VISUALIZATION BASED ON INTERACTIVE CLIPPING: APPLICATION TO CONFOCAL DATA

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science in Systems Science

In

The Interdepartmental Program in Systems Science

By
Gaurav Khanduja
B.E. Computer Science & Engg, Gorakhpur University 2002
May, 2005

Acknowledgements

I would like to express my sincere gratitude to Dr. Bijaya B. Karki, my advisor, for his invaluable guidance and encouragement extended throughout the study. His tenacious supervision, helpful suggestion, patience and time deserve a special mention. I have learnt the basics of scientific visualization under his guidance and would like to acknowledge his command in this area of research.

I would like to express my appreciation to my committee members Dr. S. Sitharama Iyengar and Dr. Rajgopal Kannan for their support and suggestions. Thanks are extended to Dr. Bijaya B. Karki for honing my fundamentals in Scientific Visualization.

I would like to express my gratitude towards my family members, friends and relatives whose encouragement and support has been a constant source of inspiration during my stay at Louisiana State University.

Last, but not the least, I would like to gratefully acknowledge Department of Computer Science, Louisiana State University for providing the resources and need during the project.

Table of Contents

Acknowledgements ii

List of Tables v

List of Figures vi

Abstract viii

Chapter 1: Introduction 1

Chapter 2: Overview of Visualization Techniques 3
- 2.1 Classification 3
- 2.2 Surface Rendering Techniques: 4
  - 2.2.1 Scalar Glyphs 4
  - 2.2.2 Isosurface 4
  - 2.2.3 Vector Glyphs 4
  - 2.2.4 Streamlines and Streaklines 5
  - 2.2.5 Textures 5
- 2.3 Volume Rendering Techniques 5
  - 2.3.1 Ray Casting 7
  - 2.3.2 Splatting 7
- 2.4 Clipping Techniques 8
  - 2.4.1 Line Clipping 8
  - 2.4.2 Polygon Clipping 10
  - 2.4.3 Box and Sphere Clipping 13

Chapter 3: Interactive Clipping 15
- 3.1 Clipping Planes In OpenGL 15
- 3.2 Additional Clipping Planes 15
- 3.3 Interactive Clipping 17
  - 3.3.1 Steps in Interactive Clipping 17
  - 3.3.2 Design and Implementation 21

Chapter 4: Application 29
- 4.1 Confocal Data 29
  - 4.1.1 Confocal Microscopy 29
  - 4.1.2 Confocal System 30
  - 4.1.3 Features of Confocal Images 30
  - 4.1.4 Application Description 30
  - 4.1.5 Rendering: Texture vs Pixel Mapping 35
- 4.2 Charge Density 37

Chapter 5: Conclusion 42

References 43

Vita……………………………………………………………………..………………..45

List of Tables

Table 4.1: Time Taken in Rendering Images 36

List of Figures

Figure 2.1: Codes for the 9 regions associated to clipping rectangle 10

Figure 2.2: Example of Cohen-Sutherland line-clipping algorithm 10

Figure 2.3 Polygon Clipping 11

Figure 2.4 Box Clipping 13

Figure 2.5 Sphere Clipping 13

Figure 3.1 Additional Clipping Planes and the Viewing Volume 16

Figure 3.2 Initial Orientation of Object 18

Figure 3.3 Object with Clipping Plane 18

Figure 3.4 Object with Rotated Clipping Plane 19

Figure 3.5 Object with Translated Clipping Plane 19

Figure 3.6 Object after First Rotation... 20

Figure 3.7 Object after Second Rotation Showing Clipped Surface 20

Figure 3.8 Object Back to Step 2.. 21

Figure 3.9 Schematic Diagram of Code... 23

Figure 4.1 Laser as Point Light Source 29

Figure 4.2 Confocal Image 31

Figure 4.3 Stacked Images... 32

Figure 4.4 Rotated Images with Clipping... 32

Figure 4.5 Translated Images... 33

Figure 4.6 Clipped Images 33

Figure 4.7 Image Showing Best View.. 34

Figure 4.8 Graph Between Pixel & Texture Rendering 36

Figure 4.9 Structure of the MgO Charge Density 38

Figure 4.10 3D View of MgO (136 atoms) Charge Density 39

Figure 4.11 Interactive Clipping 39

Figure 4.12 Rotation with Interactive Clipping 40

Figure 4.13 Interactive Clipping – Movement of Clipping Plane 40

Abstract

We have explored how clipping can be exploited in an interactive manner to visualize massive three-dimensional datasets. In essence, the proposed interactive clipping approach involves the dynamic adjustment of the clipping plane to expose any cross-section of the volume data and subsequent adjustment of the clipped surface to the best view position using a combination of rotation and translation. The thesis describes the design, implementation and application of our interactive-clipping-based visualization system. The implementation is done with OpenGL and C++, thus resulting in a highly portable and flexible system. For illustration, two types of scientific datasets, confocal data of a plant stem and calculated electronic charge density distributions are successfully visualized. The data are displayed using pixel- and texture-based rendering; the latter is shown to give a better performance.

Chapter 1: Introduction

Scientific visualization deals with the representation of data in a graphical manner to provide understanding and insight into the data. It is also sometimes called to as visual data analysis which is nothing but rendering and visualizing the data graphically. From a computer science perspective, scientific visualization is part of a greater field called visualization and involves research in computer graphics, image processing, bio-medical imaging, high performance computing, material analysis and other areas closely related to scientific visualization. The same tools that are used for scientific visualization may be applied to other fields discussed above animation, or multimedia presentation.

As a science, scientific visualization is the study concerned with the interactive display and analysis of data which can be of any dimension or type. Researchers often like the ability to do real-time visualization of data from any source. As an emerging science, its strategy is to develop fundamental ideas leading to general tools for real applications which can be used by the researchers or the end users.

The major advantages for scientific visualization are as follows: it will compress a lot of data into one picture (data browsing), it can reveal correlations between different quantities both in space and time, it can furnish new space-like structures beside the ones which are already known from previous calculations, and it opens up the possibility to view the data selectively and interactively in 'real time'. By following the formation and the deformation as well as the motions of these structures in time using the dataset obtained as a result of simulations, one can gain insight into the complicated dynamics. Simulation codes can also be integrated into a visualization environment in order to

analyze the data 'real time' and to by-pass the need to store every intermediate result for later analysis.

The objective of this project was to visualize the set of confocal images and the charge density of the MgO. Confocal images are the set of images generated by using confocal microscopy. These images are rendered using two different schemes; pixel based rendering and texture rendering and after this the process of interactive clipping developed as part of this project is applied to the data. The interactive clipping guarantees that a user will be able to view every possible cross section of the 3D data. This technique uses the clipping plane generated with the help of the OpenGL and then applying some transformations to it to visualize the structure. The clipping plane can then be given necessary rotation and translation to get the desired view of the object. This method of Interactive clipping technique is applied to electronic charge density distribution of MgO. It can be used in this case to visualize the structure of the atom when it is having a vacancy. This technique is just not limited to these objects; it can be extended to any 3D structure which needs to be visualized. Further in this project, we also study and analyze the rendering schemes namely texture based rendering and pixel based rendering using examples.

Chapter 2: Overview of Visualization Techniques

2.1 Classification

Classification of visualization techniques is based on the dimension of the domain of the quantity that is visualized, i.e. the number of independent variables of the domain on which the quantity acts, and on the type of the quantity, i.e. scalar, vector, or tensor. For example in molecular dynamics, two scalar quantities occur, viz. temperature (or pressure) and density, and two vector quantities viz. magnetic field and velocity field. These quantities are defined on a four-dimensional domain which is spanned up by the space and time coordinates. The time dependence is treated different than other dependencies. In particular, animation is used to visualize this dependency.

Visualization techniques can also be divided into surface rendering techniques, (direct) volume rendering techniques and clipping techniques. Surface rendering is an indirect geometry based technique which is used to visualize structures in 3D scalar or vector fields by converting these structures into surface representations first and then using conventional computer graphics techniques to render these surfaces. Direct volume rendering is a visualization technique for 3D scalar data sets without a conversion to surface representations or without first fitting geometric primitives to the samples. Clipping techniques involves cutting planes and orthogonal slicers to view the data. These can be used in conjunction with the scalar and volume rendering techniques to visualize the data. Various types of these techniques are described in brief below to provide the overview.

2.2 Surface Rendering Techniques

This section briefly describes a general set of 3D scalar and vector surface rendering techniques. The first four descriptions deal with scalar field techniques and the other two with vector field techniques.

2.2.1 Scalar Glyphs

Scalar glyphs is a technique which puts a sphere or a diamond on every data point. The scale of the sphere or diamond is determined by the data value. The scalar glyphs may be colored according to the same scalar field or according to another scalar field. In this way correlations can be found. As no interpolations are needed for this technique as it consumes few CPU seconds.

2.2.2 Isosurface

This technique produces surfaces in the domain of the scalar quantity on which the scalar quantity has the same value, the so-called isosurface value. The surfaces can be colored according to the isosurface value or they can be colored according to another scalar field using the texture technique. The latter case allows for the search for correlation between different scalar quantities.

There are different methods to generate the surfaces from a discrete set of data points. All methods use interpolation to construct a continuous function. The correctness of the generated surfaces depends on how well the constructed continuous function matches the underlying continuous function representing the discrete data set.

2.2.3 Vector Glyphs

This technique uses needle or arrow glyphs to represent vectors at each data point. The direction of the glyph corresponds to the direction of the vector and its

magnitude corresponds to the magnitude of the vector. The glyphs can be colored according to a scalar field.

2.2.4 Streamlines and Streaklines

This is a set of methods for outlining the topology, i.e. the field lines, of a vector field. Generally, one takes a set of starting points, finds the vectors at these points by interpolation, if necessary, and integrates the points along the direction of the vector. At the new positions the vector values are found by interpolation and one integrates again. This process stops if a predetermined number of integration steps have been reached or if the points end up outside the data volume. The calculated points are connected by lines.

The difference between streamlines and streaklines is that the streamlines technique considers the vector field to be static whereas the streaklines technique considers the vector field to be time dependent. Hence, the streakline technique interpolates not only in the spatial direction, but also in the time direction.

2.2.5 Textures

This is a technique to color arbitrary surfaces, e.g. those generated by the isosurface techniques, according to a 3D scalar field. An interpolation scheme is used to determine the values of the scalar field on the surface. A color map is used to assign the color.

