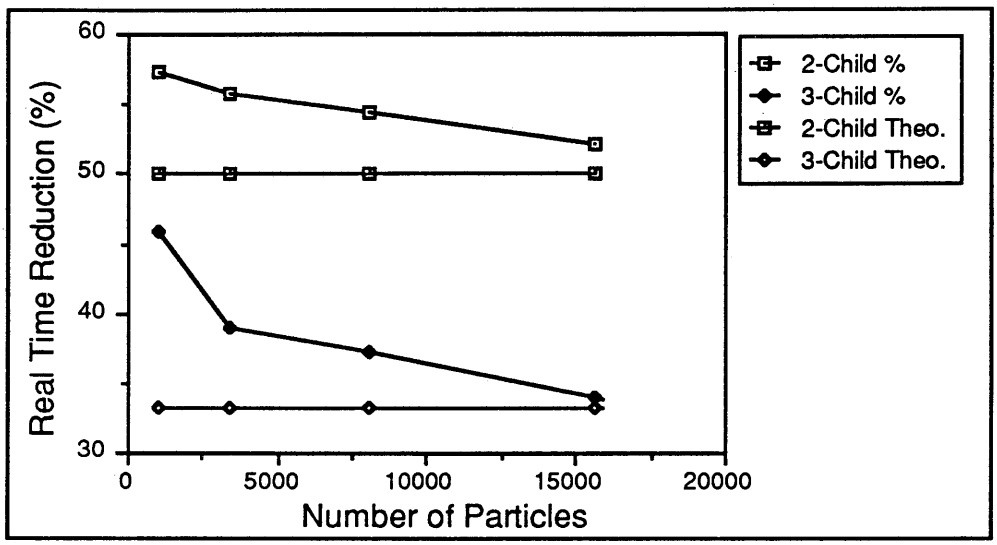
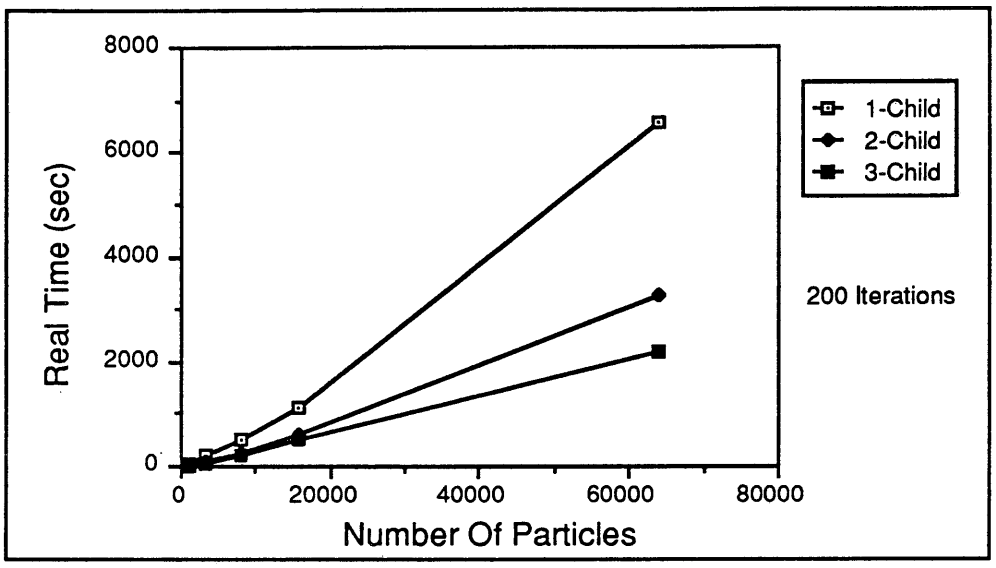


Figure 2.15: Percent Reduction For Varying Particle Counts



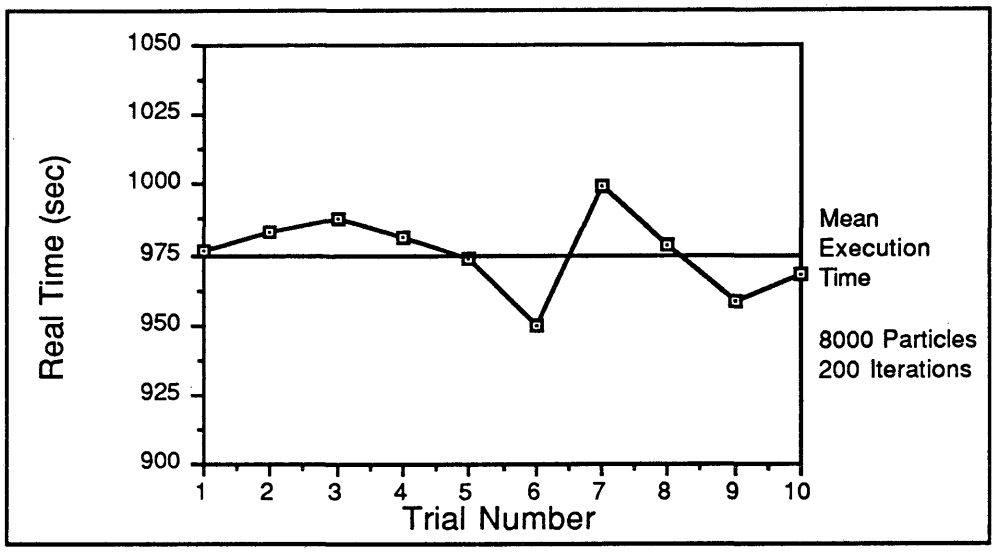
The third test also provides support to Boris' claim, that his vector sort scales linearly with the number of simulated particles. Boris supported his claim through empirical tests [Boris 1985]. Figure 2.16, which is an extension of Figure 2.15 to larger particle counts, supports this claim. The straight lines in the figure indicate two things: first, that as the number of particles simulated increases the time required to perform the updating and sorting of the particles increases in an equal proportion, and second that as the problem is partitioned the sorting of the partitions also scale linearly with the number of particles. The fact that no existing sort scales as order(n) makes this feature of Boris' vector sort quite intriguing.

Figure 2.16: Execution Times For Varying Particle Counts (Extended)



A final test was performed to determine the ELXSI's influence on execution time. The test involved executing the same case ten times to determine whether execution time was relatively constant. The case selected was for 8000 particles and 200 iterations, using the three-child configuration. Results of the test (see Figure 2.17) indicate external factors influence execution time slightly. The impact of this variability on this research has not been established, but assumptions are the impact should be minor and possibly favorable to performance of the partitionable MLG data structure.

Figure 2.17: Execution Times For Multiple Trials

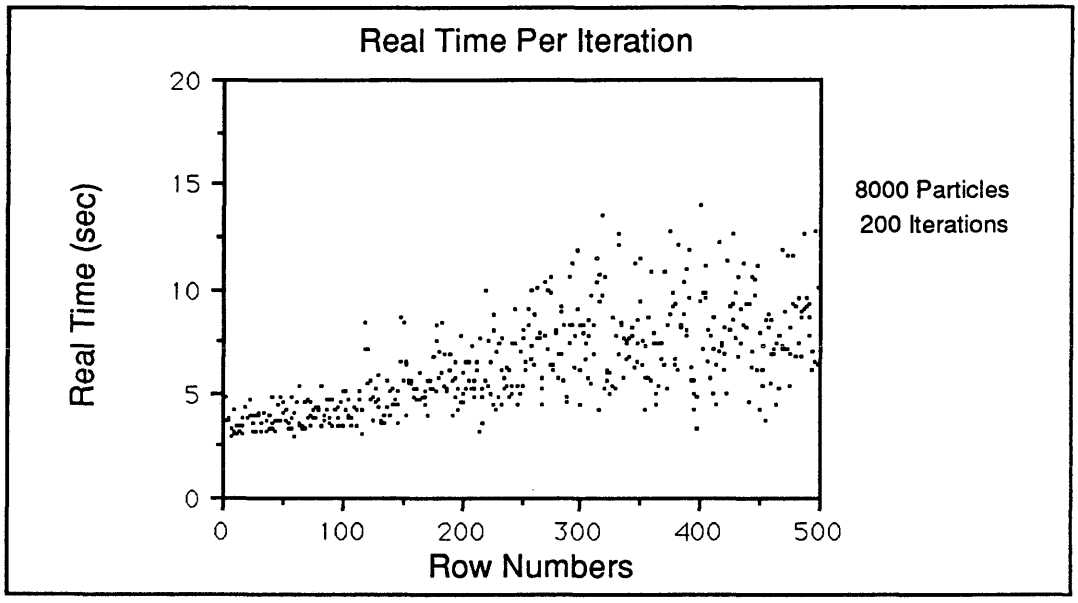


One final issue, identified too late to establish its cause, relates to the execution time per iteration. Figure 2.18 depicts the execution time per iteration for the 8000 particle, 200 iteration case. The increased variability in the execution time per iteration, after the first 100 iterations, indicates a problem with the MLG's stability may exist. Resolution of the cause of the increased variability in per iteration execution time needs further investigation.

The tests performed on the partitionable MLG data structure indicate great potential lies in its use. Three performance relationships relating to using the partitionable MLG data structure have been identified. First, the reduction in execution time achievable using the MLG data structure approaches near theoretical limits. Second, as the

number of iterations increases the performance of the MLG improves. Third, as the number of particles simulated increases the performance of the MLG improves. The results of these tests provide considerable incentive to further pursue development of the MLG as a partitionable data structure.

Figure 2.18: Real Time Per Iteration
(3-Child Configuration)



These tests support use of the MLG data structure for data partitioning in large-scale physical simulation models. Data partitioning enables developers of large-scale physical simulation models to take advantage of modern supercomputer architectures. The performance evaluation shows the benefits

achievable through use of the partitionable MLG data structure in a particle dynamics framework. Efforts are continuing toward completing integration and testing of the mesh generation and reconfiguration algorithm (discussed in the next chapter) with the partitionable MLG data structure to determine its performance with respect to spatial representation models.

CHAPTER III

MESH GENERATION AND RECONFIGURATION ALGORITHM FOR SPATIAL REPRESENTATION

Introduction

Many of today's large scale physical simulation models simulate spatial regions. Models of this type, such as finite element models and hydrodynamics models, simulate the spatial region by dividing it into a collection of space filling sub-elements. Many man-hours are expended annually in developing techniques to perform this task or to generate meshes manually. The process of creating space filling sub-elements is referred to as mesh generation. Development of a fully automated mesh generation algorithm is needed to support spatial simulation modeling. Furthermore, a reconfiguration algorithm which identifies and reconfigures deformed sub-elements is needed to support dynamic spatial simulation models.

Mesh generation is an extremely tedious and time consuming task. Existing computer algorithms require considerable user interaction to define or refine the mesh. These algorithms place unnecessary burden on the user in defining the sub-elements. Development of automated mesh

generation algorithms is a topic frequently addressed in the literature, but currently no robust algorithm exists which can eliminate the user from mesh generation.

Modeling dynamic spatial regions requires the development of algorithms to identify and reconfigure deformed sub-elements. As the spatial region being simulated moves, the sub-elements representing it move and their shape deforms. Deformed sub-elements introduce inaccuracies in the computations performed, which may influence simulation results. Mesh reconfiguration algorithms which identify and reconfigure deformed sub-elements prevent propagation of numerical inaccuracies into the simulation's results.

This chapter presents algorithms developed to solve the problems of mesh generation and reconfiguration. These algorithms, used in conjunction with a partitionable data structure, make possible multi-processing of spatial representation models. Intentions of this research were to: identify algorithms which could create and maintain a tetrahedral sub-element mesh, integrate the mesh algorithms with the partitionable MLG data structure, and evaluate the partitionable MLG data structure in a spatial representation framework. Unfortunately, after an extensive literature search, no mesh algorithm could be identified which would meet requirements; therefore, it was decided to develop one.

Development of the mesh generation algorithm consumed more time than expected, and thus precluded its integration with the partitionable MLG data structure. Fortunately, the algorithms developed provide capabilities sorely needed for spatial representation, independent of their use with the partitionable MLG data structure.

This chapter presents the automated mesh generation and reconfiguration algorithms developed and their implementation in a Pascal program. The Pascal program implementation resides on an ELXSI 6400. Graphics support was provided through DI-3000 graphics software.

