VISUALIZATION OF LATTICE FIELDS

by

David A. Jablonski

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

BACHELOR OF SCIENCE IN ELECTRICAL SCIENCE AND ENGINEERING

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1989

Signature of Author [illegible]
Department of Electrical Engineering and Computer Science
June 5, 1989

Certified by [illegible]
Professor Sanjoy K. Mitter
Thesis Supervisor

Accepted by ______________________
Leonard A. Gould
Chairman, Undergraduate Thesis Committee

VISUALIZATION OF LATTICE FIELDS

by

David A. Jablonski

Submitted to the Department of Electrical Engineering and Computer Science on June 5, 1989 in partial fulfillment of the requirements for the degree of Bachelor of Science in Electrical Science and Engineering.

Abstract

The Laboratory of Information and Decision Systems is involved in the development of a number of models of probabilistic systems based on three dimensional lattice configurations. Proper development of these models involves computer simulation to test the theoretical structure. This thesis examines some of the problems associated with visualizing the resulting lattice fields in a form that is both understandable and useful. Software was developed to render and provide some analysis of lattice fields.

Thesis Supervisor:	Professor Sanjoy K. Mitter
Title:	Professor of Electrical Engineering and Computer Science

Dedication

I would like to thank Professor Sanjoy Mitter for his guidance, Dr. Peter Doerschuk for his unlimited assistance and my mother and father for all their loving support and encouragement.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Objective

Developing a realistic probabilistic model of a lattice field is a difficult matter. Aside from whatever theoretical analysis is necessary, it is often helpful if not essential to develop computer simulations to check the validity of the model. The goal of this thesis is to provide a set of tools that will assist in the development, analysis and verification of these models.

A major problem with the development of models of probabilistic systems greater then two dimensions is the difficulty in interpreting and gaining insight from the resulting data. Reading lists of data points and graphing two dimensional slices of the results are helpful but do not reflect the true dimensionality of the results. Often symmetries and correspondences are misinterpreted or missed altogether. By graphing the results of these models using the multidimensional rendering capabilities of modern supercomputers, it is possible to reveal relationships that were missed and increase the reliability of the constructed model.

The software presented in this thesis provides a number of utilities that are helpful in the algorithm development of computer simulations of lattice field models. In addition to presenting a method for visualizing simulation results, techniques are offered that allow the graphical representation of results while they are being generated by the simulation. This feature is useful in analyzing the way the simulation actually generates its results. The software presented here is discussed in relation to two specific classes of lattice fields; however, in a broader sense this thesis is concerned with methods of visualizing and rendering understandable images out of complex structures.

1.2 Sparsely Occupied vs. Densely Occupied Lattice Fields

Two classes of lattice fields were examined as this software was created. While there are similarities between the two, each presents some unique features that require special attention. Through the analysis of the requirements of these two simulations, visualization techniques were developed that could be used for a broad range of models.

Both classes of models are based on configurations of three dimensional lattice coordinates. Through various techniques, each model manipulates the sites in the lattice to arrive at some optimal solution. The fundamental difference between the two is the number of lattice sites that are occupied in the final solution to the simulation.

Models which contain only a few occupied sites are referred to as sparse models. These models are usually concerned with the interactions between specific sites, or the effect of particular sites on the overall model. Consequently, successful visualization of these models involves techniques for rendering individual sites and their relationships to other sites in the model.

Models which contain many occupied sites (50% or more) are referred to as dense models. Usually these models simulate very large lattices, and are therefore concerned with the effects of groups of sites rather then each individual site. Visualization techniques used for this class of models involve definition and rendering of clusters of sites and representation of the interaction of these clusters with the overall model.

Chapter 2

Background

2.1 The Ardent Computer

The software for this thesis was developed is the Titan II supercomputer produced by the Ardent Corporation. The Titan is capable of running 4 CPU's with 64 Megabytes of addressable memory. Tests have shown peak performance speeds of 16 MIPS and the capability to render 400,000 3D vectors per second. [Titan Architecture 88] The particular machine used to create this software package contained 2 CPUs with 64 Megabytes of addressable RAM. This provided the capability of producing realtime rotations and translations of relatively complex models.

The graphics software designed to run on the Ardent is the Dore Graphics language. Dore is an object oriented language accessible through FORTRAN and C that handles most of the overhead necessary in rendering graphics images. Models are defined as objects and given to Dore to render. Keyboard, mouse and knob input can be configured to provide control over Dore's rendering parameters. Additional software was created on the M.I.T.'s Athena computer network to allow reformatting and printing of the graphics images generated on the Ardent.

2.2 Coordinate Systems

Each lattice field model may be associated with a different coordinate system. The sparsely occupied lattice field example that will be investigated is a representation of organic molecules. The sample data that was collected lies at real valued locations on a lattice with an angle of 94 degrees. For simplicity, this data has been discretized and translated to lie at integer valued location on a lattice with all angles equal to 90 degrees. While this simplifies the code and slightly increases execution speed, some detail is lost by truncating the actual site locations to integer coordinates. Figure 2-1. shows an example of the loss of information. Provisions have been made for the display of models with floating point coordinates to allow more realistic representations of the models, but this code does not account for the inherent symmetries in non-cartesian coordinate systems.

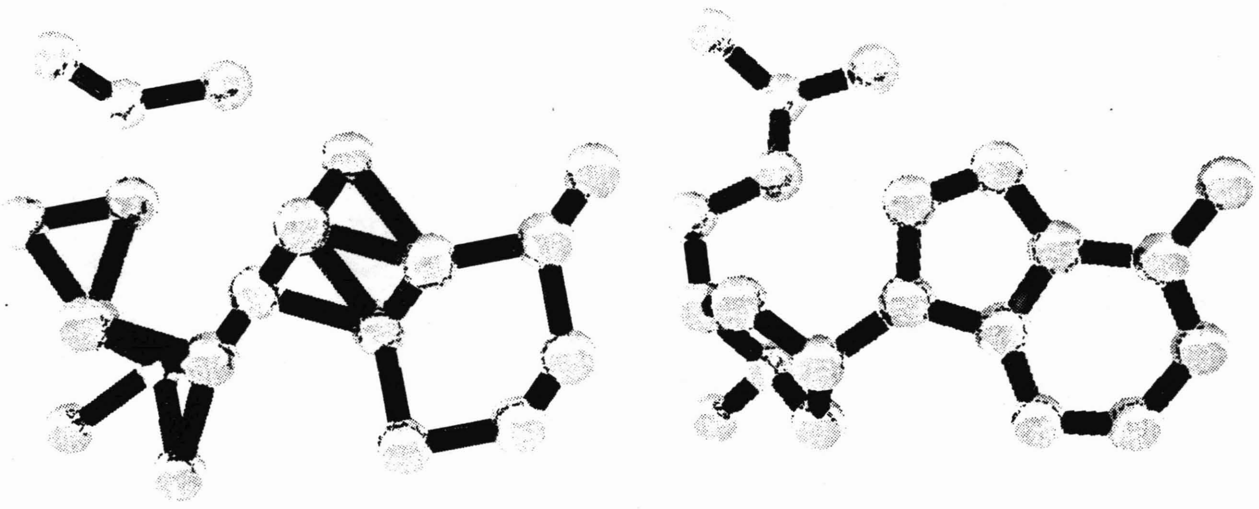

Figure 2-1: Molecule in integer coordinates (left), floating point coordinates (right)

The densely occupied lattice field example used in this thesis can be completely described in cartesian coordinates but contains two interlocking coordinate grids. Each Lattice site can be represented by an integer coordinates in cartesian space. The coordinates $\{X_{lm}, Y_{lm}, Z_{lm}\}$ which describe this location are referred to as lattice coordinates. Additionally, the dense model contains representations of the surfaces which bound clusters of similar sites. These surfaces lie between lattice coordinates and will be referred to by their locations in surface coordinates $\{X_{sm}, Y_{sm}, Z_{sm}\}$. The relationship between lattice and surface coordinates is represented below.

$$0 < X_{lm}, Y_{lm}, Z_{lm} \leq m \quad ; \quad 0 < X_{sm}, Y_{sm}, Z_{sm} \leq m+1$$

$$X_{sm} = X_{l(m-1)} + 0.5 = X_{lm} - 0.5$$
$$Y_{sm} = Y_{l(m-1)} + 0.5 = Y_{lm} - 0.5$$
$$Z_{sm} = Z_{l(m-1)} + 0.5 = Z_{lm} - 0.5$$

Each lattice coordinate has two corresponding surface coordinates that bound it- located at $m \pm 0.5$ in lattice space and at m and $m+1$ in surface space.

2.3 Terminology

There are a number of terms used throughout this thesis that require some definition or explanation. These terms are not necessarily used universally, but clear explanation of each should help to make this document understandable.

site - A particular location within a lattice. Each site can be represented by the integer vector {x, y, z} where each variable represents a coordinate in lattice coordinates.

nearest neighbor - A site which lies closest to another site in the dense model. The distance between sites is the linear distance between the centers of the two sites. In cartesian space each site has six nearest neighbors- one in the positive and negative directions of each of the x, y, and z directions.

cluster - A group of sites within a dense model in which each member is a nearest neighbor to at least one other member of the group.

occupied site - Any site with probability greater then 0.0 in the sparse model. Any site containing an atom with spin up in the dense model.

unoccupied site - Any site with probability equal to 0.0 in the sparse model. Any site containing an atom with spin down in the dense model.

Chapter 3

Sparsely Occupied Lattice Models

3.1 Modeling of Biological Molecules

The model that was chosen to examine the effectiveness of the sparse model visualization is an excellent example of many of the characteristics that make the sparse models different from dense models. Given the diffraction data from x-ray crystallography, this model attempts to generate the original crystal structure. [Doerschuk 88]

Each site in the model's lattice represents a possible location for an atom in the crystal structure. Each site has a probability associated with it- the probability that an atom is located at that site. Factors such as nuclear repulsion between close atoms and covalent attraction between distant atoms are used to generate that probability of each site's occupancy.

The visualization goals for this particular model is the accurate representation of the relationships between the sites that fall within a given probability range. However, this example is an excellent illustration of generalized sparsely occupied lattice fields. Each occupied site can be thought of as an atom and successful visualization will achieve recognizable relationships between specific sites. In this case, the hexagonal and pentagonal rings characteristic to organic molecules are the important relationship.

3.2 Assisting Visualization

The actual representation of the sparse model requires far less effort then the task of making the representation understandable and useful to the user. Since there are relatively few occupied sites and the individual sites are the focus of the investigation, it is a relatively simple task to represent each site as a primitive object. For compactness and added depth perception, each site is represented by a colored sphere. The location of the sphere represents the location of the site within the lattice field, and the color represents the probability of that site being occupied. For simplicity only sites with non-zero probability are visible.

This simple representation of the model has a number of immediate benefits. With proper perspective calculated, the position of each occupied site can be discerned by its location(indicating x and y position) and its size (indicating depth or z position). In addition, those sites whose probability of occupancy is high appear redder while those sites whose probability of occupancy is low appear greener.

Yet there are a number of difficulties with this visualization of the image. As the number of occupied sites increases, the computer requires much more time to render the image when it is translated. Furthermore, when there are more then 20 or 30 occupied sites on the screen, it becomes difficult to clearly see the position of the individual sites, and the relationships between sites.

In order to overcome some of these problems, and to further enhance the information that can be perceived through the visual representation of the model, the following methods were developed.

3.2.1 Visualization Planes

The first visualization technique involves the selective isolation of sections of the model for examination and manipulation. Often it is not necessary to look at the entire model to determine the necessary results. The user has the option of selecting clipping planes that will isolate sections of the model for viewing. By selecting the distance between the planes it is possible to look at the entire model, or only a thin slice of it. Through the use of the knob box, these clipping planes can be translated through the model to render successive slices of the model. This simple technique both increases rendering speed and limits the number of sites displayed, thereby reducing confusion.

3.2.2 Probability Thresholding

While visualization plane techniques allow the user to isolate data according to their spatial dimensions, probability thresholding allows the isolation of data according to the probabilities associated with each site. As before, it may only be necessary to view those sites with probability of 1/2 and above. By selectively rendering only those sites with the desired probability, speed is increased and the image is made clearer. Once again, the user may specify the maximum and minimum probabilities that are to be rendered and the knob box may be used to dynamically alter the minimum probability.

Since probability is represented by color, care was taken to use the full range of color values. Color is controlled by specifying the percentage of red, green and blue that is used to display each object. At each site the color of the rendered sphere is calculated with the following equations:

$$Red = \frac{P(x) - P_{min}}{1 - P_{min}}$$

$$Green = 1 - \frac{P(x) - P_{min}}{1 - P_{min}}$$

$$Blue = 0$$

Where $P(x)$ is the probability associated with the site and P_{min} is the minimum visible probability. Thus, even as the probability range is modified, those sites with the maximum probability will remain pure red, and those sites with the minimum probability will be pure green. With this technique, sites of similar probability can be easily detected.

3.2.3 Connectors

In addition to selectively displaying portions of a model, it is useful to determine the relationship between individual sites. In the case of the organic crystal model, where the atoms are held together with covalent bonds, the natural repulsive force between the atoms will limit the proximity of atoms, and the natural attractive forces between the atoms will limit the farthest distance they can be separated while remaining part of the same molecule.

These interactions can be represented on the screen by visual links between the sites. These links are represented by cylinders that join the centers of each pair of sites. Once again, maximum and minimum values for the length of these links can be specified. In order to reduce complexity and increase speed, those links that are longer than the maximum are not rendered. Those links that are within the thresholds are colored blue, and those links which are below the minimum threshold are colored yellow. By appropriate choice of maximum and minimum lengths it becomes a simple matter to determine which sites are too close or to far away from other sites to properly interact in the model. Additionally, underlying symmetries such as the hexagonal ring structure discussed earlier become visible.

3.2.4 Dynamic Models

Finally, it is often beneficial to watch the development of a model as it calculates new sites. In order to facilitate this, a program has been written that accepts data from a file or active simulation and dynamically alters the model that is displayed on the screen.

As a simulation is run, it can be configured to print each site that is examined and the success or failure of the trial. The display software can then read that output data a few sites at a time and display the resulting effect on the model. If a site was examined but not altered, it briefly appears blue. If a site was altered, it briefly appears yellow then returns to the color determined by its new probability value. By examining this visual representation of the output of a simulation for a few iterations, it is not only possible to determine how the simulation is parsing through the model, but it is possible to see the model evolve from its initial to its final state.

3.3 Information and Statistics

The image that is displayed on the screen is the result of the underlying simulation model and the interactive input of the user. The are a number of statistics associated with the rendered model that are available to the user. The table below lists each parameter and its definition.

Sparse Model Statistics	
Parameter	Definition
file0	Current model displayed in left window
file0	Current model displayed in right window
dist_mx	Maximum bond length
dist_mn	Minimum bond length
planeat	Current position of front clipping plane
dev	Current distance between clipping planes
prob_min	Minimum probability displayed
MAXX,Y,Z	Lattice field dimensions

Chapter 4

Densely Occupied Lattice Fields

4.1 The Ising Model

The second class of lattice fields that can be rendered through the use of this software are those models which contain a large number of occupied sites. These models form a different class because they do not require the visualization of each of the specific sites. Instead, the emphasis is usually placed on clusters of sites, and the interaction between these clusters.