2.3 Volume Rendering Techniques

Volume rendering techniques have been developed to overcome problems of the accurate representation of surfaces in the isosurface techniques. In short, these problems are related to making a decision for every volume element whether or not the

surface passes through it and this can produce false positives (spurious surfaces) or false negatives (erroneous holes in surfaces), particularly in the presence of small or poorly defined features. Volume rendering does not use intermediate geometrical representations, in contrast to surface rendering techniques. It offers the possibility for displaying weak or fuzzy surfaces. This frees one from the requirement to make a decision whether a surface is present or not.

Volume rendering involves the following steps: the forming of an RGBA volume from the data, reconstruction of a continuous function from this discrete data set, and projecting it onto the 2D viewing plane (the output image) from the desired point of view. An RGBA volume is a 3D four-vector data set, where the first three components are the familiar R, G, and B color components and the last component, A, represents *opacity*. An opacity value of 0 means totally transparent and a value of 1 means totally opaque. Behind the RGBA volume an opaque background is placed. The mapping of the data to opacity values acts as a classification of the data one is interested in. Isosurfaces can be shown by mapping the corresponding data values to almost opaque values and the rest to transparent values. The appearance of surfaces can be improved by using shading techniques to form the RGB mapping. However, opacity can be used to see the interior of the data volume too. These interiors appear as clouds with varying density and color. A big advantage of volume rendering is that this interior information is not thrown away, so that it enables one to look at the 3D data set as a whole. Disadvantages are the difficult interpretation of the cloudy interiors and the long time, compared to surface rendering, needed to perform volume rendering.

2.3.1 Ray Casting

Several implementations exist for ray casting. We describe the implementation used in Visualization Data Explorer. For every pixel in the output image a ray is shot into the data volume. At a predetermined number of evenly spaced locations along the ray the color and opacity values are obtained by interpolation. The interpolated colors and opacities are merged with each other and with the background by *compositing* in back-to-front order to yield the color of the pixel. These compositing calculations are simply linear transformations. Specifically, the color of the ray *Cout* as it leaves each sample location, is related to the color *Cin* of the ray, as it enters, and to the color *c(xi)* and the opacity *a(x)* at that sample location by the transparency formula :

$$C_{out} = C_{in}(1 - a(x)) + c(xi) * a(x) \qquad \textbf{Eq 2.1}$$

Performing this formula in a back-to-front order, i.e. starting at the background and moving towards the image plane, will produce the pixel color. It is clear from the above formula that the opacity acts as a data selector. For example, sample points with opacity values close to 1 hide almost all the information along the ray between the background and the sample point and opacity values close to zero transfer the information almost unaltered. This way of compositing is equal to the dense-emitter model, where the color indicates the instantaneous emission rate and the opacity indicates the instantaneous absorption rate.

2.3.2 Splatting

This technique was developed to improve the speed of calculation of volume rendering techniques like ray casting, at the price of less accurate rendering. It differs from ray casting in the projection method. Splatting projects voxels, i.e. volume

elements, on the 2D viewing plane. It approximates this projection by a so-called *Gaussian splat*, which depends on the opacity and on the color of the voxel (other splat types, like linear splats can be used also). A projection is made for every voxel and the resulting splats are composted on top of each other in back-to-front order to produce the final image.

2.4 Clipping Techniques

2.4.1 Line Clipping

This section treats clipping of lines against rectangles. Although there are specialized algorithms for rectangle and polygon clipping, it is important to note that other graphic primitives can be clipped by repeated application of the line clipper.

2.4.1.1 Clipping Individual Points

If the x coordinate boundaries of the clipping rectangle are Xmin and Xmax, and the y coordinate boundaries are Ymin and Ymax, then the following inequalities must be satisfied for a point at (X, Y) to be inside the clipping rectangle:

$$Xmin < X < Xmax \qquad \text{and} \qquad Ymin < Y < Ymax$$

If any of the four inequalities does not hold, the point is outside the clipping rectangle.

2.4.1.2 Solve Simultaneous Equations

To clip a line, only endpoints need to be considered. If both endpoints of a line lie inside the clip rectangle, the entire line lies inside the clip rectangle and can be trivially accepted. If one endpoint lies inside and one outside, the line intersects the clip rectangle and we must compute the intersection point. If both endpoints are outside the clip rectangle, the line may or may not intersect with the clip rectangle, and then further calculations are needed to determine whether there are any intersections.

The brute-force approach to clipping a line that cannot be trivially accepted is to intersect that line with each of the four clip-rectangle edges to see whether any intersection points lie on those edges; if so, the line cuts the clip rectangle and is partially inside. For each line and clip-rectangle edge, we therefore take the two mathematically infinite lines that contain them and intersect them. Next, we test whether this intersection point is "interior" -- that is, whether it lies within both the clip rectangle edge and the line; if so, there is an intersection with the clip rectangle. In the first example, intersection points G' and H' are interior, but I' and J' are not.

2.4.1.3 The Cohen-Sutherland Line-Clipping Algorithm

The more efficient Cohen-Sutherland Algorithm performs initial tests on a line to determine whether intersection calculations can be avoided.

- **Steps for Cohen-Sutherland Algorithm**

1. End-points pairs are checked for trivial acceptance or trivial rejection using out code.
2. If not trivial-acceptance or trivial-rejected, divided into two segments at a clip edge.
3. Iteratively clipped by testing trivial-acceptance or trivial-rejected, and divided into two segments until completely inside or trivial-rejected.

- **Trivial acceptance/reject test**

To perform trivial accept and reject tests, we extend the edges of the clip rectangle to divide the plane of the clip rectangle into nine regions. Each region is assigned a 4-bit code determined by where the region lies with respect to the outside half planes of the clip-rectangle edges. Each bit in out code is set to either 1 (true) or 0 (false); the 4 bits in the code correspond to the following conditions:

- Bit 1 : outside half plane of top edge, above top edge Y > Ymax

- Bit 2 : outside half plane of bottom edge, below bottom edge Y < Ymin
- Bit 3 : outside half plane of right edge, to the right of right edge X > Xmax
- Bit 4 : outside half plane of left edge, to the left of left edge X < Xmin

1001	1000	1010
0001	0000	0010
0101	0100	0110

Figure 2.1: Codes for the 9 regions associated to clipping rectangle[23]

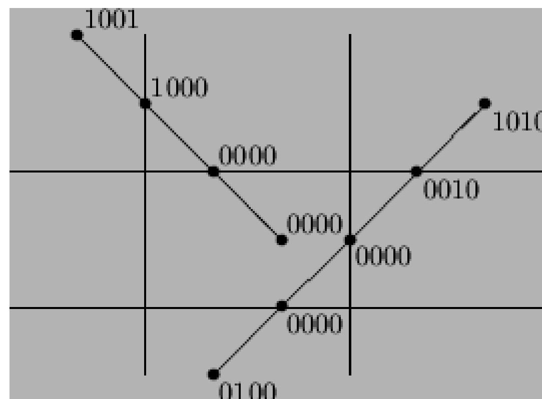

Figure 2.2: Example of Cohen-Sutherland line-clipping algorithm[23]

2.4.2 Polygon Clipping

An algorithm that clips a polygon must deal with many different cases. The case is particularly note worthy in that the concave polygon is clipped into two separate polygons. All in all, the task of clipping seems rather complex. Each edge of the polygon must be tested against each edge of the clip rectangle; new edges must be added, and existing edges must be discarded, retained, or divided. Multiple polygons may result from clipping a single polygon. We need an organized way to deal with all these cases.

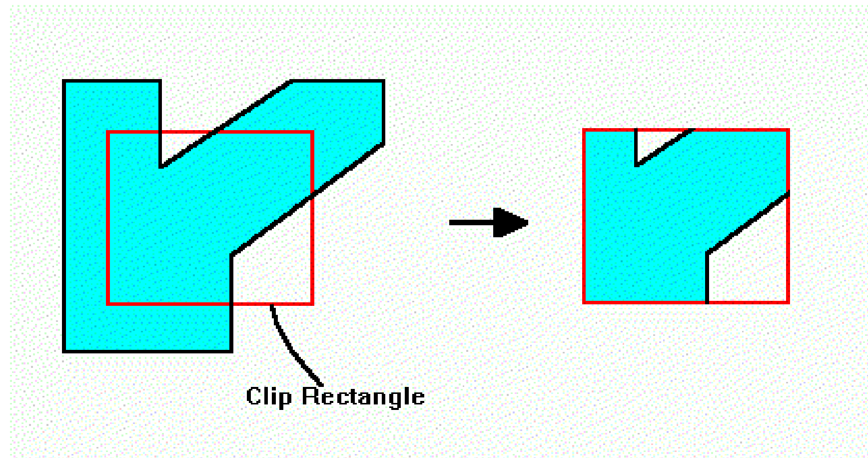

Figure 2.3 Polygon Clipping[7]

Sutherland and Hodgman's polygon-clipping algorithm uses a divide-and-conquer strategy: It solves a series of simple and identical problems that, when combined, solve the overall problem. The simple problem is to clip a polygon against a single infinite clip edge. Four clip edges, each defining one boundary of the clip rectangle, successively clip a polygon against a clip rectangle.

- **Steps of Sutherland-Hodgman's polygon-clipping algorithm**
  - Polygons can be clipped against each edge of the window one at a time. Windows/edge intersections, if any, are easy to find since the X or Y coordinates are already known.
  - Vertices which are kept after clipping against one window edge are saved for clipping against the remaining edges.
  - Note that the number of vertices usually changes and will often increases.
  - We are using the Divide and Conquer approach.

- **Four Cases of Polygon Clipping Against One Edge**

The clip boundary determines a visible and invisible region. The edges from vertex i to vertex i+1 can be one of four types:

- Case 1 : Wholly inside visible region - save endpoint
- Case 2 : Exit visible region - save the intersection
- Case 3 : Wholly outside visible region - save nothing
- Case 4 : Enter visible region - save intersection and endpoint

Because clipping against one edge is independent of all others, it is possible to arrange the clipping stages in a pipeline. The input polygon is clipped against one edge and any points that are kept are passed on as input to the next stage of the pipeline. These way four polygons can be at different stages of the clipping process simultaneously. This is often implemented in hardware.

- **Cutting Planes**

This technique makes it possible to view scalar data on a cross-section of the data volume with a cutting plane. One defines a regular, Cartesian grid on the plane and the data values on this grid are found by interpolation of the original data. A convenient color map is used to make the data visible.

