2009

Multiple dataset visualization (MDV) framework for scalar volume data

Gaurav Khanduja
Louisiana State University and Agricultural and Mechanical College, gkhand1@tigers.lsu.edu

MULTIPLE DATASET VISUALIZATION (MDV) FRAMEWORK FOR SCALAR VOLUME DATA

A Dissertation

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
Doctor of Philosophy

In

The Department of Computer Science

By
Gaurav Khanduja
B.E., Madan Mohan Malaviya Engg College, 2002
M.S., Louisiana State University, 2005
May 2009

To My Parents, Family

And

My beloved wife, Swati

ACKNOWLEDGMENTS

Foremost of all, I would like to thank my major advisor Dr. Bijaya B. Karki for guiding me through various stages of uncertainties and doubts. When I did not know what I was doing, he was the one who supported me, believed in me and guided me. I would also like to thank Dr. S. Sitharama Iyengar, Dr. Jianhua Chen and Dr. Brygg Ullmer for agreeing to be in my dissertation committee and providing valuable feedback. I would like to thank Dr. Charles Delzell, the dean's representative, for his interest and constantly asking questions about the status of the work. I would like to acknowledge Dr. Ashok Verma and Dipesh Bhattarai for providing a stimulating intellectual environment in our laboratory and acting as a sounding board when I was not sure about the worth of my own thoughts. I would also like to express my thanks to Ms. Vera Watkins and Ms. Amy Fowler for taking care of the administrative details so that I could focus on my work. I would also like to express my gratitude to the whole Department of Computer Science for providing such an excellent environment for intellectual pursuit.

I would also like to acknowledge the support from National Science Foundation and NASA for this research.

I would not be here with this work if it were not for my family. Their excitement was always source of my energy that kept on motivating me through these four and half years. Last but not the least; I would like to express my sincere gratitude to my parents and my beloved wife, Swati, for pushing me hard so that I could finish this dissertation. They would not settle for less than the finished work. If it was not for them, I would not have been able to finish. This work belongs to them as much as it does to me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Many applications require comparative analysis of multiple datasets representing different samples, conditions, time instants, or views in order to develop a better understanding of the scientific problem/system under consideration. One effective approach for such analysis is visualization of the data. In this PhD thesis, we propose an innovative *multiple dataset visualization* (MDV) approach in which two or more datasets of a given type are rendered concurrently in the same visualization. MDV is an important concept for the cases where it is not possible to make an inference based on one dataset, and comparisons between many datasets are required to reveal cross-correlations among them. The proposed MDV framework, which deals with some fundamental issues that arise when several datasets are visualized together, follows a multithreaded architecture consisting of three core components, data preparation/loading, visualization and rendering. The visualization module - the major focus of this study, currently deals with isosurface extraction and texture-based rendering techniques. For isosurface extraction, our all-in-memory approach keeps datasets under consideration and the corresponding geometric data in the memory. Alternatively, the only-polygons- or points-in-memory only keeps the geometric data in memory. To address the issues related to storage and computation, we develop adaptive data coherency and multiresolution schemes. The inter-dataset coherency scheme exploits the similarities among datasets to approximate the portions of isosurfaces of datasets using the isosurface of one or more reference datasets whereas the intra/inter-dataset multiresolution scheme processes the selected portions of each data volume at varying levels of resolution. The graphics hardware-accelerated approaches adopted for MDV include volume clipping, isosurface extraction and volume rendering, which use 3D textures and advanced per fragment operations. With appropriate user-defined threshold criteria, we find that various MDV techniques maintain a linear time-N relationship, improve the geometry generation and rendering time, and increase the maximum N that can be handled (N: number of datasets). Finally, we justify the effectiveness and usefulness of the proposed MDV by visualizing 3D scalar data

(representing electron density distributions in magnesium oxide and magnesium silicate) from parallel quantum mechanical simulations.

1. INTRODUCTION

1.1 What Is Visualization?

Visualization is an act of representing data in graphical forms that allows insight into otherwise hidden details that is easily conceivable by humans and aids the process of analysis and decision making. The visualization process transforms the symbolic representation of data into the visual representation, enabling scientists to observe their simulations and computations [1]. It offers a method for getting insight in the otherwise obscure details in the complex data and often accelerates the process of scientific discovery by making their outputs more accessible and easier to interpret. The goal of the visualization process is thus to leverage existing scientific methods by providing new scientific insight via visual methods.

The following are three minimal criteria that any visualization is expected to fulfill. A good visualization certainly has to do more, but these criteria are useful in judging the effectiveness of the visualization [2]. One purpose of visualization is communication of data. This means that the data must come from something that is abstract or at least not immediately visible (like the inside of the human body). Visualization transforms from the invisible into the visible. Another purpose of visualization is to produce the image. It may seem obvious that visualization has to produce an image, but that is not always so clear. Finally, the results must be readable and understandable.

Scientific visualization is a subset of the much larger field of visualization, which includes image analysis, computer graphics and other areas. It is the use of computer-generated graphics to explore and understand data and concepts related to science and engineering applications. The physical representation of the data is important for scientific visualization. For example, visualization of MRI data of the human bodies requires the correct reproduction of the shape of skeletons. There is another commonly recognized category of computer-assisted visualization, known as information visualization. Unlike scientific visualization, information visualization is focused on the visualization of the

abstract data that does not have inherent 2D or 3D geometry structure — for example, visualization of the fluctuations of stock prices or file structure of the operating system. Scientific visualization is widely used in essentially all areas of mathematics, science, engineering and technology. Visualization in itself is not a science but proves to be an important tool in understanding the intricacies of the data.

1.2 Multiple Dataset Visualization

Visualization of scalar data has been studied extensively in the past two decades with appearance of many techniques in the context of the single scalar dataset. These techniques can be broadly classified into surface rendering (isosurface extraction [3, 4]) and volume rendering techniques (ray casting [5], splatting [6], shear-warp [7] and texture mapping [8]). Surface rendering techniques deal with the surface representation of the data while volume-rendering techniques map the scalar data to RGBA values before rendering them. Details of these techniques can be found in [9].

The last decade has witnessed exponential increase in the technology in terms of computing power, storage and networking. Advances in computer architecture and high performance computing have allowed an ever increasing number of simulations of the complex scientific phenomena and systems, which produce massive amounts of data [10]. This has opened up new research areas in the field of scientific visualization, including multiple dataset visualization studied in this thesis. A tremendous challenge is posed by the current size of the scientific data: Either a given dataset is too massive for exhaustive visualization or there are multiple datasets to be visualized simultaneously. While we have increased the amounts of data, the amount of time to explore and understand the data has remained the same. Researchers still need the same interactivity and turn-around time available for relatively smaller data sets to get insight into the larger and more complex datasets they generate now.

The focus of this work is on the multiple dataset scenario, that is, *multiple dataset visualization* (MDV), which means that more than one dataset are visualized simultaneously in the same visualization [11]. MDV is an important concept for the cases

where it is not possible to make an inference based on a single dataset and comparison between different datasets is required. It allows handling of multiple datasets at the same time, representing multiple cases of interest so that important relationships, differences and the effects of different parameters can be better understood. In essence, MDV is an effective tool for comparative data analysis. It provides the capabilities and performance improvement to effectively visualize several datasets.

For visualization to be considered as MDV, it should fulfill the following criteria:

- It should allow multiple datasets to be rendered in the same visualization. This is an important criterion and can be satisfied through overlapping of the datasets or through spreadsheet based representation.
- Visualization should allow interactivity with the dataset. With increasing number of datasets, the large memory requirement and the lack of interactivity become major bottlenecks. Visualization should thus address issues such as performance and interactivity.
- Last but not the least, visualization should allow easy understanding of the datasets under consideration and allow for the comparative analysis of the datasets.

1.3 MDV Techniques

Scientific visualization has moved from the exploration of a single scalar data set to multivariate, time-varying, and comparative data [12-14]. Figure 1 shows various possible approaches for MDV. In the realm of time-varying data, animation is a common way of exploring time series data. While animation is adequate for expressing time evolution, it requires effort from the user to analyze space-time-value relationships. For example, if the user wants to see the relationship between two time steps A and B, he/she has to fast-forward and reverse the animation, or jump between animation frames. Even then, the relationships between value and space in time are not entirely clear due to the loss of positional information. An alternative method to explore multi-field, time-varying, and comparative data sets is to combine all data into one volume [15]. Unfortunately, the user is limited to how many time steps or datasets can be effectively visualized at the same time

because of color composition techniques used. The number of time steps that can be visualized at once is limited due to space filling and occlusion, and it might not be possible to even see volume intersection areas.

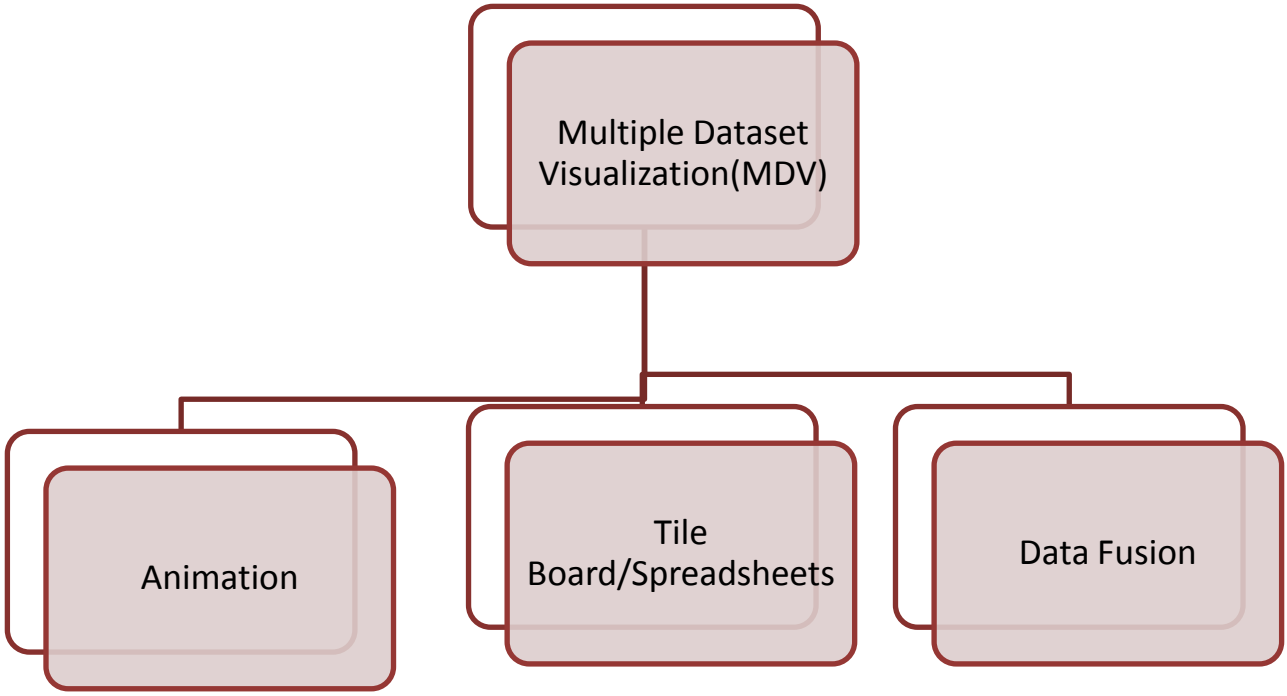


Figure 1: Ways for multiple dataset visualization

Tile boarding/spreadsheet scheme with the increased level of interactivity can be effective for MDV [11, 16] and has been effectively used in framework. A similar concept for 2D images has been studied widely [17-19] for over a decade. These techniques demonstrate the usefulness of the spreadsheet-based interface in image processing and visualization. Spreadsheets provide a tabular layout of the related datasets and operators to manipulate them. Levoy [17] proposed the spreadsheet based scheme for 2D images. Visualization explorer [19] provides an intelligent way of visual representation of the data using varying color maps, opacity values, shading and viewing position. Visualization explorer also

provides certain operators such as histogram generation, union to manipulate the data. Similarly, Ed Huai-hsin Chi [18] proposed the spreadsheet based visualization tool for information visualization. These applications mainly focus on the user interface and do not give insight into the underlying challenges associated with MDV (discussed in Section 1.4), especially in terms of memory requirement and interactivity (*frames-per-second*). The MDV framework deals with the performance issues and system behavior with increasing number of datasets and probable solutions to maintain interactivity in that environment. The challenges in MDV are at the data loading level as well as the visualization level. Our technique uses the spreadsheet-based scheme for the user interface.

1.4 Challenges In MDV

At the first look, visualization of multiple datasets might seem to be a simple extension of the visualization process for a single dataset or a variation of the process involved in the larger single dataset visualization. However, this is not entirely true. Although there are striking similarities between MDV and visualization of a single dataset, applications (providing comparative analysis) and issues such as data preparation, memory requirement, performance and interactivity make MDV interesting and challenging. This section provides insight as to how MDV is significantly different from a normal visualization.

Extensive research has been done in visualization of a single dataset. Most of these techniques are expected to be applicable to the multiple dataset scenario. It is important to note that the performance for 3D datasets degrades by a factor of N or higher for N datasets as compared to a single dataset. This is mainly due to the increase in memory requirement. Even when the size of a single dataset is equal to the total size of multiple datasets together, the memory requirement for MDV can be larger.

1.4.1 Memory Requirement

Consider a scenario where we have a single closed 3D surface and a collection of multiple isolated 3D surfaces whose total area determines the number of required geometric

primitives such as polygons and points. For simplicity, we will consider two cubic volumes V_1 and V_2 of dimensions of L_1 x L_1 x L_1 and L_2 x L_2 x L_2, respectively, with the relation $L_1 = n^{1/3}L_2$.

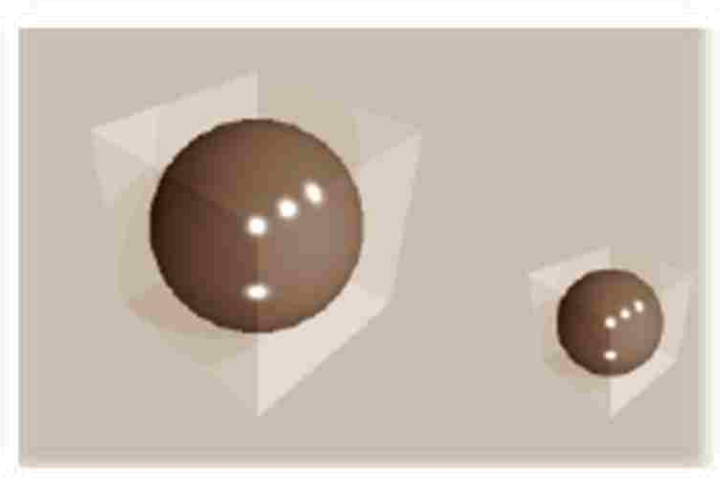


Figure 2: Two cubic volumes with spherical isosurface

Assume that the isosurface of interest in both cases is a single largest sphere that can fit into the volume. For the single volume dataset V_1, the effective isosurface area is A_1

$$A_1 = \pi L_1^2 \qquad \text{Equation 1}$$

However, the area in the case of multiple (n) volumes V_2 is A_2

$$A_2 = n\pi L_2^2 = n\pi \left(\frac{L_1}{\sqrt[3]{n}}\right)^2 = \sqrt[3]{n}\pi {L_1}^2 = \sqrt[3]{n}A_1 \qquad \text{Equation 2}$$

If n = 8, then $A_2 = 2A_1$, implying that a twice the number of polygons or points are needed in MDV than in the single dataset case to represent the isosurfaces. Above scenario justifies the high memory requirement for the case of multiple datasets. MDV requires significantly larger amount of geometric data to be stored and subsequently moved through the graphics bus. Thus, not only the size of the datasets can be very large but also the size of the extracted isosurface can eventually pose a tremendous challenge.