The sample model that was used to test this software and for which many of the visualization aspects were developed is a three dimensional extension of the Ising model of ferromagnetism. [Huang 63] A simple explanation of the Ising model will add some clarity to the overall visualization goals.

Certain metals exhibit a phenomenon known as ferromagnetism in which the spins of some percentage of the atoms in the metal become spontaneously polarized, giving rise to observable magnetic fields throughout the material. This phenomenon occurs only when the temperature of the material falls below a characteristic temperature known as the Curie temperature. The Ising model was developed to simulate the structure of these ferromagnetic materials.

The model consists of a n-dimensional lattice of N fixed sites arranged in some periodic or symmetric fashion. As in the case of the organic crystal model, simple cubic lattice fields are examined for simplicity. Each site has associated with it, a spin variable s_i which is either 0 or 1 representing spin up and spin down respectively; and a set of nearest neighbors $<ij>$. The energy of the system for a given configuration is defined to be

$$E\{s_i\} = -\sum_{<ij>} \varepsilon_{ij} s_i s_j - B \sum_{i=1}^{N} s_i$$

where the interaction energy ε_{ij} and the external magnetic field B are given constants. The simulation consists of an algorithm that minimizes the energy of the system. [Huang 63]

The Ising model gives a very accurate treatment of ferromagnetism in statistical mechanics. It is desirable to see the actual results of the simulation given different parameters. This thesis offers a number of visualization methods that allow the rendering and analysis of such models.

4.2 Grouping

One of the goals of this software is to assist the user in visualization of extremely large lattices. It is desirable to study lattices with hundreds of thousands or even millions of sites. Yet rendering lattices of this size is a time consuming task for even the most powerful supercomputers. Therefore it is necessary to develop methods for approximating the data that will reduce the rendering time but will not significantly alter the data that is displayed.

4.2.1 The Grouping Algorithm

The grouping algorithm is not optimized for speed but does achieve correct results for every configuration with which it was tested. It begins by examining each site to determine if it has already been included in a cluster. If the site is not a member of another cluster, it is inserted into a new cluster.

Next, each of this site's nearest neighbors is checked to see it they have the same spin. Each time a nearest neighbor is found with the same spin, it is added to the cluster and its nearest neighbors are tested. Care is taken that no site is added to more then one cluster, or to the same cluster more then once. After every member of a cluster has had all of its nearest neighbors tested, the cluster is closed and the program returns to parsing each site to see if it is a member of a cluster.

4.3 Approximations

After the initial clusters have been formed, they tend to have a number inconsistencies. Since the models are usually very large, representing the finest detail of every cluster is little better then representing the individual sites themselves. Often, the model is large enough that the small effects caused by a few sites can be completely ignored. Sometimes there is enough error in the model itself that looking at the behavior of a small clusters of sites does not necessarily represent realistic results.

In order to further increase rendering speed and to eliminate the small perturbations caused by surface detail, the software includes two techniques for rendering approximated or smoothed data. Judicious use of these algorithms can greatly enhance the rendered image as well as the dynamic rendering time.

4.3.1 Coarse Approximation

The first technique used to generate an approximation is similar to digitizing the information in three dimensions with a user controlled step size. In its simplest form, each data point is sampled individually to determine whether the site is occupied.

Using this technique, the user controls the size of the approximation and the threshold percentage that controls the spin. For example the data may be examined as 2x2x2 groups for a total of eight lattice sites per rendered site. If the threshold is set to 50% then the rendering site would be occupied only if four or more of the lattice sites in the group were occupied.

By specifying a large approximation, the entire lattice could be tested to create a single rendering site that represents the percentage of occupied sites in the lattice. By specifying a small approximation each lattice site could be examined to create a rendered site that represents the occupancy of that lattice site.

This technique is particularly useful in determining which sections of the lattice contain the largest number of occupied sites. A particularly large lattice may be approximated by a few rendered sites for easy manipulation, then the resolution may be reduced to allow the closer study of a particular section of interest.

4.3.2 Group Size Approximation

A second technique used to develop smoother, more quickly renderable lattice fields, involves limiting the size of individual clusters. In a simulated lattice field containing millions of sites, it is doubtful that each lattice site is in its correct state. Furthermore, in a large cluster of occupied sites, a few isolated unoccupied sites can greatly complicate the rendering algorithms, thereby adding unnecessary delay. Therefore it is often beneficial to ignore small isolated clusters altogether.

This software package allows the user to select a minimum size for each of the clusters that will be displayed. Once the lattice has initially been broken into clusters, a second routine parses each cluster and determines whether it is larger then the the minimum size. If the cluster contains fewer then the minimum number of sites, the spin of each site in that cluster is inverted. Finally, the program returns to the grouping algorithm and regroups the clusters within the lattice. While the number of occupied sites may not have changed significantly, the number of groups will usually decrease dramatically.

4.4 Rendering Clusters

Once the preliminary approximation and grouping is completed, we are at last faced with the problem of displaying the lattice field. The simplest technique would be to render each occupied site as a sphere. But this technique has a number of drawbacks. As was pointed out in the previous section, rendering individual spheres for each of hundreds of thousands of data points requires far more time then is realistically allowable. Even more importantly, since the final product is viewed on a two dimensional screen, even carefully chosen perspective drawing does little to eliminate confusion when viewing complex figures. Thus arrays of thousands of spheres are far too complex to yield any useful information.

Instead, we return to the notion of our data as clusters of information. Since it is more useful to understand the relationship between clusters as opposed to the individual sites themselves, the software uses a number of techniques for rendering the clusters as individual objects. Each of the following algorithms offers unique advantages and disadvantages and are suitable for different applications.

4.4.1 Primitive enclosures

The first technique is based on the traditional graphics technique of primitive construction. The ideal surface is one that retains all the minutest surface variations of the group, but is smooth enough for the eye to discern its boundaries. A group of sites that form a cube is most closely approximated by a cube with rounded corners and edges. The generation of such a complex continuous surface, particularly for any arbitrary size or shape, is an extremely difficult matter. However primitive construction offers a relatively simple solution to this problem.

Rather then generating the entire surface as a continuous object it can be approximated by combining a number of simple primitives. The simplest case would be to enclose each lattice site with a cube. The walls of the cube lie at X_{sm} and $X_{s(m+1)}$ while the cube is centered at the lattice site $\{X_{lm}, Y_{lm}, Z_{lm}\}$. The junction between two adjoining cubes will appear seamless. Testing can be done to eliminate all cubes that do not form group boundaries i.e. cubes sites whose nearest neighbors are occupied in all directions. An example of this technique appears in Figure 4-1.

The necessary algorithm is easily written, but the resulting images are less then optimal. Since there are still a large number of primitives to render, the images move slowly. Also since flat surfaces are usually rendered with continuous shading, surface textures and edges do not readily appear on the bounding surfaces.

A slightly better solution is the use of variable primitives. Here each occupied site is represented by a sphere, each pair of adjoining sites is linked together with a cylinder and each set of four occupied sites that lie in the same plane X is bounded by planes at $X \pm 0.5$. The spheres and cylinders all have radius 0.5 so that any combination of these primitives will form a seamless package that bounds each group. Once again testing is performed to eliminate all primitives that do not form cluster boundaries. The resulting smooth surfaces offer much more visual information then the use of simple cubes. Due to the varying shading of the curved primitives, boundary textures and edges are much more readily visible, yet since these smooth surfaces often require more primitives to define them they tend to react more slowly to transformations. Generally, this techniques is most useful when used in conjunction with the raytracing renderer to generate picture perfect smooth images. An example of this technique also appears in Figure 4-1

4.4.2 Flat Plane Bounding surfaces

A third technique for rendering bounding surfaces achieves visual results similar to those achieved with the primitive cube technique, but offers increased speed and functions well when clusters are made transparent.

In this algorithm, each surface is defined as a combination of planes that surround a cluster. The surfaces are created by testing each occupied lattice site to determine if it is adjacent to an unoccupied site. In each instance where an occupied site is adjacent to an unoccupied site, a square is placed in the interval between the sites. This test is performed simply, without any regard for the specific groups. The resulting surfaces are flat with sharp corners and edges that surround clusters. Since these planes are thin, they can be made transparent to allow the user to view any group boundaries that lie within another group. Figure 4-1 shows an example of this technique.

4.4.3 Beta-Splines

A fourth method used to render groups of atoms uses the generation of beta-spline surface to enclose regions of the same spin. Beta-splines have many advantages not offered by the other techniques. They allow the generation of smooth surfaces that offer more uniform transparency and softer looking groups then the flat plane method. They achieve a continuous surface look that was not attainable using the primitive combination techniques.

The Dore graphics package only allows the use of Non-uniform rational beta-spline surfaces. With proper control of the knot vectors uniform b-spline surfaces can be generated. Similarly, fixing the *w* component of the control point vector to unity allows the generation of non-rational b-splines.[1]. Calculating the knot vector for a continuously varying closed surface is both difficult and time intensive. For this reason I chose to generate the bounding surfaces from a number of simple Bezier patches [Newman and Sproull 79] that are combined to form a closed surface. As will be shown below, the technique can achieve a uniform surface free of discontinuities if the number of patches is allowed to go to infinity.

[1]For further discussion of topics related spline surface generation see [Barsky 85]

The algorithm used to calculate the patches consists of the following steps. A group of atoms of similar spin is selected and bounding planes are calculated that completely enclose the group. This surface is defined by six Bezier patches that form the six sides of a cube. Each patch may be broken into many smaller patches to allow more accurate representation of the surface. For the sake of simplicity we shall assume that each side of the cube comprises a single patch. Since each plane is a fourth order Bezier patch, sixteen control points must be defined that describe the curvature of the patch. This selection of the initial control points is easily accomplished by dividing the patch into a four by four array of points. [Rational B-Splines 83] Initially, all of these points lie on the side of the enclosing cube.

Next the center of the enclosing cube is calculated by finding the midpoint of the max and min values in each of the x, y and z directions. The vector that defines each control point relative to the center of the cluster is then defined by the following equations.

$$\rho = \sqrt{x^2 + y^2 + z^2}$$

$$\tan\phi = \frac{\sqrt{x^2 + y^2}}{z}$$

$$\tan\theta = \frac{y}{x}$$

A second set of equations relate this vector to lattice coordinates.

$$X_{sm} = \rho sin\phi cos\theta$$

$$Y_{sm} = \rho sin\phi sin\theta$$

$$Z_{sm} = \rho cos\theta$$

$$\{X_{lm}, Y_{lm}, Z_{lm}\} = F(\{X_{sm} + 0.5, Y_{sm} + 0.5, Z_{sm} + 0.5\}i[])]$$

where the function *F(x)* returns the largest integer not greater the *x*. Remember that each lattice site includes all the area within 0.5 units of its location.

The algorithm calculates $F(\{X_{sm} + 0.5, Y_{sm} + 0.5, Z_{sm} + 0.5\}i[])]$ to determine if the control vertex lies within an occupied site. If it does, then that location will sufficiently represent a bounding surface for the cluster. If the vertex lies in an unoccupied site, the length of the defining vector is decreased by Δl and a new control vertex is calculated. Once again the vertex is checked to determine it it lies within an occupied site. If Δl is sufficiently small, a control vertex that falls within an occupied site will closely approximate the wall which bounds this site.

The over all effect can be likened to the shrinkable plastic that is used to enclose many commercial products. The enclosing surface starts far outside the group and as each control point is moved closer to the center of the object, the bounding surface appears to shrink down to fit snugly around the cluster.

4.5 Assisting Visualization

Once the image has been properly prepared for rendering, there are a number of useful operations that may be performed on the image to further enhance the visualization of the model. While much is learned simply by seeing the result of the of the simulation brought to life, there are additional features that allow the user to dissect the results in front of him, and further understand the image.

4.5.1 Transparency

Through knob box input the user can dynamically control the transparency of the images he displays. It is often useful to see clusters that reside within one another. By making successive clusters appear transparent, it is possible to see clusters that are hidden within or behind other clusters.

4.5.2 Spinup vs. Spindown Clusters

The software also offers control over the orientation of the clusters that are rendered on the screen. Each model can be specified to render the spinup, spindown or both types of clusters. Figure 4-2 shows the different spin clusters associated with a single model.

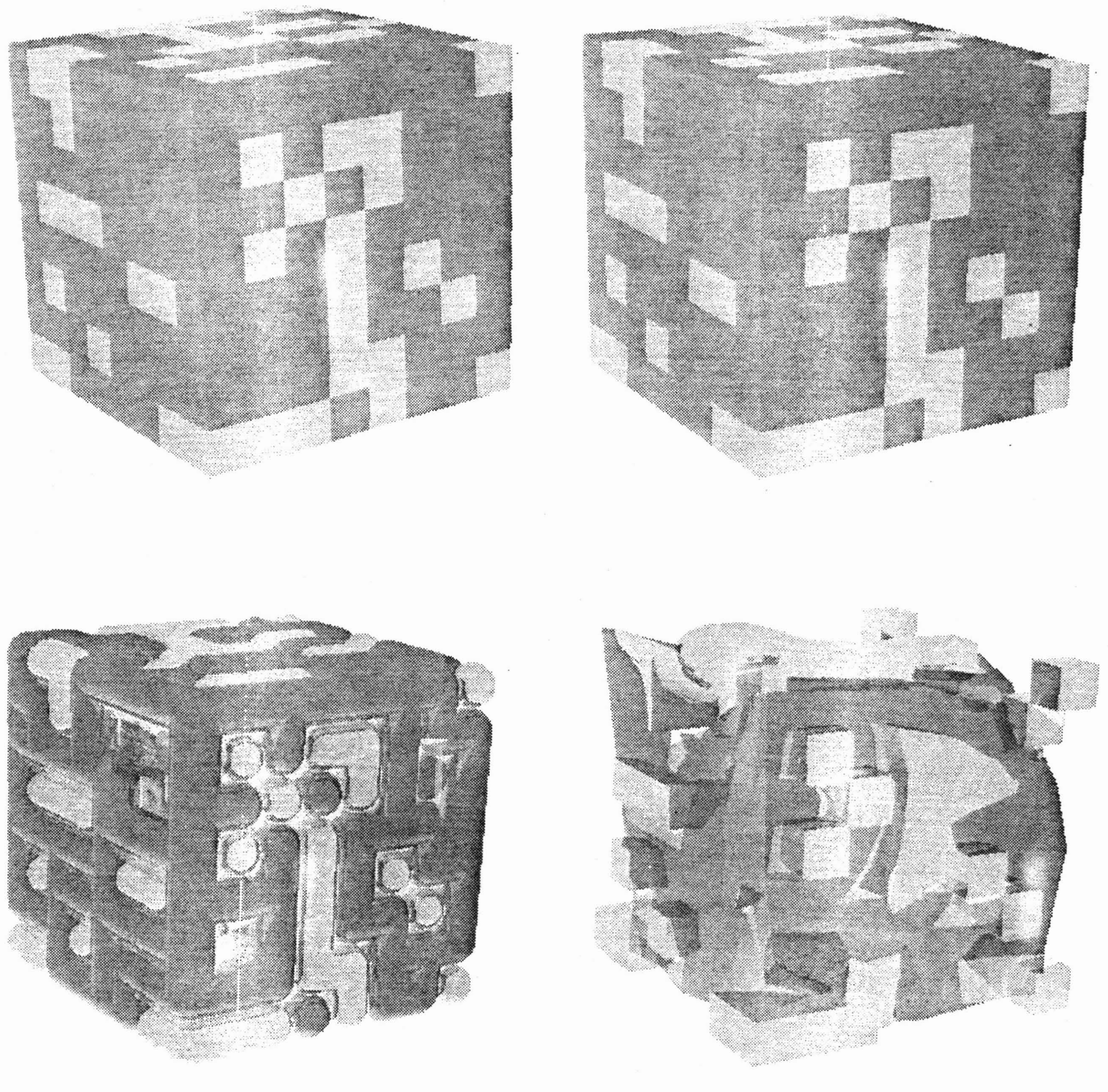

Figure 4-1: Primitive cubes (top left), Variable primitives (bottom left), Flat planes (top right), Beta Spline Surface (bottom right)