Derivation of the Combined Mesh Generation Algorithm

The mesh generation algorithm presented in this chapter is a combination of two mesh generation algorithms identified during the literature search. The algorithm presented here will be referred to as the combined mesh generation algorithm. The first algorithm is a two-dimensional mesh generation algorithm developed by M. J. Fritts and J. P. Boris [Fritts and Boris 1979], the second is a two- or three-dimensional algorithm developed by J. C. Cavendish, D. A. Fields and W. H. Frey [Cavendish, Fields and Frey 1985]. Each of the algorithms had significant advantages and, unfortunately, some disadvantages. As a result of the disadvantages, the combined algorithm, which employs the best features of both algorithms, has been developed. Without the concepts and techniques derived from these algorithms, though, it is plausible to assume that completion of a totally new mesh generation algorithm might never have occurred.

Fritts and Boris provide a relatively simple algorithm for triangulating a two-dimensional region with irregular boundaries and interior holes. The ability of Fritts and Boris' algorithm to handle irregular boundaries and interior

holes is a unique feature in mesh generators which has been integrated into the combined algorithm. The major disadvantages of Fritts and Boris' algorithm are its inability to extend to three-dimensions and its computational expense.

The algorithm developed by Cavendish, et al creates a mesh of well shaped sub-elements in two-dimensional and three-dimensional spatial regions. Cavendish's algorithm utilizes the concepts of the Dirichlet tessellation, the Delaunay tessellation, and Voronoi polygons. The algorithm has several advantages which resulted in it playing a major role in the combined mesh generation algorithm. The first advantage of Cavendish's algorithm is its assurance that the collection of sub-elements generated will be as well proportioned as possible, given a set of vertexes. The second advantage is its extensibility from two-dimensions to three dimensions, and even higher dimensions if applications exist. The third advantage is its applicability to reconfiguration of local areas of deformed sub-elements. The algorithm reconfigures deformed sub-elements without having to reconfigure the entire mesh. These features of Cavendish's algorithm gave it a central role in the development of the combined algorithm. Cavendish's algorithm

provided all but one desired feature, the ability to handle irregular regions.

The integration of concepts from Cavendish's algorithm and Fritts' algorithm into the combined algorithm provides a flexible, fully automated mesh generation and reconfiguration capability. Cavendish's algorithm provided techniques useful and efficient in defining well proportioned two-dimensional and three-dimensional sub-elements. Fritts' algorithm provided techniques for handling irregular regions. The combination of these techniques into the combined algorithm provides the needed capability to generate and maintain a mesh for both static and dynamic spatial regions.

An Overview of the Combined Mesh Generation Algorithm

The combined mesh-generation algorithm constructs a three-dimensional tetrahedral mesh from a set of user defined vertexes. The vertexes lie in a set of planes which slice through (thus defining cross sections) the spatial region. Location of the planes is determined by the user. The combined algorithm requires no specific spacing of planes. For each plane, the user must define all vertexes which lie on the plane. These vertexes are then used to define mesh sub-elements. Each plane's vertexes are divided into two groups: boundary vertexes, and non-boundary vertexes. The boundary vertexes define the spatial region's cross sectional boundary. The non-boundary vertexes determine the density of sub-elements inside the cross sectional boundary.

The combined mesh generation algorithm consists of four phases. The first phase is boundary vertex triangulation, the second phase is non-boundary vertex integration, the third phase is inter-plane tetrahedron formation, and the fourth phase is inter-plane tetrahedron refinement.

Boundary vertex triangulation defines for each plane a collection of triangles which fill the cross-sectional region. Techniques borrowed from Fritts' algorithm are used

to generate the boundary vertex triangles. Completion of this phase results in a collection of triangles, for each plane, which fill the cross-sectional area of the spatial region.

Non-boundary vertex integration incorporates each plane's additional vertexes into the boundary vertex triangulation. Techniques borrowed from Cavendish's algorithm are used in this phase to add the non-boundary vertexes to the boundary vertex triangulation, and to refine the shape of the triangles in the triangulation. Refinement of triangles is performed to refine the shape of triangles created in the boundary vertex triangulation. Completion of this phase results in a collection of well shaped triangles,for each plane, which fill the cross-sectional area of the spatial region.

Inter-plane tetrahedra formation and refinement creates tetrahedra which comprise the initial mesh. Techniques similar to those used in defining the boundary vertex triangulation are used in this phase to form the initial collection of inter-plane tetrahedra. Then, in the final phase techniques borrowed from Cavendish's algorithm are used to refine the inter-plane tetrahedra. Completion of these phases results in a collection of well shaped tetrahedral sub-elements which completely fill the spatial region. The

mesh of sub-elements formed can be used as the initial mesh in static and dynamic spatial region simulations.

Simulation of dynamic spatial regions requires sub-element reconfiguration. Movement of the spatial region deforms sub-elements. The combined algorithm's reconfiguration algorithm employs techniques borrowed from Cavendish's algorithm. The reconfiguration algorithm identifies deformed sub-elements and reconfigures local areas where the deformed sub-element is located.

The following sections of this chapter will provide a detailed description of the Pascal implementation of the combined mesh generation algorithm. The next section provides a description of the input required by the program and the data structure used in maintaining the mesh. The following four sections give a detailed description of the combined algorithm's four phases. The final two sections provide a detailed description of the vertex insertion and mesh reconfiguration portions of the combined algorithm.

Combined Mesh Generation Algorithm Input and Data Structure

The combined mesh generation algorithm has been implemented in a Pascal program which requires the user to define all vertexes which will be used to generate the tetrahedral mesh. The program requires as input: the number of planes which slice through the region, the number of vertexes on each plane, the number of boundary vertexes for each plane, and the three-dimensional position vector of each vertex. The program makes no assumptions relating to the location of planes in three-dimensional space, but it does assume the planes are parallel, with the order defined in the input file determining the sequence of neighboring planes.

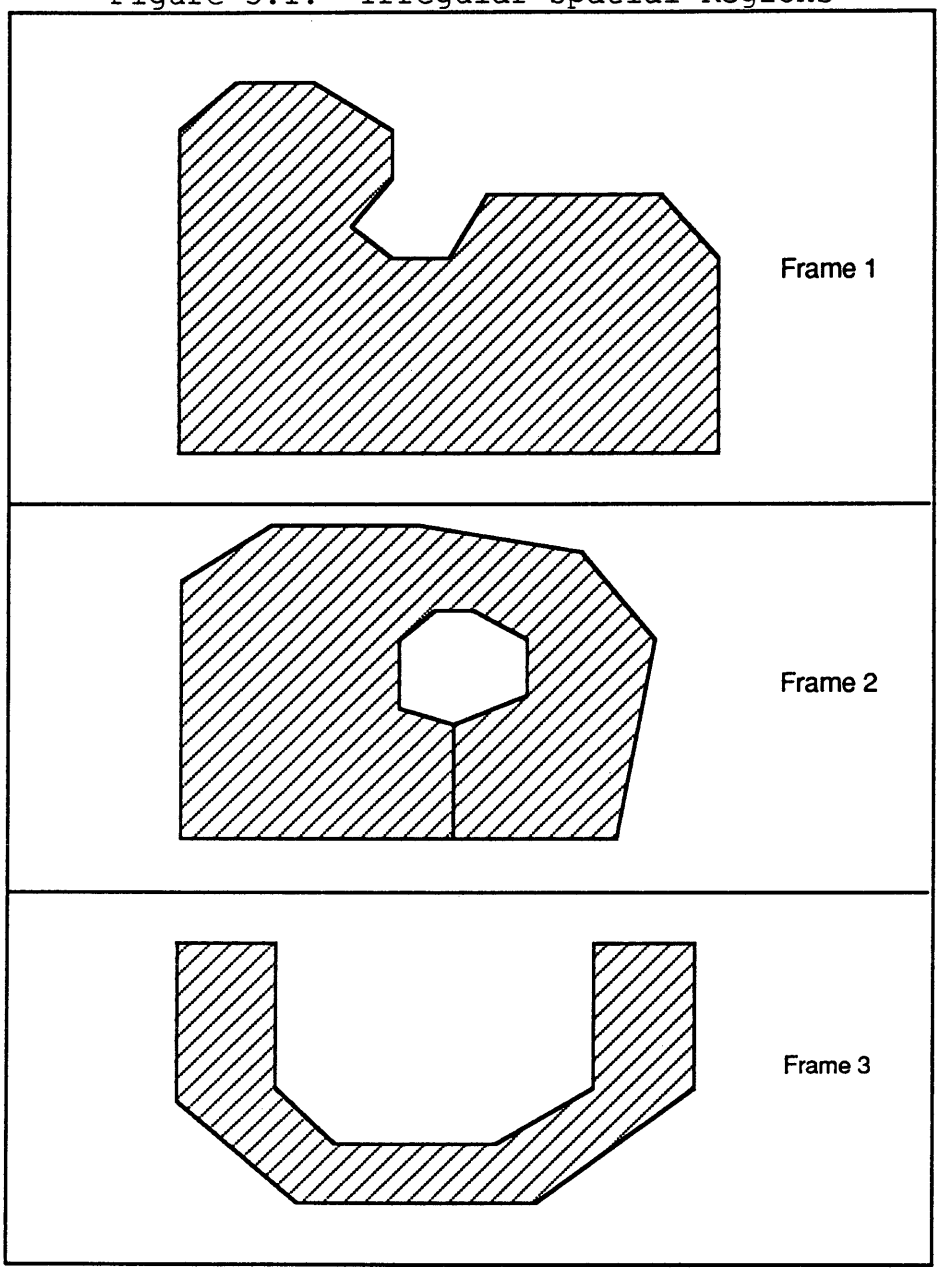
The program's data structure makes extensive use of data records to store data relating to mesh elements. Mesh elements are: vertexes, sides, triangles, tetrahedra, planes, and zones. Each different type of mesh element has a related data record defined. Each element in the mesh has all of its related data stored on its unique record. All records, relating to a type of element, are stored in that element type's array. Thus, there is an array for each element type.

Boundary Vertex Triangulation

Boundary vertex triangulation creates a collection of triangles which fill the cross-sectional region defined by the slicing planes. The program assumes boundary vertexes are defined in the order which, when connected sequentially, form the entire boundary. The boundary formed may be regular or irregular in shape. Examples of irregular shaped regions are provided in Figure 3.1.

The boundary is used to determine whether or not sides required in forming triangles are inside or outside the region. The boundary includes both outer boundaries and interior boundaries. Definition of the boundary is defined by continuous and unique connection of a plane's boundary vertexes. Continuous implies connection of the vertexes must be performed without gaps. Unique implies connection between vertexes can occur only once, with one exception: two connections between the same vertexes are required when moving between outer and interior boundaries (see Frame 2 of Figure 3.1).

Figure 3.1: Irregular Spatial Regions



Connections between vertexes are called sides. The sides formed when boundary vertexes are connected are called boundary sides. Boundary sides are used to determine whether or not a proposed side (a side required to form a triangle) has any portion outside of the cross-sectional region. If a proposed side is outside the planar region the side is not feasible and thus the triangle cannot be formed.

The following is a summary of the steps used by the combined algorithm to create boundary vertex triangles:

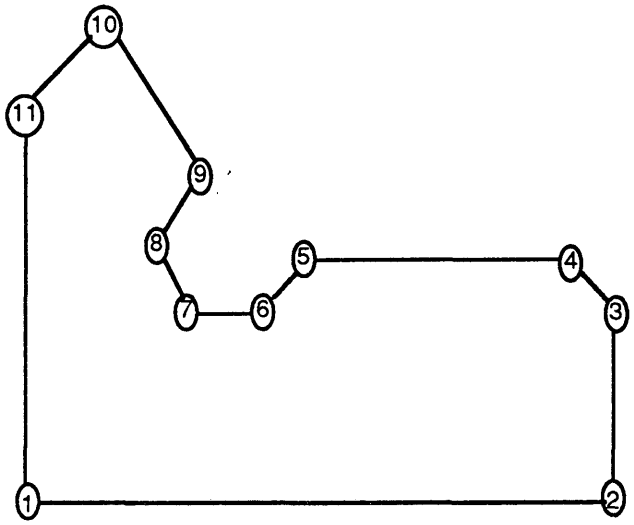
1) create all boundary sides,
2) select a boundary side,
3) select a boundary vertex and test it,
4) if selected boundary vertex is infeasible return to step 3,
5) if selected boundary vertex is feasible create triangle and return to step 2,
6) if sub-region triangulation is not complete repeat steps 2-5.