1.4.2 Visualization Display

Similarly, the number of pixels that need to be processed can be larger in MDV. Consider the view direction, which is normal to one of the faces of the cubic volumes. Now assume that the viewport is of size L_1 x L_1 pixels so that the isosurface fits the image area in the case of the volumeV_1. The number of the pixels, which are covered by the circular region on the image plane, is P_1, given by

$$P_1 = \frac{\pi}{4} {L_1}^2 \qquad \text{Equation 3}$$

For the case of multiple (n) volumes V_2, which requires n viewports of size L_2 x L_2 pixels, the number is P_2, given by

$$P_2 = n\frac{\pi}{4} L_2^2 = \sqrt[3]{n} P_1 \qquad \text{Equation 4}$$

If n = 8, we have $P_2 = 2P_1$, thereby involving two times more pixels in MDV than single dataset visualization. This can be an overhead when there are per-fragment operations such as depth test to be carried out.

1.4.3 Performance and Interactivity

Performance is one of the major issues that need to be addressed as it directly affects the speed and interactivity. This problem becomes bottleneck with an increasing number of datasets and required memory to handle these datasets. We try to address some of these issues using the techniques such as data-coherency, multi-resolution and off-screen rendering. The major factors that affect the performance are rendering and memory requirement (if data is locally stored, processed and analyzed). Interactivity also refers to the operations that can be performed on the 2D images (and views of 3D dataset) and 3D datasets.

Flexible interactive option plays a crucial role in any visualization and even more so in MDV because it gives the user with visual feedback from different prospects. MDV needs to be controlled globally as well as locally at the three levels: The *first interaction level*

includes the basic operations such as rotation, translation, scaling, shading, peeling and selection on multiple isosurfaces displayed on the screen, which can spread in a 3D space and contain complex internal structures. The graphics hardware is involved requiring data transfer through the graphics bus. The relevant time is the time for rendering polygons or points denoted as T_R which is desirable to be in fractions of a second for acceptable frame rates.

The *second interaction level* includes changing the isovalue: A wide range of isovalues often needs to be examined to decide on a subset of isovalues of actual interest. The user might be interested to extract the isosurfaces for the same isovalue or for the different isovalues from different datasets, which requires changing isovalue for selected or all datasets. A change of isovalue uses the memory bus during geometry generation and the graphics bus during the subsequent rendering. The relevant timing involves the rendering (T_R) and generation (T_G) times, which together is expected to give acceptable response time (in the order of seconds).

Finally, the *third interaction level* allows loading of the volume data in the process of adding more datasets in MDV. Here, the I/O disk operations need to be executed unlike in the interaction levels 1 and 2. All three times, loading (T_L), geometry generation (T_G) and rendering (T_R) are relevant. Acceptable values for T_L can range from several seconds to several minutes. The flexibility and the ease with which the number of operations can be carried out while maintaining the performance of the application will determine interactivity and its applicability to real world problems.

1.5 Areas of Application

Examples of multiple datasets, which require simultaneous visual analysis, are abundant. Many domains require comparative analysis to understand the relationships among multiple related datasets. One domain where this work is applicable involves materials simulation data, which are used in this study.

Consider 3D charge density distribution in real material systems, which are investigated on routine basis by parallel quantum mechanical simulations [20]. Individual density datasets

of interest might represent different samples, different temperatures, different pressures, or different simulation times.

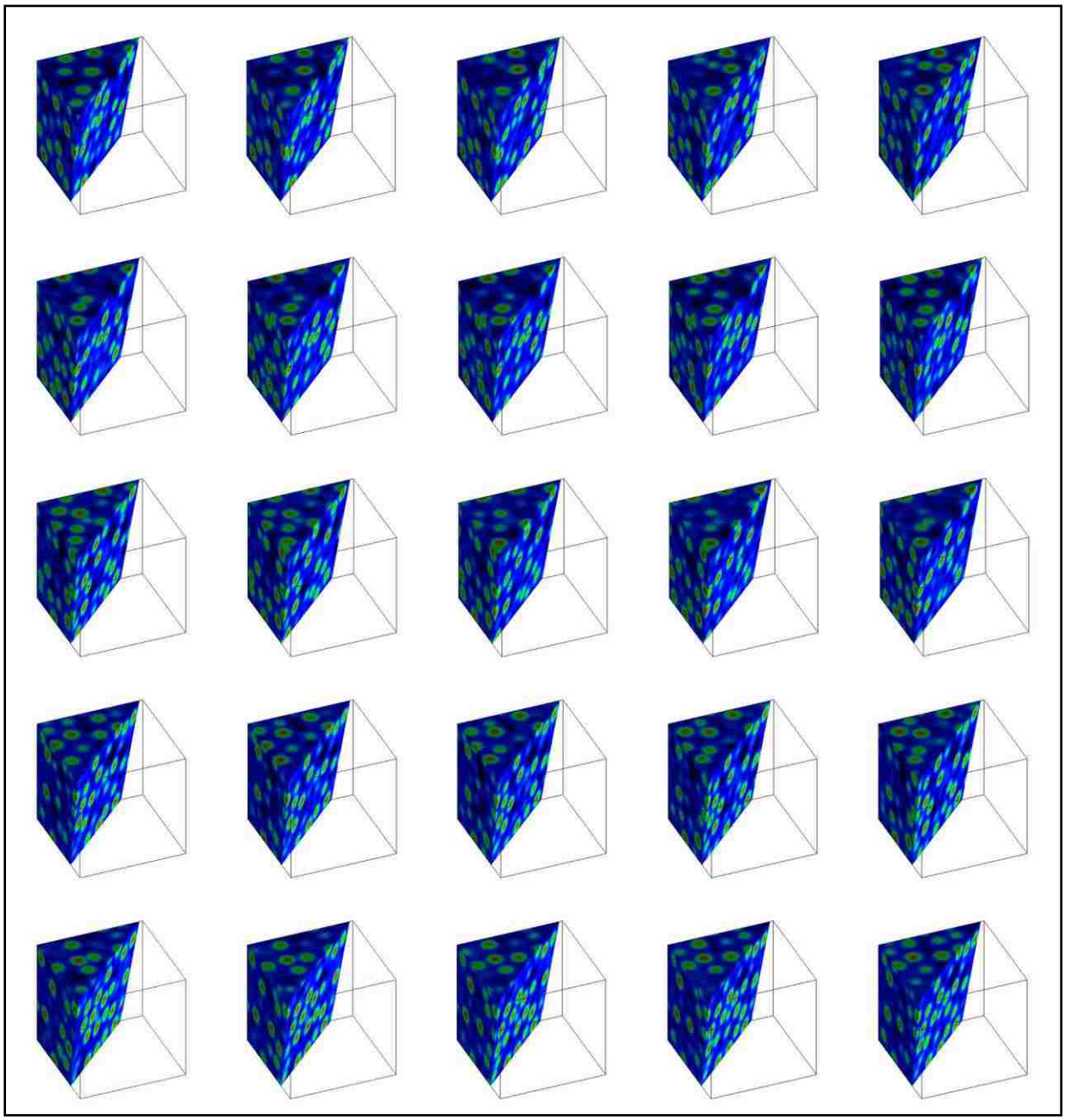


Figure 3: MDV of 25 sets of electronic charged density using 3D surface texture mapping with planar clipping. The color and opacity values for each pixel are based on the density value associated with that pixel. A multiscale RGB color mapping is used. B represents values from 0 to 0.05, G is added to represent values up to 0.4 and then R is increased and both B and G are decreased for higher values.

One might be interested in comparing the charge distribution for different types of vacancy defects in a given crystal, say, Mg, Si, and O defects in an important Earth forming mineral $MgSiO_3$ (magnesium silicate). The interest might be in investigating the effect of pressure by displaying multiple datasets corresponding to different pressure conditions (say Mg-defect at eight different pressures) at the same time. Alternatively, one might need to visualize together ten different datasets as a function of temperature. On the other hand, if one is interested to look at outputs taken at different times of simulation, the number of datasets can be arbitrarily large. Another example is flow visualization for which one might be interested in simultaneously extracting isosurfaces for various flow-associated scalar data including pressure, temperature, density, winding angle and helicity. MDV is expected to be an effective-efficient solution for understanding correlation and relationship between such related multiple datasets. It provides a way to find subtle differences in these related datasets for understanding the system behavior. Figure 3 shows MDV of electron density data. Detail analysis and visualization of materials simulation data are given in chapter 6.

Similarly, in the medical field, multiple sets of MRI volume data can be associated to a given sample taken at different times or some particular structure among different samples or, the data might be showing various parameters. In drug discovery process, one can have data to study the effects of drugs with different composition.

1.6 Thesis Organization

The rest of the dissertation is organized as follows: The chapter 2 describes the previous work related to multiple datasets, computer graphics and visualization techniques. The chapter 3 presents the MDV framework. The chapters 4 and 5 present our MDV approaches for isosurface extraction and texture-based visualization, respectively. The chapter 6 discusses the application of MDV for visualization of electron density data produced by materials simulations. Finally, conclusions and important future directions are presented in the chapter 7.

2 RELATED WORK

"A picture is worth thousand words" [21]. It has been said throughout time that a picture could be worth a thousand words. Images have been the mode of education, communication and knowledge transfer. It is now most ironic that in our present, mostly literate society, we still see the use of images as a major means of influence in television, movies, public service ads, and especially political campaigns [22].

The simple reason why visualization is becoming important is the human desire to understand what is happening in world and beyond from micro to macro level. The surge in the computing power has increased the amounts of data available and thus posing the need to extract the information from these data. Visualization has evolved as a technique for understanding of the data and conveying the information to others. Visualization acts as a superb tool for presenting the information in a way that can be grasped. It helps researcher in finding errors in their simulations and helps in understanding the complex data. The visual representation is known to be more effective communicators of information than the numbers and descriptions.

2.1 Multiple Datasets

Complex multi-dimensional datasets are now pervasive in science and elsewhere in society. This leads to requirement of better interactive tools for visual data exploration so that patterns and relationships in data may be easily discovered, data can be proofread, comparative analysis and subsets of data can be chosen for algorithmic analysis. One can find the increase in research works [10, 17, 19, 23-26] in the context of multiple dataset visualization with a broad range of applications. MDV tools such as iVici and VisCose have been developed for finding the correlations in protein structures. Ecolens, another MDV application aims at visualizing the several datasets. Along with these tools, the techniques such as spreadsheet based visualization [19, 27]; data fusion [12, 15] and animation [10] have been developed to address the issues related to representation of multiple datasets.

2.2 Computer Graphics

The phrase “Computer Graphics” was coined by William Fetter in 1960’s to describe his research at Boeing. Nevertheless, the advancement in computer graphics is attributed to Ivan Sutherland. Early computer graphics was Vector graphics, composed of thin lines whereas modern day graphics are Raster based using pixels. The disadvantages to vector files are that they cannot represent continuous tone images and are limited in the number of colors available. Raster formats on the other hand work well for continuous tone images and can reproduce as many colors as needed. The 60’s and 70’s saw the emergence of computer graphics. 70’s also saw the use of computer graphics in the television. Until 90’s the graphical subsystem of a computer used to be tightly coupled with the software application. This resulted in the duplication of the efforts of managing the graphics hardware. In late 80’s Voorhies [28], showed the feasibility of decoupling the graphics subsystem from the computer system. This led to evolution in the computer graphics and we see the emergence of graphics libraries to ease the use of graphics hardware. Developing graphics applications is easier to develop than before due to the availability of hardware independent graphics programming languages such as OpenGL [29] and DirectX [30]. In the 90’s we see the use of computer graphics in the motion picture and computer games. This trend continues in 00’s and beyond with the programmable GPU’s.

2.2.1 Coordinate System in 3D Computer Graphics

Computer graphics uses homogeneous coordinate system. Homogeneous coordinates were introduced by August Ferdinand Möbius [31], to allow affine transformations such as rotation, scaling and transformation to be easily represented by a 4x4 matrix. Further use of homogeneous coordinates make the calculations possible in projective space just as cartesian coordinates do in a Euclidean space.

A point in the 3D Euclidean space represented by (x, y, z). The point in the homogeneous coordinate system is represented by (x, y, z, w). The matrices used to transform points in 3D space are of size 4x4, each of the rows representing a homogenous vector (x, y, z, w).

The plane at infinity is usually identified with the set of points with $w = 0$. Away from this plane, we can use $(x/w\,, y/w\,, z/w)$ as an ordinary Cartesian system.

The matrix below shows the Translation matrix, d_x, d_y and d_z shows the translation along x, y and z-axis respectively.

$$\begin{pmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$ Equation 5

The matrix below shows the Scaling Matrix. s_x, s_y and s_z shows the translation along x, y and z axis respectively.

$$\begin{pmatrix} s_x & 0 & 0 & 1 \\ 0 & s_y & 0 & 1 \\ 0 & 0 & s_z & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$ Equation 6

The matrices below define rotation matrix along x, y and z respectively. These rotations are non-commutative; rotating an object in the y-axis before rotating it in the x-axis will have a different result to rotating it in the x-axis before rotating it in the y-axis.

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos x & -\sin x & 0 \\ 0 & \sin x & \cos x & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$ Equation 7

$$\begin{pmatrix} \cos y & 0 & \sin y & 0 \\ 0 & 1 & 0 & 0 \\ -\sin y & 0 & \cos y & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$ Equation 8

$$\begin{pmatrix} \cos z & -\sin z & 0 & 0 \\ \sin z & \cos z & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$ Equation 9

2.2.2 3D Graphics Rendering Pipeline

A projection of 3D scene in to the 2D images goes through series of operations that are part of graphics pipeline. The Figure 4 shows a view of 3D graphics rendering pipeline used in modern graphics processors. Each object comprising a 3D scene is specified in its local object coordinate. The object coordinates are transformed into eye coordinates (modeling transformation and viewing transformation), using the model-view matrix to form a composite scene. At this stage, complete scene is described using a single coordinate system, commonly called world coordinate system. After this the coordinates goes through the lighting stage. This needs to be done before projection transformation since z values is needed for the shading and illumination. The eye coordinates are transformed in to clip coordinates by the projection matrix (projection transformation).