- **Orthogonal Slicers**

It often occurs that one wants to focus on the influence of only two independent variables (i.e. coordinates). Thus, the other independent variables are kept constant. This is what the orthogonal slicer method does. For example, if the data is defined in spherical coordinates and one wants to focus on the angular dependences for a specific radius, the orthogonal slicer method constructs the corresponding sphere. No interpolation is used since the original grid with the corresponding data is inherited. A convenient color map is used to make the data visible.

2.4.3 Box and Sphere Clipping

Volume clipping plays a decisive role in understanding 3D volumetric data sets because it allows to, cut away selected parts of the volume based on the position of voxels in the data set. Very often clipping is the only way to uncover important, otherwise hidden details of a data set.

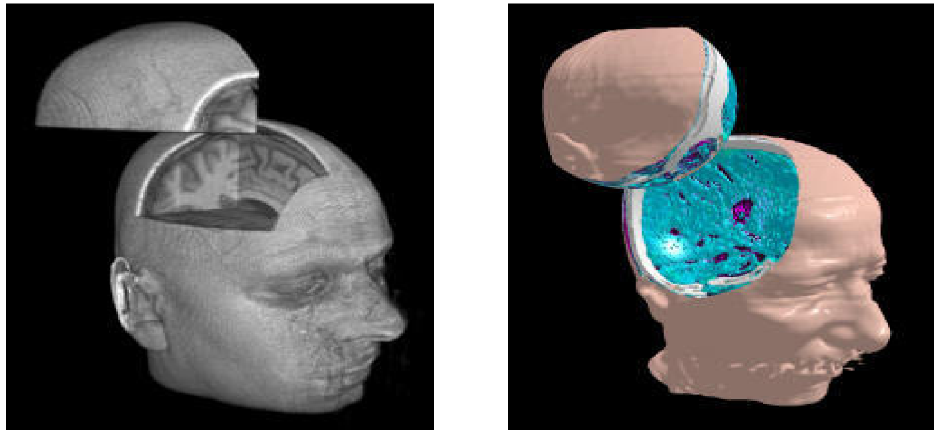

Figure 2.4 Box Clipping[27]

The basic idea is to store the depth structure of the clip geometry in 2D textures whose texels have a one-to-one correspondence to pixels on the viewing plane. In the second approach, a clip object is voxelized and represented by an additional volume data set. Clip regions are specified by marking corresponding voxels in this volume.

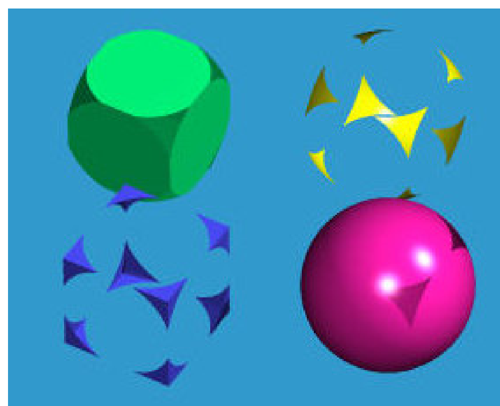

Figure 2.5 Sphere Clipping[22]

In addition to the core clipping techniques for volume visualization, issues related to volume shading are of specific interest. Volume shading, in general, extends mere volume rendering by adding illumination terms to the volume rendering integral. Lighting introduces further information on the spatial structure and orientation of features in the volume data set and can thus facilitate a better understanding of the original data. On one hand, the orientation of the clipping surface should be represented. One the other hand, properties of the scalar field should still influence the appearance of the clipping surface.

Chapter 3: Interactive Clipping

3.1 Clipping Planes In OpenGL

In OpenGL, one can define planes that are used to clip geometry. There are at least six clipping planes defined in all implementations of OpenGL; they are named using the identifiers: GL_CLIP_PLANE0, GL_CLIP_PLANE1, GL_CLIP_PLANE2.

Clipping planes are specified using the function glClipPlane, which takes two arguments. The first is one of the clipping plane identifiers. The second is an array of four GLdoubles which define the plane (using the equation $Ax + By + Cz + D = 0$). The array stores A, B, C and D (in order).

Recall that the vector $n = (A, B, C)$ is the unit-length normal vector for this plane. The retained geometry is on the side of this plane in which the normal vector is pointing; where $Ax + By + Cz + D > 0$. Also, given any point on the plane p it is easy to show that $D = -p . n$.

To actually perform clipping, a particular clipping plane must be enabled using glEnable. It will continue to be active until a corresponding call to glDisable. All geometry specified when a clipping plane is enabled will be clipped. When they are defined, clipping planes are transformed using the current viewing matrix. This means that the clipping planes are transformed just like all other geometry.

3.2 Additional Clipping Planes

In addition to the six clipping planes of the viewing volume (left, right, bottom, top, near, and far), one can define up to six additional clipping planes to further restrict the viewing volume, as shown in figure below . This is useful for removing extraneous objects in a scene - for example, if you want to display a cutaway view of an object.

Each plane is specified by the coefficients of its equation:

$$A\mathbf{x}+B\mathbf{y}+C\mathbf{z}+D = 0.$$

The clipping planes are automatically transformed appropriately by modeling and viewing transformations. The clipping volume becomes the intersection of the viewing volume and all half-spaces defined by the additional clipping planes. Polygons that get clipped automatically have their edges reconstructed appropriately by OpenGL.

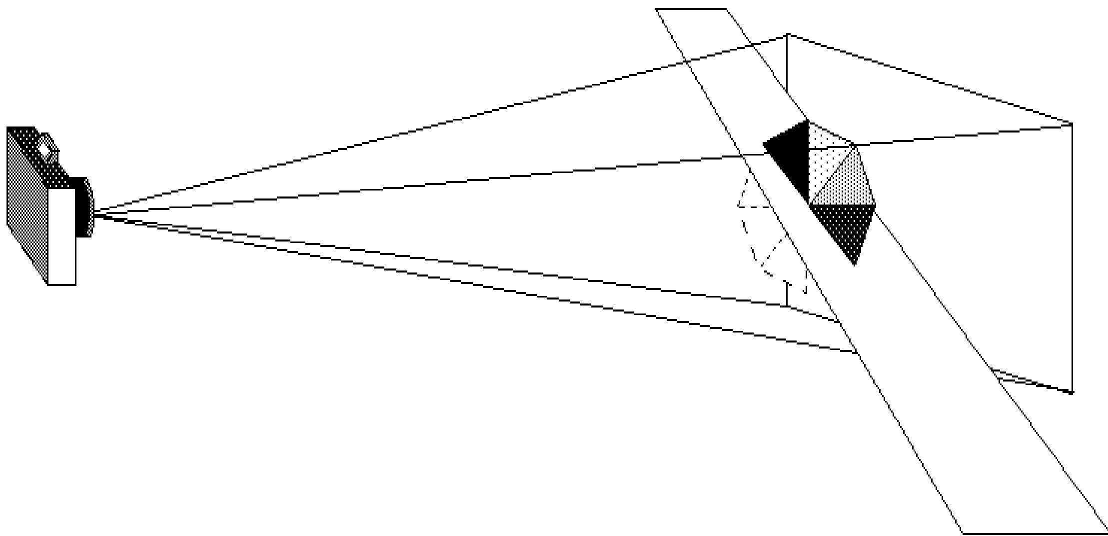

Figure 3.1 Additional Clipping Planes and the Viewing Volume[26]

void **glClipPlane**(GLenum *plane*, const GLdouble **equation*);

The above command defines a clipping plane. The *equation* argument points to the four coefficients of the plane equation, $A\mathbf{x}+B\mathbf{y}+C\mathbf{z}+D = 0$. All points with eye coordinates (xe, ye, ze, we) that satisfy (A B C D)M-1 (xe ye ze we)T >= 0 lie in the half-space defined by the plane, where M is the current modelview matrix at the time **glClipPlane()** is called. All points not in this half-space are clipped away. The *plane* argument is GL_CLIP_PLANE*i*, where *i* is an integer between 0 and 5, specifying which of the six clipping planes to define.

Clipping performed as a result of **glClipPlane()** is done in eye coordinates, not in clip coordinates. This difference is noticeable if the projection matrix is singular (that is, a real projection matrix that flattens three-dimensional coordinates to two-dimensional ones). Clipping performed in eye coordinates continues to take place in three dimensions even when the projection matrix is singular. Each additional clipping plane that is defined needs to be enabled and disabled using:

glEnable(GL_CLIP_PLANE*i*); and glDisable(GL_CLIP_PLANE*i*);

Some implementations may allow more than six clipping planes which can be found by using **glGetIntegerv()** with GL_MAX_CLIP_PLANES to find how many clipping planes are supported.

3.3 Interactive Clipping

Here we introduce the idea of interactive clipping. As the name suggests the clipping method described here is totally interactive. It gives the user the flexibility to interact with the object and visualize the interior portion of the object with the ease.

The interactive clipping consists of the clipping plane generated by the OpenGL as described in previous chapter, plus it provides the ability to rotate and translate the clipping plane, thus making sure that every possible cross section can be viewed. The user can either move the object after clipping to the desired position or use a hot key to reach the best view that is perpendicular to the clipping plane.