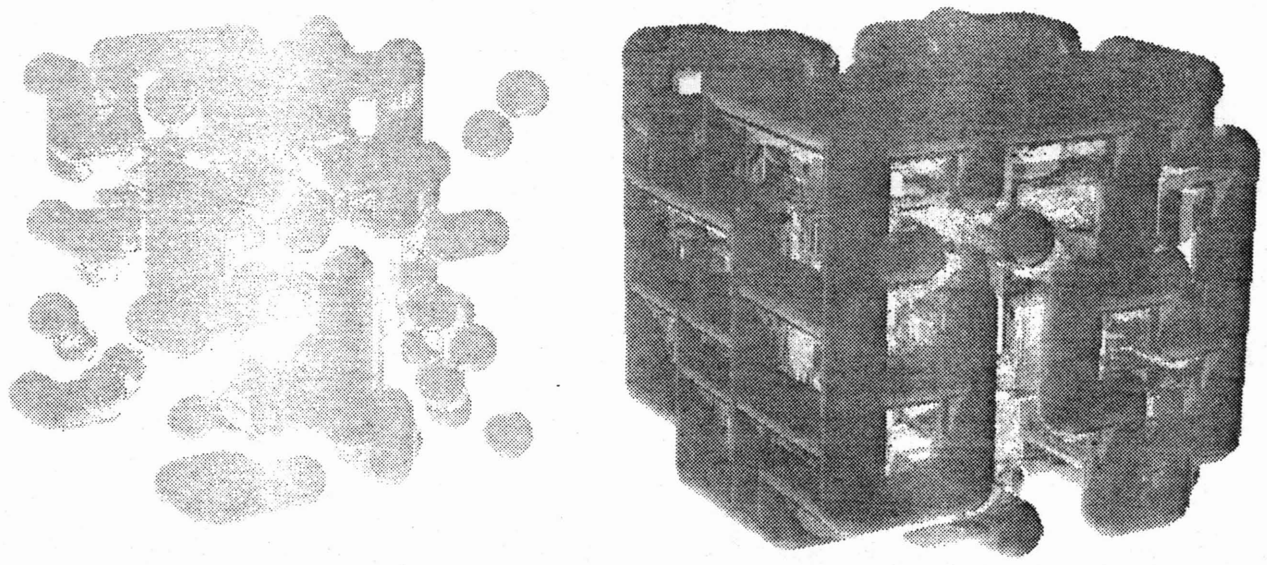

Figure 4-2: Spin up (left), Spin down (right)

4.6 Information and Statistics

While the visualization of complex lattice structures offers many advantages that are simply not available through two-dimensional examination of the data, it is still advantageous to have at hand a number of statistical parameters that define the lattice field. Many of these statistics are associated with the parameters that were selected to achieve the three-dimensional view of the lattice. Yet others are collected and are functions of the lattice itself. These statistics are presented in the following Table.

Dense Model Statistics	
Param	Definition
file0	Current model displayed in left window
file0	Current model displayed in right window
grpmax	Number of spinup and spindown clusters in each model
grpcount	Number of sites in each cluster
dens	Current size of coarse approximation
size	Current minimum cluster size
intensity	Current transparency percentage

Chapter 5

Conclusions

5.1 Results

In addition to helpful information gleaned from examination of the statistics generated by each model, a number of observations were made through use of the software in the examination of the models that were chosen to test it.

Effective visualization of the sparse model requires the solution of a number difficult problems. As the number of primitive objects on the screen increases, it takes significantly longer to render each new version of the model. While it is possible to limit the number of objects on the screen through the use of the viewing plane and probability thresholding, lack of speed can prove to be an annoyance. Secondly, due to the inherent flatness of the screen it is still somewhat difficult to comprehend the three dimensional representations of the models. Careful choice of perspective parameters and use of connectors as lines of reference are helpful, but the problem will always persist to some degree.

The dense model also can create difficulties with rendering speed, but use of beta-splines and carefully chosen approximation parameters can significantly increase speed. Despite these advantages, much information is lost when data is represented by splines. The inherent approximations associated with spline calculations result in the loss of much of the information.

5.2 Additional Research

In many ways the completion of this work has at last furnished the tools to understand and begin to answer the original question. The capabilities and limitations of modern supercomputers have been explored. Specific examples of lattice field models have been examined and evaluated. Still the task remains of the defining what makes one particular visualization technique successful and developing additional techniques for producing understandable images.

All of the various visualization and rendering methods examined in this project form only a small subset of the possible methods. While it is possible to develop methods that exploit similarities in groups of models and will be useful to all of them, each model has slightly different requirements. This work is not meant to be the definitive study of visualization possibilities, rather it offers suggestions and explores some of the visualization options in depth given the constraints and requirements of the models that were selected for testing.

There are still significant exploration possibilities available in the areas of enhancing performance speed and the clarity of the rendered image. Additionally, work can be done to define optimal solutions for the visualization parameters and the optimal B-spline surface for representation of clusters.

Appendix A
Users manual

A.1 Overview

The software developed for this thesis was written in the C programming language. The code follows the format required by the Ardent user-interface programs. Lattice field models are defined as objects and fed to Dore to render. Input is accepted from the keyboard, mouse and dial box and is parsed to determine each input command. The code that performs these functions for the sparse and dense models is provided in appendices B and C.

Portions of the code were built on top of the user-interface software that was developed by Dore. Modifications were made to this software to allow twin viewing windows and multiple input windows. The modifications that were made are shown in Appendix D.

A.2 The Program Structure

The location of the individual programs is represented in figureA-1

The code that generates the models and parses the input of the sparse and dense lattice field models are located in their respective directories. Each directory also contains a **Makefile** that should be used to recompile the files when modifications are made, and a file called **script** that will run the necessary programs.

The modified user interface code resides in the **display** directory. Additional user interface programs are located in the directory **/opt/demo/dui** and have not been modified.

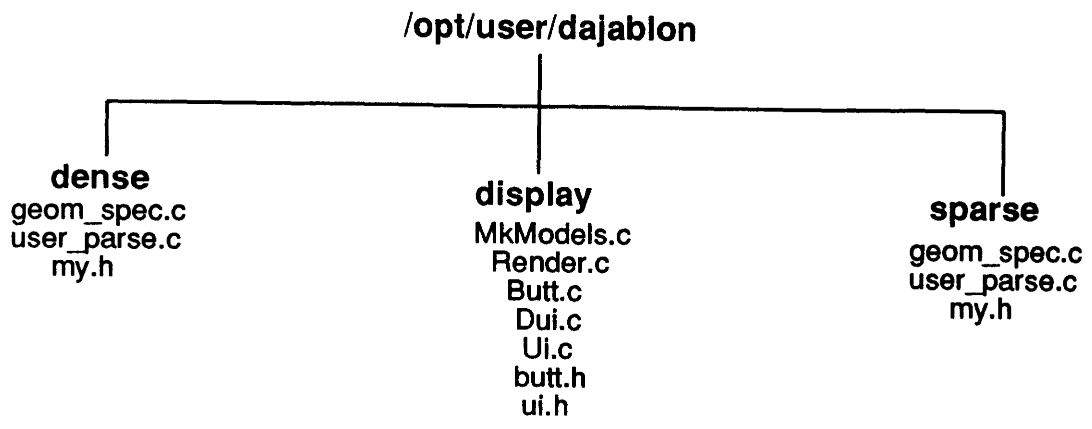

Figure A-1: The directory structure

A.3 Using the Program

The Software can be used with and Lattice field simulation that meets the following simple requirements:

1- The output of the simulation must be stored in a file accessible to the display programs.

2- The output files for the sparse models must conform to the following format:

```
Line 1   :%d %d %d (Integer values for the X, Y, and Z sizes)
Line 2-on:(%d, %d, %d) %f\n  (Integer lattice coordinates,
        floating point probability associated with that site)
```

3- The output files for the dense models must conform to the following format:

```
Line 1  :%d %d %d  (Integer values for the X, Y, and Z sizes)
Line 2- :(%d, %d, %d) %d\n  (Integer lattice coordinates,
               integer probability associated with that site)
```

A.4 Running the Programs

Each program is activated by changing to the appropriate directory and typing the command `script`. The User interface is automatically created and default test models are created and displayed. New models can be read from files and displayed using the appropriate commands.

Once the user interface is activated, commands can be input from the dial box, mouse or keyboard. The current status of each dial is displayed in the lower righthand corner of the screen. Specific commands can be activated by pressing the mouse buttons when the cursor resides on the appropriate button in the upper righthand portion of the screen. Additionally, commands may be typed directly to the program via the command window in the lower lefthand corner of the screen. A complete list of Runtime commands is given in the following tables.

Standard Runtime Commands	
Command	Definition
a	Turns auto x, y, & z rotation on/off
ax	Turns auto x rotation on/off
ay	Turns auto y rotation on/off
az	Turns auto z rotation on/off
c	Turns on/off all polygons with back facing normals
d	Switched drawing type between surface, points,& line
debug	Turns debug printout on/off
h	Turns specular highlights on/off
help	Prints this message
i	Decreases the intensity of the lights
I	Increases the intensity of the lights
q	Exit this program (same as x)
s	Switch between smooth and flat shading
V	Turns video recording on
v	Turns video recording off
x	Exit this program (same as q)
z	Zoom in
Z	Zoom out

Dense Model Visualization Commands	
Command	Definition
w	change active window both/right/left,
m0	Select enclosing surface type in left window
m1	Select enclosing surface type in right window
G	Print statistics about clusters
S0	Perform coarse approximation on currently active windows
S1	Perform group size approximation on currently active windows
I	List current state of user defined parameters
Dh	Reset transparency percentage
Y0	Select spin models for left window up/down/both
Y1	Select spin models for right window up/down/both
Mf	Read new model from file into currently open windows

Sparse Model Visualization Commands	
Cmd	Definition
w	change active window both/right/left
I	List current state of user defined parameters
Dh	Reset Probability threshold
Dv	Change position of clipping planes
Md	Change distance between clipping planes
Mf	Read new model from file into currently open windows
Mn	Change minimum bond length
Mx	Change maximum bond length
A	Turns dynamic model input on/off