These steps are discussed in the following paragraphs along with a discussion of how vertex feasibility is determined.

Boundary Side Creation. The initial step of the boundary vertex triangulation is identification and creation of boundary sides. Figure 3.2 provides an eleven vertex example along with the defined boundary sides. The eleven points are connected (much like connect-the-dots) starting with vertex number one and proceeding around the boundary to vertex number eleven. Vertex number eleven is then connected to vertex one to complete the region's boundary. For each connection between vertexes a side record is created and linked into the side array. The side record defines the vertexes which it connects by pointers to vertex records. Sides in this array are tested against any proposed sides to determine whether or not the proposed sides are feasible.

Boundary Side Selection. After the boundary sides are created one of the boundary sides is selected for triangle identification and creation. The program initiates triangle identification by selecting two boundary sides (which connect two pairs of vertexes) and defining these sides' vertexes as a low pair of vertexes and a high pair of vertexes. In the example depicted in Figure 3.2, the low pair is defined as vertexes 1 and 2. The high pair is defined as vertexes 11 and 10. It should be noted that the order of these pairs is significant, with vertexes 1 and 11 being assigned the

Figure 3.2: Boundary Vertex Triangulation (Step 1)



Current Pairs:

Low Pair	High Pair	Solution Pair
[1,2]	[11, 10]	----

Previous Pairs:

Low Pair	High Pair	Solution Pair
----	----	----

Failed Stack Pairs:

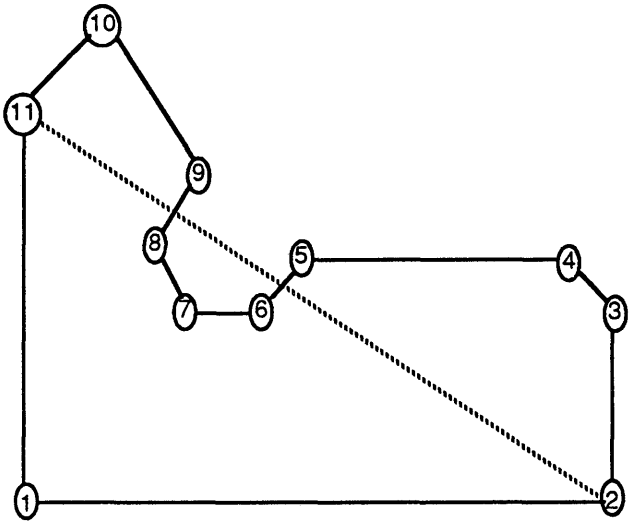
Low Pair	High Pair	Solution Pair
----	----	----

primary position, and vertexes 2 and 10 being assigned the secondary position.

A primary pair designation is also assigned to either the low or the high pair of vertexes. The primary pair designation defines the pair of vertexes, or the boundary side, which is selected for triangle formation. The other pair of vertexes is designated as the secondary pair. In Figure 3.2, the primary pair (cross hashed boxes) is designated as the low pair, vertexes 1 and 2, and the secondary pair is the high pair, vertexes 11 and 10. Once the primary pair is defined the selection of the third boundary vertex, needed to complete the triangle, is begun.

Boundary Vertex Selection And Testing. Vertex selection starts with the primary vertex of the secondary pair. In Figure 3.2 the primary vertex of the secondary pair is identified as vertex 11. The selected vertex is tested to determine feasibility of any proposed (i.e. non-existent) sides which are needed to form a triangle with the selected side (defined by the primary pair of vertexes). If all of the proposed sides are feasible, the initial vertex of the secondary pair is used to form the triangle. If any of the new sides are infeasible, the triangle cannot be

Figure 3.3: Boundary Vertex Triangulation (Step 2)



Current Pairs:

Low Pair	High Pair	Solution Pair
[1,2]	[10, 9]	----

Previous Pairs:

Low Pair	High Pair	Solution Pair
[1,2]	[11, 10]	----

Failed Stack Pairs:

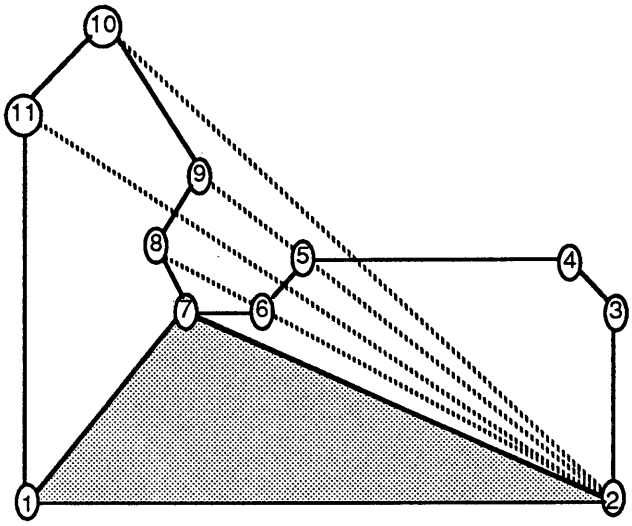
Low Pair	High Pair	Solution Pair
[1, 2]	[11, 10]	----

formed. In either case a shifting of vertex pairs is performed.

Shifting of vertexes introduces a new, but existing, boundary vertex into one of the vertex pairs. If a triangle is formed the primary pair of vertexes is shifted. If a triangle is not formed the secondary pair of vertexes is shifted. In both cases, shifting of vertexes is performed to introduce different boundary vertexes for triangle formation.

Shifting of vertexes is performed differently depending upon whether the vertex pair is the low pair or the high pair. If the shifted pair is the low pair, the vertex in the secondary location is advanced to the primary location, and the next higher vertex is assigned the secondary location. If the shifted pair is the high pair, the vertex in the secondary location is advanced to the primary location, and the next lower vertex is assigned the secondary location. Figure 3.3 shows the shifting which occurs as a result of proposed side 2-11 violating side 5-6. Since the high pair is the secondary pair, and since triangle formation failed, the high pair is shifted. Shifting of the high pair of vertexes is achieved by assigning vertex 10 the primary location and vertex 9 the secondary location. This shifting of the high pair can be seen by comparing pairs in

Figure 3.4: Boundary Vertex Triangulation (Step 3)



Current Pairs:

Low Pair	High Pair	Solution Pair
[2, 3]	[7, 6]	----

Previous Pairs:

Low Pair	High Pair	Solution Pair
[1,2]	[7, 6]	[7, 6]

Failed Stack Pairs:

Low Pair	High Pair	Solution Pair
[1, 2]	[11, 10]	[7, 6]

the current pairs section and the previous pairs section of the figure.

Boundary Vertex Infeasible. Failure of the initial vertex of a secondary pair of vertexes to be used to form a triangle results in the creation of a failed record. A failed record records the state of the vertex pairs at the time of the failure. The failed records are stored in a stack data structure, which can be understood by thinking about a stack of books. To access information about a book at the bottom of the stack the books on top must be removed. Similarly for the stack data structure, to access data stored on a record at the bottom of the stack, each record above it must be removed. In computer parlance this is referred to as popping the stack. The placing of records on the top of the stack is referred to as pushing the stack. Both these terms will be used throughout the remainder of this section.

A single failed record is created for a selected side irrespective of how many selected vertexes fail. Figure 3.4 shows the vertexes selected and tested for side 1-2. Vertexes 11, 10, 9 and 8 were tested and determined to be infeasible before feasible vertex 7 was selected. Even though four infeasible vertexes were identified only one failed record was created for selected side 1-2.

Once a feasible vertex is identified the solution pair is stored on the failed record. The solution pair is defined as the secondary pair at triangle formation. The failed stack pairs shows the failed record created for side 1-2. The low pair is the primary pair of vertexes and the high pair is the secondary pair of vertexes at the time of the initial selected vertex failure (i.e. vertex 11). The solution pair is the secondary pair of vertexes when a feasible vertex is identified (i.e. vertex 7). The failed record is created when the initial vertex is determined to be infeasible, but not all fields on the record are defined until a feasible vertex is identified. This can be seen by comparing the failed stack pairs in Figures 3.3 and 3.4. In 3.3 the solution pair is empty, in Figure 3.4 after several infeasible vertexes have been tested and a feasible vertex has been identified the solution pair is defined.

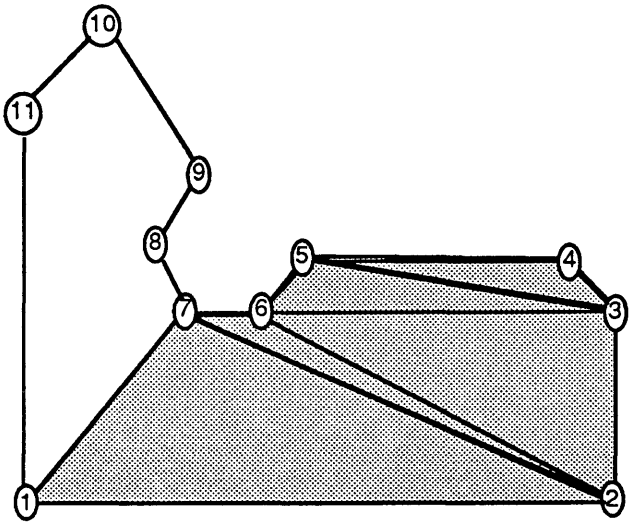
The failed stack maintains a history of boundary sides which have failed their initial triangle formation attempts. The failed stack is used to skip over sub-regions which cannot be triangulated. The data recorded by the failed records enables the boundary vertex triangulation algorithm to return to sub-regions where triangulation was halted and thus complete triangulation.

Boundary Vertex Solution. Identification of a feasible vertex results in triangle formation, primary pair shifting, and pair designation switching. Figure 3.4 shows the above operations as a result of vertex 7 being feasible. A triangle 1-2-7 is formed which is defined by vertexes 1, 2 and 7 and sides 1-2, 1-7 and 2-7. The shifting of the primary pair of vertexes, 1 and 2, is seen by comparing the previous and current pairs sections. In the previous pairs section the primary pair is the low pair. The low pair is shifted from vertexes 1 and 2 to vertexes 2 and 3. The pair designation switching is also seen by comparing the previous and current pairs sections. The cross-hatched pairs in these sections switch from the low pair to the high pair. The last two operations are performed in preparation of the next iteration. The primary pair shifting is performed to introduce a new vertex into the vertex pairs. Primary pair designation is performed to reduce the time required to triangulate the region. Once these operations are complete, formation of a new triangle is begun using the newly selected primary pair.

Sub-Region Triangulation Termination. The process outlined above continues until overlapping of the primary and secondary pairs occurs. Overlapping of the primary and secondary pairs is defined to occur whenever the primary vertex of the primary pair is the same as the secondary vertex of the secondary pair, and the primary vertex of the secondary pair is the same as the secondary vertex of the primary pair. This is difficult to follow, but review of the current pairs section of Figure 3.5 should help. The current pairs section shows the low pair to consist of vertexes 4 and 5 and the high pair to consist of vertexes 5 and 4. This overlapping of pairs indicates triangulation has continued in its current direction as far as possible.

The occurrence of overlapping pairs indicates the completion of triangulation in the current direction, but it does not imply completion of the triangulation. Completion of the triangulation process occurs when the vertex pairs overlap and the failed stack is empty. An empty failed stack indicates all regions have been triangulated. If the failed stack is not empty the stack must be popped to obtain the status at the time of the most recent triangle formation failure.