3.3.1 Steps in Interactive Clipping

1. For clipping we choose two points in the XY plane assuming initially that Z component of the plane is 0. The two points chosen are (x1, y1) and (x2, y2). From

this we calculate the coefficients of the equation $Ax + By + Cz + D = 0$. Initially we take two assumptions,

a. First that the plane is parallel to z axis
b. Second assumption is the plane passes through origin.
c. The coefficients are thus calculated as A = y2- y1; B = -(x2 – x1); C = 0; D = 0

These coefficients are passed to the following command

```
glClipPlane (GL_CLIP_PLANE3,  eqn);
```

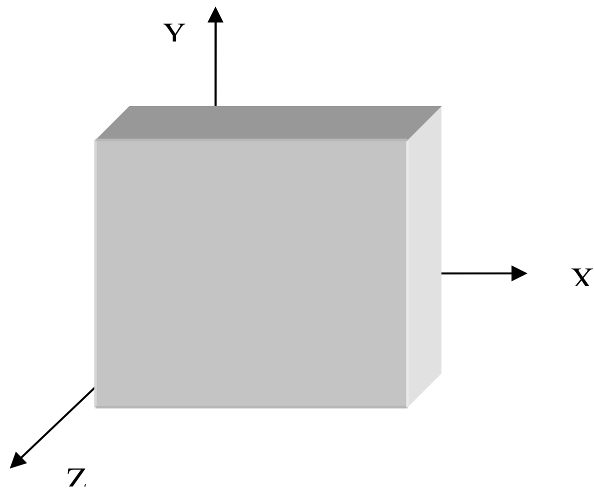

Figure 3.2 Initial Orientation of Object

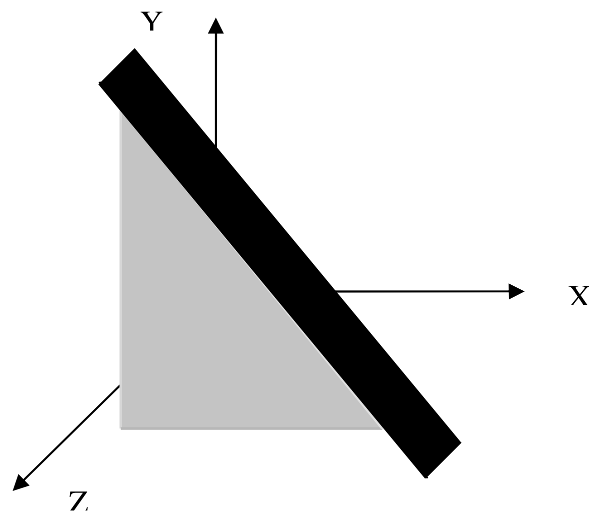

Figure 3.3 Object with Clipping Plane

2. Rotate and translate to adjust the clipping plane.

a. For rotating the clipping plane we change the coefficients A and B. Increasing A by n, the resulting the plane will move in the clockwise direction

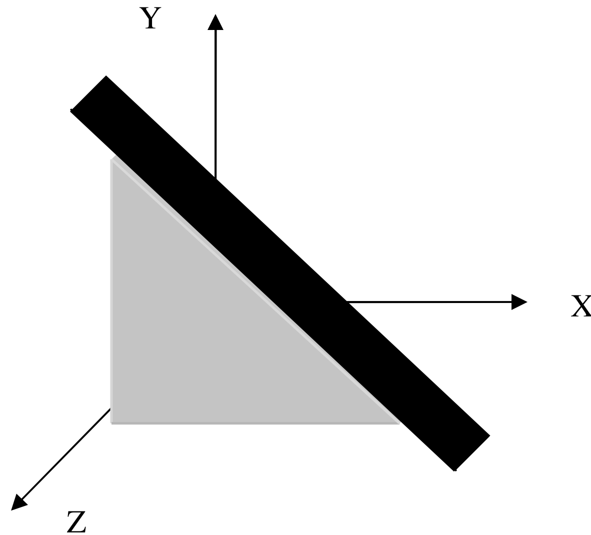

Figure 3.4 Object with Rotated Clipping Plane

b. For translation, the coefficient D needs to be increased or decreased. If suppose n is a positive number and we increase D by n the plane will move away from the origin, if D >= 0.

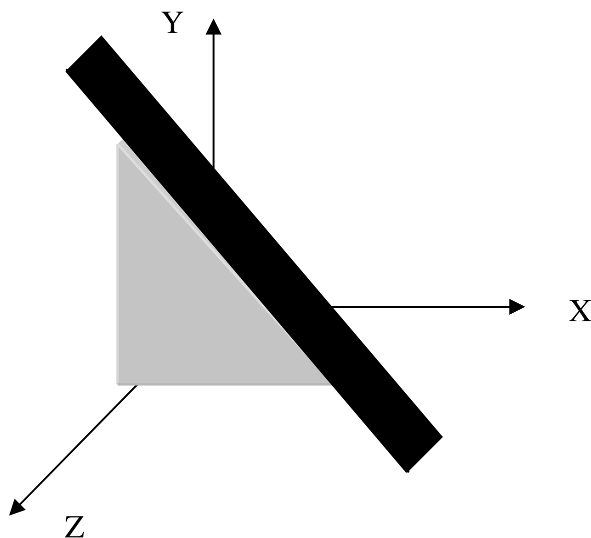

Figure 3.5 Object with Translated Clipping Plane

3. Bringing the clipped object to the best view requires the two rotations.

a. First rotation is around x-axis since clipping plane is in XY plane.

b. Second rotation is around the vector parallel to the clipping plane which is defined by the $Ax + By + Cz + D = 0$. The unit vector parallel to axis is found by the getting the normal vector and using this normal vector to find the desired parallel vector.

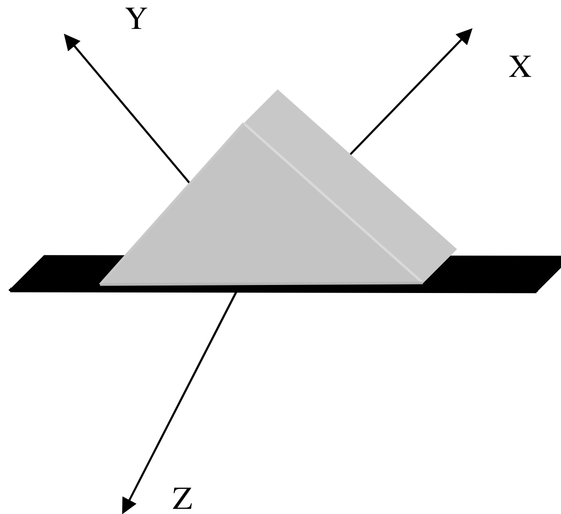

Figure 3.6 Object After First Rotation

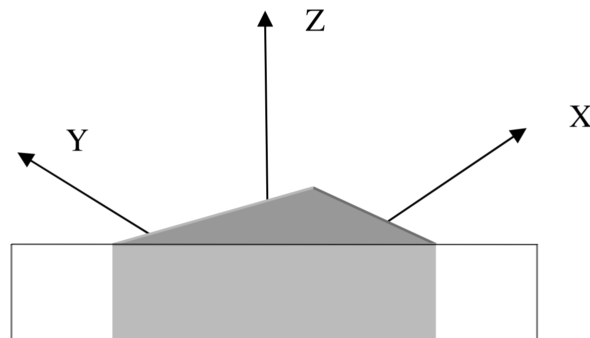

Figure 3.7 Object after Second Rotation Showing Clipped Surface

4. After getting to the best view the change in clipping plane is brought by bringing back the object which is in this case is the set of images back to the initial position and then making the changes.

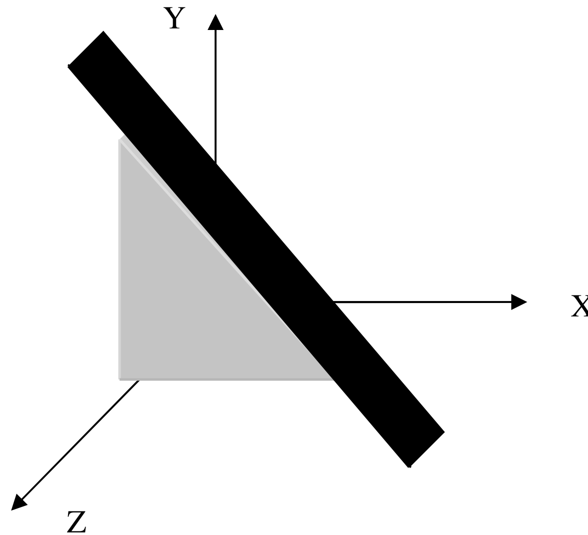

Figure 3.8 Object Back to Step 2

5. Repeat the step 2 or end.

The above steps give the step by step procedure of the interactive clipping. Object can be replaced by anything which the user wants to visualize.

3.3.2 Design and Implementation

The modules which provide interactive clipping are written using C++, OpenGL and GLUT library. The detail about these modules is given in the next section.

3.3.2.1 OpenGL, GLU and GLUT

The approach is implemented using C++, OpenGL and GLUT library. OpenGL is a software interface to graphics hardware. The OpenGL API (Application Programming Interface) began as an initiative by SGI to create a single, vendor-independent API for the development of 2D and 3D graphics applications. Prior to the introduction of OpenGL, many hardware vendors had different graphics libraries. This situation made it

expensive for software developers to support versions of their applications on multiple hardware platforms, and it made porting of applications from one hardware platform to another very time-consuming and difficult. SGI saw the lack of a standard graphics API as an inhibitor to the growth of the 3D marketplace and decided to lead an industry group in creating such a standard[24]. This interface consists of about 150 distinct commands that you use to specify the objects and operations needed to produce interactive three-dimensional applications. OpenGL is designed as a streamlined, hardware-independent interface to be implemented on many different hardware platforms. OpenGL doesn't provide commands for performing windowing tasks or obtaining user input and high-level commands for describing models of three-dimensional objects. A sophisticated library that provides these features could certainly be built on top of OpenGL. The OpenGL Utility Library (GLU) provides many of the modeling features, such as quadric surfaces and NURBS curves and surfaces. GLU is a standard part of every OpenGL implementation.

GLUT is the OpenGL Utility Toolkit, a window system independent toolkit for writing OpenGL programs. It implements a simple windowing application programming interface (API) for OpenGL. GLUT provides a portable API so you can write a single OpenGL program that works across all PC and workstation OS platforms. GLUT is designed for constructing small to medium sized OpenGL programs. The GLUT library has C, C++, FORTRAN, and Ada programming bindings.

3.3.2.2 Modules

The code consists of the four parts are shown in Figure 3.9.