Appendix B
The Code for the Sparse Model

B.1 User_parse.c

```
#ident  "@(#)user_parse.c      1.1"    9/1/88

#include "/opt/user/dajablon/display/ui.h"
#include "/opt/user/dajablon/display/butt.h"
#include "my.h"

/*USER_PARSE() - Checks input as it is received and performs
                 the required task*/
int user_parse(str)
char *str;     /*String to be parsed*/
{
  char c;
  float value;
  char string[100];
  int x, y, z;
  float prob;

  if (debug)
    printf("User_parse(%s);\n",str);

  c = *(str++);
    switch (c) {

  case 'd': /*Change active window from 0=both, 1=right, 2=left*/
    if (!mouse_hit) tap_button(0);
    mouse_hit = 0;
    Act_Wind++;
    if (Act_Wind == 3)
      Act_Wind = 0;
    return (1);
  case 's': /*Change 'Surface Type' icon when command is typed*/
    if (!mouse_hit) tap_button(1);
    mouse_hit = 0;
    return (0);
  case 'h': /*Change 'Highlights' icon when command is typed*/
    if (!mouse_hit) tap_button(2);
    mouse_hit = 0;
    return (0);
  case 'c': /*Change 'Backface Culling' icon when command is typed*/
    if (!mouse_hit) tap_button(3);
    mouse_hit = 0;
    return (0);

  case 'z': /*Change state of connectors 0=off, 1=on*/
  if (!mouse_hit) tap_button(8);
  mouse_hit = 0;
  Connects++;
  if (Connects == 2)
    Connects = 0;
  reset_files();
  return (1);

case 'I': /*Print list of current parameters*/
  printf("*******Current Parameters*******\n");
  printf("Lattices Displayed\n");
  printf("   Left Window: %s\n", file0);
  printf("   Right Window: %s\n", file1);
  printf("Connector Parameters\n");
  printf("   Maximum Length = %f\n", dist_mx);
  printf("   Minimum Length = %f\n", dist_mn);
  printf("Disection plane\n");
  printf("   Front plane = %5.1f%%\n", planeat[2]);
  printf("   Thickness = %5.1f%%\n", dev[2]);
  return(1);

case '?':
  printf("*******Keyboard Commands********\n");
  printf("I   Print current parameters\n");
  printf("Md x.xx  -Set viewplane thickness to x.xx\n");
  printf("Mf0 'file'  -Display data in 'file' in right window\n");
  printf("Mf1 'file'  -Display data in 'file' in left window\n");
  printf("Mn x.xx  -Set minimum bond length to x.xx\n");
  printf("Mx x.xx  -Set maximum bond length to x.xx\n");
  printf("]   -enable/disable filereading\n");
  printf("?   -Print this list\n");
  return(1);

case 'D':  /* Parse a commands from dial box */

  c = *(str++);
  switch (c) {

  case 'h': /*Change Probability Threshold*/
    sscanf(str, " %f", &value);
    if(debug) printf("User 'Dh' value = %f\n",value);
    prob_min = value;
    reset_files();
    return (1);

  case 'v': /*Change position of clipping planes*/
    sscanf(str, " %f", &value);
    if(debug) printf("User 'Dv' value = %f\n",value);
    planeat[0] = ((value/100) * (zmx[0] - zmn[0])) + zmn[0];
    planeat[1] = ((value/100) * (zmx[1] - zmn[1])) + zmn[1];
    planeat[2] = value;
    reset_files();
    return (1);
  default:
    return(0);
  }
case 'a': /*Change state of 'auto-rotate' buttons when cmd is typed*/
  if(mouse_hit) {mouse_hit = 0;  return(0); }
  c = *(str++);
  switch (c) {

  case 'x': /*Auto-rotate around X-axis*/
    tap_button(5);
    return (0);

  case 'y': /*Auto-rotate around Y-axis*/
    tap_button(6);
    return (0);

  case 'z': /*Auto-rotate around Z-axis*/
    tap_button(7);
    return (0);
  }
case 'M':  /*Change various viewing parameters */

  c = *(str++);
  switch (c) {

  case 'd': /*Change viewplane thickness*/
    sscanf(str, " %f", &value);
    if(debug) printf("User 'Dv' value = %f\n",value);
    dev[0] = (value/100) * (zmx[0] - zmn[0]);
    dev[1] = (value/100) * (zmx[1] - zmn[1]);
    dev[2] = value;
    reset_files();
    return (1);

  case 'f': /*Read new models into all windows currently*/
    sscanf(str, " %s", &string);   /*open for update*/
    if(Act_Wind == 0) {
      strcpy(file0, string);
      strcpy(file1, string);
      readfile(0);
      readfile(1);
    }
    else if(Act_Wind ==1) {
      strcpy(file1, string);
      readfile(1);
    }
    else if(Act_Wind ==2) {
      strcpy(file0, string);
      readfile(0);
    }
    reset_files();
    return (1);
  case 'n': /*Change minimum bond length*/
    sscanf(str, " %f", &value);
    if(debug) printf("User 'Dv' value = %f\n",value);
    dist_mn = value;
    reset_files();
```

```
        return (1);
      case 'x': /*Change maximum bond length*/
        sscanf(str, " %f", &value);
        if(debug) printf("User 'Dv' value = %f\n",value);
        dist_mx = value;
        reset_files();
        return (1);
      }
    case 'A': /*Turn on/off automatic file reading*/
      if (autohit) {
        autohit = 0;
        return(0);
      }
      else {
        autohit = 1;
        autoread(1);
        reset_files1();
        return(0);
      }
    }

    return(0); /*Default return if command was not accepted*/

}  /* End of user_parse function */

/*AUTOREAD() - Loads initial state of model into memory
               and recalculates center*/
autoread(num)
int num;
{
  int i, x, y, z, prob;
  float xmax, ymax, xmin, ymin;
  char filename[60];

  xmax = ymax = zmx[num] = -1000.0;
  xmin = ymin = zmn[num] = 1000.0;
  printf("What is the name of the input file?");
    scanf("%s", filename);
/*Clear memory array*/
  for(x=1; x<=(MAXX[num]); x++) {
    for(y=1; y<=(MAXY[num]); y++) {
      for(z=1; z<=(MAXZ[num]); z++) {
        m[x][y][z][num] = 0.0;
      } } }
/*Open new file for reading*/
  if ((autosfp = fopen(filename, "r")) == NULL)
    printf("Couldn't open %s for reading\n", filename);
/*Read Maximum lattice values*/
  fscanf(autosfp, "%d %d %d\n", &MAXX[num], &MAXY[num], &MAXZ[num]);
/*Read number of sites in initial state*/
  fscanf(autosfp, "%d\n", &MAXARRAY[num]);
/*Read each site in initial model*/
  for (i=0; i<MAXARRAY[num]; i++) {
    fscanf(autosfp, "%d %d %d %d\n", &x, &y, &z, &prob);
    m[x+1][y+1][z+1][num] = (float)prob;
    if (x >= xmax)
      xmax = x;
    if (x <= xmin)
      xmin = x;
    if (y >= ymax)
      ymax = y;
    if (y <= ymin)
      ymin = y;
    if (z >= zmx[num])
      zmx[num] = z;
    if (z <= zmn[num])
      zmn[num] = z;
  }
/*Calculate center of model*/
  xmid[num] = MAXX[num] / 2;
  ymid[num] = MAXY[num] / 2;
  zmid[num] = MAXZ[num] / 2;
  planeat[num] = zmid[num];  /*Set viewing plane in center of model*/
}


/*PIPE_CHECK() - read in required number of sites for dynamic display*/
pipe_check()
{
  int i, num, x, y, z, x1, y1, z1, prob, step;
  float hold;

  num = 1;
  step = 10000;
/*Search data until success is found or enough samples have been tried*/
  for(i=0; i< step; i++) {
    if (fscanf(autosfp, "%d %d %d %d %d %d %d\n",
               &x, &y, &z, &x1, &y1, &z1, &prob) == EOF)   return;
    if (prob == 1) i = step;/*Stop if a successful trial is found*/
  }
/*Set probability of sites in successful trial to 99 or 101 MAKE_MESH()
  will set the color to yellow and reset probabilities to 0 or 1*/
  if(prob) {
    hold = m[x+1][y+1][z+1][num];
    m[x+1][y+1][z+1][num] = m[x1+1][y1+1][z1+1][num] + 100.0;
    m[x1+1][y1+1][z1+1][num] = hold + 100.0;
  }
/*Set probability of sites in successful trial to 9 or 11 MAKE_MESH()
  will set the color to blue and reset probabilities to 0 or 1*/
  else {
    m[x+1][y+1][z+1][num] = m[x+1][y+1][z+1][num] + 10.0;
    m[x1+1][y1+1][z1+1][num] = m[x1+1][y1+1][z1+1][num] + 10.0;
  }
  reset_files();
}

/*RESET_FILES() - Recalculates model in active window each time a change
                  is made*/
reset_files()
{
  if (Act_Wind != 1) { /*If right window is to be updated*/
    DgEmpty(models[0]); /*Empty old Model*/
    DgOpen(models[0]);
    DgAddObj(make_mesh(0)); /*Create new model*/
    DgClose();
    if (Connects) { /*If connectors are on*/
      DgEmpty(models[0]);  /*Empty old model*/
      DgOpen(models[0]);
      DgAddObj(make_mesh(0)); /*Create new model*/
      DgAddObj(make_conn(0)); /*Add connectors*/
      DgClose();
    } }
  if (Act_Wind != 2) { /*If left window is to be updated*/
    DgEmpty(models[1]); /*Empty old Model*/
    DgOpen(models[1]);
    DgAddObj(make_mesh(1)); /*Create new model*/
    DgClose();
    if (Connects) {
      DgEmpty(models[1]); /*Empty old Model*/
      DgOpen(models[1]);
      DgAddObj(make_mesh(1)); /*Create new model*/
      DgAddObj(make_conn(1)); /*Add connectors*/
      DgClose();
    } }
}

/*TAP_BUTTON() - Resets button icons when commands are typed*/
tap_button(butslid)
int butslid; /*The number of the button to be reset*/
{
  draw_button(butslid_window,xyloc[butslid][0],xyloc[butslid][1],
            buttdn[butstat[butslid]], "", 0, "", 0);
  do {
    if((++butstat[butslid]) > NumButtStates)
      butstat[butslid] = 0;

  } while(buttons[butslid].exec[butstat[butslid]][0] == '\0');

  draw_button(butslid_window,xyloc[butslid][0],xyloc[butslid][1],
            buttup[butstat[butslid]],
            buttons[butslid].text, 0x000000,
            buttons[butslid].label[butstat[butslid]],0x000000);

  XFlush(display);

}
```

B.2 Geom_spec.c

```
#ident  "@(#)geom_spec.c       1.1"    9/1/88

#include <dore.h>
#include <stdio.h>
#include <math.h>
#include "my.h"

#define NMODELS 4
int nmodels = NMODELS;
int firstcycle = 1;
int lastcycle = 2;

/*GEOM_SPEC() - Initializes models and viewing parameters*/
geom_spec()
{
  DtObject DoGroup();
  DtReal value;
  static DtPoint3 border[] = { -10.0, -10.0, 0.0,
                                 -10.0,  10.0, 0.0,
                                 10.0,  10.0, 0.0,};

  if (debug) printf("In geom_spec...\n");

/*Initialize parameters*/
  Res = 1.3;
  dens = 1.0;
  dist_mn = 1.0;
  dist_mx = 2.5;
  dev[0] = dev[1] = dev[2] = 15;
  strcpy(file0, "data.strt.int");
  strcpy(file1, "data.cube");
/*Read initial models from files*/
  readfile(0);
  readfile(1);

/*Create initial model for left window*/
  if (debug) printf("In geom_spec...Read file\n");
  models[0] = DoGroup(DcTrue);
    DgAddObj(DoLabel(0));
    DgAddObj(DoDiffuseColor(DcRGB, bg));
    DgAddObj(DoRepType(DcPoints));
    DgAddObj(DoPolyline(DcRGB, DcLoc, 3, border));
    DgAddObj(DoLabel(1));
  DgClose();
  DsHoldObj(models[0]);

/*Create initial model for right window*/
  models[1] = DoGroup(DcTrue);
    DgAddObj(DoLabel(0));
    DgAddObj(DoDiffuseColor(DcRGB, bg));
    DgAddObj(DoRepType(DcPoints));
    DgAddObj(DoPolyline(DcRGB, DcLoc, 3, border));
    DgAddObj(DoLabel(1));
  DgClose();
  DsHoldObj(models[1]);
}  /* End of geom_spec function */

/*READFILE() - Read data from file*/
readfile(num)
int num; /*0 = left window, 1 = right window*/
{
  int i;
  float xmax, ymax, xmin, ymin;
  int x, y, z;
  float  prob;
  FILE *sfp;

/*Clear max/min values of model*/
  xmax = ymax = zmx[num] = -1000.0;
  xmin = ymin = zmn[num] = 1000.0;
/*Open data files*/
  if (num == 0) {
    if ((sfp = fopen(file0,"r")) == NULL)
      printf("Couldn't open testfile0 for reading\n");
  }
  if (num == 1) {
    if ((sfp = fopen(file1,"r")) == NULL)
      printf("Couldn't open testfile1 for reading\n");
  }
/*Read maximum array values*/
  fscanf(sfp, "%d %d %d\n", &MAXX[num], &MAXY[num], &MAXZ[num]);
/*Clear arrays to 0*/
  for(x=0; x<=(MAXX[num]+1); x++) {
    for(y=0; y<=(MAXY[num]+1); y++) {
      for(z=0; z<=(MAXZ[num]+1); z++) {
        m[x][y][z][num] = 0;
      } } }
/*Read valu of each occupied site*/
  MAXARRAY[num] = (MAXX[num])*(MAXY[num])*(MAXZ[num]);
  for (i=0; i<=MAXARRAY[num]; i++) {
    fscanf(sfp, "(%d, %d, %d) %f\n", &x, &y, &z, &prob);
    if (prob >= 1.000001)
      prob = 0.95;
    m[x+1][y+1][z+1][num] = prob;
    if (x >= xmax)
      xmax = x;
    if (x <= xmin)
      xmin = x;
    if (y >= ymax)
      ymax = y;
    if (y <= ymin)
      ymin = y;
    if (z >= zmx[num])
      zmx[num] = z;
    if (z <= zmn[num])
      zmn[num] = z;
  }
```

```
/*Calculate center of model*/
  xmid[num] = MAXX[num] / 2;
  ymid[num] = MAXY[num] / 2;
  zmid[num] = MAXZ[num] / 2;
  (void) fclose(sfp);
  planeat[num] = zmid[num];
}

/*MAKE_MESH() - Create model object to be rendered by Dore*/
DtObject make_mesh(num)
int num;
{
  int x, y, z;
  DtReal color[3];

  for(x=1; x<=(MAXX[num]); x++) { /*Test each site in model*/
    for(y=1; y<=(MAXY[num]); y++) {
      for(z=1; z<=(MAXZ[num]); z++) {
        if(m[x][y][z][num] <= prob_min) /*do not render if below*/
          continue;                     /*probability threshhold*/
        if((z - zmid[num]) > planeat[num]) /*Do not render if in front*/
          continue;                        /*of viewing plane*/
        if((z - zmid[num]) <= (planeat[num] - (3 * dev[num])))
          continue;            /*Do not render if behind viewing plane*/

        if (m[x][y][z][num] > 50) {
          color[0] = yellow[0];  /*If PIPE_CHECK() marked as*/
          color[1] = yellow[1];  /*successful trial color yellow*/
          color[2] = yellow[2];  /*and reset probability*/
          m[x][y][z][num] = m[x][y][z][num] - 100;
        }
        else if (m[x][y][z][num] > 5) {
          color[0] = blue[0];  /*If PIPE_CHECK() marked as*/
          color[1] = blue[1];  /*successful trial color blue*/
          color[2] = blue[2];  /*and reset probability*/
          m[x][y][z][num] = m[x][y][z][num] - 10;
        }
        else {  /*otherwise calculate probability color*/
          color[0] = ((m[x][y][z][num] - prob_min) /
                      (1.0 - prob_min));
          color[1] = 1. - ((m[x][y][z][num] - prob_min) /
                           (1.0 - prob_min));
          color[2] = 0.03;
        }
    /*Add new site to model*/
        DgAddObj(DoPushMatrix());
        DgAddObj(DoDiffuseColor(DcRGB, color));
        DgAddObj(DoRepType(DcSurface));
        if((z - zmid[num]) <= (planeat[num] - dev[num]))
          DgAddObj(DoRepType(DcWireframe));
        if((z - zmid[num]) <= (planeat[num] - (2 * dev[num])))
          DgAddObj(DoRepType(DcPoints));
        DgAddObj(DoTranslate(((float)x - xmid[num]),
                  ((float)y - ymid[num]),((float)z - zmid[num])));
        DgAddObj(DoScale(0.5, 0.5, 0.5));
        DgAddObj(DoPrimSurf(DcSphere));
        DgAddObj(DoPopMatrix());
      } } }
  return (DoLabel(50)); /*Mark end of model*/
}

/*MAKE_CONN() - Create connectors in desired object*/
DtObject make_conn(num)
int num;
{
  int x, y, z, x1, y1, z1;
  float Dist, a, b, c, d;
  DtReal Theta, Chi;
  DtObject cyl();

  if (debug) printf("Make connectors!!!!");

  for(x=1; x<=MAXX[num]; x++) { /*Try each site*/
    for(y=1; y<=MAXY[num]; y++) {
      for(z=1; z<=MAXZ[num]; z++) {
        if(m[x][y][z][num] <= prob_min)
          continue;
        if((z - zmid[num]) > planeat[num])
          continue;
        if((z - zmid[num]) <= (planeat[num] - (3 * dev[num])))
          continue;
        for(x1=1; x1<=MAXX[num]; x1++) { /*Test against each of the*/
          for(y1=1; y1<=MAXY[num]; y1++) {  /*other sites*/
            for(z1=1; z1<=MAXZ[num]; z1++) {
              if(m[x1][y1][z1][num] <= prob_min)
                continue;
              if((z1 - zmid[num]) > planeat[num])
                                      continue;
              if((z1 - zmid[num]) <= (planeat[num] - (3 * dev[num])))
                continue;
              a = (float)x - (float)x1;
              b = (float)y - (float)y1;
              c = (float)z - (float)z1;
              Dist = sqrt(fabs(a * a + b * b + c * c));
              if(Dist >= dist_mx)
                continue;
              d = hypot(a, c);
              Theta = atan2(a, c);
              Chi = -atan2(b, d);
              DgAddObj(DoPushMatrix());
              DgAddObj(DoDiffuseColor(DcRGB, blu));
              if(Dist <= dist_mn)
                DgAddObj(DoDiffuseColor(DcRGB, yellow));
              DgAddObj(DoRepType(DcSurface));
              if((z1 - zmid[num]) <= (planeat[num] - dev[num]))
                DgAddObj(DoRepType(DcWireframe));
              if((z1 - zmid[num]) <= (planeat[num] - (2 * dev[num])))
                DgAddObj(DoRepType(DcPoints));
              DgAddObj(DoTranslate(((float)x - xmid[num]),
                     ((float)y - ymid[num]), ((float)z - zmid[num])));
                DgAddObj(DoRotate(DcYAxis, Theta));
                DgAddObj(DoRotate(DcXAxis, Chi));
                DgAddObj(DoScale(0.3, 0.3, -Dist));
                DgAddObj(DoPrimSurf(DcBox));
                DgAddObj(DoPopMatrix());
              } } }
       } } }
  return (DoLabel(100)); /*Mark end of connector group*/
}
```