Figure 3.5: Boundary Vertex Triangulation (Step 4)



Current Pairs:

Low Pair	High Pair	Solution Pair
[4, 5]	[5, 4]	----

Previous Pairs:

Low Pair	High Pair	Solution Pair
[3, 4]	[5, 4]	[5, 4]

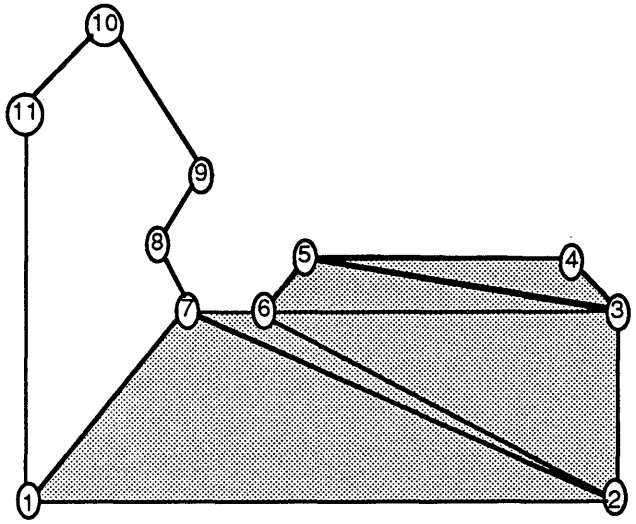
Failed Stack Pairs:

Low Pair	High Pair	Solution Pair
[1, 2]	[11, 10]	[7, 6]

From Figure 3.5 it can be seen that the failed stack consists of only a single record. This record is popped off the stack and its data is used to re-initialize the current pairs of vertexes.The current pairs are re-initialized to allow the process to return to the un-triangulated portion of the region. Re-initialization performs the following actions: define the current secondary pair as the failed record secondary pair, define the current primary pair as the failed record primary pair, re-define the vertex in the secondary location of the current primary pair to be the primary vertex in the solution pair, and specify the current pair designated as the primary pair to be the pair designated as the primary pair on the failed record.

Figure 3.6 portrays the results of these operations, which result in an empty failed stack and vertexes 1 and 7 becoming the primary pair. The popping of the failed stack and the redefinition of the pairs of vertexes enable processing to continue to un-triangulated regions. After the failed stack is popped and the pairs of vertexes are redefined processing proceeds as usual. Again if failure to form a triangle occurs the data is pushed onto the stack and processing proceeds until overlapping of pairs occurs.

Figure 3.6: Boundary Vertex Triangulation (Step 5)



Current Pairs:

Low Pair	High Pair	Solution Pair
[1, 7]	[11, 10]	----

Previous Pairs:

Low Pair	High Pair	Solution Pair
[4, 5]	[5, 4]	----

Failed Stack Pairs:

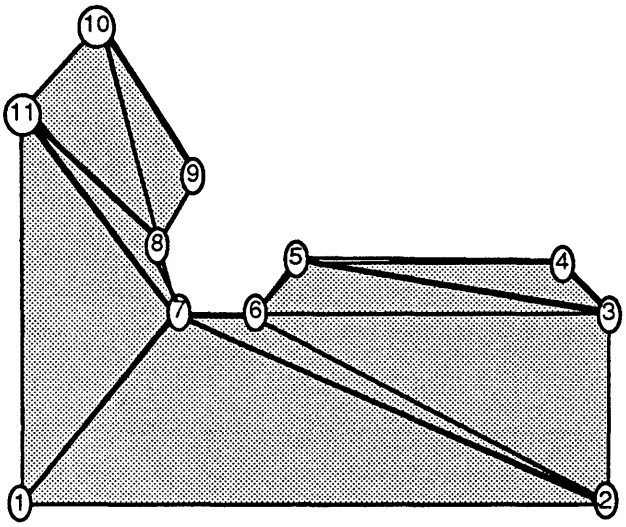
Low Pair	High Pair	Solution Pair
----	----	----

This continues until overlapping of vertexes occurs and the failed stack is empty, at which point boundary vertex triangulation is complete and the addition of non-boundary vertexes is begun.

Figure 3.7 shows the completed boundary vertex triangulation, along with a history of the vertex pairs and their outcome. The vertex history provided in the figure indicates overlapping occurring when the low pair, 7 and 8, overlap with the high pair, 8 and 7. Overlapping with an empty stack indicates boundary vertex triangulation is complete. The process described above is performed for each plane of vertexes defined by the user.

Vertex Feasibility Determination. Vertex feasibility is determined by testing each of the vertex's proposed sides to assure none of them violates any existing side. Violation of an existing side is determined by looking at a convex cone formed using one of the proposed side's vertexes. The vertex is used to define the vertex of a convex cone, with the cone's extreme vectors defined by two vectors which emanate from the cone's vertex to the vertexes of the existing side.

Figure 3.7: Boundary Vertex Triangulation (Complete)

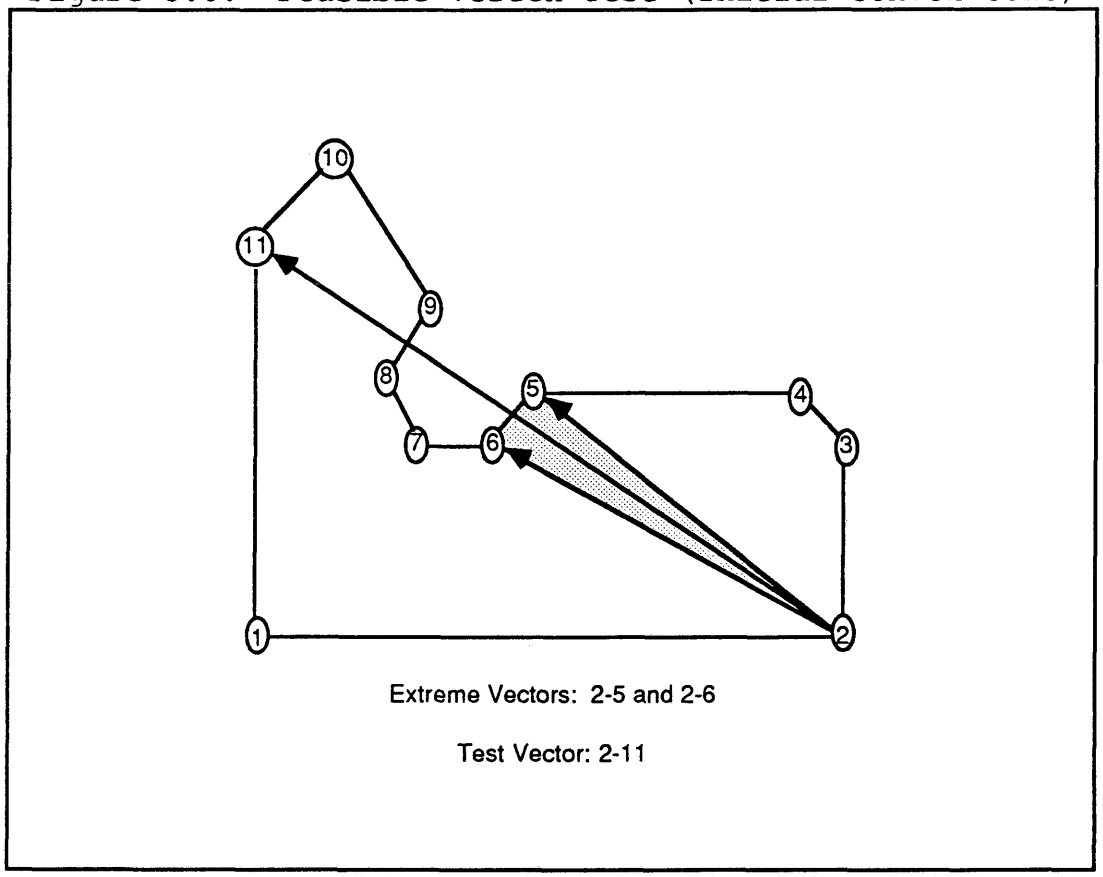


Pair History And Outcome:

Low Pair	High Pair	Outcome
[1, 7]	[11, 10]	Δ1-7-11
[7, 8]	[11, 10]	Failed
[8, 9]	[11, 10]	Δ11-10-9
[8, 9]	[10, 9]	Δ 8-9-10
[9, 10]	[10, 9]	Overlap
[7, 8]	[11, 8]	Δ11-8-7
[7, 8]	[8, 7]	Overlap

Figure 3.8 gives an example of this for the proposed side 2-11. A convex cone is formed with vertex 2 as the vertex and the vectors 2-6 and 2-5 as the extreme vectors. A test vector, which represents the proposed side, is created. In the example, the test vector is 2-11. The extreme vectors are used to determine whether or not the test vector is in the interior of the convex cone.

Figure 3.8: Feasible Vertex Test (Initial Convex Cone)

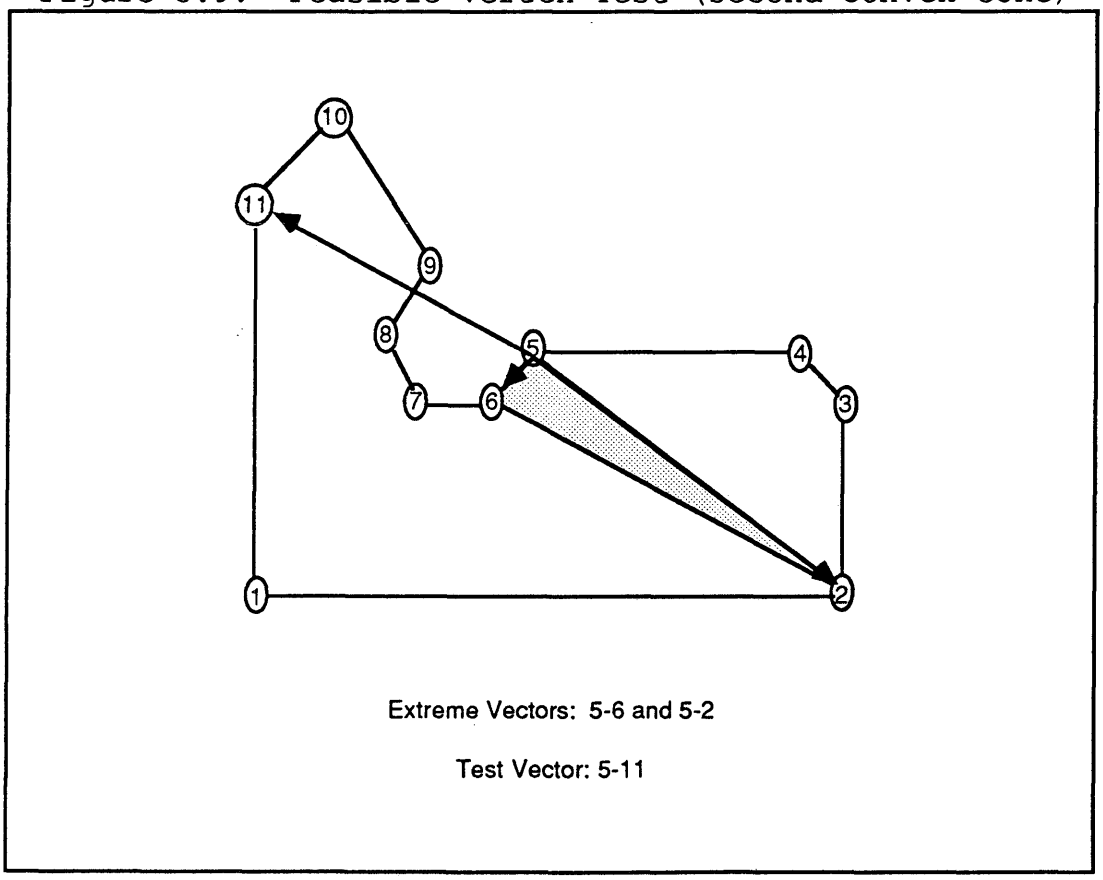


A vector will lie in the convex cone if a strictly positive linear combination of the extreme vectors can be found [Marlow 1978]. A vector will lie on the extreme vector if one of the linear combination's scalar multiples is zero. A matrix is formed with the extreme vectors as the first two columns and the negative of the test vector as the third. The null space of the matrix, calculated using row reduction, and the basis of the null space are inspected to determine the value of the scalar multiples. Non-negative scalar multiples indicate the test vector lies in the interior of the convex cone.

A test vector which lies in the interior of the convex cone requires a second check to be performed to determine whether it penetrates the existing side. It is possible for the test vector to lie in the interior of the cone without penetrating the existing side. Penetration of the existing side is determined in the same manner as above with the exception that a new convex cone is defined for the test. The new convex cone has the initial vertex of the existing side as its vertex, and the extreme vectors emanate to the second vertex of the existing side and the initial vertex of the proposed side.