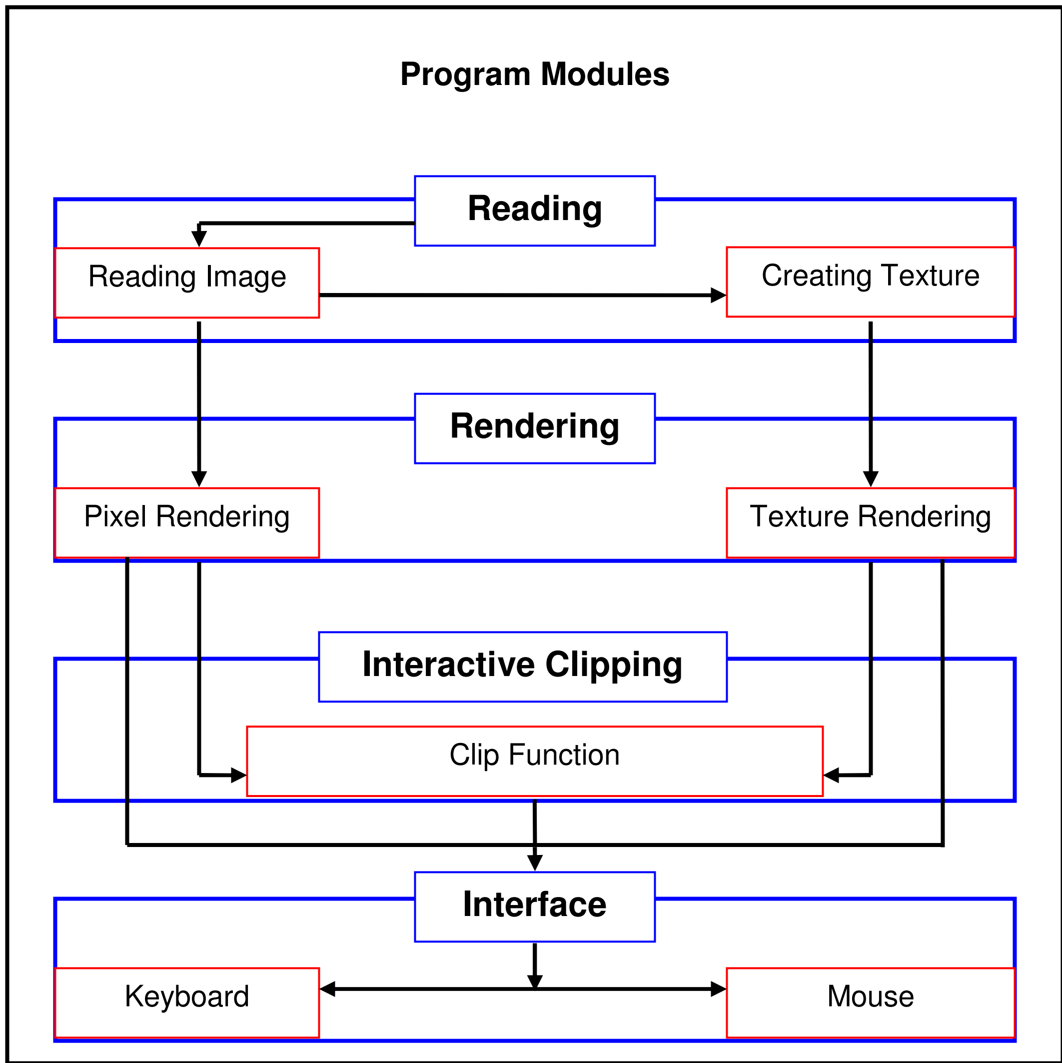

Figure 3.9 Schematic Diagram of Code

The modules are shown using the blue color. The basic functions which constitute a particular module are shown by red color and the flow of information and interaction between functions is shown by the black arrows in the figure below. Functions which are common in all OpenGL programs are not shown here. The whole process of rendering is done in orthographic projection. With an orthographic projection, the viewing volume is a rectangular parallelepiped. This type of projection is used for applications where it's crucial to maintain the actual sizes of objects and angles between them as they're projected.

- **Reading**

Reading module consists of two functions:

a. One for reading the image: This function reads the complete data of the image like its width, height, and RGB component of the each and every pixel. This process is repeated for every image that needs to be rendered.

```
Function Load_TIFF(char *filename)
Begin

    // Step 1 Open the file to be read – TIFFOPEN is the function provided in Libtiff library
    TIFF* tif = TIFFOpen(filename, "r");
    // Step 2 Check if image is opened properly
    if (!tif)
        return NULL;
    // Step 3 Allocate a memory for structure ImageType to hold image data
    ImageType *Image = new ImageType;
    // Step 4 Read the Image Date
    TIFFGetField(tif, TIFFTAG_IMAGEWIDTH,  &(Image->width));
    TIFFGetField(tif, TIFFTAG_IMAGELENGTH, &(Image->height));
    size_t npixels = Image->width * Image->height;
    // Step 5 Read the RGBA component of the pixels of the image
    uint32* raster = new uint32[npixels];  // (uint32*) malloc (npixels * sizeof (uint32));
    if (!TIFFReadRGBAImage (tif, Image->width, Image->height, raster, 0))
    {
        printf ("Error!  Unable to read Tiff image data correctly!\n");
        delete raster;
        delete Image;
        return NULL;
    }
    // Step 6 Close the file and return the structure holding the image Data
    TIFFClose (tif);
    Image->rgbadata = (RGBAType*) raster;
    return Image;
End
```

b. For generating the texture: This functions uses the images read in step 1a. The texture has the details of image such as height and width. But the RGBA component of every pixel is merged to generate the texture out of it and it is loaded in the memory. It is hardware dependent. The texture is generated using the following command glGenTextures (1, &ID)

```
Function CreateTexture (ImageType* Image)
Begin

    //Step 1 Set the storage mode of the pixel
    glPixelStorei (GL_UNPACK_ALIGNMENT, 1);

    //Step 2 glGenTextures function returns n texture names in the textures parameter.
    &ID is a //pointer to the first element of an array in which the generated texture names
    are stored.
        glGenTextures (1, &ID);

    //Step 3 The glBindTexture function enables you to create a named texture
    glBindTexture (GL_TEXTURE_2D, ID);

    //Step 4 The gluBuild2DMipmaps function obtains the input image and generates all
    mipmap images so the input image can be used as a mipmapped texture image.

    if (0 != gluBuild2DMipmaps (GL_TEXTURE_2D,
        4, //GL_RGBA,  //GL_ALPHA,
        Image->width,
        Image->height,
        GL_RGBA,  // GL_RGB,  //
        GL_UNSIGNED_BYTE,
        Image->rgbadata) )
        {
                printf ("Error!  2D mipmaps couldn't be built!\n");
          return 0;
        }
        return 1;
End
```

- **Rendering**

Rendering is creating an image of objects designed in a three-dimensional modeling program. Rendering can simulate the appearance of real-world textures, colors, surface shadows, highlights and reflections. It helps to understanding the complex systems which would have been impossible. In case of pixel rendering the data read in step 1.a is used to render each and every pixel. In case of texture rendering the texture generated in step 1.b is displayed by using the glBindTexture (GL_TEXTURE_2D, ID); The part of function that is used to do both pixel rendering and texture rendering is shown.

```
// FOR TEXTURE RENDERING

glEnable (GL_TEXTURE_2D);
glPushMatrix ();
glTranslatef (0, 0, -0.5 * (state->nslices-1) * (state->slice_thickness * ls) );
for (int s = 0; s < state->nslices; s++)
{
        Setactive call the function glBindTexture( GL_TEXTURE_2D, ID);
        ogltex[s]->SetActive();  //
        glBegin (GL_QUADS);
        glNormal3f (0, 0, 1);
        glColor4f (1, 1, 1, 1);
        glTexCoord2f (1, 1);
        glVertex3f (hls, hls, 0);
        glTexCoord2f (0, 1);
        glVertex3f (-hls, hls, 0);
        glTexCoord2f (0, 0);
        glVertex3f (-hls, -hls, 0);
        glTexCoord2f (1, 0);
        glVertex3f (hls, -hls, 0);
        glEnd ();
        glTranslatef (0, 0, /*s * */ (state->slice_thickness * ls) );
}
glDisable(GL_CLIP_PLANE3);
glPopMatrix ();
glDisable (GL_TEXTURE_2D);



// FOR PIXEL RENDERING
for (int s = 0; s < state->nslices-1; s++)
{
        wd = ogltex[s]->Imagedata->width ;
        ht = ogltex[s]->Imagedata->height ;

        for(e = wd -1 ; e != -1; e--)
        {
                for(c = 0; c < ht ; c++)
                {
                        k = e * wd + c;
                        R = TIFFGetR(ogltex[s]->Imagedata->raster[k]);
                        B = TIFFGetB(ogltex[s]->Imagedata->raster[k]);
                        G = TIFFGetG(ogltex[s]->Imagedata->raster[k]);
                        glColor3f(R*0.01,G*.01,B*.01);
                        glVertex3f(c*0.005 - 1.2 ,e*0.005 - 1.2,s*0.01);
                }
        }

        glTranslatef (0, 0, /*s * */ (state->slice_thickness * ls) );
}
```

- **Clipping**

The clipping module has one function which generates the clipping plane as described in the previous chapter and then the whole structure is displayed to see the affect of the clip plane.

```
Function clip (integer x1, integer y1, integer x2, integer y2, float d)

Begin
        //(x1,y1) and (x2, y2) are two points used to generate the clipping plane
        float a=0, b=0, c=0, a1, b1;
        a1=x2-x1;
        b1=y2-y1;
        a = b1;
        b= - a1;
        // array eqn holds the coefficients of Ax + By + Cz + D = 0
        //  C = 0 since we assume plane is parallel to the z axis
        //  D = 0 initially since we assume plane passes through the origin
        eqn[0]=a; eqn[1]=-b; eqn[2]=c;   eqn[3]=d;
        Display(); // uses array eqn, array eqn is declared globally

End
```

- **Interface**

It consists of functions for handling keyboard and mouse events. Before these callback functions can be used they need to be registered which provide the interactive nature to the application. As soon any action is taken by user the associated code performs the desired function. These functions provide the rotation and translation of the object and the clipping plane. Function names with the parameters they require are shown below.

```
void MotionFunc (int x, int y)

void MouseFunc (int button, int buttonstate, int x, int y)

void KeyboardFunc (unsigned char key, int x, int y)
```

- **Rotation, Translation of Clipping Plane**

For rotation and translation of the clipping plane we need to change the coefficients of the clipping plane defined by Ax + By + Cz + D = 0.

```
//Rotation //m1, m2, m3, m4 are parameters passed to the clip function
        //clockwise
                m3=m3 - 6 ; m4= m4 - 6;
                clip(m1,m2,m3,m4,eqn[3]);
        //anti clockwise
                m3=m3 + 6 ; m4= m4 + 6;
                clip(m1,m2,m3,m4,eqn[3]);

//Translation
        eqn[3] = eqn[3] + 5;  OR   eqn[3] = eqn[3] - 5;
        clip(m1,m2,m3,m4,eqn[3]);
```

- **Best View**

For best view the object is rotated such that the viewing direction is normal to the clipping plane. The following code takes care of that

```
//Best View
        sq = sqrt(eqn[0]*eqn[0] + eqn[1]*eqn[1]+ eqn[2] * eqn[2]);
        if (test ==1) // this reverses the first and second rotation
        {
                glRotatef (theta, eqn[1]/sq, - eqn[0]/sq, 0);
                glRotatef ( phi, 0, 0, 1);
                glMultMatrixd (state->rot_matrix);
                test =  2;
        }
        if (test==0)
        {
                angle = (eqn[0]/eqn[1]); // Slope of the line Ax + By = 0
                phi= ((atan(angle)* 180 * 7)/22);
                if (phi < 0)
                        phi = phi * (-1);
                else
                        phi = 180-phi;
                //first rotation
                glRotatef (- phi, 0, 0, 1);
                //second rotation
                glRotatef (-theta, eqn[1]/sq, - eqn[0]/sq, 0);
                glMultMatrixd (state->rot_matrix);
                test =1;
        }
```

Chapter 4: Application

We have implemented interactive clipping on two types of scientific datasets, confocal data of a plant stem and calculated electronic charge density distributions.