B.3 My.h

```
#ident  "@(#)my.h      1.1"    9/1/88

#define MAX 25  /*Maximum length of the side of an array*/
#define Pi 3.1415927
#define MaxButs 12 /*Maximum number of buttons*/

int MAXARRAY[2], MAXX[2], MAXY[2], MAXZ[2];
float xmid[2], ymid[2], zmid[2];
extern DtInt debug;
extern DtObject models[4];
extern DtReal bg[3];
float dist_mx;
float dist_mn;
char file0[100];
char file1[100];
float m[MAX][MAX][MAX][2];
float zmx[2], zmn[2], dens;
float prob_min;
float planeat[3];
float dev[3];
float Res;
int Act_Wind, Connects;
DtObject models[4];
int count;
int readinput, count, autohit;
FILE *autosfp;

static DtReal white[] = {1.0, 1.0, 1.0}; /*Define some colors*/
static DtReal red[] = {1., 0.01, .03};
static DtReal yellow[] = {1., 1.0, 0.0};
static DtReal blue[] = {0., 0.0, 1.0};
```

Appendix C
The Code for the Dense Model

C.1 User_parse.c

```
#ident   "@(#)user_parse.c      1.1"    9/1/88

#include "/opt/user/dajablon/display/ui.h"
#include "/opt/user/dajablon/display/butt.h"
#include "my.h"

/*USER_PARSE() - Checks input as it is received and performs
                 the  required task*/
int user_parse(str)
char *str;     /*String to be parsed*/
{
  char c;
  float value;
  int valint, a;
  char string[100];

  if (debug)
    printf("User_parse(%s);\n",str);

  c = *(str++);
  switch (c) {

  case 'w':/*Change active window from 0=both, 1=right, 2=left*/
    if (!mouse_hit) tap_button(0);
    mouse_hit = 0;
    Act_Wind++;
    if (Act_Wind == 3)
      Act_Wind = 0;
    return (1);
  case 's':/*Change 'Surface Type' icon when command is typed*/
    if (!mouse_hit) tap_button(2);
    mouse_hit = 0;
    return (0);
  case 'd':/*Change 'Surface Rep' icon when command is typed*/
    if (!mouse_hit) tap_button(1);
    mouse_hit = 0;
    return (0);
  case 'm':/*Modify bounding surface type*/
    c = *(str++);
    switch (c) {

    case '0':/*Left window*/
      modtype0++;
      if (modtype0 == 4) modtype0 = 0;
      reset_files();
      return (1);
    case '1':/*Right window*/
      modtype1++;
      if (modtype1 == 4) modtype1 = 0;
      reset_files();
      return (1);
    }
  case 'G':/*Print information about groups*/
    for(gnum=0; gnum<4; gnum++) {
      printf("Lattice number %d has %d groups\n", gnum, grpmax[gnum]);
      for(a=0; a<grpmax[gnum]; a++) {
        printf("Group %d has %d members\n", a+1, grpcount[a][gnum]);
      } }
  case 'S':/*Make approximation of group structures*/
    c = *(str++);
    switch (c) {

    case '0':/*Do course approximation*/
      sscanf(str, " %d", &valint);
      if(debug) printf("User 'S0' value = %d\n",valint);
      dens = (float)valint;
      printf("%f\n", dens);
      if(Act_Wind == 0) {
        approx(0);
        approx(1);
      }
      else if(Act_Wind ==1) {
        approx(1);
      }
      else if(Act_Wind ==2) {
        approx(0);
      }
      reset_files();
      return (1);
    case '1':/*Do group size approximation*/
      sscanf(str, " %d", &valint);
      if(debug) printf("User 'S1' value = %d\n",valint);
      if(Act_Wind == 0) {
        smooth(0, valint);
        smooth(1, valint);
      }
      else if(Act_Wind ==1) {
        smooth(1, valint);
      }
      else if(Act_Wind ==2) {
        smooth(0, valint);
      }
      reset_files();
      return (1);
    }
  case 'I': /*Print list of current parameters*/
    printf("*******Current Parameters*******\n");
    printf("Lattices Displayed\n");
    printf("   Left Window: %s\n", file0);
  printf("   Right Window: %s\n", file1);
  printf("Connector Parameters\n");
  printf("   Maximum Length = %f\n", dist_mx);
  printf("   Minimum Length = %f\n", dist_mn);
  printf("Disection plane\n");
  printf("   Front plane = %5.1f%%\n", planeat[2]);
  printf("   Thickness = %5.1f%%\n", dev[2]);
  return(1);

case 'D': /* Parse a commands from dial box */
  c = *(str++);
  switch (c) {

  case 'h': /*Reset transparency value*/
    sscanf(str, " %f", &value);
    if (debug) printf("User 'Dh' value = %f\n",value);
    intensity = value;
    update_modified = DcFalse;
    return (1);
  default:
    return(0);
  }
case 'a': /*Change state of 'auto-rotate' buttons when cmd is typed*/
  if(mouse_hit) {mouse_hit = 0;  return(0); }
  c = *(str++);

  switch (c) {
  case 'x': /*Auto-rotate around X-axis*/
    tap_button(4);
    return (0);

  case 'y': /*Auto-rotate around Y-axis*/
    tap_button(5);
    return (0);

  case 'z': /*Auto-rotate around Z-axis*/
    tap_button(6);
    return (0);
  }
case 'Y':  /* Select spinup, spindown or both models */
  c = *(str++);
  switch (c) {

  case '0': /*Left Window*/
    spin[0]++;
    if (spin[0] == 3) spin[0] = 0;
    reset_files();
    return (1);
  case '1': /*Right Window*/
    spin[1]++;
    if (spin[1] == 3) spin[1] = 0;
    reset_files();
    return (1);
  }
case 'M': /*Change various viewing parameters */
```

```
    c = *(str++);
    switch (c) {

    case 'f': /*Read new models into all windows currently open*/
      sscanf(str, " %s", string);
      if(Act_Wind == 0) {
        strcpy(file0, string);
        strcpy(file1, string);
        readfile(0);
        readfile(1);
      }
      else if(Act_Wind ==1) {
        strcpy(file1, string);
        readfile(1);
      }
      else if(Act_Wind ==2) {
        strcpy(file0, string);
        readfile(0);
      }
      reset_files();
      return (1);
    }
  }
  return(0); /*Default return if command was not accepted*/

}  /* End of user_parse function */

/*TRANSPAREN_CALLBACK() - Dynamicly alter transparency as requested*/
transparen_callback()
{
  DtObject DoTranspIntens();

  /*printf("transparent intensity = %lf\n",intensity);*/
  if (intensity <= 0.25)
    DsExecuteObj(DoTranspSwitch(DcOff));
  else {
    DsExecuteObj(DoTranspSwitch(DcOn));
    DsExecuteObj(DoTranspColor(DcRGB, white));
    DsExecuteObj(DoTranspIntens(intensity));
  }
}  /* End of transparen_callback function */

/*RESET_FILES() - Recalculates model in active window each time a change
                  is made*/
reset_files()
{
  if (Act_Wind != 1) {
    DgEmpty(models[0]);
    DgOpen(models[0]);
    DgAddObj(DoReflectionSwitch(DcOff));
    DgAddObj(DoCallback(transparen_callback,DcNullObject));
    if(modtype0 == 0)  make_mesh(0);
    else if(modtype0 == 1)  make_conn(0);
    else if(modtype0 == 2)  make_conn2(0);
    else if(modtype0 == 3)  bsurf(0);
    DgClose();
  }
  if (Act_Wind != 2) {
    DgEmpty(models[1]);
    DgOpen(models[1]);
    DgAddObj(DoReflectionSwitch(DcOff));
    DgAddObj(DoCallback(transparen_callback,DcNullObject));
    if(modtype1 == 0)  make_mesh(1);
    else if(modtype1 == 1)  make_conn(1);
    else if(modtype1 == 2)  make_conn2(1);
    else if(modtype1 == 3)  bsurf(1);
    DgClose();
  }
}

/*TAP_BUTTON() - Resets button icons when commands are typed*/
tap_button(butslid)
     int butslid;  /*The number of the button to be reset*/
{
  draw_button(butslid_window,xyloc[butslid][0],xyloc[butslid][1],
              buttdn[butstat[butslid]], "", 0, "", 0);
  do {
    if((++butstat[butslid]) > NumButtStates)
      butstat[butslid] = 0;
  } while(buttons[butslid].exec[butstat[butslid]][0] == '\0');
  draw_button(butslid_window,xyloc[butslid][0],xyloc[butslid][1],
              buttup[butstat[butslid]],
              buttons[butslid].text, 0x000000,
              buttons[butslid].label[butstat[butslid]],0x000000);
  XFlush(display);
}
```