Figure 3.9 depicts this cone, with vertex 5 as the cones vertex, and vectors 5-6 and 5-2 as its extreme vectors. Vector 5-11 is the test vector. A matrix with the vectors as columns is formed, and the scalar multiples are obtained. If the scalar multiples are non-negative the the proposed side does not violate the existing side. If either scalar multiple is negative then the proposed side violates the existing side.

Figure 3.9: Feasible Vertex Test (Second Convex Cone)



Failure to pass both of the above tests indicates the proposed side violates an existing side. Both tests are performed on each of a selected vertex's proposed sides. Failure of any proposed side indicates the selected vertex is infeasible, and the triangle cannot be formed.

The final result of the boundary vertex triangulation is a collection of triangles which may be irregular in shape. The approach used to define the boundary vertex triangles does not include any methodology to improve the shape of the triangles. The approach is solely concerned with creating some set of triangles using the boundary vertexes. Improving the shapes of the triangles is performed during the non-boundary vertex integration.

Non-Boundary Vertex Integration

Non-boundary vertex integration incorporates new vertexes into the boundary vertex triangulation. The new vertexes can be located either in the interior or along the boundary of the region. The combined algorithm's approach for non-boundary vertex integration is: define for each existing triangle a circumscribing circle, for each new vertex use the circumscribing circle to identify an insertion polygon, eliminate all sides in the interior of the insertion polygon, and connect all remaining sides to the new vertex to form a set of triangles. The following paragraphs provide a detailed example of the algorithm. This approach was presented in Cavendish's description of his algorithm.

The algorithm is based on a property developed by D. F. Watson relating circumscribing circles and triangles. Watson identified the property that three vertexes, from a collection of given vertexes, will form a Delaunay triangle if and only if the circumscribing circle defined by these vertexes contains no other vertexes in its interior [Watson 1981]. A Delaunay triangulation is a collection of triangles each of which is as nearly equilateral as possible for the given set of vertexes. Cavendish's algorithm utilizes this property by rejecting all triangles which contain any

vertexes in their circumscribing circles interior, and keeping those triangles which do not contain any vertexes in the interior of their circumscribing circles.

The Delaunay triangulation is generated in the following manner. Assume the positions of n distinct vertexes $p_1 \ldots p_n$, lying in a plane, are known. The set of all points closest to a particular vertex defines a territory known as a Voronoi polygon, defined as follows:

$$V_i = \left\{x: |x-p_i| < |x-p_j| \text{ for all } j \neq i\right\}$$

where $|\cdot|$ is the Euclidean distance and p_i is the generating vertex. The set of all Voronoi polygons is called the Dirichlet tessellation. In two dimensions, each corner point of each Voronoi polygon is shared by two other Voronoi polygons, and by connecting the three generating vertexes a triangle is formed. The set of all triangles formed in this manner generates a Delaunay triangulation, which is the dual of the Dirichlet tessellation. Figure 3.10 gives an example of the Dirichlet tessellation and the resulting Delaunay triangulation for a set of 13 vertexes. Watson proved the Delaunay triangulation defines a collection of triangles which are as close to equilateral as possible, thus avoiding ill-conditioned and thin triangles whenever possible.

The initial step in integrating non-boundary vertexes into the triangulation is to define for existing triangles their circumscribing circles. The circumscribing circles are completely defined by the center of the circle and the circle's radius. The center of the circumscribing circle is calculated by finding the intersection of the perpendicular bisectors of the triangle's sides. The radius is then derived by finding the distance from the center of the circle to one of the triangle's vertexes. Appendix B provides a detailed derivation of this calculation.

The second step in non-boundary vertex integration is identification of the new vertex's insertion polygon. The insertion polygon consists of those triangles whose circumscribing circles have the new vertex in their interior. Identification of triangles in the insertion polygon is performed by calculating the distance between the new vertex and the center of each triangle's circumscribing circle and comparing the distance to the radius of the circle. If the distance is less than the radius of the circle the new vertex is in the interior of the circle, and the triangle is added to the list of insertion polygon member triangles. Sides and vertexes which form the member triangles will be referred to as member sides and member vertexes.

Figure 3.10: Delaunay Triangulation
Copied from figure in [Cavendish, Field and Frey 1985]

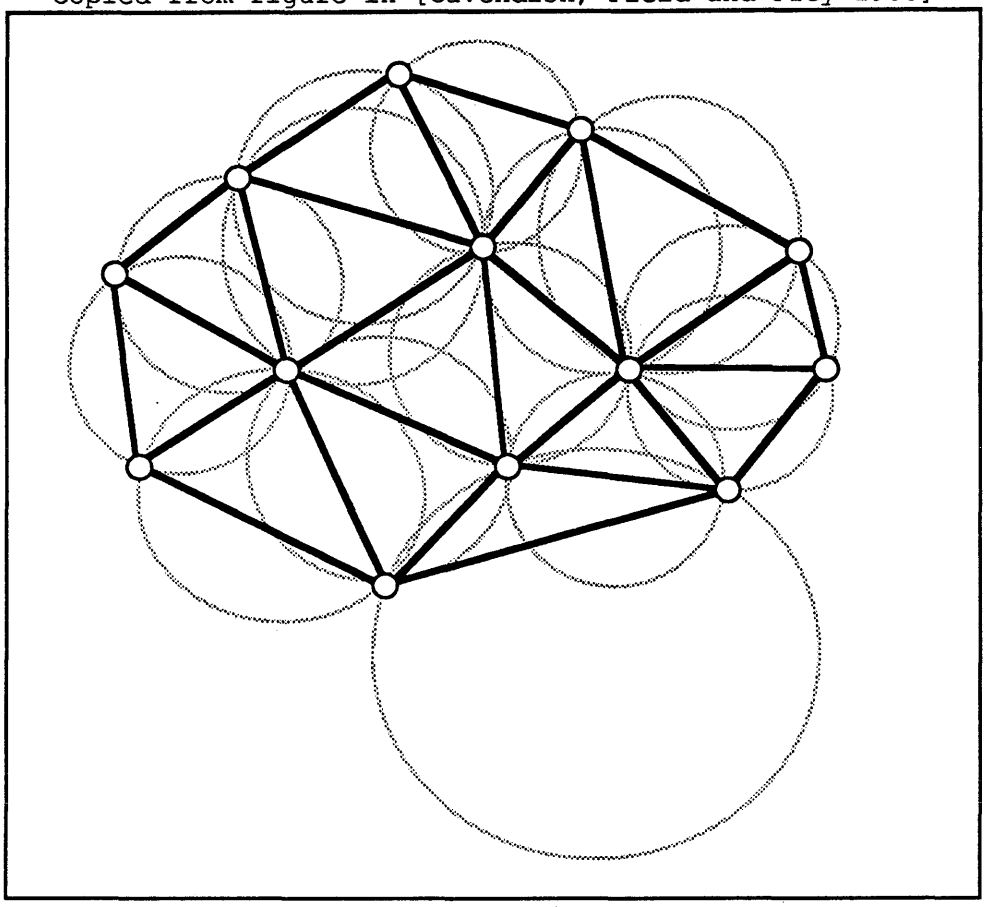
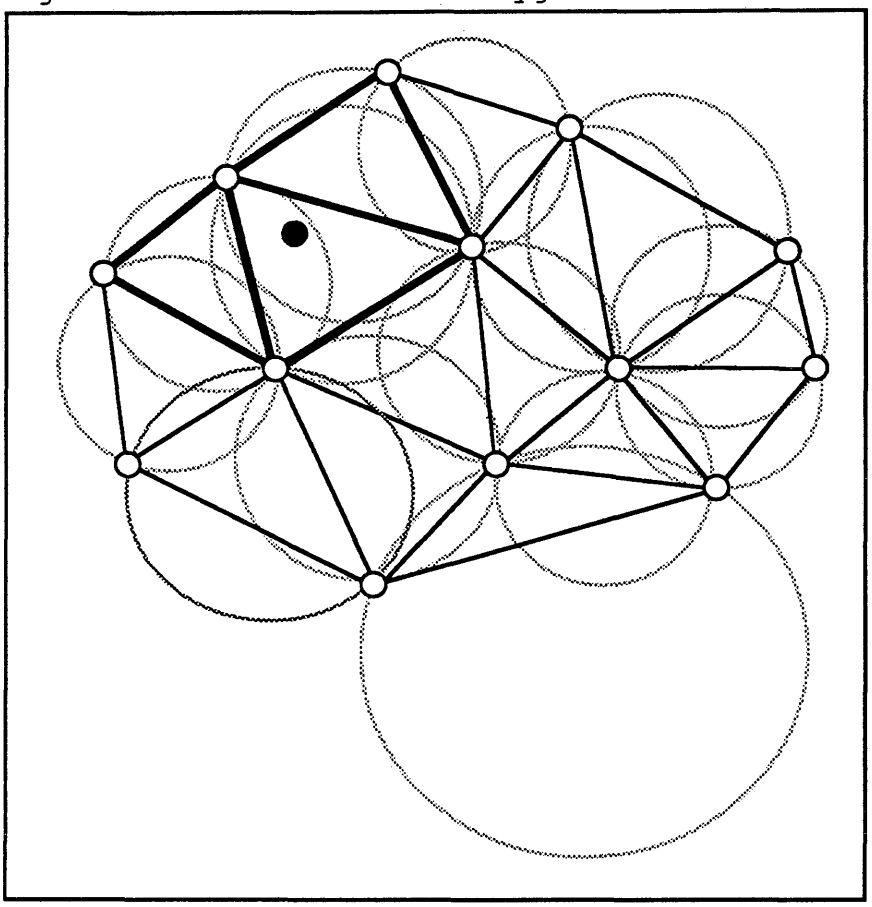


Figure 3.11 gives an example of an insertion polygon. After the new vertex's insertion polygon is identified, the sides in the interior of the insertion polygon must be identified. Interior sides are identified by determining whether the vector from the new vertex to each member vertex penetrates any member side. If any member side is penetrated the side is in the interior of the insertion polygon. The computation performed is the same as that described in the

boundary vertex triangulation: a member side is selected, a convex cone with the new vertex as its vertex and the vectors emanating to the vertexes of the selected side as its extreme vectors is calculated, a member vertex is selected and a test vector is calculated, a linear combination of the extreme vectors is calculated and the scalar multipliers are evaluated to determine whether or not the test vector is in the interior of the convex cone. If inside the convex cone, a similar test is performed to determine whether or not the test vector penetrates the selected side. If the test vector penetrates the selected side the selected side is in the interior of the insertion polygon. A convex cone is calculated for each member side and compared to each member vertex's test vector. Whenever a test vector violates a member side the member side is removed from the insertion polygon.

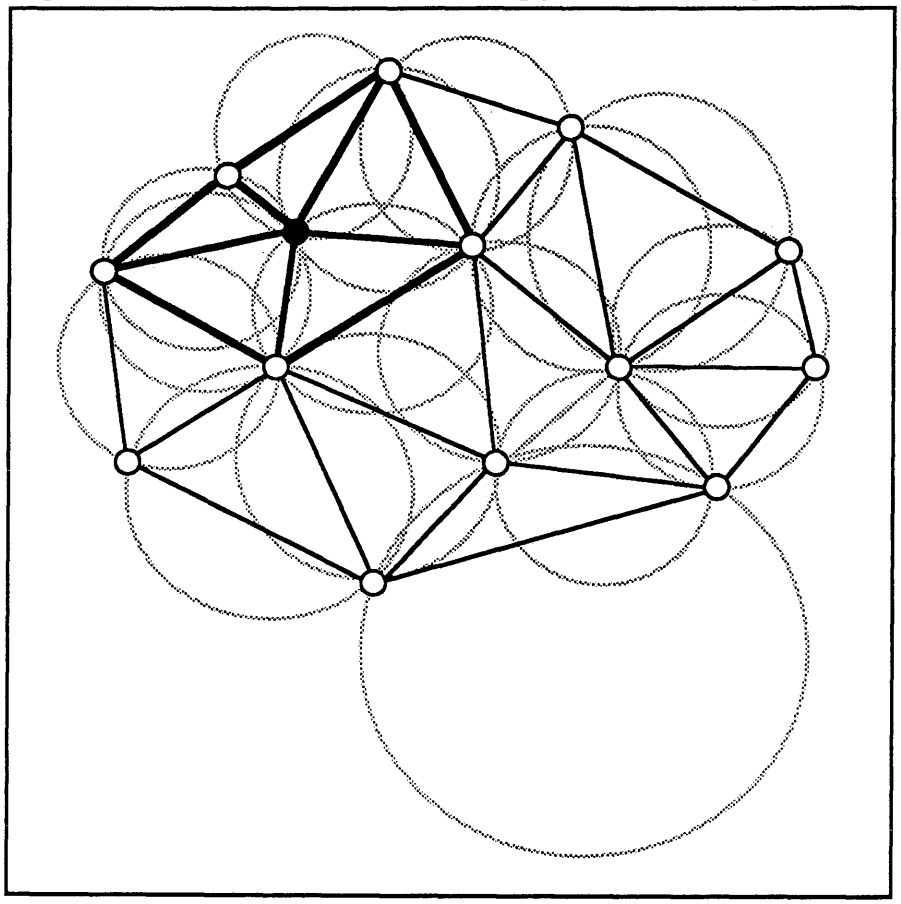
The final step in integrating non-boundary vertexes is definition of the new set of triangles. New triangles are formed by creating the sides needed to connect the remaining member sides to the new vertex. Figure 3.12 gives an example of this reconnection process.