4.1 Confocal Data

4.1.1 Confocal Microscopy

In confocal microscopy, point light from under-focal-plane is focused at a plane behind the pinhole such is blocked away by the pinhole plate. The light from above-focal-plane will be focused before the pinhole and is blocked away by the pinhole too. Only the light from focal plane is just focused at the pinhole thus can reach the image detector.

The size of pinhole determines how thick an optical slice will be. The smaller the pinhole, thinner is the slice. But the thickness will not go down indefinitely. It is also limited by all those factors affecting resolution of the lens. Point light source illumination within the specimen will improve image quality by eliminating stray light interference.

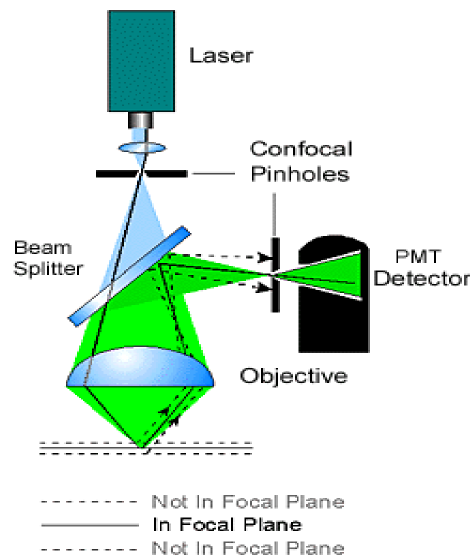

Figure 4.1 Laser as Point Light Source[25]

4.1.2 Confocal System

1. A point light source for illumination,
2. A point light focus within the specimen,
3. A pinhole at the image detecting plane.

These three points are optically conjugated together and aligned accurately to each other in the light path of image formation, this is confocal. Confocal effects result in suppression of out-of-focal-plane light, suppression of stray light in the final image.

4.1.3 Features of Confocal Images

1. Void of interference from lateral stray light: higher contrast.
2. Void of superimpose of out-of-focal-plane signal: less blur, sharper image.
3. Images derived from optically sectioned slices (depth discrimination)
4. Improved resolution (theoretically) due to better wave-optical performance.

4.1.4 Application Description

The application takes the images and renders them using pixel based mapping and texture based mapping. The interactive clipping is developed which involves the rotation, translation of the clipping plane to allow users to see all possible cross-sections of the volume. Apart from this it allows the user to go the best view after clipping. The drawback with the texture mapping is that the texture is not visible when the complete object is rotated by 90 degrees.

The process involves reading the set of confocal images and then stacking them together to generate the 3D volume. The images are now stacked above each other. Now these set of images can be rotated or translated according to the user choice. The clipping plane can be used to clip the images to remove a portion of the images for

understanding and gaining insight to the structures. The clipping plane can be further rotated or translated to get the different cross section view according to the users choice. The rotation and translation of the clipping enables interactive clipping. Further more the whole object can be rotated to give the best view. This best view can further be moved according to the convenience. The images below show the features described above.

4.1.4.1 2D-Images

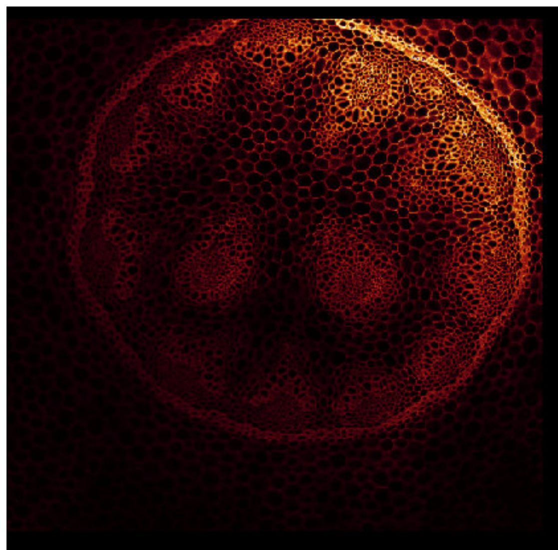

Figure 4.2 Confocal Image

This is one of the set of images. These set of images are stacked together to give resemblance to the 3D structure to which they belong. The set of images is obtained by observing the 3D structure using the confocal microscope at different depths. The images are taken are different depths without actually dissecting the original structure. These images are further stacked above each other in the order to give the user the actual representation of the 3D structure. They may need to be aligned in case they are not, to reconstruct the exact copy of the 3D structure. This process can be done for any 3D volume for which images can be obtained or reconstructed

4.1.4.2 Stacked Images

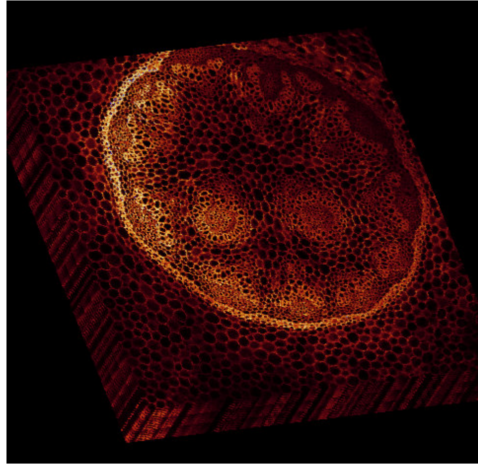

Figure 4.3 Stacked images:

The set of images are stacked together and this is the original orientation of the axes. The images shown above are rendered using texture mapping. In texture mapping the complete image is stored in to the memory and then rendered. The quality of the image is lost, but the operations like rotation, clipping etc are pretty much faster as compared to the pixel based rendering.

4.1.4.3 Rotated Images

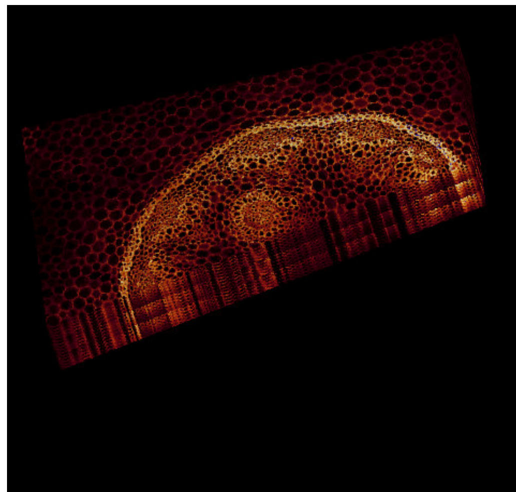

Figure 4.4 Rotated Images with clipping

The images are rotated along with the axes. The array rot is associated with the rotation and keeps the current state of the images that how much it has rotated from the original orientation of the axes.

4.1.4.4 Translated Images

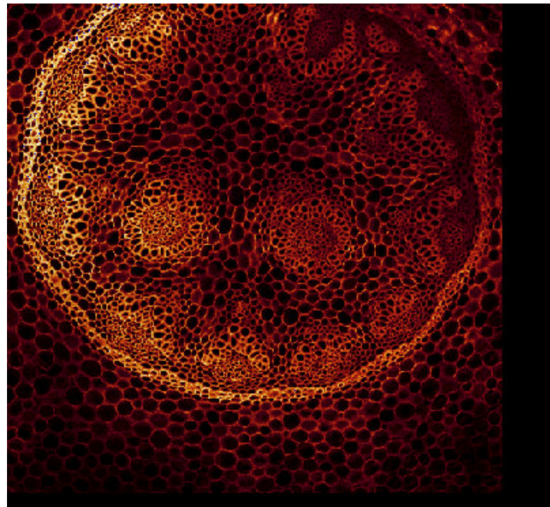

Figure 4.5 Translated Images

The image shown above is showing the translation of the object in the XY plane. The translation is the process of moving the object.

4.1.4.5 Clipped Images

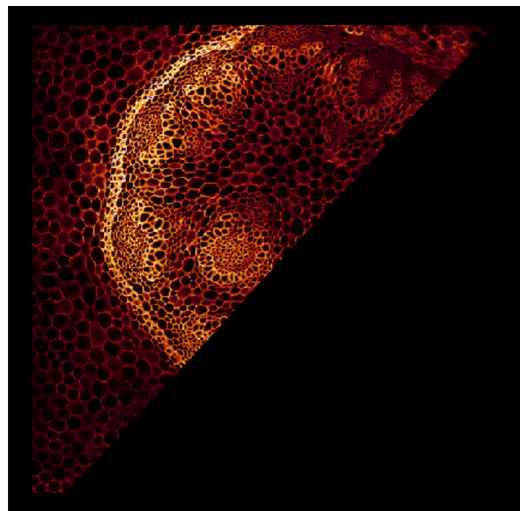

Figure 4.6 Clipped Images

The function in the clipping module is used to clips the set of images by selecting the two points in the XY plane as described in the above chapter. The clipping plane generated by it is parallel to the z-axis and in the xy plane. The clipping plane can be rotated and translated. This rotation and translation gives user the flexibility to view all possible cross sections and thus providing the interactive clipping.