C.2 Geom_spec.c

```
#ident  "@(#)geom_spec.c      1.1"    9/1/88

#include <dore.h>
#include <stdio.h>
#include <math.h>
#include "my.h"

#define NMODELS 4
int nmodels = NMODELS;
int firstcycle = 1;
int lastcycle = 2;

/*GEOM_SPEC() - Initializes models and viewing parameters*/
geom_spec()
{
  DtObject DoGroup();
  static DtPoint3 border[] = { -10.0, -10.0, 0.0,
                               -10.0,  10.0, 0.0,
                                  10.0,  10.0, 0.0};

  if (debug) printf("In geom_spec...\n");

  /*Initialize parameters*/
  Res = 1.3;
  dens = 1.0;
  dist_mx = 1.1;
  dist_mn = 1.0;
  intensity = 0.1;
  modtype0 = modtype1 = 0;
  dev[0] = dev[1] = dev[2] = 15;
  strcpy(file0, "data4.3");
  strcpy(file1, "data4.3");
  /*Read initial models from files*/
  readfile(0);
  readfile(1);

  /*Create initial model for left window*/
  if (debug) printf("In geom_spec...Read file\n");
  models[0] = DoGroup(DcTrue);
  DgAddObj(DoLabel(0));
  DgAddObj(DoDiffuseColor(DcRGB, bg));
  DgAddObj(DoRepType(DcPoints));
  DgAddObj(DoPolyline(DcRGB, DcLoc, 3, border));
  DgAddObj(DoLabel(1));
  DgClose();
  DsHoldObj(models[0]);

  /*Create initial model for right window*/
  models[1] = DoGroup(DcTrue);
  DgAddObj(DoLabel(0));
  DgAddObj(DoDiffuseColor(DcRGB, bg));
  DgAddObj(DoRepType(DcPoints));
  DgAddObj(DoPolyline(DcRGB, DcLoc, 3, border));
  DgAddObj(DoLabel(1));
  DgClose();
  DsHoldObj(models[1]);
}  /* End of geom_spec function */

/*READFILE() - Read data from file*/
readfile(num)
     int num; /*0 = left window, 1 = right window*/
{
  int i;
  float xmax, ymax, xmin, ymin;
  int x, y, z, prob;
  FILE *sfp;

  /*Clear max/min values of model*/
  xmax = ymax = zmx[num] = -1000.0;
  xmin = ymin = zmn[num] = 1000.0;
  /*Open data files*/
  if (num == 0) {
    if ((sfp = fopen(file0,"r")) == NULL) {
      printf("Couldn't open testfile0 for reading\n");
      return;
    }
  }
  if (num == 1) {
    if ((sfp = fopen(file1,"r")) == NULL) {
      printf("Couldn't open testfile1 for reading\n");
      return;
    }
  }
  /*Read maximum array values*/
  fscanf(sfp, "%d %d %d\n", &MAXX[num], &MAXY[num], &MAXZ[num]);
  MAXARRAY[num] = (MAXX[num])*(MAXY[num])*(MAXZ[num]);
  MAXX[num+2] = MAXX[num];
  MAXY[num+2] = MAXY[num];
  MAXZ[num+2] = MAXZ[num];
  /*Clear arrays to -1000*/
  for(x=0; x<=MAXX[num]+1; x++) {
    for(y=0; y<=MAXY[num]+1; y++) {
      for(z=0; z<=MAXZ[num]+1; z++) {
        m[x][y][z][num] = -1000;
        m[x][y][z][num+2] = -1000;
      } } }
  /*Read valu of each occupied site*/
  for (i=0; i<=MAXARRAY[num]; i++) {
    fscanf(sfp, "(%d, %d, %d) %d\n", &x, &y, &z, &prob);
    m[x+1][y+1][z+1][num] = prob;
    m[x+1][y+1][z+1][num+2] = prob;
    if (x >= xmax)
      xmax = x;
    if (x <= xmin)
      xmin = x;
    if (y >= ymax)
      ymax = y;
    if (y <= ymin)
      ymin = y;
    if (z >= zmx[num])
      zmx[num] = z;
    if (z <= zmn[num])
      zmn[num] = z;
  }
  /*Calculate center of model*/
  xmid[num] = -1.0*(dens/2.0*MAXX[num])+(dens/2.0);
  ymid[num] = -1.0*(dens/2.0*MAXY[num])+(dens/2.0);
  zmid[num] = -1.0*(dens/2.0*MAXZ[num])+(dens/2.0);
  (void) fclose(sfp);
  planeat[num] = zmid[num];
  group(num); /*Define clusters*/
}

/*TREE() - Given a starting point calculate of the members
           of the cluster*/
tree(num, grpnum, posneg)
     int num, grpnum, posneg;
{
```

```
int a, b, c, d, cont, x, y, z;

gnum = num + 2*posneg;
a = grpnum;
b = 0;
do {
  /*Take site off the top of the culster list*/
  x = grp[gnum][a][b][0];
  y = grp[gnum][a][b][1];
  z = grp[gnum][a][b][2];
  /*Test each of its nearest neighbors*/
  if (m[x-1][y][z][num] == posneg) {
    cont = 0;
    for(c=0; c<=grpcount[a][gnum]; c++) {
      if ((x-1 == grp[gnum][a][c][0]) && (y == grp[gnum][a][c][1])
          && (z == grp[gnum][a][c][2]))
        cont = 1;
    }
    /*If a test is positive add neighbor to cluster list*/
    if (!cont) {
      d = grpcount[a][gnum];
      grp[gnum][a][d][0] = x-1;
      grp[gnum][a][d][1] = y;
      grp[gnum][a][d][2] = z;
      grpcount[a][gnum]++;
    } }
  /*check another nearest neighbor*/
  if (m[x+1][y][z][num] == posneg) {
    cont = 0;
    for(c=0; c<=grpcount[a][gnum]; c++) {
      if ((x+1 == grp[gnum][a][c][0]) && (y == grp[gnum][a][c][1])
          && (z == grp[gnum][a][c][2]))
        cont = 1;
    }
    if (!cont) {
      d = grpcount[a][gnum];
      grp[gnum][a][d][0] = x+1;
      grp[gnum][a][d][1] = y;
      grp[gnum][a][d][2] = z;
      grpcount[a][gnum]++;
    } }
  /*Again*/
  if (m[x][y-1][z][num] == posneg) {
    cont = 0;
    for(c=0; c<=grpcount[a][gnum]; c++) {
      if ((x == grp[gnum][a][c][0]) && (y-1 == grp[gnum][a][c][1])
          && (z == grp[gnum][a][c][2]))
        cont = 1;
    }
    if (!cont) {
      d = grpcount[a][gnum];
      grp[gnum][a][d][0] = x;
      grp[gnum][a][d][1] = y-1;
      grp[gnum][a][d][2] = z;
      grpcount[a][gnum]++;
```

```
        } }
      if (m[x][y+1][z][num] == posneg) {
        cont = 0;
        for(c=0; c<=grpcount[a][gnum]; c++) {
          if ((x == grp[gnum][a][c][0]) && (y+1 == grp[gnum][a][c][1])
              && (z == grp[gnum][a][c][2]))
            cont = 1;
        }
        if (!cont) {
          d = grpcount[a][gnum];
          grp[gnum][a][d][0] = x;
          grp[gnum][a][d][1] = y+1;
          grp[gnum][a][d][2] = z;
          grpcount[a][gnum]++;
        } }
      if (m[x][y][z-1][num] == posneg) {
        cont = 0;
        for(c=0; c<=grpcount[a][gnum]; c++) {
          if ((x == grp[gnum][a][c][0]) && (y == grp[gnum][a][c][1])
              && (z-1 == grp[gnum][a][c][2]))
            cont = 1;
        }
        if (!cont) {
          d = grpcount[a][gnum];
          grp[gnum][a][d][0] = x;
          grp[gnum][a][d][1] = y;
          grp[gnum][a][d][2] = z-1;
          grpcount[a][gnum]++;
        } }
      if (m[x][y][z+1][num] == posneg) {
        cont = 0;
        for(c=0; c<=grpcount[a][gnum]; c++) {
          if ((x == grp[gnum][a][c][0]) && (y == grp[gnum][a][c][1])
              && (z+1 == grp[gnum][a][c][2]))
            cont = 1;
        }
        if (!cont) {
          d = grpcount[a][gnum];
          grp[gnum][a][d][0] = x;
          grp[gnum][a][d][1] = y;
          grp[gnum][a][d][2] = z+1;
          grpcount[a][gnum]++;
        } }
      b++;
    } while (b < grpcount[a][gnum]);
}

/*GROUP() - Controls the calculation of clusters*/
group(num)
     int num;
{
  int x, y, z, a, b, c;

  gnum = num;
  /*Check for spinup and spin down clusters*/
  for(posneg=0; posneg<2; posneg++) {
    /*Clear out old cluster values*/
    for(a=0; a<=grpmax[gnum]; a++) {
      for(b=0; b<=grpcount[a][gnum]; b++) {
        grp[gnum][a][b][0] = grp[gnum][a][b][1] =
          grp[gnum][a][b][2] = -1000;
      } }
    for(a=0; a<=grpmax[gnum]; a++) {
      grpcount[a][gnum] = 0;
    }
    grpmax[gnum] = 0;
    /*check each site to see if it is a part of a group*/
    for(x=1; x<=MAXX[num]; x++) {
      for(y=1; y<=MAXY[num]; y++) {
        for(z=1; z<=MAXZ[num]; z++) {
          if (m[x][y][z][num] != posneg)
            continue;
          c = 1000;
          for(a=0; a<=grpmax[gnum]; a++) {
        for(b=0; b<=grpcount[a][gnum]; b++) {
          if ((x == grp[gnum][a][b][0]) && (y == grp[gnum][a][b][1])
              && (z == grp[gnum][a][b][2]))
            c = a;
        } }
          /*If site is not in a cluster star a new cluster*/
          if (c == 1000) {
            c = grpmax[gnum];
            grpmax[gnum]++;
            b = grpcount[c][gnum] = 0;
            grp[gnum][c][b][0] = x;
            grp[gnum][c][b][1] = y;
            grp[gnum][c][b][2] = z;
            grpcount[c][gnum]++;
            printf("%d, %d, %d Starts Group %d\n", x-1, y-1, z-1, c);
            tree(num, c, posneg); /*Determine entire cluster*/
          }
        } } }
    gnum = gnum + 2;
  }
}

/*SMOOTH() - Perform group size approximation*/
smooth(num, newsize)
     int num, newsize;
{
  int a, b, x, y, z;
  /*If newsize is smaller then old size, recalculate initial clusters*/
  if (newsize < size) {
    for(x=1; x<=MAXX[num]; x++) {
      for(y=1; y<=MAXY[num]; y++) {
        for(z=1; z<=MAXZ[num]; z++) {
          m[x][y][z][num] = m[x][y][z][num+2];
        } } }
    group(num);
  }
  /*Invert spin vales of members of clusters smaller then newsize*/
  for(gnum=num; gnum<=num+2; gnum=gnum+2) {
    for(a=0; a<grpmax[gnum]; a++) {
      if (grpcount[a][gnum] <= newsize) {
        for(b=0; b<grpcount[a][gnum]; b++) {
     m[grp[gnum][a][b][0]][grp[gnum][a][b][1]][grp[gnum][a][b][2]][num]=
     !m[grp[gnum][a][b][0]][grp[gnum][a][b][1]][grp[gnum][a][b][2]][num];
        } } }
  }
  size = newsize;
  group(num); /*Calculate new clusters*/
}

/*Generate B-Spline Surface*/
bsurf(num)
     int num;
{
  DtReal xmin, ymin, zmin, xmax, ymax, zmax;
  DtReal xavg, yavg, zavg;
  int a, b, r, s, t, i;
  DtReal x, y, z;
  DtReal surf[4][4][4][3];
  DtReal r1, xoc, yoc, zoc, thea, ph;
  static DtReal subdivspc[5]={0.0617};
  DtReal v[16][4];
  static DtArea unit_square={0.,0., 10.,10.};
  static double bezier_knots[]={0.,0.,0.,0.,1., 1.,1.,1.};

  gnum = num;
  for(posneg=0; posneg<2; posneg++) {
    if (posneg == spin[num]) {gnum=gnum+2;  continue;}
    if (posneg == 0) DgAddObj(DoDiffuseColor(DcRGB, red));
    if (posneg == 1) DgAddObj(DoDiffuseColor(DcRGB, blue));
    for(a=0; a<grpmax[gnum]; a++) {
      if (grpcount[a][gnum] <= size)   continue;
      xmin = ymin = zmin = 100000.0;
      xmax = ymax = zmax = -100000.0;
      for(b=0; b<grpcount[a][gnum]; b++) {
        x = (double) grp[gnum][a][b][0];
        y = (double) grp[gnum][a][b][1];
        z = (double) grp[gnum][a][b][2];
        if (x >= xmax)
          xmax = x;
        if (x <= xmin)
          xmin = x;
        if (y >= ymax)
          ymax = y;
        if (y <= ymin)
        ymin = y;
        if (z >= zmax)
          zmax = z;
        if (z <= zmin)
          zmin = z;
    }
      /*Select point down to which the surface will collapse*/
```

```
xavg = grp[gnum][a][0][0];
yavg = grp[gnum][a][0][1];
zavg = grp[gnum][a][0][2];
if (m[(int)floor(xavg)][(int)floor(yavg)][(int)floor(zavg)][num]
    != posneg)
  printf("the center of group %d is unoccupied\n", a);
for(r=0; r<4; r++) {
  for(s=0; s<4; s++) {
    for(t=0; t<4; t++) {
      surf[r][s][t][0] = xmin + (r*(xmax - xmin)/3);
      surf[r][s][t][1] = ymin + (s*(ymax - ymin)/3);
      surf[r][s][t][2] = zmin + (t*(zmax - zmin)/3);
    } } }
for(r=0; r<4; r++) {
  for(s=0; s<4; s++) {
    for(t=0; t<4; t++) {
      if (!((r==0) || (r==3) || (s==0) || (s==3)
           || (t==0) || (t==3)))
        continue;
      x = surf[r][s][t][0];
      y = surf[r][s][t][1];
      z = surf[r][s][t][2];
      if (floor(x) == floor(x - 0.5))
        xoc = floor(x) + 1;
      else  xoc = floor(x);
      if (floor(y) == floor(y - 0.5))
        yoc = floor(y) + 1;
      else  yoc = floor(y);
      if (floor(z) == floor(z - 0.5))
        zoc = floor(z) + 1;
      else  zoc = floor(z);
      if (m[(int)xoc][(int)yoc][(int)zoc][num] == posneg) {
        continue;
      }
      x = surf[r][s][t][0] - xavg;
      y = surf[r][s][t][1] - yavg;
      z = surf[r][s][t][2] - zavg;
      r1 = sqrt(x*x + y*y + z*z);
      thea = atan2(y, x);
      ph = atan2(hypot(x, y), z);
      do {
        /*printf("r1 = %f\n", r1);*/
        x = r1*sin(ph)*cos(thea) + xavg;
        y = r1*sin(ph)*sin(thea) + yavg;
        z = r1*cos(ph) + zavg;
        if (floor(x) == floor(x - 0.5))
          xoc = floor(x) + 1;
        else  xoc = floor(x);
        if (floor(y) == floor(y - 0.5))
          yoc = floor(y) + 1;
        else  yoc = floor(y);
        if (floor(z) == floor(z - 0.5))
          zoc = floor(z) + 1;
        else  zoc = floor(z);
        r1 = r1 - (r1/50);
      }  while ((m[(int)xoc][(int)yoc][(int)zoc][num] != posneg)
                && (r1 > 0.5));
      surf[r][s][t][0] = x;
      surf[r][s][t][1] = y;
      surf[r][s][t][2] = z;
    } } }
/*Generate each of the B-Spline surfaces*/
r = i = 0;
for(s=0; s<4; s++) {
  for(t=0; t<4; t++) {
    v[i][0] = surf[r][s][t][0] + xmid[num] - dens;
    v[i][1] = surf[r][s][t][1] + ymid[num] - dens;
    v[i][2] = surf[r][s][t][2] + zmid[num] - dens;
    v[i][3] = 1.;
    i++;
  } }
DgAddObj(DoNURBSurf(DcRGB,DcCtr,unit_square,
                    4,8,bezier_knots,
                    4,8,bezier_knots,
                    4,4,v));
i = 0;
r = 3;
for(s=0; s<4; s++) {
  for(t=0; t<4; t++) {
    v[i][0] = surf[r][s][t][0] + xmid[num] - dens;
    v[i][1] = surf[r][s][t][1] + ymid[num] - dens;
    v[i][2] = surf[r][s][t][2] + zmid[num] - dens;
    v[i][3] = 1. ;
    i++;
  } }
DgAddObj(DoNURBSurf(DcRGB,DcCtr,unit_square,
                    4,8,bezier_knots,
                    4,8,bezier_knots,
                    4,4,v));
s = i = 0;
for(r=0; r<4; r++) {
  for(t=0; t<4; t++) {
    v[i][0] = surf[r][s][t][0] + xmid[num] - dens;
    v[i][1] = surf[r][s][t][1] + ymid[num] - dens;
    v[i][2] = surf[r][s][t][2] + zmid[num] - dens;
    v[i][3] = 1.;
    i++;
  } }
DgAddObj(DoNURBSurf(DcRGB,DcCtr,unit_square,
                    4,8,bezier_knots,
                    4,8,bezier_knots,
                    4,4,v));
i = 0;
s = 3;
for(r=0; r<4; r++) {
  for(t=0; t<4; t++) {
    v[i][0] = surf[r][s][t][0] + xmid[num] - dens;
    v[i][1] = surf[r][s][t][1] + ymid[num] - dens;
    v[i][2] = surf[r][s][t][2] + zmid[num] - dens;
    v[i][3] = 1.;
```

```
          i++;
        } }
      DgAddObj(DoNURBSurf(DcRGB,DcCtr,unit_square,
                          4,8,bezier_knots,
                          4,8,bezier_knots,
                          4,4,v));
      t = i = 0;
      for(r=0; r<4; r++) {
        for(s=0; s<4; s++) {
          v[i][0] = surf[r][s][t][0] + xmid[num] - dens;
          v[i][1] = surf[r][s][t][1] + ymid[num] - dens;
          v[i][2] = surf[r][s][t][2] + zmid[num] - dens;
          v[i][3] = 1.;
          i++;
        } }
      DgAddObj(DoNURBSurf(DcRGB,DcCtr,unit_square,
                          4,8,bezier_knots,
                          4,8,bezier_knots,
                          4,4,v));
      i = 0;
      t = 3;
      for(r=0; r<4; r++) {
        for(s=0; s<4; s++) {
          v[i][0] = surf[r][s][t][0] + xmid[num] - dens;
          v[i][1] = surf[r][s][t][1] + ymid[num] - dens;
          v[i][2] = surf[r][s][t][2] + zmid[num] - dens;
          v[i][3] = 1.;
          i++;
        } }
      DgAddObj(DoNURBSurf(DcRGB,DcCtr,unit_square,
                          4,8,bezier_knots,
                          4,8,bezier_knots,
                          4,4,v));
    }
    gnum= gnum +2;
  }
}

/*APPROX() - Do Course Approximation*/
approx(num)
     int num;
{
  int x, y, z, a, b, c, densnum;
  float prob;

  densnum = num + 2;
  MAXX[num] = MAXX[densnum]/(int)dens;
  MAXY[num] = MAXY[densnum]/(int)dens;
  MAXZ[num] = MAXZ[densnum]/(int)dens;
  for(x=0; x<=MAXX[num]+1; x++) {
    for(y=0; y<=MAXY[num]+1; y++) {
      for(z=0; z<=MAXZ[num]+1; z++) {
        m[x][y][z][num] = 0;
      } } }
  for(x=1; x<=MAXX[num]; x++) {
    for(y=1; y<=MAXY[num]; y++) {