Figure 3.11: Insertion Polygon Identification



Integration of non-boundary vertexes is performed for each user defined plane. Once complete, a set of region filling well-shaped triangles are defined. After all planes have been triangulated the formation of inter-plane tetrahedra is begun.

Figure 3.12: Insertion Polygon Reconfiguration



Inter-Plane Tetrahedra Configuration

Inter-plane tetrahedra configuration creates the tetrahedra which connect triangles on two neighboring planes. The formation of tetrahedra between planes parallels the approach used in performing boundary vertex triangulation. The combined algorithm's approach to inter-plane tetrahedra formation is:

1) select two neighboring planes,
2) select a triangle from one of the planes
3) identify the available vertex in the neighboring plane nearest the selected triangle,
4) determine whether sides proposed (or required) to form a tetrahedron between the triangle and the vertex will violate any existing inter-plane triangles,
5) if all proposed sides are feasible form the tetrahedron,
6) if the formation of any proposed side violates an existing inter-plane triangle perform steps 3-6
6) repeat steps 1-6 for each triangle in each plane.

The convex cone methodology defined earlier is used in determining whether or not a proposed side violates any existing inter-plane triangles. The convex cone is now a three-dimensional cone rather than the planar cone in the previous discussions. For each proposed side and each existing inter-plane triangle a convex cone is calculated. The first vertex of the proposed side is used as the vertex

of the cone, and the vertexes of the inter-plane triangle are used to form its extreme vectors. A test vector emanating from the cone's vertex to the second vertex of the proposed side is calculated. A linear combination of the extreme vectors which equates to the test vector is calculated. The scalar multiples of the linear combination are reviewed to determine whether or not the test vector is in the interior of the convex cone. If all scalar multiples are strictly positive then the test vector is inside the convex cone, if any scalar multiple is negative the test vector is outside the convex cone. If inside the convex cone, a similar test is performed to determine whether or not the proposed side penetrates the inter-plane triangle.

Penetration of the inter-plane triangle is determined using the same approach as defined in the previous paragraph with a change in selection of the cone's vertex. The choice of cone vertex is changed to one of the inter-plane triangle's vertexes, and the convex cone's extreme vectors emanate to the remaining vertexes of the triangle and the initial vertex of the proposed side. The test vector formed is a vector emanating from the cone's vertex to the second vertex of the proposed side. Again the scalar multiples of the linear combination are used to determine whether or not penetration occurs. If the scalar multiples are all non-

negative then the proposed side does not penetrate the inter-plane triangle. If any of the scalar multiples are negative then the proposed side does penetrate the inter-plane triangle, and the selected opposite plane vertex is infeasible.

The formation of inter-plane tetrahedra is performed for each pair of neighboring planes. After all inter-plane tetrahedra are formed the final mesh generation stage is begun. The final mesh generation stage refines the collection of defined inter-plane tetrahedra.

Inter-plane Tetrahedra Refinement

Inter-plane tetrahedra refinement reviews the tetrahedra formed in the previous stage to identify and reconfigure any ill-shaped tetrahedra. The combined algorithm's inter-plane tetrahedra refinement is based on concepts developed by Watson and implemented by Cavendish. Watson proved that Delaunay triangulation, in two-dimensions, generates a set of triangles as close to equilateral triangles as possible for the given set of vertexes. Watson also extended the concept to three-dimensions, where it implies that the triangular faces of the tetrahedra are as near equilateral triangles as possible. These concepts were implemented by Cavendish in

his mesh generation algorithm. The combined algorithm uses the same techniques as developed by Cavendish for inter-plane tetrahedra refinement.

The combined algorithm's tetrahedra refinement algorithm closely parallels the non-boundary vertex integration phase described earlier. The refinement process is: identify any vertex which is in the interior of another tetrahedron's circumscribing sphere, identify the vertex's insertion polygon, identify and remove any triangles in the insertion polygon's interior, and reconnect remaining member triangles to the vertex. Each of these steps is described in the following paragraphs.

Identification of a vertex's insertion polygon in three-dimensions requires identification of all tetrahedra which contain the vertex in the interior of their circumscribing sphere. A circumscribing sphere is completely specified by its center and radius. The calculation of a tetrahedron's circumscribing sphere is analogous to the calculation of a triangle's circumscribing circle. The center of a sphere is determined by the intersection of the perpendicular bisectors for the faces of the tetrahedron. This calculation is described in detail in Appendix B. The radius of the sphere is the distance between the center of the sphere and one of the sphere's vertexes. The distance between the center of

each tetrahedron's circumscribing sphere and the test vertex is calculated and compared to the radius of the sphere. Whenever the radius is greater than the distance calculated, the vertex is in the interior of the sphere. All tetrahedra with the test vertex in the interior of their circumscribing sphere are linked into a list of member tetrahedra. The vertexes, sides, and triangles which comprise the tetrahedra are also placed in member lists.

Identification of an insertion polygon's interior triangles in three-dimensions is analogous to the identification of an insertion polygon's interior sides in two-dimensions. For each member triangle in the insertion polygon a convex cone is defined, with the insertion polygon's generating vertex as the vertex. The vectors emanating to the triangle's vertexes define the extreme vectors. A collection of test vectors is defined which emanate to each member vertex. These test vectors are checked to see if any of them penetrate the member triangle. If any test vector penetrates the member triangle the triangle is in the interior of the insertion polygon and removed from the member list.

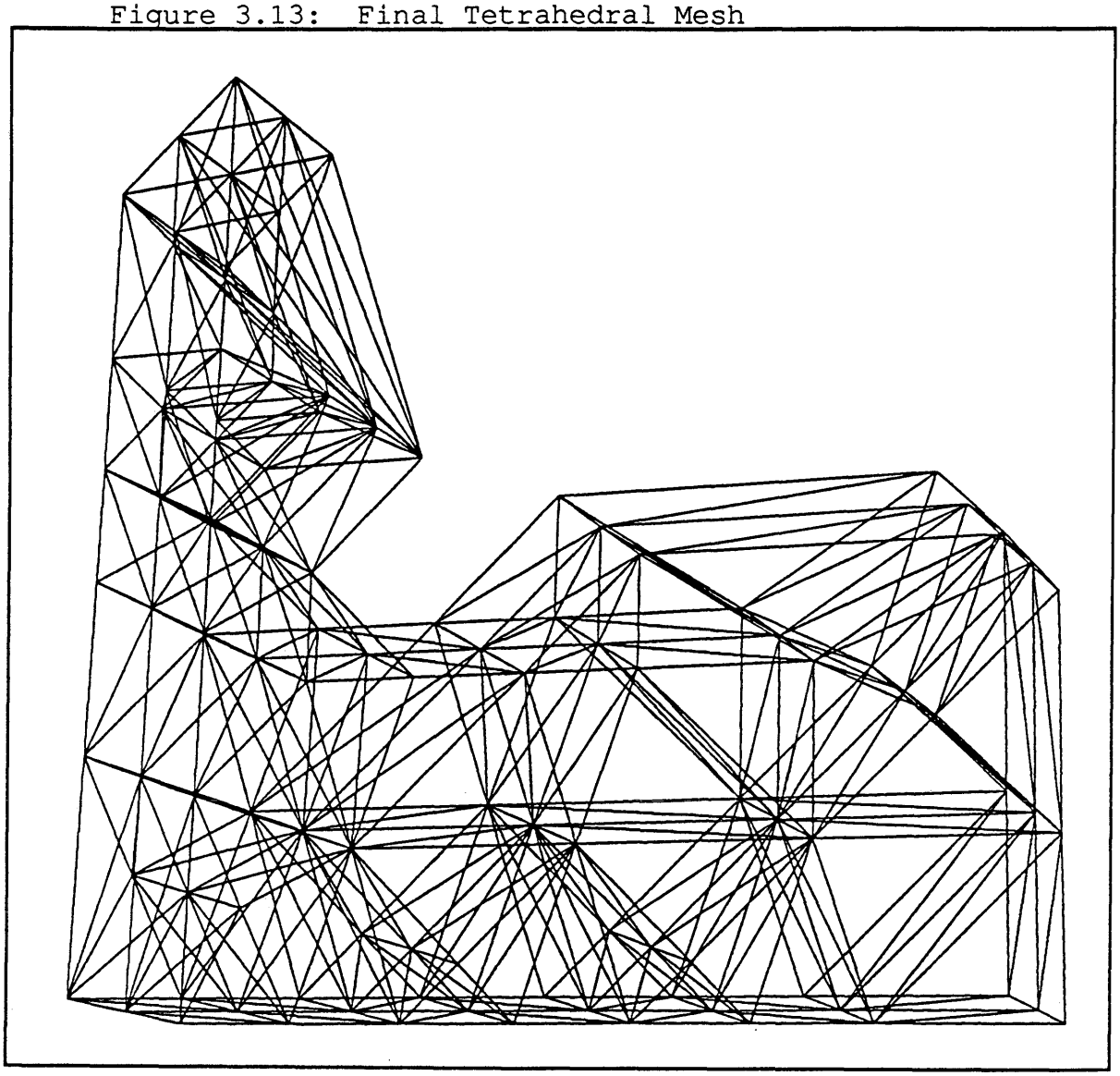
After all interior triangles have been removed the remaining member triangles are connected to the generating vertex to form new tetrahedra. The generating vertex and

each member triangle are connected by creating the required sides and triangles needed to form the tetrahedra.

The inter-plane tetrahedra refinement phase is complete after all vertexes have been tested and refined where required. Completion of this final phase of the combined mesh algorithm results is a collection of well-shaped tetrahedra which completely fill the spatial region. The completion of the combined mesh algorithm provides a tetrahedral mesh which can be used in both static and dynamic spatial representation models.

After the initial mesh is generated (an example is provided in Figure 3.13) the user may desire to add additional vertexes to the mesh. The combined mesh generation algorithm enables the user to integrate these new vertexes without re-calculating the entire mesh. The next section discusses how the combined mesh algorithm integrates additional vertexes into the mesh.

Figure 3.13: Final Tetrahedral Mesh



Integration of New Vertexes Into The Tetrahedral Mesh

After the initial mesh is generated, the user may decide a particular region of the mesh requires greater fidelity. To achieve the greater fidelity a higher density of sub-elements may need to be created. To create the additional sub-elements requires the integration of new vertexes in the specific region. By integrating new vertexes additional sub-elements will be created, and the computational fidelity of the region will be increased.

The integration of new vertexes into the tetrahedral mesh is performed in a manner analogous to that used in integrating non-boundary vertexes into the boundary vertex triangulation. The new vertex's position is used to identify its three-dimensional insertion polygon. Once the insertion polygon is identified its interior triangles are identified and removed, and a new set of tetrahedra are formed with the remaining triangles of the insertion polygon. By adding additional vertexes the user can improve the representation of the spatial region without reconfiguring the entire mesh.