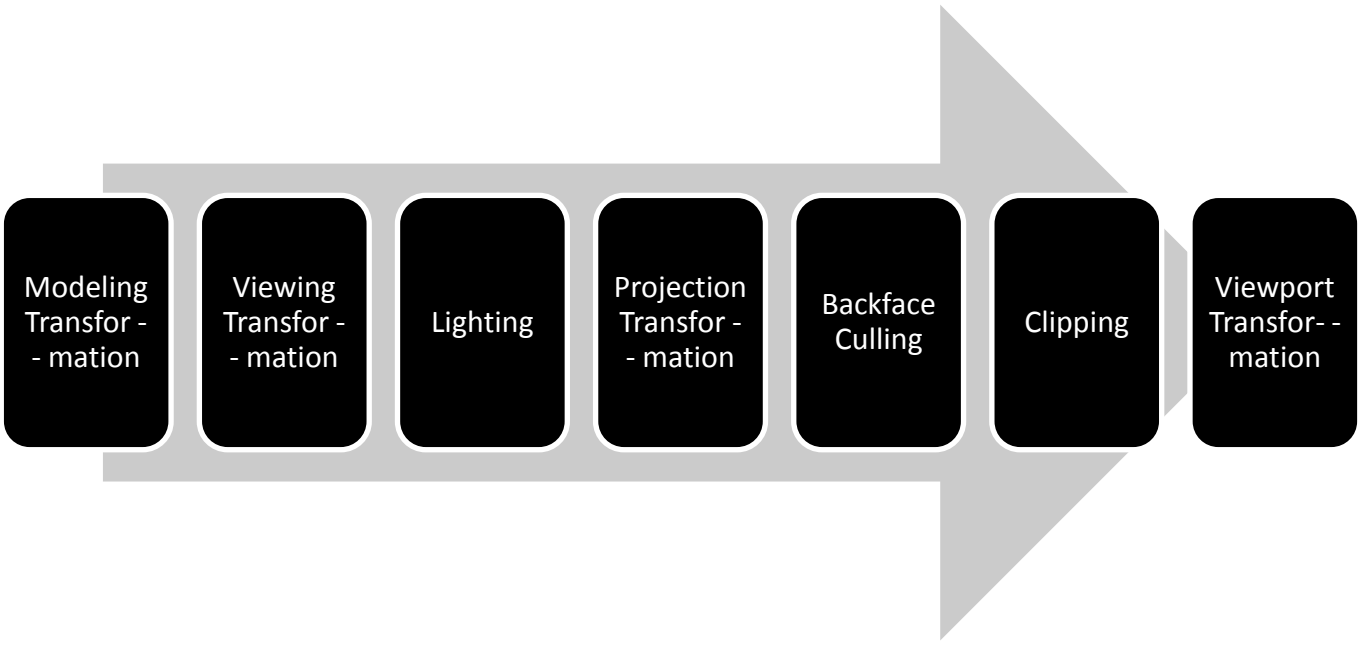


Figure 4: Graphics Rendering Pipeline

There are two major projection mechanisms: orthogonal projection and perspective projection. The orthogonal projection simply maps the 3D scene into a 2D screen without taking the real world distortion that makes the objects that are farther smaller into

account. The perspective projection considers that distortion and an image generated using perspective projection looks more realistic than an image generated using orthographic projection. The resulting coordinates go through the backface culling and clipping. During these stages, the portions of the scene falling outside of the view volume are clipped away. As a final step, the clip coordinates are mapped into window coordinates using the viewport matrix (viewport transformation).

Modern GPU's provide the ability to program certain stages of the rendering pipeline. The GPU (graphical processing unit) of the modern systems provides parallel environment. The GPU connects the north bridge and the display. It provides high memory bandwidth for faster transfer of data to the GPU. The major components of the new GPU architectures are vertex program, geometry program and fragment program. The vertex program allows operation on the vertices and allows lighting computations, texturing and transformation on per-vertex basis. The geometry program operates on each primitive. After a primitive passes through geometry program, it is rasterized and converted to fragments. The fragment program can then operate on individual fragments and allow to perform lighting, texturing, depth based computations and shading. GPU is discussed in detail in section 5.1.

2.3 Overview of Visualization Techniques

In multiple dataset visualization, we process several datasets simultaneously. Two natural approaches for MDV can be a) an extension of the standard visualization methods (available for single dataset) to handle multiple datasets and b) a parallel and distributed processing (using multiple CPUs and/or multiple display screens) for visualization. The second approach obviously requires more resources to deal with more datasets; however, such resources may not be readily available. On the other hand, the first approach enables one to perform MDV in a more convenient way with easily available resources such as PC desktops. In this thesis, we study the first approach and believe that one can adopt our techniques in the parallel programming environment with certain modifications.

Several visualization methods are available for volumetric scalar dataset. Common examples of such techniques include the isosurface extraction [3, 4], ray casting [5],

splatting [6], shear-warp [7] and texture mapping [8]. Indirect methods extract intermediate geometric representation of the surfaces from the volume data and render those surfaces via conventional surface rendering methods, e.g., isosurfaces [4]. On the other hand, direct methods render the data without generating any intermediate representation and as such, they are more general and flexible, e.g., texture-based rendering [8, 32]. Both the strengths and weaknesses of these techniques have been assessed on a wide variety of single dataset [9]. We choose to adopt the isosurface and texture-based visualization methods for MDV for the following reasons: First, the former is an indirect method whereas the latter is a direct method. Second, the former is software-based approach whereas the latter is the hardware-assisted approach. Third, isosurface is widely used technique whereas texture mapping is fast volume visualization techniques and is becoming increasingly popular in recent years [9, 33-35] and has applications in number of areas.

2.3.1 Ray Casting

Ray casting [36] approach was developed by Levoy [37] for producing high quality images. 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, we obtain the color and opacity values 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 C_{out} as it leaves each sample location, is related to the color C_{in} of the ray, as it enters, and to the color $c(x_i)$ and the opacity $a(x)$ at that sample location by the transparency formula :

$$C_{out} = C_{in}\big(1 - a(x)\big) + c(x_i) * a(x) \qquad \text{Equation 10}$$

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.

Ray casting is not ray tracing. Both are image order algorithms and render the scene in two-dimensional space by following the rays of lights from the eye of the observer to a light source. Ray casting has the performance advantage over ray tracing, as it does not compute the news rays after intersection of the original ray with the surface.

2.3.2 Splatting

Splatting was developed by Westover [6, 38] to improve the speed of at the expense of less accurate rendering. It differs from ray casting in the projection method. These splats are rendered as disks whose properties (color and transparency) vary diametrically in normal (Gaussian) manner. Flat disks and those with other kinds of property distribution are also used depending on the application. Splatting projects voxels, i.e. volume elements, on the 2d viewing plane. It approximates this projection by a Gaussian splat, which depends on the opacity and on the color of the voxel (other splat types, like linear splats can be used). A projection is made for every voxel and the resulting splats are composted on the top of each other in back-to-front order to produce the final image.

2.3.3 Shear-Warp

Shear-warp was developed by Cameron and Undrill and was made popular by Lacroute and Levoy [7, 39] using the volume and image encoding techniques. In this technique, the viewing transformation is adjusted so as to align the nearest face of the volume becomes axis with an off-screen image buffer having fixed scale of voxels to pixels. The volume is rendered into this buffer using the far more favorable memory alignment with fixed scaling and blending factors. Once all slices of the volume have been rendered, the buffer is then warped into the desired orientation and scale in the displayed image. The encoding scheme is coupled with the simultaneous traversal of the volume and image. The opaque image regions and transparent voxel are skipped during visualization process. In the preprocessing step, the voxels are coded using run-length encoding based on the opacities.

This requires the construction of a separate encoded volume for each of the three major viewing directions. The rendering is performed using a ray casting like scheme, which is simplified by shearing the appropriate encoded volume such that the rays are always perpendicular to the volume slices. The rays obtain their sample values via bilinear interpolation within the traversed volume slices. A final warping step transforms the volume-parallel base plane image into the correct screen image.

2.3.4 2D and 3D Textures

Texture mapping was first given by Catmull [40]. The texture mapping process lays the image over the polygons. In case of 2D texture, mapping the volume is divided in to volume slices, which are mapped from back to front to provide the volume rendering. In 3D texture mapping, the whole volume is treated as a single 3D texture and the mapping is done to the polygons that are view aligned. Texture mapping is covered in detail in section 2.5. The wide availability and high-performance of texture mapping makes it a desirable rendering technique for achieving a number of effects that are normally obtained with special purpose drawing hardware.

2.3.5 Isosurface Extraction

The isosurface extraction computes and draws a surface within a volumetric data field to cover all locations containing a given single scalar value. The value against which the surface is drawn is termed as *isovalue* and the surface is termed as *isosurface.* Details about the isosurface extraction process are given in the next section. For the work presented here, we consider a marching cube algorithm.

2.4 Isosurface Rendering Technique

2.4.1 Introduction

Software based approach uses CPU based isosurface extraction algorithm to extract the isosurfaces. An isosurface is a 3D surface representation of points with equal values in a 3D

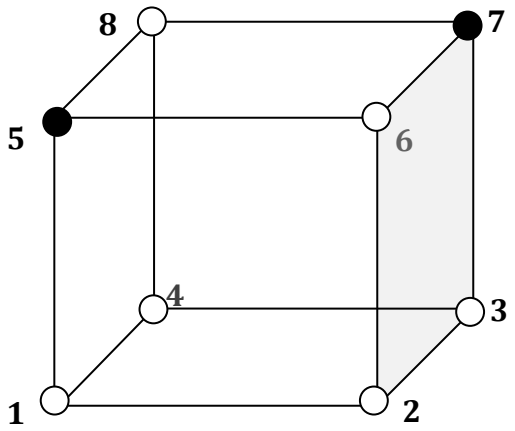
data distribution. These points represent a constant value (isovalue) (e.g. pressure, temperature, velocity, density) within a volume of space. These points are joined together, forming a 3D surface representing the isovalue. A lot of work for isosurface extraction [4, 41-46] has been done over last two decades. Westermann [47] proposed 3D texture based technique for isosurface extraction. GPU based ray casting [41, 48] approach has been effectively used for isosurface extraction.

Isosurface extraction is one of the most widely used methods for visualization of 3D scalar volume data across several disciplines including medicine, geophysics, materials modelling and computational fluid dynamics [1]. The isosurface extraction approaches can be put into three categories based on the space in which a given approach operates: First, the geometric space decomposition category includes the well-known Marching Cubes algorithm for generating isosurface polygons on a voxel-by-voxel basis [4] and the Dividing Cubes approach of subdividing threshold voxels into smaller cubes at the resolution of pixels [45]. It also includes the accelerated approaches based on octree [49] and adaptive reconstruction [50]. Second, the value-space decomposition category includes the NOISE algorithm [3]. Finally, the image-space decomposition category includes mesh reduction [46], ray tracing with an analytic isosurface intersection computation [42] and view-dependent isosurface extraction [3]. A large number of different isosurfacing techniques can be found in the literature [1, 3], majority of which are aimed for the fast extraction of isosurfaces from large-scale data as well as time-varying data. We have used marching cubes algorithm [4] for extraction of isosurfaces from multiple sets of data at the same time.

2.4.2 Marching Cubes Algorithm

Marching Cubes [4, 51-55] is an algorithm for rendering isosurfaces in volumetric data. We use the octree data structure for representing the dataset. In *Marching Cubes* basic notion, a voxel (cube) is defined by the pixel values at the eight corners of the cube. If one or more scalar values of a cube have value less than the user-specified isovalue, and one or more have value greater than this value, then the voxel must contribute some component of the isosurface. *Marching cubes* uses a divide-and-conquer approach to locate the surface in a

logical *cube* created from these eight vertices. The algorithm determines how the surface intersects this cube, then moves (or *marches)* to the next cube.



Bit Vector 00001010

Figure 5: Marching Cube cell

The first step in the algorithm is to look at each vertex of the volume element and determine whether it is above or below the given isovalue. If the value is above the given isovalue, the vertex is assigned 1 otherwise 0.

The second step in the algorithm is the computation of exact locations along the edges. It is done using the simple linear interpolation between the vertices *i* and *j.*

$$x = x(i) + fac * \big(x(i) - x(j)\big) \qquad \text{Equation 11}$$

$$y = y(i) + fac * \big(y(i) - y(j)\big) \qquad \text{Equation 12}$$

$$z = z(i) + fac * \big(z(i) - z(j)\big) \qquad \text{Equation 13}$$

Where fac is given by,

$$fac = \left(\frac{\sigma(j) - \sigma_{iso}}{\sigma(j) - \sigma(i)}\right) \qquad \text{Equation 14}$$

(x, y, z) represents the computed point along the edges where the isosurface intersects. $\sigma(j)$ & $\sigma(i)$ correspond to the scalar values at vertices j and i respectively. σ_{iso} is the isovalue at which the isosurface is being calculated.

Since there are eight vertices in each cube, there are 2^8 = 256 ways a surface can intersect the cube. By enumerating these 256 cases, look up table for determining surface-edge intersections is created, given the labeling of a cubes vertices. The table contains the edges intersected for each case. Two different symmetries of the cube reduce the problem from 256 cases to 15 patterns [4]. Figure 6 shows the triangulation for the 15 patterns.

The simplest pattern, 0, occurs if all vertex values are above (or below) the selected value and produces no triangles. The next pattern, 1, occurs if the surface separates on vertex from the other seven, resulting in one triangle defined by the three edge intersections. Other patterns produce multiple triangles. By determining which edges of the cube are intersected by the isosurface, we can create triangular patches that divide the cube between regions within the isosurface and regions outside. By connecting the patches from all cubes on the isosurface boundary, a surface representation of the isosurface is achieved. The result of the algorithm is the facets, which approximate the isosurface within the given volume. Smooth shading is enabled for giving the isosurface more realistic view. For shading, the surface normal needs to be calculated. This is done using the gradient of the scalar value as

$$\left[\frac{d(F(x,y,z)}{dx} \quad \frac{d(F(x,y,z)}{dy} \quad \frac{d(F(x,y,z)}{dz}\right] \qquad \text{Equation 15}$$

The Marching Cube algorithm is easy to implement but it generates disconnected isosurface. There are also the ambiguity cases where it is difficult to decide the structure of the isosurface [1]. There exist several approaches for resolving this problem such as marching cube tetrahedral, asymptotic decider and extended MC cases. The marching cube tetrahedral takes care of the ambiguous cases but generates much more triangles, and another problem is regarding the orientation of the tetrahedral when expressing the cell as a tetrahedron, which may result in bumps along face diagonals. The other approach [51] asymptotic decider is based on an analysis of the variation of the scalar value across the

ambiguous face. The analysis decides how the edges are to be connected. Another solution, which is simple and effective, extends the 15 cases [1] of marching cube.