      for(z=1; z<=MAXZ[num]; z++) {
        prob = 0;
        for(a=((x-1)*dens)+1; a<=(x*dens); a++) {
          for(b=((y-1)*dens)+1; b<=(y*dens); b++) {
            for(c=((z-1)*dens)+1; c<=(z*dens); c++) {
              prob = prob + m[a][b][c][densnum];
            } } }
        prob = prob/(dens*dens*dens);
        if(prob >= 0.25)
          m[x][y][z][num] = 1;
        else
          m[x][y][z][num] = 0;
      } } }
  xmid[num] = -1.0*(dens/2.0*MAXX[num])+(dens/2.0);
  ymid[num] = -1.0*(dens/2.0*MAXY[num])+(dens/2.0);
  zmid[num] = -1.0*(dens/2.0*MAXZ[num])+(dens/2.0);
}

/*MAKE_CONN2() - Produce Thin surface Bounding surrfaces*/
make_conn2(num)
     int num;
{
  int x, y, z;
  DtReal color[3];
  static DtReal squarex[] = {0.5, 0.5, 0.5,
                             0.5, 0.5, -0.5,
                             0.5, -0.5, -0.5,
                             0.5, -0.5, 0.5};
  static DtReal squarey[] = {0.5, 0.5, 0.5,
                             0.5, 0.5, -0.5,
                             -0.5, 0.5, -0.5,
                             -0.5, 0.5, 0.5};
  static DtReal squarez[] = {0.5, 0.5, 0.5,
                             0.5, -0.5, 0.5,
                             -0.5, -0.5, 0.5,
                             -0.5, 0.5, 0.5};

  for(posneg=0; posneg<2; posneg++) {
    if (posneg == spin[num])  continue;
    if (posneg == 0) DgAddObj(DoDiffuseColor(DcRGB, red));
    if (posneg == 1) DgAddObj(DoDiffuseColor(DcRGB, blue));

    for(x=0; x<=MAXX[num]; x++) {
      for(y=0; y<=MAXY[num]; y++) {
        for(z=0; z<=MAXZ[num]; z++) {
          if((m[x][y][z][num]) == (m[x+1][y][z][num]))
            continue;
          if((m[x][y][z][num]!=posneg)&&(m[x+1][y][z][num]!=posneg))
            continue;
          DgAddObj(DoPushMatrix());
          DgAddObj(DoTranslate(((x-1)*dens+xmid[num]),
                   ((y-1)*dens+ymid[num]),((z-1)*dens+zmid[num])));
          DgAddObj(DoScale(dens, dens, dens));
          DgAddObj(DoSimplePolygon(DcRGB, DcLoc, 4, squarex, DcConvex));
```

```
            DgAddObj(DoPopMatrix());
          } } }
      for(x=0; x<=MAXX[num]; x++) {
        for(y=0; y<=MAXY[num]; y++) {
          for(z=0; z<=MAXZ[num]; z++) {
            if((m[x][y][z][num]) == (m[x][y+1][z][num]))
              continue;
            if((m[x][y][z][num]!=posneg)&&(m[x][y+1][z][num]!=posneg))
              continue;
            DgAddObj(DoPushMatrix());
            DgAddObj(DoTranslate(((x-1)*dens+xmid[num]),
                       ((y-1)*dens+ymid[num]),((z-1)*dens+zmid[num])));
            DgAddObj(DoScale(dens, dens, dens));
            DgAddObj(DoSimplePolygon(DcRGB, DcLoc, 4, squarey, DcConvex));
            DgAddObj(DoPopMatrix());
          } } }
      for(x=0; x<=MAXX[num]; x++) {
        for(y=0; y<=MAXY[num]; y++) {
          for(z=0; z<=MAXZ[num]; z++) {
            if((m[x][y][z][num]) == (m[x][y][z+1][num]))
              continue;
            if((m[x][y][z][num]!=posneg)&&(m[x][y][z+1][num]!=posneg))
              continue;
            DgAddObj(DoPushMatrix());
            DgAddObj(DoTranslate(((x-1)*dens+xmid[num]),
                       ((y-1)*dens+ymid[num]),((z-1)*dens+zmid[num])));
                 DgAddObj(DoScale(dens, dens, dens));
            DgAddObj(DoSimplePolygon(DcRGB, DcLoc, 4, squarez, DcConvex));
            DgAddObj(DoPopMatrix());
          } } }
  }
}

/*MAKE_MESH() - Produce primative cubes bounding surface*/
make_mesh(num)
     int num;
{
  int x, y, z;
  DtReal color[3];

  for(x=1; x<=(MAXX[num]); x++) {
    for(y=1; y<=(MAXY[num]); y++) {
      for(z=1; z<=(MAXZ[num]); z++) {
        /*printf("(%d, %d, %d)%d %d\n", x, y, z, num, m[x][y][z][num]);*/
        if(m[x][y][z][num] == spin[num]) continue;
        if(m[x][y][z][num] == 0) DgAddObj(DoDiffuseColor(DcRGB, red));
        if(m[x][y][z][num] == 1) DgAddObj(DoDiffuseColor(DcRGB, blue));
        DgAddObj(DoPushMatrix());
        DgAddObj(DoTranslate(((x-1)*dens + xmid[num]),
                   ((y-1)*dens + ymid[num]), ((z-1)*dens+zmid[num])));
        DgAddObj(DoTranslate((-dens/2.0), (-dens/2.0), (-dens/2.0)));
        /*DgAddObj(DoTranslate((-.5), (-.5), (-.5)));*/
        DgAddObj(DoScale(dens, dens, dens));
        DgAddObj(DoPrimSurf(DcBox));
        DgAddObj(DoPopMatrix());
      } } }
  return (DoLabel(50));
}

/*MAKE_CONN() - Produce complex primatives bounding surface*/
make_conn(num)
     int num;
{
  int x, y, z;
  static DtReal white[] = {1.0, 1.0, 1.0};
  DtObject cyl();
  static DtReal squarexy[] = {0.0, 0.0, 0.0,
                              1.0, 0.0, 0.0,
                              1.0, 1.0, 0.0,
                              0.0, 1.0, 0.0};
  static DtReal squareyz[] = {0.0, 0.0, 0.0,
                              0.0, 1.0, 0.0,
                              0.0, 1.0, 1.0,
                              0.0, 0.0, 1.0};
  static DtReal squarezx[] = {0.0, 0.0, 0.0,
                              1.0, 0.0, 0.0,
                              1.0, 0.0, 1.0,
                              0.0, 0.0, 1.0};

  if (debug) printf("Make connectors!!!!");

  for(posneg=0; posneg<=1; posneg++) {
    if (posneg == spin[num])  continue;
    if (posneg == 0) DgAddObj(DoDiffuseColor(DcRGB, red));
    if (posneg == 1) DgAddObj(DoDiffuseColor(DcRGB, blue));

    for(x=1; x<=MAXX[num]; x++) {
      for(y=1; y<=MAXY[num]; y++) {
        for(z=1; z<=MAXZ[num]; z++) {
          if(m[x][y][z][num] != posneg)
            continue;
          if((m[x+1][y][z][num] == posneg) &&
             (m[x-1][y][z][num] == posneg))
            continue;
          if((m[x][y+1][z][num] == posneg) &&
             (m[x][y-1][z][num] == posneg))
            continue;
          if((m[x][y][z+1][num] == posneg) &&
             (m[x][y][z-1][num] == posneg))
            continue;
          DgAddObj(DoPushMatrix());
          DgAddObj(DoTranslate(((x-1)*dens+xmid[num]),
                   ((y-1)*dens+ymid[num]),((z-1)*dens+zmid[num])));
          DgAddObj(DoScale((.5*dens), (.5*dens), (.5*dens)));
          DgAddObj(DoPrimSurf(DcSphere));
          DgAddObj(DoPopMatrix());
        } } }
    for(y=1; y<=MAXY[num]; y++) {
      for(z=1; z<=MAXZ[num]; z++) {
        for(x=1; x<=MAXX[num]-1; x++) {
```

```
        if(m[x][y][z][num] != posneg)
          continue;
        if(m[x+1][y][z][num] != posneg)
          continue;
        if((m[x][y+1][z][num] == posneg) &&
           (m[x+1][y+1][z][num] == posneg)) {
          if((m[x][y-1][z][num] == posneg) &&
             (m[x+1][y-1][z][num] == posneg))
            continue;
        }
        if((m[x][y][z+1][num] == posneg) &&
           (m[x+1][y][z+1][num] == posneg)) {
          if((m[x][y][z-1][num] == posneg) &&
             (m[x+1][y][z-1][num] == posneg))
            continue;
        }
        DgAddObj(DoPushMatrix());
        DgAddObj(DoTranslate(((x-1)*dens+xmid[num]),
                  ((y-1)*dens+ymid[num]),((z-1)*dens+zmid[num])));
        DgAddObj(DoRotate(DcYAxis, -Pi/2));
        DgAddObj(DoRotate(DcXAxis, 0.0));
        DgAddObj(DoScale((.5*dens), (.5*dens), (-1.0*dens)));
        /*DgAddObj(DoScale(0.5, 0.5, -1.0));*/
        DgAddObj(DoSubDivSpec(DcSubDivRelative, 4.0));
        DgAddObj(DoPrimSurf(DcCylinder));
        DgAddObj(DoPopMatrix());
      } } }
  for(x=1; x<=MAXX[num]; x++) {
    for(z=1; z<=MAXZ[num]; z++) {
      for(y=1; y<=MAXY[num]-1; y++) {
        if(!m[x][y][z][num] == posneg)
          continue;
        if(!m[x][y+1][z][num] == posneg)
          continue;
        if((m[x+1][y][z][num] == posneg) &&
           (m[x+1][y+1][z][num] == posneg)) {
          if((m[x-1][y][z][num] == posneg) &&
             (m[x-1][y+1][z][num] == posneg))
            continue;
        }
        if((m[x][y][z+1][num] == posneg) &&
           (m[x][y+1][z+1][num] == posneg)) {
          if((m[x][y][z-1][num] == posneg) &&
             (m[x][y+1][z-1][num] == posneg))
            continue;
        }
        DgAddObj(DoPushMatrix());
        DgAddObj(DoTranslate(((x-1)*dens+xmid[num]),
                   ((y-1)*dens+ymid[num]),((z-1)*dens+zmid[num])));
        DgAddObj(DoRotate(DcYAxis, Pi/2));
        DgAddObj(DoRotate(DcXAxis, Pi/2));
        DgAddObj(DoScale((.5*dens), (.5*dens), (-1.0*dens)));
        DgAddObj(DoSubDivSpec(DcSubDivRelative, 4.0));
        DgAddObj(DoPrimSurf(DcCylinder));
        DgAddObj(DoPopMatrix());
      } } }
  for(x=1; x<=MAXX[num]; x++) {
    for(y=1; y<=MAXY[num]; y++) {
      for(z=1; z<=MAXZ[num]-1; z++) {
        if(!m[x][y][z][num] == posneg)
          continue;
        if(!m[x][y][z+1][num] == posneg)
          continue;
        if((m[x+1][y][z][num] == posneg) &&
           (m[x+1][y][z+1][num] == posneg)) {
          if((m[x-1][y][z][num] == posneg) &&
             (m[x-1][y][z+1][num] == posneg))
            continue;
        }
        if((m[x][y+1][z][num] == posneg) &&
           (m[x][y+1][z+1][num] == posneg)) {
          if((m[x][y-1][z][num] == posneg) &&
             (m[x][y-1][z+1][num] == posneg))
            continue;
        }
        DgAddObj(DoPushMatrix());
        DgAddObj(DoTranslate(((x-1)*dens+xmid[num]),
                   ((y-1)*dens+ymid[num]),((z-1)*dens+zmid[num])));
        DgAddObj(DoScale((.5*dens), (.5*dens), (1.0*dens)));
        DgAddObj(DoRotate(DcYAxis, 0.0));
        DgAddObj(DoRotate(DcXAxis, 0.0));
        DgAddObj(DoSubDivSpec(DcSubDivRelative, 4.0));
        DgAddObj(DoPrimSurf(DcCylinder));
        DgAddObj(DoPopMatrix());
      } } }
  for(z=1; z<=MAXZ[num]; z++) {
    for(x=1; x<=MAXX[num]-1; x++) {
      for(y=1; y<=MAXY[num]-1; y++) {
        if(!m[x][y][z][num] == posneg)
          continue;
        if(!m[x+1][y][z][num] == posneg)
          continue;
        if(!m[x][y+1][z][num] == posneg)
          continue;
        if(!m[x+1][y+1][z][num] == posneg)
          continue;
        if((m[x][y][z+1][num] == posneg) &&
           (m[x+1][y][z+1][num] == posneg)) {
          if((m[x][y+1][z+1][num] == posneg) &&
             (m[x+1][y+1][z+1][num] == posneg))
            continue;
        }
        DgAddObj(DoPushMatrix());
        DgAddObj(DoTranslate(((x-1)*dens+xmid[num]),
                   ((y-1)*dens+ymid[num]),((z-1)*dens+zmid[num])));
        DgAddObj(DoTranslate(0.0, 0.0, (0.5*dens)));
        DgAddObj(DoScale((dens), (dens), (1.0)));
        DgAddObj(DoSimplePolygon(DcRGB, DcLoc, 4, squarexy, DcConvex));
        DgAddObj(DoPopMatrix());
      } } }
```

```
for(z=1; z<=MAXZ[num]; z++) {
  for(x=1; x<=MAXX[num]-1; x++) {
    for(y=1; y<=MAXY[num]-1; y++) {
      if(!m[x][y][z][num] == posneg)
        continue;
      if(!m[x+1][y][z][num] == posneg)
        continue;
      if(!m[x][y+1][z][num] == posneg)
        continue;
      if(!m[x+1][y+1][z][num] == posneg)
        continue;
      if((m[x][y][z-1][num] == posneg) &&
         (m[x+1][y][z-1][num] == posneg)) {
        if((m[x][y+1][z-1][num] == posneg) &&
           (m[x+1][y+1][z-1][num] == posneg))
          continue;
      }
      DgAddObj(DoPushMatrix());
      DgAddObj(DoTranslate(((x-1)*dens+xmid[num]),
                 ((y-1)*dens+ymid[num]),((z-1)*dens+zmid[num])));
      DgAddObj(DoTranslate(0.0, 0.0, (-0.5*dens)));
      DgAddObj(DoScale((dens), (dens), (1.0)));
      DgAddObj(DoSimplePolygon(DcRGB, DcLoc, 4, squarexy, DcConvex));
      DgAddObj(DoPopMatrix());
    } } }
for(x=1; x<=MAXX[num]; x++) {
  for(y=1; y<=MAXY[num]-1; y++) {
    for(z=1; z<=MAXZ[num]-1; z++) {
      if(!m[x][y][z][num] == posneg)
        continue;
      if(!m[x][y+1][z][num] == posneg)
        continue;
      if(!m[x][y][z+1][num] == posneg)
        continue;
      if(!m[x][y+1][z+1][num] == posneg)
        continue;
      if((m[x+1][y][z][num] == posneg) &&
         (m[x+1][y+1][z][num] == posneg)) {
        if((m[x+1][y][z+1][num] == posneg) &&
           (m[x+1][y+1][z+1][num] == posneg))
          continue;
      }
      DgAddObj(DoPushMatrix());
      DgAddObj(DoTranslate(((x-1)*dens+xmid[num]),
                 ((y-1)*dens+ymid[num]),((z-1)*dens+zmid[num])));
      DgAddObj(DoTranslate((0.5*dens), 0.0, 0.0));
      DgAddObj(DoScale((1.0), (dens), (dens)));
      DgAddObj(DoSimplePolygon(DcRGB, DcLoc, 4, squareyz, DcConvex));
      DgAddObj(DoPopMatrix());
    } } }
for(x=1; x<=MAXX[num]; x++) {
  for(y=1; y<=MAXY[num]-1; y++) {
    for(z=1; z<=MAXZ[num]-1; z++) {
      if(!m[x][y][z][num] == posneg)
        continue;
        if(!m[x][y+1][z][num] == posneg)
          continue;
        if(!m[x][y][z+1][num] == posneg)
          continue;
        if(!m[x][y+1][z+1][num] == posneg)
          continue;
        if((m[x-1][y][z][num] == posneg) &&
           (m[x-1][y+1][z][num] == posneg)) {
          if((m[x-1][y][z+1][num] == posneg) &&
             (m[x-1][y+1][z+1][num] == posneg))
            continue;
        }
        DgAddObj(DoPushMatrix());
        DgAddObj(DoTranslate(((x-1)*dens+xmid[num]),
                    ((y-1)*dens+ymid[num]),((z-1)*dens+zmid[num])));
        DgAddObj(DoTranslate((-0.5*dens), 0.0, 0.0));
        DgAddObj(DoScale((1.0), (dens), (dens)));
        DgAddObj(DoSimplePolygon(DcRGB, DcLoc, 4, squareyz, DcConvex));
        DgAddObj(DoPopMatrix());
      } } }
  for(y=1; y<=MAXY[num]; y++) {
    for(z=1; z<=MAXZ[num]-1; z++) {
      for(x=1; x<=MAXX[num]-1; x++) {
        if(!m[x][y][z][num] == posneg)
          continue;
        if(!m[x+1][y][z][num] == posneg)
          continue;
        if(!m[x][y][z+1][num] == posneg)
          continue;
        if(!m[x+1][y][z+1][num] == posneg)
          continue;
        if((m[x][y+1][z][num] == posneg) &&
           (m[x+1][y+1][z][num] == posneg)) {
          if((m[x][y+1][z+1][num] == posneg) &&
             (m[x+1][y+1][z+1][num] == posneg))
            continue;
        }
        DgAddObj(DoPushMatrix());
        DgAddObj(DoTranslate(((x-1)*dens+xmid[num]),
                    ((y-1)*dens+ymid[num]),((z-1)*dens+zmid[num])));
        DgAddObj(DoTranslate(0.0, (0.5*dens), 0.0));
        DgAddObj(DoScale((dens), (1.0), (dens)));
        DgAddObj(DoSimplePolygon(DcRGB, DcLoc, 4, squarezx, DcConvex));
        DgAddObj(DoPopMatrix());
      } } }
  for(y=1; y<=MAXY[num]; y++) {
    for(z=1; z<=MAXZ[num]-1; z++) {
      for(x=1; x<=MAXX[num]-1; x++) {
        if(!m[x][y][z][num] == posneg)
          continue;
        if(!m[x+1][y][z][num] == posneg)
          continue;
        if(!m[x][y][z+1][num] == posneg)
          continue;
        if(!m[x+1][y][z+1][num] == posneg)
```

```
            continue;
          if((m[x][y-1][z][num] == posneg) &&
             (m[x+1][y-1][z][num] == posneg)) {
            if((m[x][y-1][z+1][num] == posneg) &&
               (m[x+1][y-1][z+1][num] == posneg))
              continue;
          }
          DgAddObj(DoPushMatrix());
          DgAddObj(DoTranslate(((x-1)*dens+xmid[num]),
                     ((y-1)*dens+ymid[num]),((z-1)*dens+zmid[num])));
          DgAddObj(DoTranslate(0.0, (-0.5*dens), 0.0));
          DgAddObj(DoScale((dens), (1.0), (dens)));
          DgAddObj(DoSimplePolygon(DcRGB, DcLoc, 4, squarezx, DcConvex));
          DgAddObj(DoPopMatrix());
        } } }
  }
  return (DoLabel(100));
}
```