Reconfiguration of Ill-Configured Tetrahedra

As a spatial region is simulated over time, the spatial region will move, and its sub-elements will become deformed. Hydrodynamics models are an example of simulations which require the ability to move the region while still maintaining the sub-elements in a reasonable configuration. The deformation of the sub-elements will eventually introduce numerical inaccuracies into the computations performed. In order to prevent numerical inaccuracies from affecting model results, deformed sub-elements must be reconfigured. The reconfiguration of the sub-elements must be performed in a manner which will not reconfigure the entire mesh. The reconfiguration must be local to the group of sub-elements near the deformed sub-element.

The first problem in sub-element reconfiguration is identification of deformed sub-elements. During this research two types of deformation have been selected for identification: narrow deformation, and fat deformation. Narrow tetrahedra are identified by calculating the ratio of the tetrahedron's inscribed sphere radius to its circumscribed sphere radius. Whenever this ratio becomes smaller than a user defined limit the tetrahedron is

considered to be too narrow and thus requires reconfiguration. Fat tetrahedra are identified by comparing the tetrahedron's circumscribed sphere radius to a user defined upper bound. Whenever a tetrahedron's circumscribed sphere radius exceeds the upper bound the tetrahedron is considered to be too fat and thus requires reconfiguration. Numerous possibilities exist for identifying ill-shaped tetrahedra. The two presented above are simple ones which were relatively easy to determine (after it was determined how to calculate the center of the inscribed sphere, which is provided in Appendix B). Typically, identification of ill-shaped tetrahedra is unique to the problem simulated, and therefore checks must be determined by the developer. Once ill-shaped tetrahedra are identified, the problem becomes how to reconfigure them.

A narrow tetrahedron is reconfigured by collapsing it to a single vertex and reconfiguring the vertex's insertion polygon. Whenever a tetrahedron becomes too narrow the tetrahedron is replaced by a single vertex located at the center of the tetrahedron. The new vertex is then integrated into the tetrahedral mesh in the same manner as described above for integrating new vertexes.

Fat tetrahedra are reconfigured by simply integrating a new vertex, located at the center of the tetrahedron, into

the mesh. The new vertex is integrated using the same approach as described in the integration of new vertexes section.

The two reconfigurations discussed above resolve most reconfiguration problems. Reconfigurations, for whatever the reason, require that either a tetrahedron be eliminated or that it be sub-divided. Therefore the developer need only identify cases where reconfiguration is required and determine whether the reconfiguration should be handled as: a narrow tetrahedron, where the tetrahedron is eliminated and represented by a single point, or as a fat tetrahedron, where the tetrahedron is sub-divided into several tetrahedra by adding a new vertex.

Conclusions

The combined mesh generation algorithm presented in this chapter provides all capabilities needed to simulate either static or dynamic spatial regions. The combined mesh generation algorithm is a fully automated mesh generator which can create a mesh for regularly and irregularly shaped regions. The only interaction required by the user is definition of the vertexes which are used to create the mesh. The combined mesh generation algorithm provides the ability to modify, by addition of new vertexes, local areas of the initial mesh without requiring complete reconfiguration of the entire mesh. The combined mesh generation algorithm also provides the ability to reconfigure local areas of the mesh where deformed tetrahedra are identified.

The combined mesh generation algorithm provides capabilities beyond those of both Fritts' algorithm and Cavendish's algorithm. Fritts' algorithm cannot perform three dimensional mesh generation. Cavendish's algorithm cannot handle irregularly shaped regions. The combined algorithm, using features borrowed from both these algorithms, solves both these problems and provides additional capabilities, primarily in the area of sub-element reconfiguration.

CHAPTER IV

CONCLUSIONS AND FUTURE DIRECTIONS

The problems addressed in this paper provide a significant step toward development of techniques which will improve the efficiency and effectiveness of large-scale physical simulation models. The analysis performed on the partitionable MLG data structure show it to be an extremely effective tool for reducing execution times of large-scale physical simulation models. The development and implementation of the combined mesh generation algorithm support the need for fully automated creation of tetrahedral sub-elements for spatial region representation. The capability to maintain the tetrahedral sub-elements in shapes which will not introduce numerical inaccuracies, using the mesh reconfiguration algorithm, supports modeling of dynamic spatial regions.

Although these algorithms and the partitionable MLG have not been integrated, considered individually they provide significant capabilities. The partitionable MLG data structure, as presented, is capable of data partitioning for concurrent processing in a particle dynamics framework. The mesh generation and reconfiguration algorithms provide the ability to represent static and dynamic spatial regions.

Finite element models can utilize the mesh generation algorithm for finite element generation. Hydrodynamics, and other dynamic spatial representation models, can utilize the mesh generation and reconfiguration algorithms to create the initial sub-element mesh and maintain them as the simulated region moves.

Algorithms developed in this research provide many different directions for additional research. The efficiency of the mesh generator needs evaluation and improvement. Thorough testing of the mesh generator needs to be performed to assure the complete filling of the spatial region. Automated user interfaces need to be developed which simplify the modification of the initial mesh. Vertex insertion algorithms need to be developed which reduce the amount of vertex data required by the user. Each of these problems, relating to mesh generation and reconfiguration, need to be investigated.

After further development of the mesh algorithms is completed, data partitioning with the MLG data structure should be pursued. Integration of the mesh algorithms with the partitionable MLG data structure will enable spatial representation models to take advantage of multi-processor architectures. Continued evaluation of the MLG data structure should be pursued in order to establish its

performance when more complex particle movers and particle update calculations are dependent upon nearest neighbors.

These problems need to be investigated and results need to be communicated to the simulation community. Effective communication will improve the performance and accuracy of large-scale physical simulation models. The improvement of these models will provide a greater body of data from which information relating to physical events can be derived.

REFERENCES CITED

Bate, R. R., Mueller, D. D., and White, J. E., 1971. Fundamentals of Astrodynamics. New York NY, Dover: 1971.

Boris, J. P., 1985. "A Vectorized 'Nearest Neighbors' Algorithm of Order N Using a Monotonic Logical Grid." NRL Memorandum Report 5570, Naval Research Laboratory, Washington DC.

Cavendish, J. C. ,Field, D. A. and Frey, W. H., 1985. "Automatic Three Dimensional Finite Element Mesh Generation." International Journal For Numerical Methods In Engineering 21 (2): 329-347.

Fritts, M. J. and Boris, J. P., 1979. "The Lagrangian Solution of Transient Problems in Hydrodynamics Using a Triangular Mesh." Journal Of Computational Physics 31 (2): 173-213.

Kelly, M. D., 1986. " Application of a Monotonic Data Structure to an Irregular Simulation Grid." Masters Thesis, Colorado School of Mines, Golden CO.

Marlow, W. H., 1978. Mathematics For Operations Research. New York, John Wiley & Sons: 1978.

Moise, E. E. and Downs, F. L., 1964. Geometry. Reading, Massachusetts, Addison-Wesley: 1965.

Watson, D. F., 1981. "Computing the n-Dimensional Delaunay Tessellation With Application to Voronoi Polytopes." Computer Journal 24 (2): 167-172.

Welchons, A. M. and Krickenberger, W. R., 1936. Solid Geometry. 2nd Edition Boston, Ginn and Company: 1950.

BIBLIOGRAPHY

Bate, R. R., Mueller, D. D., and White, J. E., 1971. Fundamentals of Astrodynamics. New York NY, Dover: 1971.

Bell, J. L. and Kelly, M. D., 1988. "A Partitionable Data Structure For Parallel Processing of Irregular Spatial Grids." Submitted for presentation at Supercomputing 88 Conference, Colorado School of Mines, Golden CO.

Boris, J. P., 1985. "A Vectorized 'Nearest Neighbors' Algorithm of Order N Using a Monotonic Logical Grid." NRL Memorandum Report 5570, Naval Research Laboratory, Washington DC.

Bower, A., 1981. "Computing Dirichlet Tessellations", Computer Journal 24 (2): 162-166.

Buneman, O., 1980. "Tetrahedral Finite Elements For Interpolation." Society for Industrial and Applied Mathematics (SIAM) Journal of Scientific and Statistical Computing, 1 (2): 223-248.

Cavendish, J. C. ,Field, D. A. and Frey, W. H., 1985. "Automatic Three Dimensional Finite Element Mesh Generation." International Journal For Numerical Methods In Engineering 21 (2): 329-347.

Fritts, M., 1979. "Three Dimensional Algorithms for Grid Restructuring In Free Lagrangian Calculations." Journal Of Computational Physics 31 (2): 173-213.

Kelly, M. D., 1986. "Application of a Monotonic Data Structure to an Irregular Simulation Grid." Masters Thesis, Colorado School of Mines, Golden CO.

Lo, S. H., 1985. "A New Mesh Generation Scheme for Arbitrary Planar Domains." International Journal For Numerical Methods In Engineering 21 (8): 1403-1426.

Marlow, W. H., 1978. Mathematics For Operations Research. New York, John Wiley & Sons: 1978.

Moise, E. E. and Downs, F. L., 1964. Geometry. Reading, Massachusetts, Addison-Wesley: 1965.

Moulton, F. R., 1914. An Introduction To Celestial Mechanics. 2nd ed. Mineola NY, Dover: 1970.

Watson, D. F., 1981. "Computing the n-Dimensional Delaunay Tessellation With Application to Voronoi Polytopes." Computer Journal 24 (2): 167-172.

Welchons, A. M. and Krickenberger, W. R., 1936. Solid Geometry. 2nd Edition Boston, Ginn and Company: 1950.

Yerry, M. A. and Shephard, M. S., 1984. "Automatic Three Dimensional Mesh Generation By The Modified-Octree Technique." International Journal For Numerical Methods In Engineering 20 (11): 1965-1990.

APPENDIX A

CODE FOR BORIS' VECTOR SORT

This appendix presents the Pascal code used to perform Boris' vector sort. The routine used to perform the sort is called SortMonoLogicGrid (Sort Monotonic Logical Grid). The parameters passed to the routine include: GV (Global Variable Record) which contains the MLG.array, StartIndex, which is a three element array of integer values which defines the indices' start values, and EndIndex, which is a three element array of integer which defines the indices' end values. The StartIndex and EndIndex values are used by the child processes to identify their assigned subsets of the MLG data structure. The routine consists of one local procedure, SwapNodes, which determines whether or not two particles are out of monotone order. If the particles are out of order the SwapNodes procedure reorders the particles' cell locations.

```
procedure SortMonoLogicGrid(
              var GV                          : GlobalVarRec;
              StartIndex                      : IntegerVec;
              EndIndex                        : IntegerVec;

var
      SwapMade                                : boolean;
      XIndex                                  : integer;
      YIndex                                  : integer;
      ZIndex                                  : integer;
      SortIndex                               : integer;

      procedure SwapNodes(
                    X1                        : integer;
                    Y1                        : integer;
                    Z1                        : integer;
                    X2                        : integer;
                    Y2                        : integer;
                    Z2                        : integer;
                    Index                     : integer);
      var
            Temp          integer;
            TestVal1      : real;
            TestVal2      : real;
      begin
        with GV do
        begin
          TestVal1 := Node[MLG[X1,Y1,Z1]].CurrPos[Index];
          TestVal2 := Node[MLG[X2,Y2,Z2]].CurrPos[Index];
          if TestVal1>TestVal2 then
          begin
            SwapNode := true;
            Temp  := MLG[X1,Y1,Z1];
            MLG[X1,Y1,Z1] := MLG[X2,Y2,Z2];
            MLG[X2,Y2,Z2] := Temp;
        end;
      end;
```

The main routine of the SortMonoLogicGrid procedure sorts the particles based on the user defined index order. For example, the use may wish the Z-index to be sorted first, the Y-index to be sorted second, and the X-index to be sorted last. This sort would result in an MLGSortIndex value of 3, 2 and 1 respectively for the three iterations of the FOR loop. The repeat loop is executed until all particles in the MLG data structure are checked to assure none are out of monotone order, thus requiring no swaps.

```
begin  { SortMonoLogicGrid }
  with GV do
  begin
    repeat
      SwapMade  := false;
      for SortIndex := 1 to 3 do
       case MLGSortIndex[SortIndex] of
      1 :
         for ZIndex := StartIndex[3] to EndIndex[3] do
         for YIndex := StartIndex[2] to EndIndex[2] do
         for XIndex := StartIndex[1] to EndIndex[1] -1 do
              SwapNode(XIndex,YIndex,ZIndex,
                XIndex+1,YIndex,ZIndex,1);
      2 :
         for ZIndex := StartIndex[3] to EndIndex[3] do
         for XIndex := StartIndex[1] to EndIndex[1]  do
         for YIndex := StartIndex[2] to EndIndex[2] -1 do
              SwapNode(XIndex,YIndex,ZIndex,
                XIndex+1,YIndex,ZIndex,2);
      3 :
         for XIndex := StartIndex[1] to EndIndex[1]  do
         for YIndex := StartIndex[2] to EndIndex[2] do
         for ZIndex := StartIndex[3] to EndIndex[3] -1 do
              SwapNode(XIndex,YIndex,ZIndex,
                XIndex+1,YIndex,ZIndex,3);
     end;
    until not SwapMade;
  end;
end;  { SortMonoLogicGrid }
```

APPENDIX B

CIRCLE AND SPHERE CALCULATIONS

Triangle Circumscribing Circle

A circle which circumscribes a triangle intersects each of the triangle's vertices. It can be shown [Moise and Downs 1964] that the circumscribing circle is unique and its center is defined by the intersection of the perpendicular bisectors of any two of the triangle's sides. In three dimensions, the center of the circumscribing circle is defined by the intersection of three planes: the plane in which the triangle's vertices lie and two planes which bisect sides of the triangle (i.e. have as its normal vector one of the triangle's sides and passes through that sides midpoint).