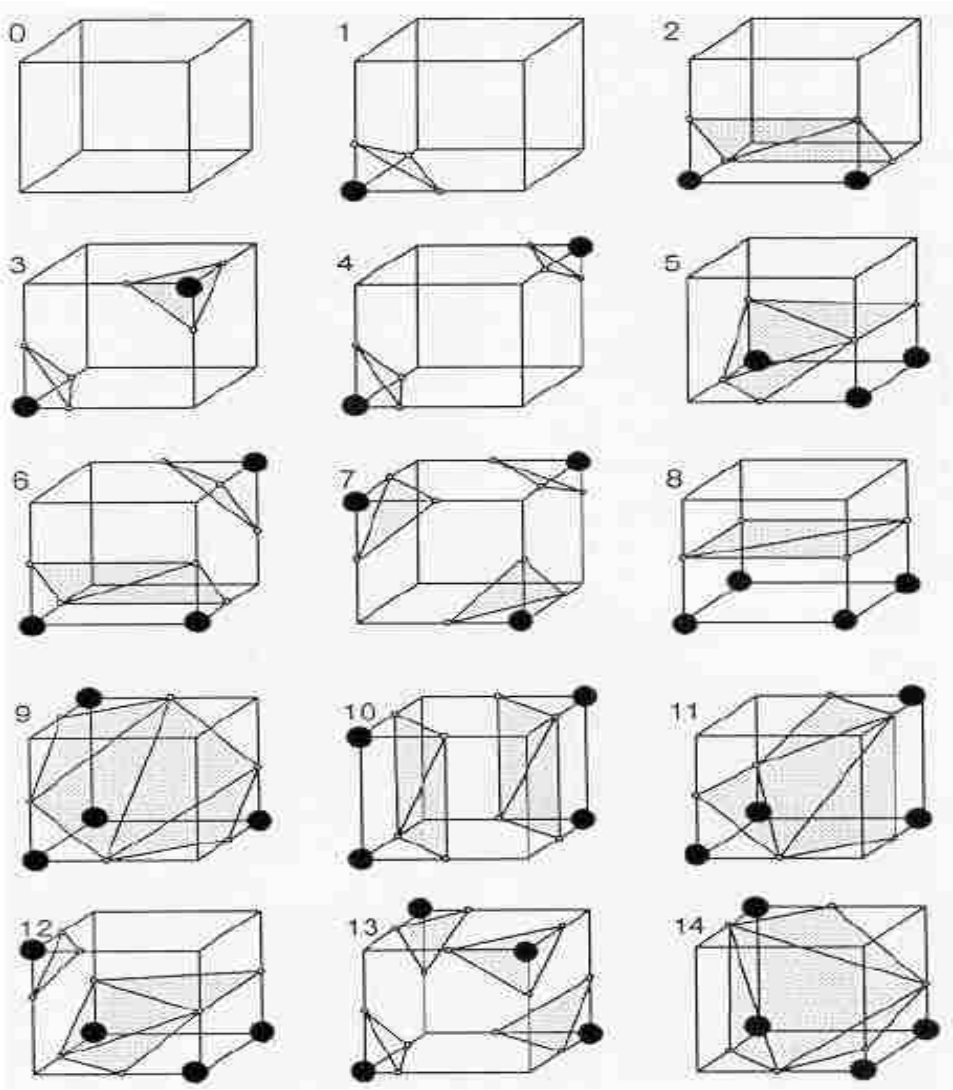


Figure 6: Different cases for intersection of surface with cube [4]

2.4.3 Dividing Cubes Algorithm

Motivation behind *the Dividing Cubes* algorithm was to eliminate the scan conversion step of polygonal display algorithm. This is important, for example, electronic charge density

dataset; there are many surface elements and triangles created by the polygonal algorithm, which may occupy only a small area on the raster display. In addition, as the size of data increases, the triangles will occupy only small area. Thus, it is more efficient in memory and time to display point primitives on the raster display directly than rendering polygons.

The dividing cubes algorithm proceeds by recursively dividing the volume cells that lie on the surface of the object such that the smaller subdivided cells are equal to the size of the pixel. Now each point generated from the subdivided cubes will correspond to the pixel on the display. Subdividing the cell is equivalent to interpolating the data in three dimensions to generate the scalar values at eight corners of the cells. This is achieved by using linear interpolation. The image resolution is limited by the display resolution rather than the voxel size. Using this process leads to generation of a large number of points. The dividing cube algorithm calculates the index for each of the subdivided cube. The vertex is termed outside (inside) if its scalar values is less (greater) than the isovalue. The cubes that lie on the surface have some values greater than and some values less than the isovalue. These cubes are used to calculate the isosurface [45]. To improve the image quality, volume shading is used. The normal at each contributing point is calculated using the central difference formula and is given by

$$g_x = [f(x_0 + a, y_0, z_0) - f(x_0 - a, y_0, z_0)]/2a \qquad \text{Equation 16}$$

$$g_y = [f(x_0, y_0 + b, z_0) - f(x_0, y_0 - b, z_0)]/2b \qquad \text{Equation 17}$$

$$g_z = [f(x_0, y_0, z_0 + c) - f(x_0, y_0, z_0 - c)]/2c \qquad \text{Equation 18}$$

Where gradient vector is given by, $G = (g_x, g_y, g_z)$. a, b & c are the dimensions of the unit cell and $f(x, y, z)$ gives the scalar value at (x, y, z) point. Unit surface normal is the gradient normal divided by its magnitude, $|G|$.

The Marching [4] and Dividing [45] cubes algorithms are used to generate polygons and points, respectively. The essence of both the algorithms remains same for MDV: It examines all voxels of each volume dataset and determines, from the arrangement of vertex values

above or below a threshold value, if and how an isosurface would pass through these elements. The algorithm thus generates isosurface geometry on a voxel-by-voxel basis. Once all the voxels of one volume dataset are processed and the corresponding isosurface is extracted, the same is repeated for each remaining volume in a given multiple set.

2.4.4 Span Space Based Approach

The span space [Figure 7] technique places the cells according to the maximum and minimum values (among the eight vertices of each cell) in the 2D place. The x axis corresponds to the minimum value and y axis corresponds to the maximum value. All the points thus lie above x = y line. All the cells that correspond to the isosurface for the particular isovalue must satisfy the following condition (x <= isovalue) and (y>= isovalue). This is the NOISE (Near Optimal IsoSurface Extraction) algorithm.

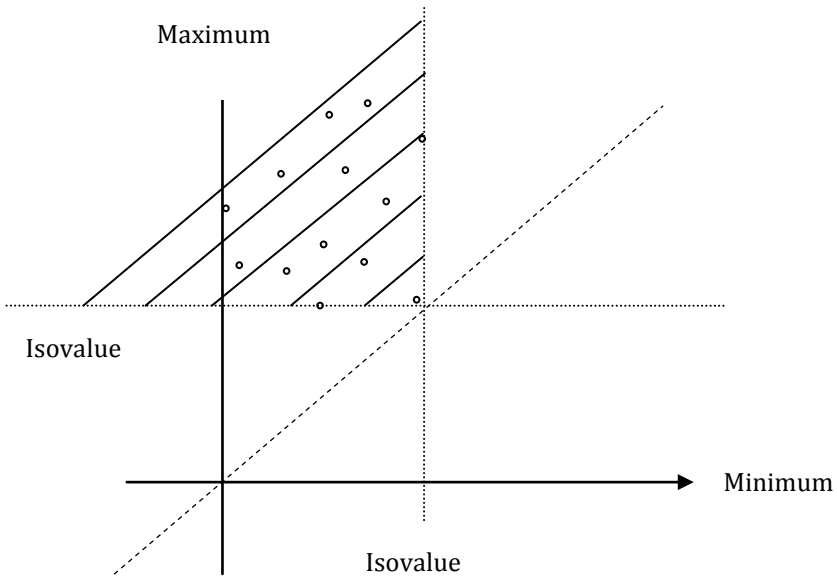


Figure 7: Span space for NIOSE algorithm

The ISSUE (Isosurfacing in Span Space with Utmost Efficiency) [Figure 8] algorithm further optimizes the NOISE algorithm by dividing the plane generated in the NOISE algorithm in to grids to speed up the search process for cell which contribute to finding isosurface.

These cells can be categorized into 5 types depending upon the position where they lie on the min-max plane resulting, which defines the type of computation needed for each of them. After searching for cells that contribute to the isosurface, the surface is evaluated in a similar way as for marching cube algorithms.

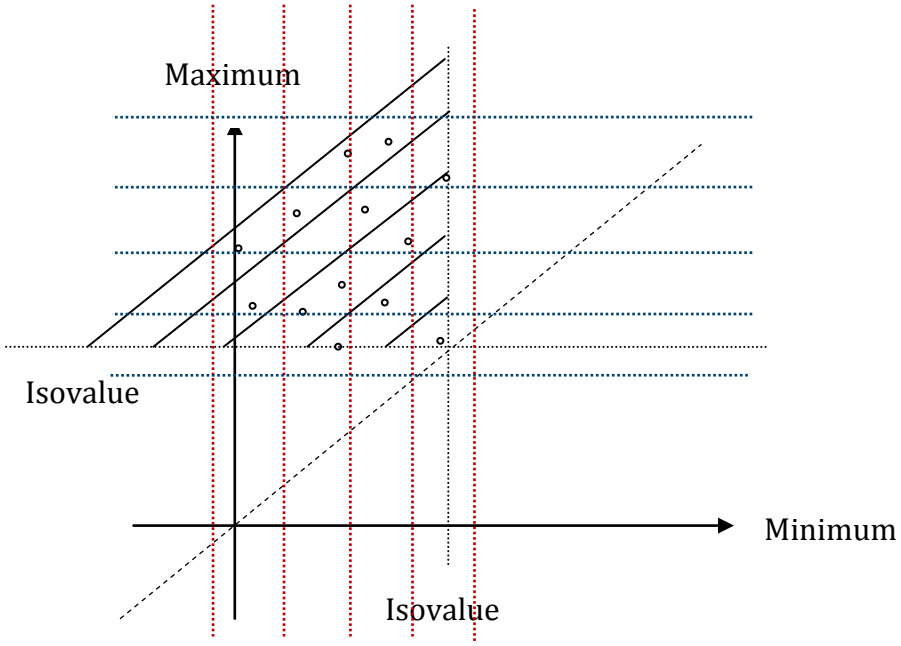


Figure 8: Division of span space for ISSUE algorithm

2.4.5 Isosurface Extraction Using Ray Tracing

Other techniques for finding isosurface involve using the Ray Tracing method [42]. The ray tracing method shoots the ray in the image plane through each pixel. Each ray, if it intersects with the volume cell contributing to the isosurface, is tested. In the first phase of the algorithm ray $\vec{a} + t\overrightarrow{b}$ is traversed through the cells, and then analytical computation of the isosurface for each cell contributing to the isosurface is performed. Finding the parameter t for the ray intersecting the isosurface is done by

$$\rho(x_a + tx_b, y_a + ty_b, z_a + tz_b) - \rho_{iso} = 0 \qquad \text{Equation 19}$$

Only the roots, which lie within the cell, are considered. If there are no roots then this signifies that the cell under consideration is not the part of the isosurface. If the equation has multiple roots the ray intersects the cell at multiple points.

2.5 Texture Mapping Technique

Texture mapping [8, 56, 57] is one of the widely used techniques in recent years for visualization of scalar data. Due to recent advances of commodity graphics hardware, texture-based rendering is able to achieve acceptable frame-rates with high image quality [32, 56, 58]. Clipping combined with texture-based rendering can exploit advanced fragment operations supported by graphics hardware and acts as an important tool for uncovering the hidden details. Planes [59] were previously used for clipping. Weiskopf et al. proposed techniques for volume clipping with complex clip geometries, which are based on the depth structure and voxelization of the clip geometry [60]. Some studies have used volume clipping based on stencil test [47], isosurface clipping and interactive clipping combined with dual-resolution texture-based volume rendering [61]. For example, Van Gelder and Kim [59] have used clip planes. Techniques for volume clipping with complex geometries, which are based on the depth structure and voxelization of the clip geometry and also involve subsequent shading of the clipped surfaces have been proposed [60, 62]. In the volume clipping based on stencil tests, stencil buffer entries are set at only those positions where the clip plane is covered by an inside part of the clip geometry [47]. There are also techniques, which have exploited isosurface clipping [63] and interactive combined with dual-resolution texture-based volume rendering [61].

Multitexturing is the use of more than one texture at a time on a polygon. For instance, a light map texture may be used to light a surface as an alternative to recalculating lighting every time the surface is rendered. Another multitexture technique is bump mapping, which allows a texture to directly control the facing direction of a surface for the purposes of its lighting calculations; it can give a very good appearance of a complex surface, such as tree bark or rough concrete, which takes on lighting detail in addition to the usual detailed

coloring. Bump mapping has become popular in recent video games as graphics hardware has become powerful enough to accommodate it.

The way the resulting pixels on the screen are calculated from the texels (texture pixels) is governed by texture filtering. The fastest method is to use the nearest-neighbor interpolation, but bilinear interpolation or trilinear interpolation between mipmaps are two commonly used alternatives which reduce aliasing. In the event of a texture coordinate being outside the texture, it is either clamped or wrapped.

2.5.1 2D And 3D Textures

Often textures are related to 2D images, but they can be 1, 2 or 3 dimensional. Texture can be treated as texture (e.g. wood, cloth) or it can be defined as a multidimensional image [64]. Textures in general are the arrays of data. Texture mapping is a process of applying function to the surface. The domain of surface can vary in dimensions (1D, 2D or 3D). The texture mapping approach uses 2D or 3D textured data slices, combined with an appropriate blending factor [8, 57]. Figure 9 shows the relationship between 2D and 3D textures.

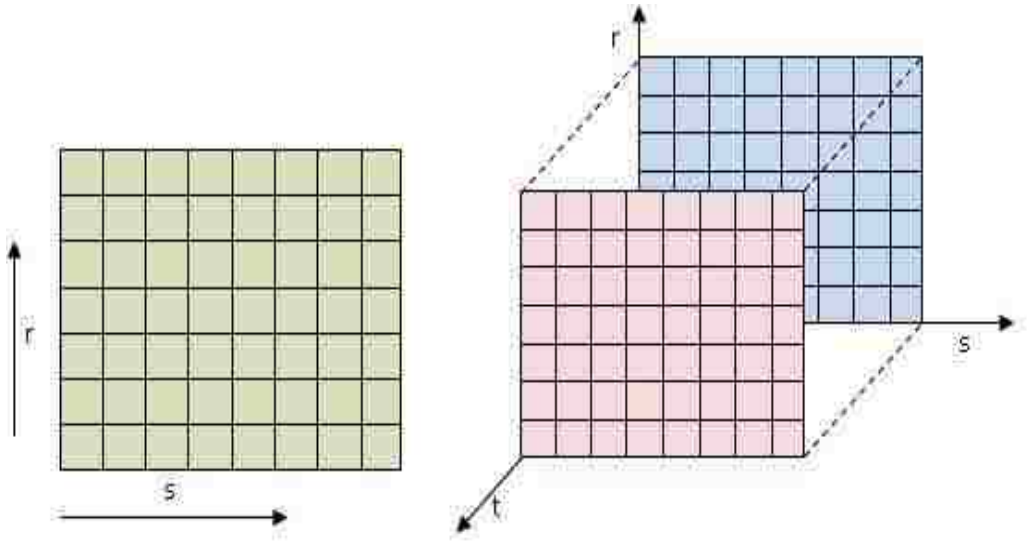


Figure 9: Relationship of 2D and 3D Textures

In case of 2D textures, the volume is represented as axis-aligned slices. These slices are mapped to the polygons to enable volume rendering. The hardware does the bilinear interpolation between the slices before rendering them. This process introduces an artifact when the volume slices are parallel to the viewing direction. In this case, there are gaps visible between the slices. Thus, for exact volume rendering three sets of slices each parallel to x, y and z-axes are generated. Out of three sets, the one that is most parallel to the viewing direction is used for rendering the volume. Unlike 2D textures, 3D textures allow arbitrary cross-sections of the volume to be mapped, thus eliminating the need of maintaining three sets. Also since the arbitrary cross-sections are allowed in the 3D texture mapping, certain operations (such as box clipping) to be performed on the volume effectively.