4.1.4.6 Best View

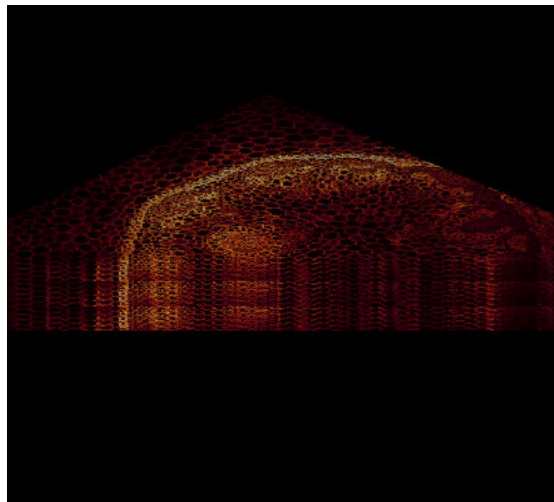

Figure 4.7 Image Showing the Best View

The image is clipped and then rotated about z-axis and then further about the unit vector passing through the center and parallel to the clipping plane. The second rotation is possible since the axes rotate with each rotation and the equation of the clipping plane remains same no matter how the object is rotated. The best view is achieved by giving the second rotation of about 75 degrees. The 90 degree rotation can't be given since in case of texture mapping there will be nothing visible to the user after 90 degree of rotation. When the clipping plane is further rotated the best view is achieved by rotating the whole objects (set of images) back to the original position and repeating the steps described above.

4.1.5 Rendering: Texture vs. Pixel Mapping

- **Texture Based Mapping**

In this the volumetric images are rendered using parallel, axis-aligned, texture –mapped planes. The images are loaded in the video memory. Texture based mapping is highly dependent on the graphics card used in the system. The major flaw of the texture based mapping is the when the images are viewed in along the edges the volumes disappears, to overcome this problem of the texture based mapping many software packages used for visualization of the of images use set of three images along the three different axes. Since this method of rendering the data is dependent on the graphics capability of the machine the performance depreciates as the number of images increases.

- **Pixel Based mapping**

In case of pixel based mapping data at each pixel is read and then rendered accordingly. Since each image has the large amount of data the over all rendering speed is slow but the quality of rendered images is much better then what we get in texture based rendering. Table 4.1 shows that the initial reading and rendering images for pixel based mapping is more than the texture based rendering. On applying operations such as rotation and translation pixel based rendering takes a longer time. Secondly when the images are rendered along the edges we can still see the pattern it makes along the edge. Thirdly since it is not highly dependent on the hardware of the machine the quality and the speed nearly remain same.

The table below shows that initial time to read and render image on a machine for a pixel based rendering is more than the texture based rendering.

Table 4.1: Time taken in Rendering Images

Number of Images	Pixel Rendering (sec)	Texture Rendering (sec)
15	4	2
30	7	4
45	11	6
60	14	8
75	17	10
90	22	12

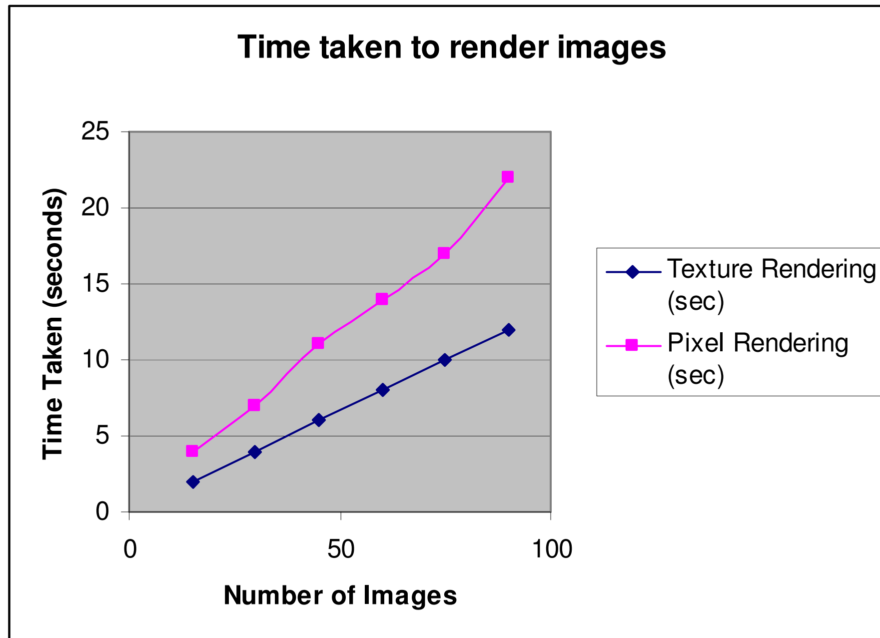

Figure 4.8 Graph Between Pixel and Texture Rendering

We can see from the above table and chart that the time taken to render images in texture based rendering is less as compared to pixel based rendering. The reason behind this behavior is, in pixel based rendering each pixel is rendered. Suppose a image is 512 * 512 pixels, then to render it using the pixel based rendering would require the information about 262144 points, on the other hand the texture based rendering doesn't require that much data. It only requires RGBA component of the whole image thus resulting in a faster rendering scheme and smooth transitions.

But the advantage of the pixel rendering scheme over texture rendering scheme is as follows:

1. Since each pixel is rendered individually there is no loss in the quality of the image generated.
2. In case of pixel based rendering we can see the pattern along the clipping plane when we are viewing it from the perpendicular direction. But in case of texture rendering we can't see anything if we are standing perpendicular the clipping plane.

4.2 Charge Density

The interactive clipping idea described in the previous chapter can be used to visualize the structure of the atoms and the charge density. Here we demonstrate the idea with the electronic charge distribution of MgO system. Magnesium oxide is an alkaline earth metal oxide. It is produced using calcinations and is also found in seawater, underground deposits of brine and deep salt beds. Magnesium is the eighth most abundant element constituting about two per cent of the earth's crust and typically 0.12% of seawater

The sources of the three-dimensional electronic density data are the massive electronic calculations performed on the Superhelix cluster of 256 processors. The simulated MgO system consists of 216 atoms and has used up to 64 processors. Each Mg atom contributes two valence electrons and each O atom contributes six valence electrons so the system contains a total number of 864 electrons. These electrons are distributed throughout the system. The simulations produce the electronic density on a fine regular Cartesian grid whose size is 136 for our datasets. Our goal is to visualize these massive datasets to gain insight into the electronic structures and their response

to the presence of defects in the system. For example, a cation vacancy is created by removing Mg core without changing the number of valence electrons, while an anion vacancy is formed by removing an O core together with eight valence electrons. Thus, the net charges of the Mg and O vacancies are 2e and 2e, respectively. These charged defects will affect the distributions of electrons which we can visualize using the resulting datasets. Here, we apply our proposed interactive clipping technique to visualize the 3D charge density datasets. This idea can be extended to the more complicated structures with non symmetric geometry. It helps in understanding the properties of a system in a real time and its behavior under different condition. In the future we further plan to investigate other approaches for real time visualization with more complex clipping objects The charge density at different region are shown by different colors (red being maximum charge and blue being minimum. The figures below show the structure and explain how interactive clipping can be applied to it.