Define the vertices of a triangle with the vectors v_1, v_2, v_3. The bisecting plane of the side connecting vertices v_1 and v_2 is defined as:

$$(1) \quad P_{12} = \left\{ x \mid \left[x - \frac{1}{2} (v_2 + v_1) \right]^T (v_2 - v_1) = 0 \right\},$$

which can be rewritten as

$$(2)\quad P_{12} = \left\{ x \mid x^T(v_2-v_1) = \frac{1}{2}(v_2+v_1)^T(v_2-v_1) \right\}.$$

Similarly, we can obtain the bisecting plane of the side connecting vertices v_1 and v_3 :

$$(3)\quad P_{13} = \left\{ x \mid x^T(v_3-v_1) = \frac{1}{2}(v_3+v_1)^T(v_3-v_1) \right\}.$$

The plane which contains all of the triangle's vertices can be defined as:

$$(4)\quad P_{123} = \left\{ x \mid x = c_1(v_2-v_1) + c_2(v_3-v_1) + v_1,\ c1, c2 \in \Re \right\}.$$

Let x^* be the intersection of the three planes, thus x^* must lie in each plane and therefore the following three equations must hold:

$$(5)\quad x^* = c_1(v_2-v_1) + c_2(v_3-v_1) + v_1$$
$$(6)\quad x^{*T}(v_2-v_1) = \frac{1}{2}(v_2+v_1)^T(v_2-v_1)$$
$$(7)\quad x^{*T}(v_3-v_1) = \frac{1}{2}(v_3+v_1)^T(v_3-v_1).$$

Taking the dot product of equation (5) with (v_2-v_1) results in:

$$(8)\quad (v_2-v_1)^T x^* = (v_2-v_1)^T\left[c_1(v_2-v_1) + c_2(v_3-v_1) + v_1\right]$$

Replacing the left hand side of equation (8) with the right hand side of equation (6) results in:

$$(9)\quad \frac{1}{2}(v_2+v_1)^T(v_2-v_1) = (v_2-v_1)^T\left[c_1(v_2-v_1) + c_2(v_3-v_1) + v_1\right]$$

which can be rewritten in the form:

$$(10)\quad \frac{1}{2}(v_2-v_1)^T(v_2-v_1) = (v_2-v_1)^T\left[c_1(v_2-v_1) + c_2(v_3-v_1)\right] .$$

Similarly, by taking the dot product of equation (5) with (v_3-v_1) and using equation (7) results in:

$$(11)\quad \frac{1}{2}(v_3-v_1)^T(v_3-v_1) = (v_3-v_1)^T\left[c_1(v_2-v_1) + c_2(v_3-v_1)\right] .$$

Simultaneous solution of equations (10) and (11) will provide the scalar multipliers c_1 and c_2 which in conjunction with equation (5) will define the center of the circumscribing circle. In matrix notation the calculation is:

$$(12)\quad \begin{bmatrix} (v_2-v_1)^T(v_2-v_1) & (v_2-v_1)^T(v_3-v_1) \\ (v_3-v_1)^T(v_2-v_1) & (v_3-v_1)^T(v_3-v_1) \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \frac{1}{2}\begin{bmatrix} (v_2-v_1)^T(v_2-v_1) \\ (v_3-v_1)^T(v_3-v_1) \end{bmatrix}$$

where a unique solution exists if the angle between (v_3-v_1) and (v_2-v_1) is not 0 or 180 degrees (i.e. the vertices are non-colinear). The radius of the circumscribing circle can be found by calculating the distance from the center of the circle to any of the triangle's vertices.

Triangle Inscribed Circle

A circle which is inscribed within a triangle must be tangent to each of the triangle's sides. The center of the inscribed circle is the intersection of the bisectors of the angles defined by the sides of the triangle [Moise and Downs 1964]. In three dimensions, the center of the inscribing circle is defined by the intersection of three planes, the plane in which the triangle's vertices lie, and two additional planes which each bisect one of the angles formed by the triangle's sides.

Define the vertices of the triangle with the vectors v_1, v_2, v_3. The plane which bisects the angle formed at vertice 1 is defined as:

$$(13)\quad P_1 = \left\{ x \mid \frac{(v_3-v_1)^T(x-v_1)}{|v_3-v_1|} = \frac{(v_2-v_1)^T(x-v_1)}{|v_2-v_1|} \right\}$$

where $|\cdot|$ represents the norm of the vector. Equation (13) can be rewritten as:

$$(14)\quad P_1 = \left\{ x \mid \left[\frac{(v_3-v_1)}{|v_3-v_1|} - \frac{(v_2-v_1)}{|v_2-v_1|} \right]^T (x-v_1) = 0 \right\}.$$

Similarly, for the plane which bisects the angle formed at vertex v_2 we have:

$$(15)\ P_2 = \left\{ x \mid \left[\frac{(v_3-v_2)}{|v_3-v_2|} - \frac{(v_1-v_2)}{|v_1-v_2|} \right]^T (x-v_2) = 0 \right\}.$$

The plane which contains all of the triangle's vertices can be defined as:

$$(16)\ P_{123} = \left\{ x \mid x = c_1(v_2-v_1) + c_2(v_3-v_1) + v_1,\ c1,c2 \in \Re \right\}.$$

Define u_{ij} to be:

$$(17)\ u_{ij} = \frac{(v_i-v_j)}{|v_i-v_j|}$$

and let x^* be the intersection of the three planes described above. Then x^* must lie in each of the three planes and the following three equations must hold:

$$(18)\ x^* = c_1(v_2-v_1) + c_2(v_3-v_1) + v_1$$

$$(19)\ x^{*T}(u_{31}-u_{21}) = v_1^T(u_{31}-u_{21})$$

$$(20)\ x^{*T}(u_{32}-u_{12}) = v_2^T(u_{32}-u_{12})$$

Taking the dot product of equation (18) with $(u_{31}-u_{21})$ and replacing the left hand side with the right hand side of

equation (19) results in:

$$(21)\quad v_1^T(u_{31}-u_{21}) = \left[c_1(v_2-v_1) + c_2(v_3-v_1) + v_1\right]^T(u_{31}-u_{21}).$$

Factoring v_1 out of the right hand side of equation (21) and subtracting from both sides results in:

$$(22)\quad \left[c_1(v_2-v_1) + c_2(v_3-v_1)\right]^T(u_{31}-u_{21}) = 0.$$

Similarly, taking the dot product of equation (18) with u_{32} and replacing the left hand side with the right hand side of equation (20) results in:

$$(23)\quad \left[c_1(v_2-v_1) + c_2(v_3-v_1)\right]^T(u_{32}-u_{12}) = (v_2-v_1)^T(u_{32}-u_{12}).$$

Simultaneously solving equations (22) and (23) for the scalar multipliers and substituting the results into equation (18) will obtain the center of the inscribed circle. In matrix notation this calculation is:

$$(24)\quad \begin{bmatrix} (v_2-v_1)^T(u_{31}-u_{21}) & (v_3-v_1)^T(u_{31}-u_{21}) \\ (v_2-v_1)^T(u_{32}-u_{12}) & (v_3-v_1)^T(u_{32}-u_{12}) \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 0 \\ (v_2-v_1)^T(u_{32}-u_{12}) \end{bmatrix}$$

The radius of the inscribed circle can be calculated by finding the minimum distance from the center of the circle to any of the triangle's sides.

Tetrahedron Circumscribing Sphere

A sphere which circumscribes a tetrahedron intersects each of the tetrahedron's vertices. It can be shown [Welchons and Krikenberger 1936] that the circumscribing sphere is unique and its center is defined by the intersection of three planes which are perpendicular bisectors of the tetrahedron's sides.

Define the vertices of a tetrahedron with the vectors v_1, v_2, v_3, v_4 . The plane which bisects the side connecting vertices v_i and v_j is defined as:

$$(25)\ P_{ij} = \left\{ x \mid \left[x-\frac{1}{2}(v_i+v_j)\right]^T (v_j-v_i) = 0 \right\}.$$

Letting,

$$i = 1 \text{ and } j \in (2,3,4)$$

and using a rewritten form of equation (25) results in the following set of simultaneous equations (in matrix notation):

$$(26)\ \begin{bmatrix} (v_2-v_1)^T \\ (v_3-v_1)^T \\ (v_4-v_1)^T \end{bmatrix} x^* = \frac{1}{2}\begin{bmatrix} (v_2+v_1)^T(v_2-v_1) \\ (v_3+v_1)^T(v_3-v_1) \\ (v_4+v_1)^T(v_4-v_1) \end{bmatrix}$$

where x^* is the center of the circumscribing sphere. The matrix on the left hand side of the equation is guaranteed to be non-singular if the tetrahedron's vertices are non-coplanar, since the determinant of the matrix represents the volume of the parallelpiped formed by the vectors (i.e. a zero volume parallelpiped implies the tetrahedron has zero volume and thus the vertices are coplanar) . The radius of the sphere can be found by calculating the distance from the center of the sphere to any of the tetrahedron's vertices.

Tetrahedron Inscribed Sphere

A sphere which is inscribed within a tetrahedron must be tangent to each of the tetrahedron's faces. It can be shown [Welchons and Krikenberger 1936] that the center of the inscribed sphere is the intersection of three (of the six possible) planes which bisect the dihedral angles generated by the tetrahedron's faces.

Define the vertices of a tetrahedron with the vectors v_1, v_2, v_3, v_4 . The plane which bisects the dihedral angle formed by the side connecting vertices v_i and v_j is defined as:

$$(27)\quad P_{ij} = \left\{ x \mid (x-v_i)^T a_{ijk} = (x-v_i)^T a_{ijl} \right\}$$

where v_k and v_l are the tetrahedron's remaining vertices, and a_{ijk} is a unit vector defined as follows:

$$(28)\quad a_{ijk} = \frac{(v_{ki} - (v_{ki} u_{ji}) u_{ji})}{\left| v_{ki} - (v_{ki}^T u_{ji}) u_{ji} \right|}$$

where

$$(29)\quad u_{ij} = \frac{(v_i - v_j)}{\left| v_i - v_j \right|} .$$

Selecting pairs of vertices and substituting into equation (27) results in three simultaneous equations as follows:

$$(30)\quad \begin{bmatrix} (a_{312}-a_{314})^T \\ (a_{231}-a_{234})^T \\ (a_{142}-a_{143})^T \end{bmatrix} x^* = \begin{bmatrix} v_3^T(a_{312}-a_{314}) \\ v_2^T(a_{231}-a_{234}) \\ v_1^T(a_{142}-a_{143}) \end{bmatrix}$$

where x^* is the center of the inscribed sphere. The radius of the sphere can be found by calculating the minimum distance from the center of the sphere to any of the tetrahedron's faces.