2.5.2 3D Texture Based Volume Rendering

3D texture approach can sample the data in the *s*, *t* or *r* directions freely so the slices can always be oriented perpendicular to the viewer's line of sight. Image quality is independent of the viewing direction. The intrinsic trilinear hardware interpolation allows us to perform super-sampling, i.e., to use an arbitrary number of slices with an appropriate resampling on the slices. Only one single 3D texture needs to be loaded thus requiring one third of the memory, compared to the case of 2D textures. In 3D texture mapping approach, first step is to load the volume data into a 3D texture; it involves simply reading a set of images or shading data points. The second step involves choosing the number of slices perpendicular to the viewing direction for each texture. The numbers of slices should be equal to the volume's dimensions, measured in texels. For instance, each dataset of 256^3 needs 256 slices. Next step is to generate the series of polygons that cut through the data perpendicular to the viewing direction. Final step is to render each slice as a textured polygon, from back to front. A blend operation is performed at each slice; the type of blend depends on the desired effect.

As the viewpoint and direction of view changes, recomputation of the data slice positions and update the texture transformation matrix is necessary. The third step is to use texture coordinate generation to texture the slice properly with respect to each 3D texture data. Finally, the textured slices are rendered from back to front, towards the viewing position, with appropriate blending performed at each slice. As the viewpoint and direction of view change, one needs to recompute the data slice positions and update the texture transformation matrix as necessary.

Affine texture mapping does not take into account the depth information about a polygon's vertices, where the polygon is not perpendicular to the viewer it produces a noticeable defect. If these texture coordinates are linearly interpolated across the screen, the result is affine texture mapping. This results in noticeable discontinuity between adjacent triangles when these triangles are at an angle to the plane of the screen. In Perspective correct texture mapping for interpolation the coordinates are divided by their depth, and the reciprocal of the depth value is also interpolated and used to recover the perspective-correct coordinate. This correction makes it so that in parts of the polygon that are closer to the viewer the difference from pixel to pixel between texture coordinates is smaller (stretching the texture wider), and in parts that are farther away this difference is larger (compressing the texture). All modern graphics hardware implements perspective correct texturing.

Affine texture mapping directly interpolates a texture coordinate u_α between two endpoints u_0 and u_1:

$$u_\alpha = (1 - \alpha)u_0 + \alpha u_1 \text{ where } 0 \leq \alpha \leq 1 \qquad \text{Equation 20}$$

Perspective correct mapping interpolates after dividing by depth z, then use its interpolated reciprocal to recover the correct coordinate:

$$u_\alpha = \frac{(1-\alpha)\frac{u_0}{z_0} + \alpha\frac{u_1}{z_1}}{(1-\alpha)\frac{1}{z_0} + \alpha\frac{1}{z_1}} \qquad \text{Equation 21}$$

Besides interactive frame rates provided by 3D texture rendering [47], there is often a need for flexible editing of the dataset. This becomes all the more important in case of MDV where the goal is comparative visualization. This can be done intuitively and in an efficient way using the clipping object.

2.5.3 Clipping

Clipping was earlier performed using a variety of clip geometry, from simple plane, to multiple planes, to complex polygons. With improvement in the graphics hardware, clipping can be done using complex clipping objects. For example, Van Gelder and Kim used clip planes [59]. Weiskopf et al. have proposed techniques for volume clipping with complex geometries, which are based on the depth structure and voxelization of the clip geometry [62] and also involve subsequent shading of the clipping surface [60]. Some studies have exploited isosurface clipping [63]. Often clipping is combined with texture-based rendering of the data to exploit the advanced fragment operations supported by graphics hardware.

2.5.3.1 Planar Clipping

Planar clipping planes are used for clipping, although arbitrary clipping object is more desirable from visualization point of view as it provides more flexibility. The clip plane in the 3D space is given by

$$Ax + By + Cz + D = 0 \qquad \text{Equation 22}$$

OpenGL [65] can be used to define the clip plane and perform all transformations the plane undergoes subsequently. The clip plane like any other geometry is transformed according to the current viewing matrix. All points with eye coordinates (xe, ye, ze, we) that satisfy the equation lie in the half-space defined by the plane

$$(A\ B\ C\ D)M^{-1}(xe, ye, ze, we) \geq 0 \qquad \text{Equation 23}$$

Here *M* is the current model-view matrix. All points not in this half-space are clipped away. Model view matrix is (4X4) matrix, which holds the current state of the scene. Scene defines the rotation, scaling and translation. Whenever rotation, scaling or translation is performed, the model view matrix is changed according to the parameters of the corresponding operations taking place.

2.5.3.2 Volume Clipping

Volume clipping in the past has been implemented using multiple clipping planes using OpenGL. There are two major drawbacks of this approach. Firstly, this approach is limited y the number of clipping planes provided by the OpenGL implementation. Secondly, clipping of arbitrary geometry is not feasible. Modern advancement in GPU, have enabled arbitrary clipping geometry from the volume data. Some of the volume rendering techniques, namely depth based clipping and voxelized clipping were developed by Weiskopf [60], exploit GPU supported per-fragment operations.

Depth Based Clipping

This approach analyzes the depth structure of the boundary representation of the clip geometry to decide which parts of the volume have to be clipped [62]. Depth buffer can store only one depth value per pixel. As such, the depth buffer allows single clipping plane and volume clipping cannot be implemented. The idea is to store the depth of the volume as the high-resolution depth texture and then use the depth texture to do volume clipping. The algorithm work as follows: In the first step, the depth values corresponding to the first boundary of the clipping geometry are stored in Z_{back} . This is achieved by rendering the back faces of the clip geometry into the depth buffer. The value in the depth buffer where clipping object is not present contains the far value of the viewing frustum. Next step is applying the depth test with value “greater” with changing the depth buffer. This process draws the volume behind the clipping object. In the second pass, the volume in front of the clipping object is drawn. This pass is similar to the first pass, except now Z_{front} is used and the depth test is performed with value set to “less”. The equations corresponding to the process are

$$d_f(z_f) = ((z_f - z_{front} \geq 0) \wedge (z_f - z_{front} \leq 1)) \wedge (z_f - z_{front} \leq z_{back} - z_{front})$$

Equation 24

Here $d_f(z_f)$ is the depth test for the fragment with the depth value z_f. Assuming that the depth values are normalized [0, 1]. By changing the values of the depth test in the first and the second pass volume cutting is achieved. The above approach is called volume probing. Volume cutting is opposite to volume probing. Here, only the portion of volume outside the clipping object remains.

Voxelized Clipping

Voxelized clipping technique [60] utilizes two textures, volume object and clipping object. The 3D texture represents the voxelized clip object in the binary form in which the alpha value of the clipping texture is set to 0 or 1 depending upon whether the voxel lies inside the clipping geometry or not. Depending upon the alpha value of the clipping texture GPU assigns the alpha value to the volume texture. The fragments lying outside the clipping texture are assigned value zero so these fragments are discarded using the alpha test for volume probing. For volume clipping, the fragments lying outside are assigned the value 1. The clip object can be moved around by mapping the texture coordinates of the clipping texture although changing the shape of the clipping object requires the loading of the new 3D texture. Since the method utilizes the binary representation there is a visible transition between the voxels having value 0 and 1.

To overcome this artifact one can utilize the distance map similar to the one given by Lorenson [4]. Evaluation of *F(x, y, z)* at any point produces a value < 0, = 0, or > 0, where *F(x, y, z)* gives the signed Euclidean distance of the voxel from the surface of the clipping object. Points that lie inside the clipping object are assigned negative values and points outside the clipping object are assigned the positive values. The points on the surface of the clipping object have a distance zero. If RGBA 3D textures are used, these distances are clamped to the range (0, 1). Floating-point textures remove any round of error due to clamping in RGBA textures. For different clipping geometries there needs to be a different

clipping 3D texture. The same 3D clipping object can be reused for the more than one dataset in case of MDV. A new 3D texture needs to be loaded if different clipping geometries are used with different datasets.

3 MDV FRAMEWORK

3.1 What Is Framework?

Framework in general is a conceptual model or structure which provides the guidelines for utilizing and expanding the concepts presented in the framework to solve similar problem. In computer system, framework usually represents the layered structure acting as a guideline for solving a problem. It may also include what components of the framework will interact with each other, and may also include actual program or offer programming tools for using the framework [66]. The framework presented here outlines the steps to solve the problem of visualizing multiple datasets. It specifically looks in to the performance issues for the case of multiple datasets.

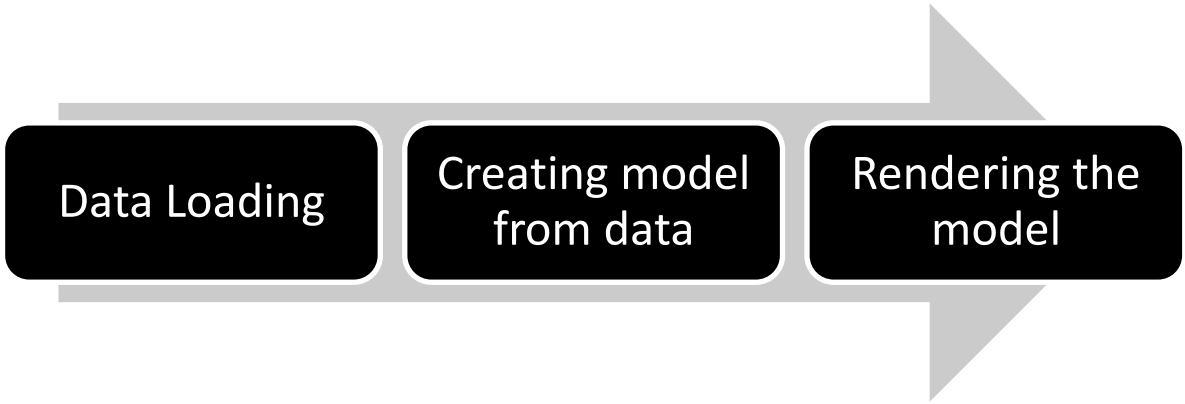


Figure 10: Basic steps in the visualization process

Visualization is the process of representing information in the form of visual imagery. Ignoring the details, visualization is a three step process [Figure 10] involving loading the dataset, creating a model and finally rendering the model. Depending upon the desired output, research goals, nature and complexity of the data the steps in visualization process can vary or may include additional steps. For example, in case of isosurface visualization the second step can involve generating the model or surface in the form of triangles or in case of texture based visualization, the second step may involve generating a 3D image of

the data, where value at each point in the dataset is represented by the some transfer function involving RGB values. The framework described in this thesis is a general guideline for MDV case and can be customized accordingly.

3.2 MDV Framework

Our proposed framework is an extension of the visualization process, to address the issues such as memory requirement and interactivity, which are critical for any multiple dataset visualization system. It is around isosurface and texture based visualization techniques, but can be extended for other visualization techniques.

3.2.1 Architecture

Our MDV framework follows a multithreaded multilayered architecture. Core modules of the framework are data preparation, visualization and rendering [

Figure 11] that run on separate threads. Apart from these core components, framework also has an analysis module. Apart from these, the framework consist of user interface and central engine to handle input going to core modules either in the form of user interaction or change in state of the dataset. The analysis module interacts with all the other modules and comprises of algorithm needed for analysis at the data level, or geometry level or image level. The data module of the architecture only interacts with the analysis and visualization modules, the visualization module acts as a bridge between the data and the rendering components and is responsible for implementing visualization algorithms such as marching cube algorithm. These algorithms transform the data in to primitives and these primitives are rendered in the off-screen buffer through the rendering module. The rendering module implements algorithm for off-screen rendering making it independent of any visualization technique under consideration. This module is also responsible for handling user input affecting the view state (rotation & translation) of the dataset.

Figure 11 shows the interaction of these components. Rendering module also implements the algorithm for the displaying the datasets in different views namely normal, comparative and analysis mode.

Since the rendering module is not directly dependent on the visualization techniques, the framework provides flexibility for adding new visualization techniques. Figure 12 and Figure 13 shows the extended view of the framework. All these modules run in parallel [Figure 12] to improve the performance and to provide much needed interactive environment for MDV. These modules follow the concept of event driven programming and are triggered internally as new datasets are added or externally when the user interacts with the datasets.

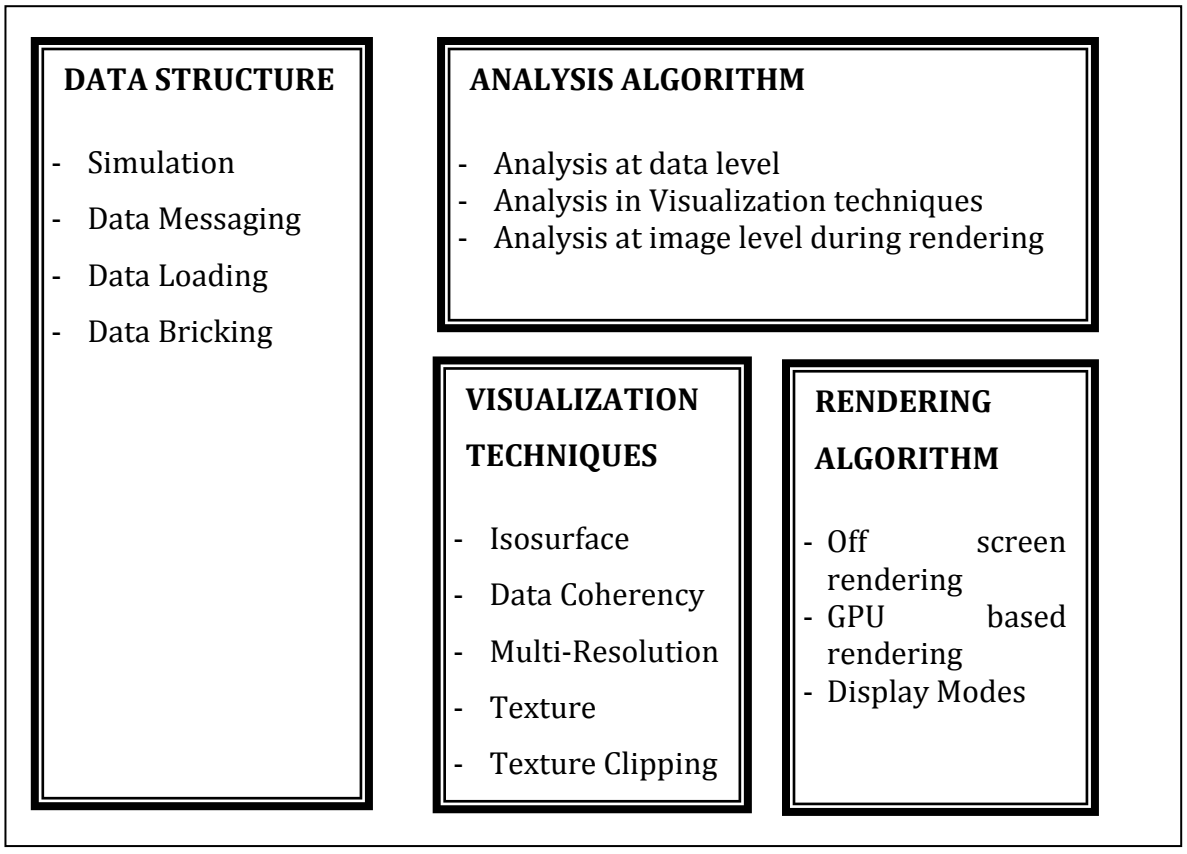


Figure 11: Multiple Dataset Visualization

At the beginning of the computation, all the threads remain in the suspended state. As the datasets are added, the data loading thread becomes active and is responsible for loading the dataset. Once the loading of dataset is completed, the data loading thread signals the visualization thread and goes in the suspended state if no more datasets are available. If more datasets are available it will continue to load other dataset and send the signal to the visualization module about the availability of the datasets. Signal from data loading thread activates the visualization thread. The visualization thread performs the computation depending upon the visualization technique and passes the signal to the rendering thread. The rendering thread renders the primitives to the off screen buffer and then to the display. Also the visualization thread remains in suspended state if no more datasets are available. The multithreading feature of the framework allows maintaining the user interactivity even while dealing with many datasets. Other major reason to maintain separate threads for each of the modules is to reduce latency and improve the responsiveness of the application. The application latency is defined as the feedback from the application to the user input.