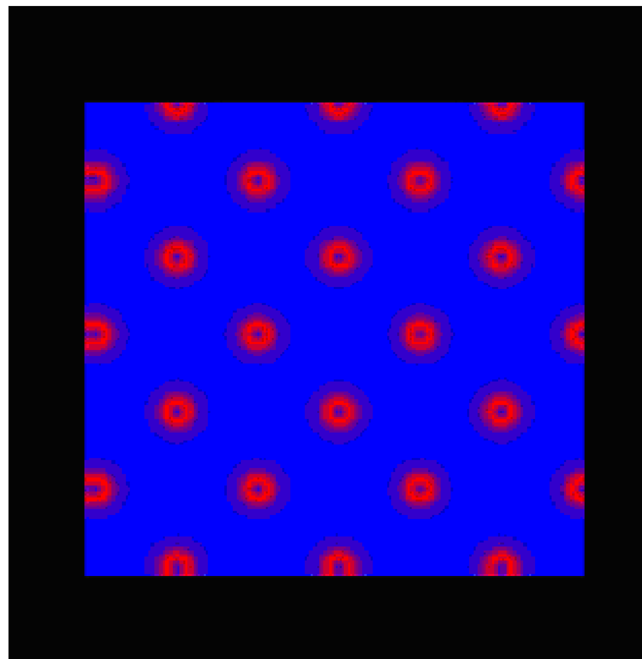

Figure 4.9 Structure of the MgO (136 atoms) Charge Density

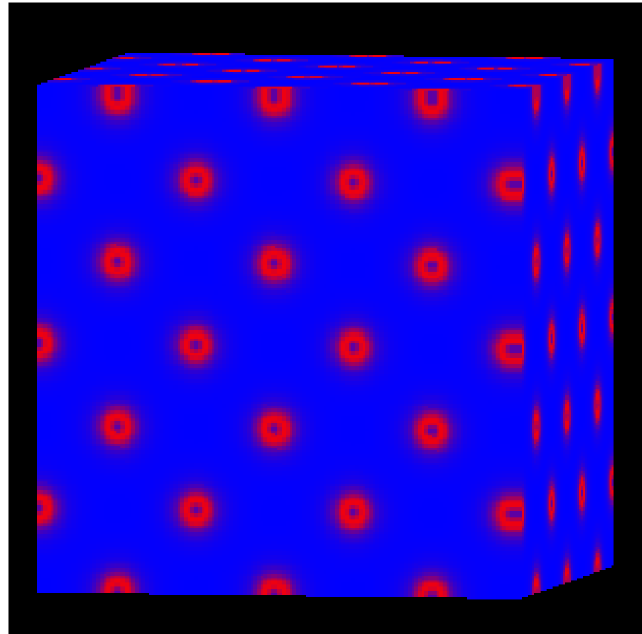

Figure 4.10 3D View of MgO (136 atoms) Charge Density

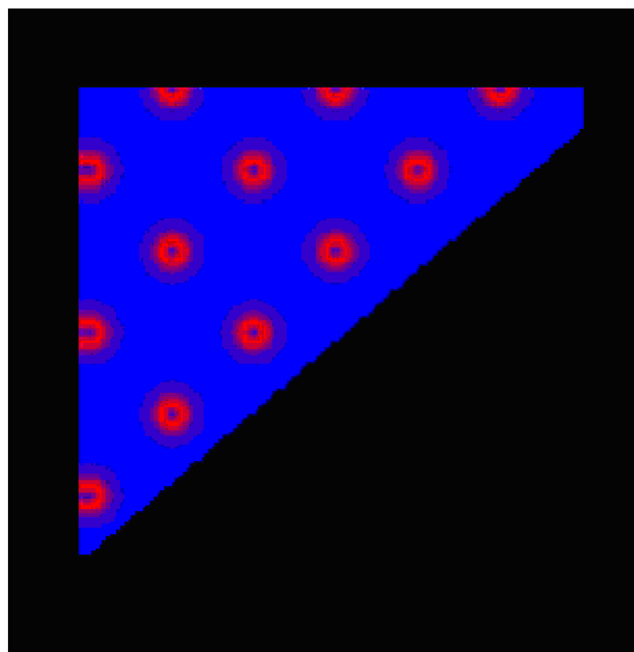

Figure 4.11 Interactive Clipping

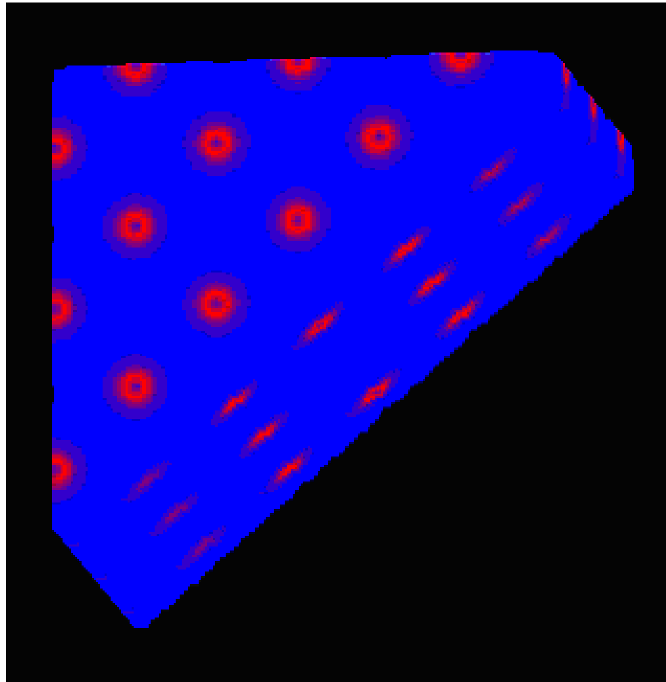

Figure 4.12 Rotation with Interactive Clipping

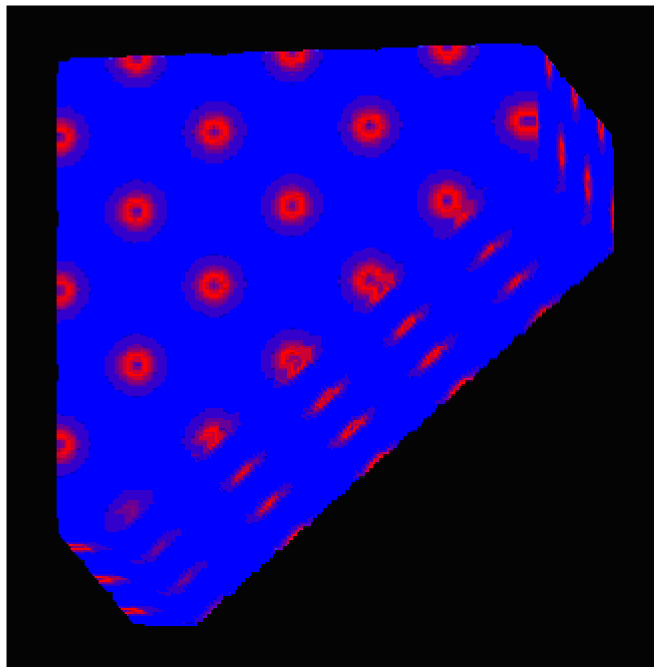

Figure 4.13 Interactive Clipping – Movement of Clipping Plane

Interactive clipping idea guarantees that the user can view every cross- section of the rendered volume. In the above structure the number of points being rendered is more than 2 million. Since with every transformation being applied the structure needs to be regenerated, the transformations are not smooth. The rendering can be smoothed by changing the structure in to a texture and then rendering it. This will result in the smooth transitions.

The structure described above has no vacancy and it the one of the simplest structure with symmetric geometry. When the structure has the vacancy there is the distortion in the atoms and the resulting charge density. This tool can be used to visualize such structures. The data for visualization can be obtained from the simulations which tell the details of the structure like location of the atoms and charge density around atom.

Chapter 5: Conclusion

The motivation behind this project was to provide researchers with the visualization tool to gain insight in to the data. The interactive clipping methodology and rendering described here can be used to visualize any 3D structures

We have explored how clipping can be exploited in an interactive manner to visualize massive three-dimensional datasets. In essence, the proposed interactive clipping approach involves the dynamic adjustment of the clipping plane to expose any cross-section of the volume data and subsequent adjustment of the clipped surface to the best view position using a combination of rotation and translation. The thesis describes the design, implementation and application of our interactive-clipping-based visualization system. The implementation is done with OpenGL and C++, thus resulting in a highly portable and flexible system. For illustration, two types of scientific datasets, confocal data of a plant stem and calculated electronic charge density distributions are successfully visualized. The data is displayed using pixel- and texture-based rendering; the latter is shown to give a better performance. Pixel rendering for large data is slow but we achieve better quality of the images generated.

Further this clipping methodology can be extended using the any curve instead of a clipping plane to visualize the data.

References

1. http://www.cc.gatech.edu/scivis/tutorial/linked/whatisscivis.html

2. “Emissive Clipping Planes for Volume Rendering.” by YON - Jan C. Hardenbergh & Yin Wu, TeraRecon, Inc.

3. “A Hardware-Assisted Hybrid Rendering Technique for interactive Volume Visualization.” By – Brett Wilson, Kwan-Liu Ma and Patrick S. McCormick

4. Indeed - Visual Concepts GmbH, Berlin –www.amiravis.com

5. Aldus (1992) TIFF revision 6.0. Aldus Corporation, Seattle, Washington.

6. de la Fraga, L. G., Dopazo, J. & Carazo, J. M. (1995) Confidence limits for resolution estimation in image averaging by random subsampling. Ultramicroscopy 60, 385-391.

7. http://www.cc.gatech.edu/grads/h/Hao-wei.Hsieh/Haowei.Hsieh/mm.html

8. Frigo, M. & Johnson, S. G. (1997) The Fastest Fourier Transform in the West. MIT-LCS-TR-728 (September 1997)

9. Frigo, M. & Johnson, S. G. (1998) FFTW: An adaptive software architecture for the FFT. In Proceedings of the ICASSP eds.), p. 1381.

10. Frigo, M. (1999) A fast fourier transform compiler. In Proceedings of the Conference on Programming Language Design and Implementation eds.), ACM SIGPLAN, Atlanta, Georgia.

11. Gobel, U., Sander, C., Schneider, R. & Valencia, A. (1994) Correlated mutations and residue contacts in proteins. Proteins 18, 309-317.

12. Hall, S. R. (1991) The STAR file: A new format for electronic data transfer and archiving. J. Chem. Inform. Comput. Sci. 31, 326-333.

13. Hall, S. R., Allen, F. H. and Brown, I. D. (1991) The Crystallographic Information File (CIF): a new standard archive file for crystallography[1]. Acta Crystall. A47, 655-685.

14. Heymann, J. B. & Engel, A. (2000) Structural clues in the sequences of the aquaporins. J. Mol. Biol. 295, 1039-1053.

15. Heymann, J. B., Müller, D. J., Landau, E. M., Rosenbusch, J. P., Pebay-Peyroula, E., Büldt, G. & Engel, A. (1999). Charting the surfaces of the purple membrane. J. Struct. Biol. 128(3), 243-249.

16. Heymann, J. B., Pfeiffer, M., Hildebrandt, V., Kaback, R., Fotiadis, D., de Groot, B., Engel, A., Büldt, G., Oesterhelt, D. & Müller, D. J. (2000). Conformations of the rhodopsin third cytoplasmic loop grafted onto bacteriorhodopsin. Structure 8, 643-653.

17. Pittet, J.-J., Henn, C., Engel, A. & Heymann, J. B. (1999) Visualizing 3D data obtained from microscopy on the Internet. J. Struct. Biol. 125, 123-132.

18. Saxton, W. O. and Baumeister, W. (1982) The correlation averaging of a regularly arranged bacterial cell envelope protein. J. Microsc. 127, 127-138.

19. Trus, B. L., Heymann, J. B., Nealon, K., Cheng, N., Newcomb, W. W., Brown, J. C., Kedes, D. H. & Steven, A. C. (2001). Capsid structure of Kaposi's sarcoma-associated herpesvirus, a gammaherpesvirus, compared to an alphaherpesvirus, herpes simplex virus type 1, and a betaherpesvirus, cytomegalovirus. J. Virol., 75(6), 2879 - 2890.

20. Unser, M., Trus, B. L. and Steven, A. C. (1987) A new resolution criterion based on spectral signal-to-noise ratios. Ultramicr. 23, 39-52.

21. Winkelmann, D. A., Baker, T. S. & Rayment, I. (1991) Three-dimensional structure of myosin subfragment-1 from electron microscopy of sectioned crystals. J. Cell Biology 114, 701-713.

22. http://www.cs.mtu.edu/~shene/COURSES/cs3621/LAB/povray/clip.html

23. http://www.mat.univie.ac.at/~kriegl/Skripten/CG/node31.html

24. http://www.sgi.com/products/software/opengl/

25. http://www.hi.helsinki.fi/amu/AMU%20Cf_tut/IMAGES/cf_tut1.gif

26. http://www.opengl.org/

27. http://www.vis.uni-stuttgart.de/eng/research/fields/current/volclipping/

VITA

Gaurav Khanduja was born in Roorkee, Uttaranchal, India, in June 1980. He is the son of Mrs. Sneh Lata Khanduja and Dr. Sita Ram Khanduja. He completed his high school studies and joined the undergraduate studies program in computer science and engineering at Gorakhpur University, Gorakhpur, India, in 1998. He finished his undergraduate studies in 2002. He joined Louisiana State University, Baton Rouge, to pursue his master's degree in systems science in August 2002. His research interest is in the area scientific visualization and computer graphics and he plans to pursue his research in scientific visualization at Louisiana State University leading to a doctorate degree. He is a candidate for the degree of Master of Science in systems science to be awarded at the commencement of May 2005.