C.3 My.h

```
#ident  "@(#)my.h      1.1"    9/1/88

#define MAX 64  /*Maximum lattice size*/
#define Pi 3.1415927

extern DtInt debug;
extern DtObject models[4];
extern DtReal bg[3];
static int MAXARRAY[2], MAXX[4], MAXY[4], MAXZ[4];
float dist_mx;
float dist_mn;
char file0[100];
char file1[100];
int m[MAX][MAX][MAX][4];
static float xmid[2], ymid[2], zmid[2];
float zmx[2], zmn[2], dens;
int size, posneg;
int prob_min;
float planeat[3];
float dev[3];
float Res;
int Act_Wind, Connects;
DtObject models[4];
int spin[2], gnum;
int modtype0, modtype1;
static DtReal white[] = {1.0, 1.0, 1.0};
static DtReal red[] = {1.0, 0.0, 0.0};
static DtReal blue[] = {0.0, 0.0, 1.0};
float intensity;
extern DtInt update_modified;
int grp[4][70][4100][3], grpmax[4], grpcount[70][4];
```

Appendix D
Modifications to the Display Programs

This software package relies heavily upon the user interface software supplied by Ardent for the creation of icons and other screen interface windows. Numerous modifications were made to the Ardent software to make it more appropriate for this use.

Including a complete listing of all the modified programs would be excessive. Instead, I have listed the sections of code that were modified in their modified form. Each section represents a different program. The number of the altered line is followed by a letter indicating whether the line was 'changed', 'added' or 'deleted'. This is followed by the corect form of the line.

D.1 Butt.c

```
13a     #include "butt.h"
15c     int      mouse_hit;
        Button   buttons[MaxButs];
        int       butstat[MaxButs];
        int      xyloc[MaxButs][2];
        int Nbuts = 0;
22c     char *buttupimg[NumButtStates] = {
29c     char *buttdnimg[NumButtStates] = {
36c     XImage    *buttup[NumButtStates];
        XImage    *buttdn[NumButtStates];
39c     XFontStruct *font_struct1 = 0, *font_struct2 = 0;
        unsigned int textheight1,textheight2;
257a    mouse_hit = 1;
```

D.2 Dui.c

```
404c    cmap = XTitanDefaultDirectColormap(display,
                     XDefaultRootWindow(display));
```

D.3 MkModels.c

```
39c     DtObject model_group[3];
116a    if (nmodels == 1) {
139a    }
145c    model_group[nmodels] = DoGroup(DcTrue);
178a    if (nmodels == 2) {
        DgAddObj(models[1]);
        }
184c    DsHoldObj(model_group[nmodels]);
```

D.4 Render.c

```
36c     DtObject device0;
        DtObject frame0;
        DtObject view0;
        DtObject device1;
        DtObject frame1;
        DtObject view1;
44a     DtReal bg[3];
65c     DtInt window0=0;
        DtInt window1=0;
150a    extern int Act_Wind;
        char rayname[50];
177c    extern DtObject studio_group, model_group[3];
189d    /*make_models(argc, argv);*/
194d    /*make_studios();*/
200c    nmodels = 1;   /*Set so MkModels uses model[0]*/
        make_models(argc, argv);
        make_studios();
        DgAddObjToGroup(DvInqDefinitionGroup(view0),studio_group);
        DgAddObjToGroup(DvInqDisplayGroup(view0),model_group[nmodels]);
        nmodels = 2;
        make_models(argc, argv);
        make_studios();
        DgAddObjToGroup(DvInqDefinitionGroup(view1),studio_group);
        DgAddObjToGroup(DvInqDisplayGroup(view1),model_group[nmodels]);
207c    DdUpdate(device0, DcFalse);
        DdUpdate(device1, DcFalse);
        geom_spec(&xbound, argc, argv);
        DdUpdate(device1, DcFalse);
        DdUpdate(device0, DcFalse);
223c    DsReleaseObj(device0);
        DsReleaseObj(device1);
342c    prsarg_get_keyword_int(argc,argv,"-w",0,&window0);
372c    window0 = ui_init(pseudo, singleb, 0);
415c    if(window0 != 0) {
417c    window0);
431c    if (!(device0 = DoDevice(devicetype,devicefile))) {
           printf("can't create device0:  type = '%s', file = '%s'\n",
437c    DdInqExtent(device0,&bound);
```

```
442c    DdSetViewport(device0,&bound);
447c    if (!(frame0 = DoFrame())) {
451c    DdSetFrame(device0,frame0);
        if (pseudo) DdSetShadeMode(device0,shade);
        DfSetBoundary(frame0,&bound);
457c    if (!(view0 = DoView())) {
461c    DvSetClearFlag(view0,DcTrue);
        DvSetRendStyle(view0,renderstyle);
464c    DgAddObjToGroup(DfInqViewGroup(frame0),view0);
        DvSetBoundary(view0,&bound);
        DvSetBackgroundColor(view0,DcRGB,backcolor);
        if (pseudo) create_color_table(device0,shade);
472c    DdInqViewport(device0,&bound);
        printf("device0=%lf,%lf,%lf\n      %lf,%lf,%lf\n",
476c    DfInqBoundary(frame0,&bound);
        printf("frame0=%lf,%lf,%lf\n      %lf,%lf,%lf\n",
480c    DvInqBoundary(view0,&bound);
        printf("view0=%lf,%lf,%lf\n      %lf,%lf,%lf\n",
484a    if(window1 != 0) {
          sprintf(dummy,"%s -display %d -window %d",devicefile,display,
                  window1);
           strcpy(devicefile,dummy);
         }
485a     if (debug) printf("DoDevice(type = '%s', file = '%s')\n",
                          devicetype,devicefile);

         if (!(device1 = DoDevice(devicetype,devicefile))) {
           printf("can't create device1:  type = '%s', file = '%s'\n",
                  devicetype,devicefile);
         exit(1);
         }

         DdInqExtent(device1,&bound);
         devicewidth = bound.fur[0] - bound.bll[0];
         deviceheight = bound.fur[1] - bound.bll[1];
         if(debug) printf("device1 width = %lf, height = %lf\n",
                         devicewidth, deviceheight);
         DdSetViewport(device1,&bound);

         if (!(frame1 = DoFrame())) {
                 printf("can't create frame\n");
                 exit(1);
         }
         DdSetFrame(device1,frame1);
         if (pseudo) DdSetShadeMode(device1,shade);
         DfSetBoundary(frame1,&bound);

         if (!(view1 = DoView())) {
                 printf("can't create view1\n");
                 exit(1);
         }
         DvSetClearFlag(view1,DcTrue);
         DvSetRendStyle(view1,renderstyle);

         DgAddObjToGroup(DfInqViewGroup(frame1),view1);
        DvSetBoundary(view1,&bound);
        DvSetBackgroundColor(view1,DcRGB,backcolor);
        if (pseudo) create_color_table(device1,shade);

        if(debug) {
                DdInqViewport(device1,&bound);
                printf("device1=%lf,%lf,%lf\n      %lf,%lf,%lf\n",
                        bound.bll[0], bound.bll[1], bound.bll[2],
                        bound.fur[0], bound.fur[1], bound.fur[2]);
                DfInqBoundary(frame1,&bound);
                printf("frame1=%lf,%lf,%lf\n      %lf,%lf,%lf\n",
                        bound.bll[0], bound.bll[1], bound.bll[2],
                        bound.fur[0], bound.fur[1], bound.fur[2]);
                DvInqBoundary(view1,&bound);
                printf("view1=%lf,%lf,%lf\n      %lf,%lf,%lf\n",
                        bound.bll[0], bound.bll[1], bound.bll[2],
                        bound.fur[0], bound.fur[1], bound.fur[2]);
        }
523c    if (Act_Wind != 1) {
               DdUpdate(device0, DcFalse);
               DdUpdate(device0, DcTrue);
        }
        if (Act_Wind != 2) {
               DdUpdate(device1, DcFalse);
               DdUpdate(device1, DcTrue);
        }
        update_modified = DcTrue;
        script_in = 1;
        parse_input("Dh 0.000000");
        script_in = 0;
561c    update = pipe_check();
623d
636c    if (Act_Wind != 1) {
          DdUpdate(device0, DcFalse);
          DdUpdate(device0, DcTrue);
        }
        if (Act_Wind != 2) {
          DdUpdate(device1, DcFalse);
          DdUpdate(device1, DcTrue);
        }
        update_modified = DcTrue;
684c    if (Act_Wind != 1) {
           DdUpdate(device0, DcFalse);
           DdUpdate(device0, DcTrue);
        }
        if (Act_Wind != 2) {
           DdUpdate(device1, DcFalse);
           DdUpdate(device1, DcTrue);
        }
        update_modified = DcTrue;
        /*update = 0;*/
711c    float value;
764c    DvSetBackgroundColor(view0,DcRGB,bg);
        DvSetBackgroundColor(view1,DcRGB,bg);
        update_modified = DcFalse;
```

```
804c     update_modified = DcFalse;
810c     update_modified = DcFalse;
837c     width = 517; height = 636;
         sscanf(command, "%s", rayname);
         printf("rayname = %s\n", rayname);
840c     DdUpdate(device0,DcFalse);
853c     sscanf(command,"%f",&value);
857c     printf("Setting subdivision level to %f\n",
875c     DdUpdate(device0,DcTrue);
911c     update_modified = DcFalse;
916c     update_modified = DcFalse;
921c     sscanf(command,"%*s %f",&value);
924c     if(debug) printf("Dial val: %f\n",value);
989c               update_modified = DcFalse;
994c               update_modified = DcFalse;
1108d
1268c    int width, height;
1272c    extern DtObject studio_group, model_group[3];
1276c    printf("Raytrace res: %dx%d file %s\n", width, height, rayname);
1277a    if (strcmp(rayname, "left") == 0)
                ray_device = device0;
         else if (strcmp(rayname, "right") == 0)
                ray_device = device1;
1282c    else {
         sprintf(arg, "-filename %s -width %d -height %d",
                rayname, width, height);
1289c   } }
1319c   DgAddObjToGroup(DvInqDisplayGroup(ray_view),model_group[1]);
1325c   DdUpdate(ray_device,DcFalse);
```

D.5 Ui.c

```
26a     Window data_window;
        Window dore_window0;
        Window dore_window1;
36a     int trying;
53c     int ui_init(pseudo, singleb, num)
        int pseudo, singleb, num;
101a    if (num == 0) {
103c    instruct_window = init_window(0,0,3,3,0,1);
116c    XMoveResizeWindow(display,xtermwindow,0,dore_hnew+spc,
                 dore_wnew,screen_h-dore_hnew);
124a    system("xterm =72x32+518+638 -name bob &");
        trying = getpid();
        trying = trying + 2;
        if (debug) printf("Window id = %d\n", trying);
125d    /*if (debug) printf("New Window.....");
        data_window = XCreateSimpleWindow(display,0,119,138,117,182,2);
            dore_wnew+spc,dore_hnew+spc,dore_wnew,screen_h-dore_hnew,1);
        XMapWindow(display, data_window);
        XFlush(display);*/
126,127c
```

```
        dore_window0 = init_window(0,0,dore_wnew,dore_hnew,pseudo,
               (singleb?1:2));
        XClearWindow(display,dore_window0);
        }
128a    else {
        if (debug) {printf("dore...");fflush(stdout);}
         dore_window1 = init_window(dore_wnew+spc,0,dore_wnew,dore_hnew,
               pseudo,(singleb?1:2));
         XClearWindow(display,dore_window1);
        }
143c    XDefineCursor(display,dore_window0,waitcurs);
        XDefineCursor(display,dore_window1,waitcurs);
152c    printf("dore_window = %#x\n",dore_window0);
161c    if (num == 0)
           return((int)dore_window0);
        if (num == 1)
           return((int)dore_window1);
380c    XDefineCursor(display,dore_window0,dore);
        XDefineCursor(display,dore_window1,dore);
416c    if (bevent.xany.window == dore_window0) {
```

D.6 butt.h

```
#define MaxButs 12

extern mouse_hit;
extern Button   buttons[MaxButs];
extern int      butstat[MaxButs];
extern int      xyloc[MaxButs][2];
extern int Nbuts;

extern char *buttupimg[NumButtStates];
extern char *buttdnimg[NumButtStates];
extern XImage   *buttup[NumButtStates];
extern XImage   *buttdn[NumButtStates];

extern XFontStruct *font_struct1, *font_struct2;
extern unsigned int textheight1,textheight2;
```

D.7 ui.h

```
41c     #define spc 2
45a     #define dore_hnew 636
        #define dore_wnew 517
```

References

[Barsky 85] Barsky, Brian A.
Computer Graphics and Geometric Modeling Using Beta-Splines.
Springer-Verlag, 1985.

[Doerschuk 88] Doerschuk, Peter.
AFOSR Renewal Proposal.
September, 1988.
Includes additional untitled notes.

[Huang 63] Huang, K.
Statistical Mechanics.
McGraw-Hill Book Company, 1963.

[Newman and Sproull 79]
Newman, William M. and Sproull, Robert F.
Principles of Interactive Computer Graphics.
McGraw-Hill Book Company, 1979.

[Rational B-Splines 83]
Tiller, Wayne.
Rational B-Splines for Curve and Surface Representation.
IEEE Computer Graphics and Applications , September, 1983.

[Starbase Graphics 88]
Starbase Graphics Techniques HP-UX Concepts and Tutorials
Edition 2.0 edition, Hewlett-Packard Corporation, 1988.

[Titan Architecture 88]
Diede, Tom et. al.
The Titan Graphics Supercomputer Architecture.
Computer , September, 1988.