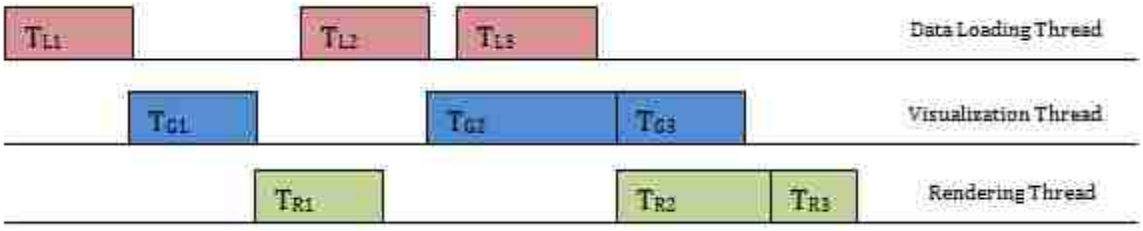


Figure 12: Multithreaded Architecture

3.2.2 Multithreaded View

Figure 13 presents the detailed view of the framework for MDV application. The major components of the multi-threaded framework are, data loading, visualization/image processing, rendering module, analysis module, OpenGL based engine and user interface. Centerpiece of the framework is the MDV engine, which controls all other modules via the user interface. The engine is responsible for initializing the data loading thread,

visualization thread, and rendering thread. The data-loading module and the visualization module are customizable according to data and user requirements. As explained in the previous sections, the data loading module dispatches the *signal* to visualization module as soon the data is loaded and is available for further processing.

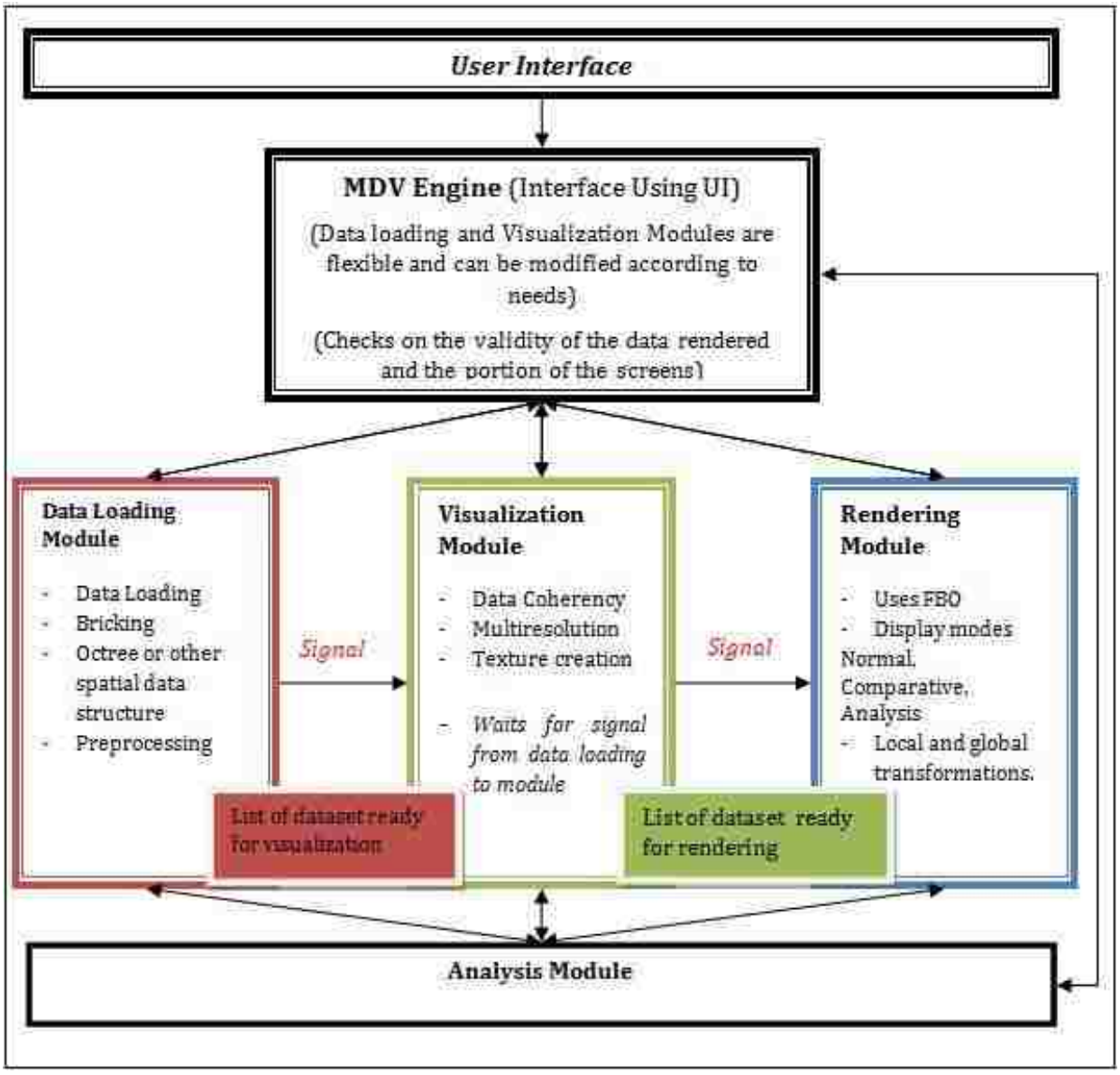


Figure 13: MDV Framework

Similarly, the *signal* event from visualization module to rendering module tells that the geometry primitives corresponding to the dataset are available for rendering. Data loading, visualization and rendering modules each run in separate threads. Analysis module is at the lowest level of the framework and allows analysis at data, geometry and rendering level. The user interface allows the user to load new datasets and analyze them. It also allows the user to control the view of the datasets and for performing transformation in the already uploaded datasets.

3.3 Data Loading Module

The data loading module is responsible for loading the datasets. The MDV engine creates the data loading thread. This thread remains in the suspended state. It is only invoked by through the user interface when the new dataset is added or some other operations which requires altering the data. The user interface, upon receiving the input from the user, signals the data loading thread. The data loading thread loads the data and is also responsible for the pre-processing of the data. Apart from loading the data, this thread is also responsible for data bricking or creating an octree in case of isosurface based technique. This module is also responsible for dataset preparation, which is critical since the input to the visualization module depends upon type of visualization technique in use. Other functions of this module are loading the data in spatial data structures such as octree and calculating min-max values for the data blocks in the octree and can extend to involve compression, wavelet transformation or any other step directly dealing and transforming the data. Once the data is loaded, it is added to the list of the available dataset. This list is shared by the visualization thread and the data loading thread. Once the data loading thread performs the requested operations, it again goes in the suspended state. Data loading thread signals the visualization thread as soon the data becomes available.

With increasing number of datasets, memory requirement increases. Graphics memory may not be large enough to hold the even the single dataset. Hence, we see two levels of paging taking place; the first one is between graphics memory and main memory and second occurs between, main memory and virtual memory. Thus, performance degradation is due to the amount of paging occurring during visualization process. To improve the

performance (or reduce the amount of paging) bricking the dataset in to smaller datasets can be used. The data needs to be organized in the memory to exploit the spatial locality. This has been shown to improve the paging of the data between virtual and main memory and graphics memory [67]for single dataset. The challenge with bricking in case of MDV is from the amount of data (decomposition and reconstruction of the dataset can be an overhead). Secondly, the advantages of bricking diminish with the increase in the datasets. We also need to take in to consideration that in case of MDV, virtual memory come in to picture, which can further slow down the process of bricking and reconstruction. Another important consideration while loading the data is selection of data structure as this will determine how effectively we can exploit the paging. The choice of data structure will depend upon the amount of speed up for isosurface extraction process and the overhead associated with the dataset. For example in case of interval trees they speed up the process but can be significantly space consuming [68].

3.4 Visualization Module

Visualization thread is responsible for responding to signal from data loading module and generating the output based on the visualization technique requested by the user. The visualization module checks the list of available datasets and performs the requested operation on the dataset. As with data loading thread, the visualization thread remains in the suspended state after creation, and can only be invoked by the data loading thread or user interface. After performing the requested task, it signals the rendering module and goes in the suspended state. This thread can be activated either through the user interaction (change in isovalue or change in transfer function for the texture) or through the signal from the data loading thread.

Currently, the visualization module implements two visualization techniques namely isosurface and texture rendering. To give a brief overview, the isosurface extraction technique implements the data coherency and multi-resolution approaches for the computation. Both approaches aim at reducing the amount of memory required by the system and thus improving the performance and interactivity of the system. For isosurface, these techniques help to reduce the number of geometric primitives defining the

isosurface. We introduce two modifications for the isosurface extraction process for MDV, namely, multi-resolution and data-coherency. While the multi-resolution technique processes the data at varying level of details, data coherency exploits the similarity among the datasets under investigation. This module is responsible for calculating and handling any change in the state of the isosurfaces or the techniques involved. On the other hand, the texture-based visualization uses 3D surface rendering and clipping for visualization of the data. The 3D surface texture mapping improves the rendering time and thus the interactivity of the application. These techniques are discussed in detail in the next chapter.

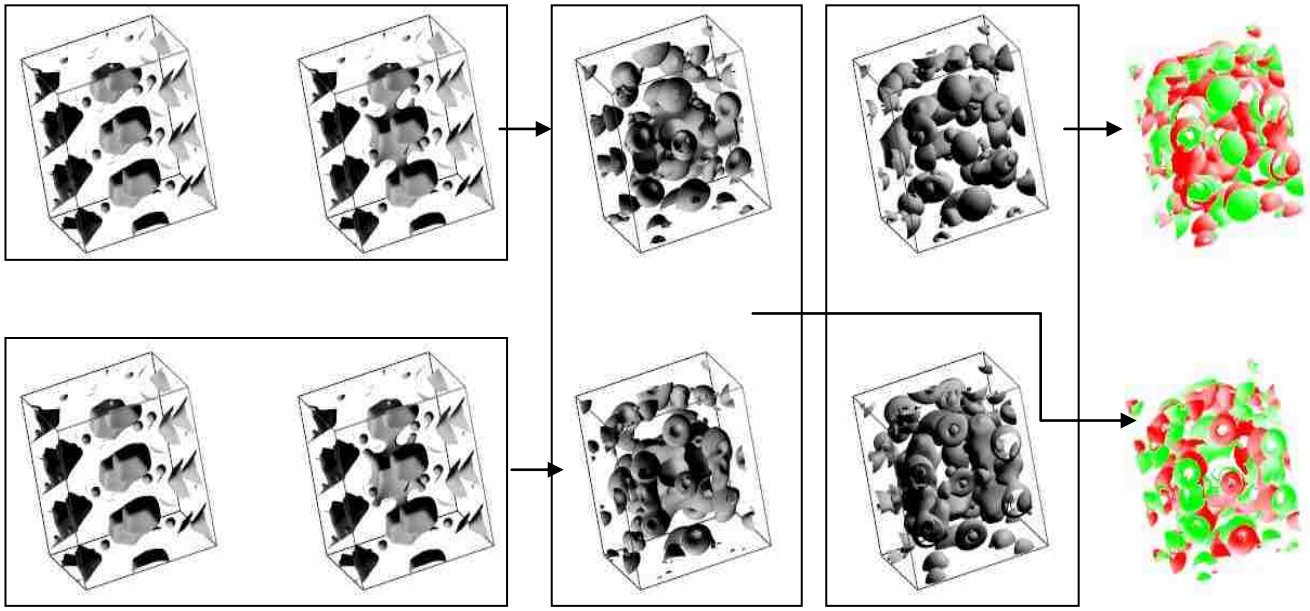


Figure 14: Isosurface difference using GPU. Arrows show the difference in the isosurface in the box. First difference is based on data and second difference is based on depth using GPU.

The visualization thread implements core components of the visualization algorithms. Once the visualization thread has generated the model, it places the generated primitives in a list shared between the visualization and the rendering module. Visualization thread is also responsible for sending the signal to the rendering thread. If more datasets are available, the visualization thread continues to compute the model for them according to the user-defined technique, otherwise it returns to the suspended state.

3.5 Rendering Module

The rendering module uses the frame buffer object or FBO's. Frame buffer objects are rectangular pieces of RAM in graphics card. FBO's can be used to render stuff off-screen (invisible to the user) that can be read back or used as textures. That means FBOs can also serve as input for rendering. It is important to note that a single FBO cannot act as input (texture) and output (render target) at the same time. It can either read from or written to it but not either simultaneously.

The rendering module is responsible for the simultaneous rendering the datasets. Upon receiving the signal from the visualization module, rendering module renders the dataset to the off-screen buffer, which is stored in the texture. This texture is rendered on the screen. Rendering to the off-screen buffer and storing them as separate texture, allows dividing the display screen. The major advantage of this approach is that we can selectively update the portion of the screen, depending upon the dataset being manipulated. Rendering to the off-screen buffer and capturing it as texture, allows only rendering the datasets that are changed and thus improving the performance and the interactivity of the system.

The rendering module provides the three display modes for interacting with the data. The "Normal", "Comparative" and "Analysis" mode allows the user to have different views. The "Normal View" provides an enlarged view of the dataset, allowing user to select the dataset and do local transformations of the dataset. In this view, the application performs traditional rendering of the dataset. It also provides scrollable snapshots of the datasets under consideration and allowing the user to select the dataset from this list for viewing. Since we use the off-screen renderer, we only need to map the texture. The texture needs to be updated only when there is some transformation that changes the view of the dataset. The "Comparative View" displays the dataset as a spreadsheet and allows the global transformations. Any transformation such as rotation or changing the isosurface threshold value affects all the datasets. In this view, all datasets are displayed and the allotted display area for each dataset becomes significantly less with increasing number of datasets. In such a scenario, "Normal View" proves to be useful. The views are updated as the

transformations are performed on the data. Finally, there is an "Analysis View" [Figure 14, Figure 15], which allows performing different types of analysis on multiple sets of data.

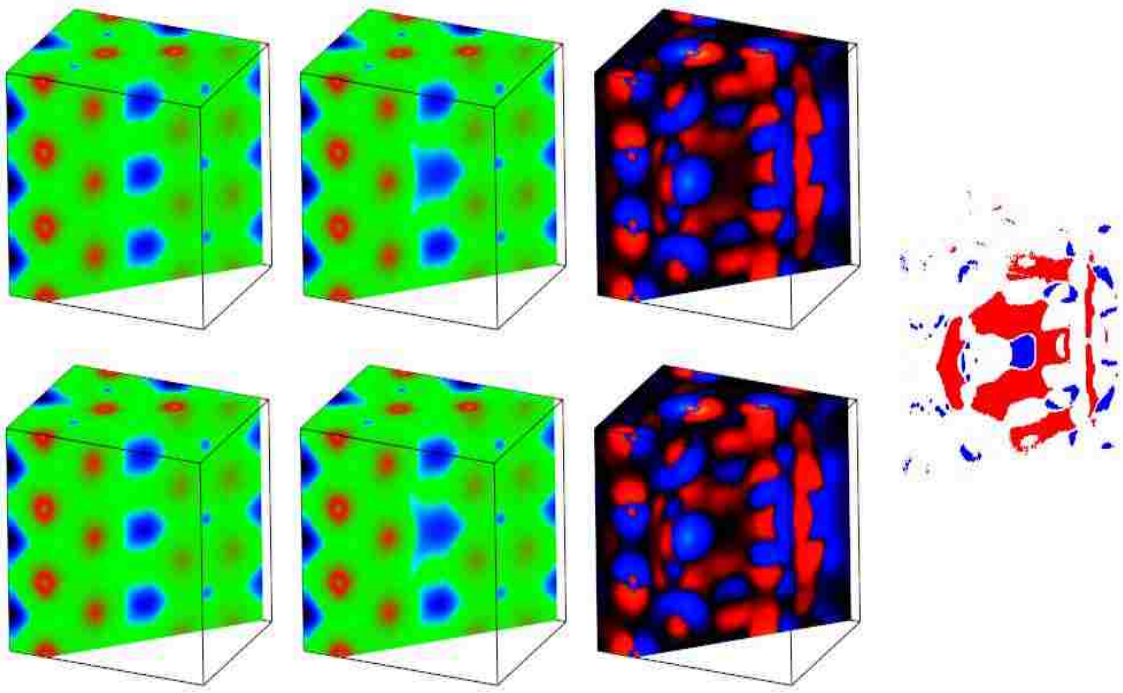


Figure 15: Texture based difference for the magnesium silicate datasets.

Analysis here means, for example, taking the difference in the isosurfaces extracted for different datasets, or performing binary operations. This can be extended or altered based upon the visualization technique being used or based upon the requirements of the datasets being used. This view allows the user to select two different datasets and perform the comparative analysis on them.

3.5.1 Rendering Time

This section presents the performance analysis of the rendering module. Since we are rendering to the off-screen buffer and using multiple threads, we discuss the best and worst case scenario. Let N be the total number of datasets, and T be the total time taken to

render them. Further, time to map the single texture is T_M and time needed to render the dataset to the off-screen buffer and capture is T_R

Best Case:

The best-case scenario is when only one dataset that is affected by the transformation. This can occur in the Normal mode or the Analysis mode provided by the visualization module. Therefore, the rendering time T is given by,

$$T = N * T_M + T_R \qquad \text{Equation 25}$$

In the analysis mode, we need to add time taken by the pixel shader to calculate the differences in the isosurfaces.

Average Case:

To look in to the average case scenario, let's consider the initial setup. If we compare the off screen rendering approach and direct rendering approach

	Using off-screen rendering	Direct rendering
For 1 dataset	$T = 1 * T_M + T_R$	$T = T_R$
When 2 datasets is available	$T = 2 * T_M + T_R$	$T = 2 * T_R$
When N datasets are available	$T = N * T_M + T_R$	$T = N * T_R$
Total initial setup	$T = N(\frac{N+1}{2} * T_M + T_R)$	$T = \frac{N * (N+1)}{2} * T_R$

This is theoretical estimation of the rendering time. However, in reality, the times will vary from the theoretical limits due to memory constraints.

Now consider changing the isovalue, for the average case. We get the similar results. Since, $T_M$is negligible, the rendering time in case of off-screen rendering is of the order of $O(N)$while in case of direct rendering it is of the $O(N^2)$ of T_R

Worst Case:

The worst case can occur in the comparative analysis mode, where the transformations can affect all the datasets. In this case the rendering time will be

$$T = N * (T_M + T_R) \qquad \text{Equation 26}$$

3.6 Results

This section presents the results of the techniques and methodology presented in the previous section. All the results were obtained for 256^3 data on Pentium IV 3.2 GHz, 1 GB RAM and 128 MB NVIDIA Geforce FX 5200 graphics card. For performance analysis, perfect magnesium crystal is used as the reference dataset while multiple copies of magnesium defect datasets are used for non-reference datasets. In this section, we look in to the results involving the comparison of the response time and the latency in case of multithreaded and single threaded versions. We have two cases, one [Figure 16] which involves data loading and isosurface generation, and the other [Figure 17] which involves changing the isovalue without involving data loading. We use the tolerance value of 0.002 and the isovalue of 0.02. Figure 16 shows the responsiveness and latency of the application for both single and multi-threaded cases. Latency refers to time when the isosurface for the first dataset is rendered. Responsiveness on the other hand refers to when all the datasets under consideration are rendered. In the case of multithreaded application, the latency is the time when first dataset is ready to be rendered whereas in the case of single threaded application, the latency is equal to the responsiveness since the rendering only starts after the isosurface for all the datasets is generated. The multithreaded application initially for up to 10 datasets is comparatively less responsive to compared to the single threaded application. When the number of datasets is such that available main memory for MDV is not sufficient and swapping starts taking place, the multithreaded application starts performing better. In the initial phase, the context switching between the threads slows down the application, while in the second phase, the context switch overhead is less as compared to swapping and we can see the advantage of using multithreads.

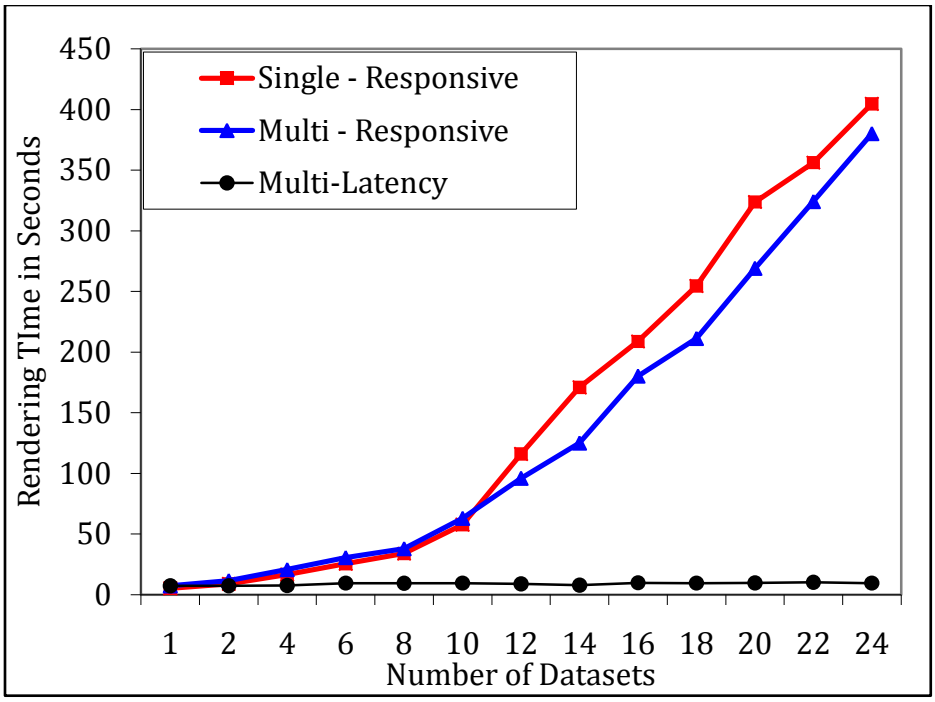


Figure 16: Responsiveness and Latency for single and multi thread scenario

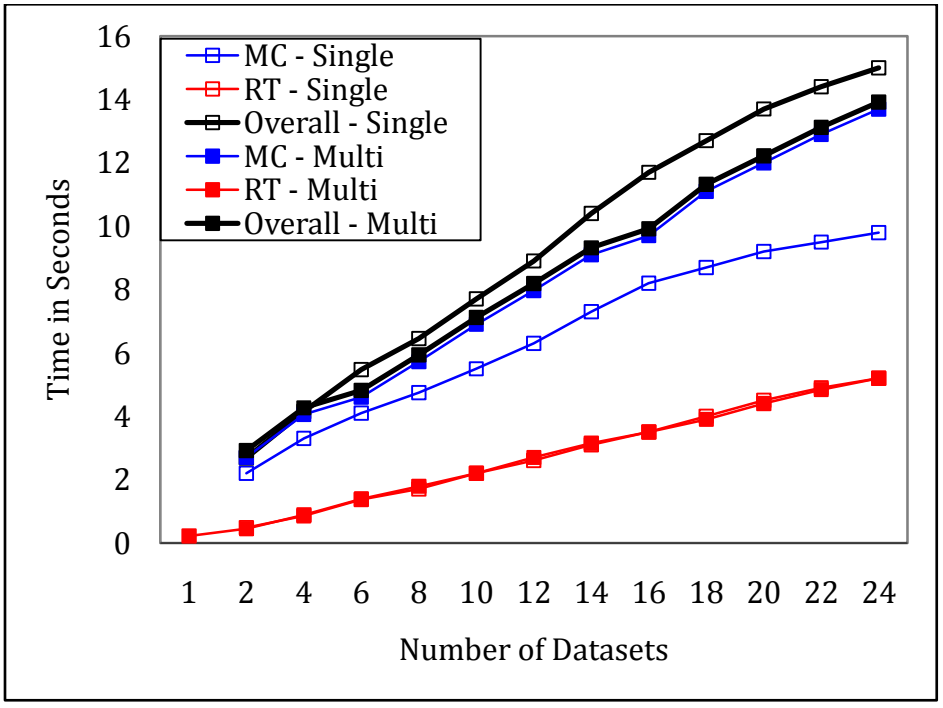


Figure 17: Single thread vs. Multi thread - Responsiveness (for marching cube and rendering time)

Figure 17 shows the second case discussed above where we perform the operation such as changing the isovalue. This only requires generating the geometry corresponding to the isovalue and rendering it. Rendering time is same for both the single and multithreaded cases. The isosurface extraction time in the case of single threaded scenario performs better as compared to multithreaded scenario. However, if we consider a combination of isosurface extraction and rendering, the multithreaded case gives a better performance.

3.7 Implementation

MDV Framework is implemented in C++, OpenGL, GLUT library and CG shading language. Implementation and performance analysis is done on Windows XP machine, which has 3.2 GHz Pentium IV processor, 1 GB RAM, NVIDIA GeForce FX 5200 graphics GPU with 128 MB graphics memory.

OpenGL is a software interface to graphics hardware. OpenGL (Open Graphics Library) is a standard specification defining a cross-language cross-platform API for writing applications that produce 3D computer graphics (and 2D computer graphics as well). The interface consists of over 250 different function calls, which can be used to draw complex three-dimensional scenes from simple primitives. OpenGL is designed as a streamlined, hardware-independent interface to be implemented on many different hardware platforms. OpenGL does not 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. Cg is shading language developed by NVIDIA. With the advancement in the GPU, architecture there has been recent trend to replace the fixed functionality pipeline with programmable pipeline. The pipelines provide the vertex and fragment shader to do some of the complex tasks. These shading languages provide programmer with the capability to express the processing that occurs in the graphics pipeline to some extent. Independently compatible units that are written in this language are called shaders. A program is a set of shaders, which are compiled and linked together.

4 ISOSURFACE EXTRACTION FOR MDV

4.1 MDV Framework and Isosurface Technique

In the previous chapter, we have presented an overview of the generalized MDV framework. There are three major components of the framework, namely data loading, visualization and rendering, which interact with the central engine and the user interface. Each of these components can vary according to the visualization technique and the nature of the data. Figure 18 shows the three components and the tasks related to the each component for the isosurface extraction-based visualization method. These tasks can also vary for load balancing among threads and according to the nature of data and desired results.

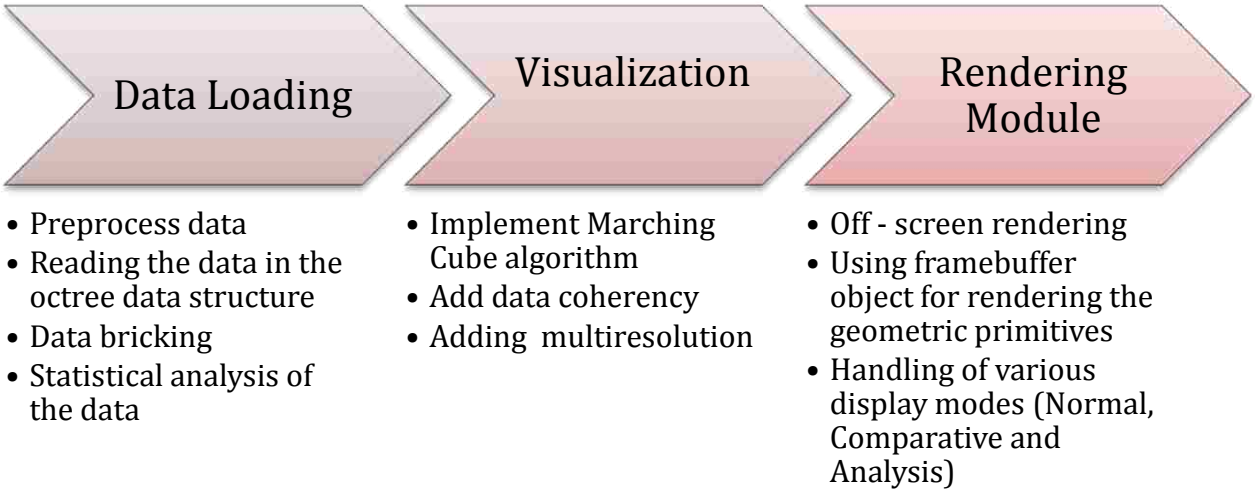


Figure 18: MDV Framework for isosurface extraction

4.2 Data Structure

The goal here is to load the data efficiently and store it in the memory in such a way that it helps up in speeding up the visualization process. This is entirely dependent on the

visualization technique we used. There are different forms in which data can be loaded to improve the performance like Octree, KD-Trees, quad trees, link list, voronoi diagrams and others [69, 70]. Selection of the representation of the data will depend upon the type of dataset whether the data is 2-dimensional or 3-dimensional or the structure in the dataset is single body or it is distributed and visualization technique. We consider the octree representation of the dataset. As mentioned above data loading can be a bottleneck as the size of data increases.

In case of isosurface, data loading module task includes but are not limited to preprocessing the data, reading the data and doing statistical analysis of the data. The preprocessing step can include interpolating the data, data messaging to remove the unwanted information from the file for faster loading. The core component of the data-loading module is responsible for reading the dataset and generating an octree. During the process of octree generation, the data-loading module also extracts the min-max value for each octant and its children.

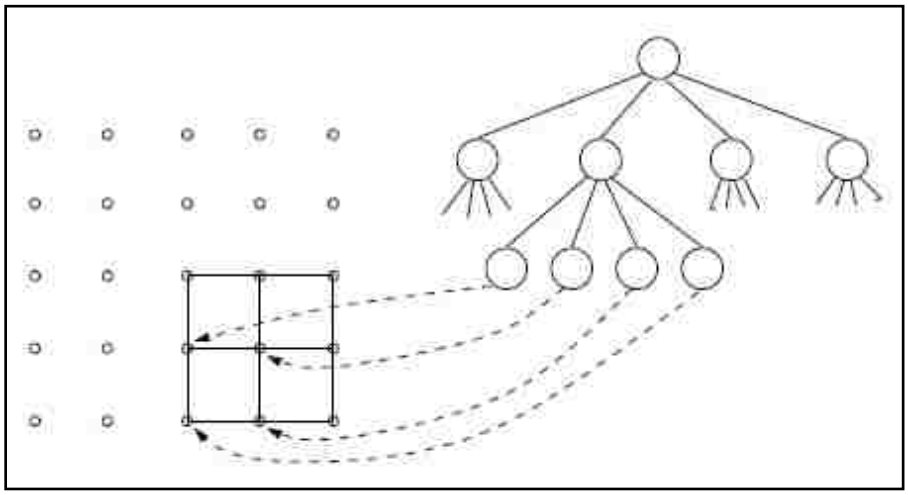


Figure 19:Quadtree for calculating isosurfaces [71]

Octree [Figure 19, Figure 20] is a hierarchical data structure, which divides the space recursively in to eight equal subspaces [49]. Since we are using isosurface technique to

visualize the data, we opt to use octree as it offers an efficient way to speed up computation of isosurfaces [71].

During the construction of the octree, some nodes are labeled as leaf nodes and others as parent nodes. Leaf nodes of the tree are the ones, which are at the lowest, level and do not have the children, while parent nodes are the ones, which have children and only hold min-max value of the children under them. The number of leaf nodes is determined by the level at which the tree is pruned. Data loading module is also responsible for data bricking and thus creating multiple smaller datasets from the original data that enables better utilization of the memory.

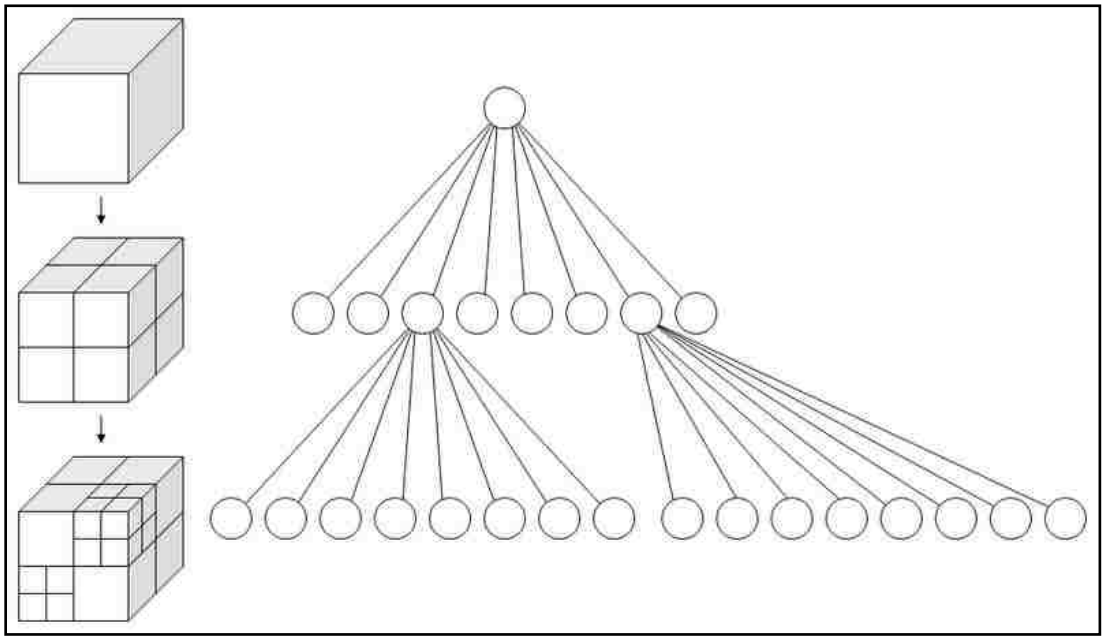


Figure 20: Octree for isosurface construction [72]

Each of the parent nodes holds the min-max value. During the process of isosurface extraction, the parent node is checked for the isovalue. Only voxel for which the isovalue is between the min-max values pair in the parent node will contribute to the final isosurface. Thus using an octree data structure can speed up the process of finding the cells that contribute to the isosurface.

4.3 Rendering Primitives

For isosurface extraction process, the datasets under consideration are loaded into the memory and then processed with the Marching Cubes (MC) algorithm. The generated geometrical primitives representing the desired isosurfaces are rendered using the graphics hardware [65].

For rendering, we consider two types of primitives, polygons and points. The basic isosurface extraction algorithm remains the same in both the cases. In case of polygons, the vertex position and connectivity information need to be maintained. Each cell containing the isosurface usually generates more than one polygon so the number of triangles can be larger than the number of cells. In the case of MDV, using the points as rendering primitives can provide following advantages:

- The memory requirement in case of points is significantly lower as compared to using triangles. There is no need to store the connectivity information. Secondly, the number of points for an isosurface will be equal to the number of cells in the volume contributing to the isosurface.
- In case of MDV, as the number of datasets increases, the number of effective pixels covered by a cell reduces. There are two cases to consider depending upon the cell size and the pixel size. Let C_{size} and P_{size} denote the cell size and pixel size respectively.

  Case 1: $C_{size} \leq P_{size}$

  Points will display exact isosurface as polygons, as all polygons in a cell will eventually converge to a pixel.

  Case 2: $C_{size} > P_{size}$

  Points will not display exact isosurface as polygons. We will see the holes in the isosurface. To overcome this problem, we can use the hybrid rendering approach using points and polygons. The other way around will be to use splats. While using splats we will need to calculate the effect of neighboring points and calculate the variance matrix. This variance matrix combined with a spherical primitive can be used to fill the gaps.

- Thirdly, when using points the amount of computation reduces as we do not have to calculate the edge intersection of cell with the isosurface and we avoid calculating the triangle.

Points give better performance by allowing loading of more number of datasets. The normal calculation is done using the central difference formula. Although the points have advantages, there is one major drawback that every voxel should exactly match the pixel on the screen. If a voxel covers more than one pixel, then the data need to be interpolated to get the continuous surface, which means more points to be stored and processed. Therefore, in the isosurface-based techniques mentioned in the next sections, triangles are used as rendering primitives. Further, the texture based isosurface extraction discussed in the section 5.7 provides the advantages of the point based isosurface extraction along with the 3D interpolation.

4.4 Visualization Techniques

For multiple datasets, isosurface extraction is performed using two schemes, namely, *All-In-Memory (AIM)* and *Only-Polygon-In-Memory (OPIM)* [16]. These methods differ in the methodologies as to how the datasets are stored. In this section we will present both the schemes.

For performance analysis for the AIM and OPIM schemes, three timings, data loading time (T_L), geometry extraction time (T_G) and rendering time (T_R) are considered. The operation being performed and the approach being used determine what timing effect the frame rates. If the basic operations such as rotation, translation, scaling, shading and selection are performed on multiple isosurfaces, which are displayed on the screen, then only rendering time (T_R) is considered, which should be small enough (in fractions of a second) to give an acceptable frame rate. If isovalue is being changed then both geometry extraction (T_G) and rendering time (T_R) are considered. A change of isovalue uses the memory bus and cache memory during polygon generation whereas the graphics bus and memory during the subsequent polygon rendering. The relevant timing involves the both rendering (T_R) and generation (T_G) times, which together should give acceptable response times (in the order

of seconds). Finally, if new dataset is loaded then all three timings, loading (T_L), geometry extraction (T_G) and rendering time (T_R) are relevant. Acceptable values for T_L can range from several seconds to a few minutes. The I/O overhead can be a bottleneck even at the interaction levels 1 and 2 if the memory space is not enough to hold the entire datasets.

4.4.1 All-In-Memory (AIM) Method

As the name stands for, the all-in-memory (AIM) method requires all datasets under consideration and the corresponding lists of polygons always to be present in the memory. Due to the availability of the input data, the isosurface can be extracted in a flexible manner and the isovalue can be changed interactively. The AIM technique works as follows (Figure 21):

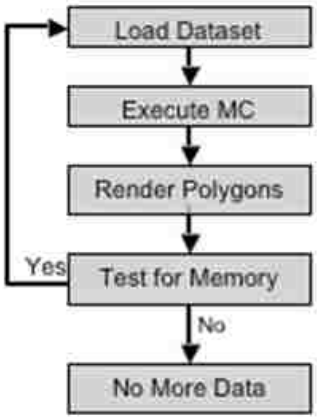


Figure 21: AIM Scheme

Step 1: Load the first dataset into the memory via, for example, data streaming process.

Step 2: Execute the MC algorithm on the volume data to generate a list of polygons (triangles), which represent the isosurface(s), for an isovalue.

Step 3: Pass the polygon data to the graphics card, which renders them in the form of 3D surface(s). The result is the desired isosurface(s) on display.

Step 4: Both the original volume and generated geometric data are held in the memory.

Estimate the total memory space used. Steps 1, 2 and 3 are repeated for the next dataset until no more datasets can be processed due to memory limitation. Let us assume that we have already processed N datasets and are now processing the $(N+1)^{th}$ dataset. The total memory space required by $N+1$ datasets can be expressed as

$$M_{N+1} = \left(1 + \frac{1}{N}\right) \sum_{i=1}^{N} \left(D_i + P_i(s)\right) \qquad \text{Equation 27}$$

Where D_i and $P_i(s)$ are the memory used by the volume data and generated geometric data (for an isovalue s) for the i^{th} dataset. Their values for $i = 1$ to N are known by the time when the $(N+1)^{th}$ dataset is being considered. The fraction $1/N$ represents the memory requirement for the volume and geometric data for the $(N+1)^{th}$ dataset, which are simply the averages taken over already processed N datasets. If $M_{N+1}$is smaller than the system's total virtual memory available for this application, we can process the $(N+1)^{th}$ dataset.

For the AIM case, to explore the complete set of data in details, the user needs to interact with the MDV process at above-mentioned three different levels. The timings for the level 1, 2 and 3 in case of AIM are, respectively, given by, $T_1 = T_R$; $T_2 = T_R + T_G$; and $T_2 = T_R + T_G + T_L$. Note that since the volume datasets are always in the memory, a change of isovalue (the level 2) does not require reloading of the data. The overall storage requirement in the AIM method can be overwhelming.

4.4.2 Only-Polygons-in-Memory (OPIM) Method

The only-polygons-in-memory method (OPIM) reduces the memory usage by keeping the scalar data in the memory until the MC algorithm is executed, after which the dataset is removed from the memory. Thus, polygonal geometry generated by MC algorithm is available. The OPIM scheme proceeds as follows [Figure 22]:

Step 1: Load one dataset at a time.

Step 2: Run the MC algorithm for a given isovalue and then delete the volume data from the memory. The generated polygon data are subsequently rendered.

Step 3: Repeat the process until all datasets are processed so that total geometric data can still fit in the memory. In effect, only one volume dataset can be the memory-resident at a time. We can estimate the total memory space required by N+1 datasets before processing the $(N+1)^{th}$ dataset using

$$M_{N+1} = \frac{1}{N}\sum_{i=1}^{N} D_i + \left(1 + \frac{1}{N}\right)\sum_{i=1}^{N} P_i(s) \qquad \text{Equation 28}$$

The first-term represents the memory needed to hold the $(N+1)^{th}$ dataset in the memory until its isosurface is extracted. The second term represents the total memory needed by the N+1 isosurfaces. If M_{N+1} is smaller than the system's total virtual memory, we can process the $(N+1)^{th}$ dataset, otherwise not.

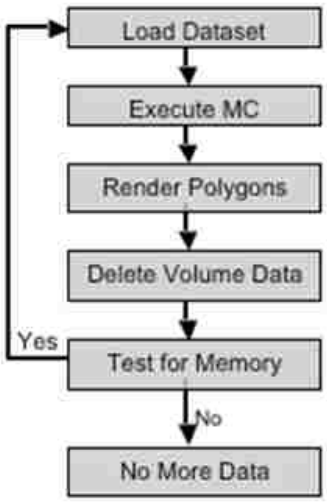


Figure 22: OPIM Scheme

Thus, only one volume dataset can be memory-resident at a time, thereby allowing more memory to store more polygon data. The associated timing factors differ from those of the AIM method. For the interaction level 1, only the rendering time is relevant:$T_1 = T_R$. However, for the interaction levels 2 and 3, both changing the isovalue and visualizing new

data involve the complete cycle consisting of data loading, MC execution and polygon rendering. Thus, the relevant timing is given by, $T_2 = T_3 = T_R + T_G + T_L$

4.4.3 Performance Analysis

Performance Analysis Using Polygons

How the three timing factors, T_R , T_G and T_L vary with the number of the datasets, *N* depend on several factors. First, the size (effective surface area) of the isosurface often varies with the isovalue [Figure 23]. The area is relatively large when not only the isosurface spreads over much of the volume but also when it occurs in the form of disconnected or weakly connected 3D surfaces. Second, the effective image area determines the number of the pixels that need to be processed. This is relevant because the multiple datasets often need to be rendered on multiple viewports preferably on the largest window. Third, the average size of the volume data under consideration makes difference at all stages of processing. Larger data size means a fewer independent datasets that can be visualized simultaneously. Finally, the hardware specifications such as CPU, cache, memory, bus bandwidth, graphics card, etc. become crucial as *N* increases.

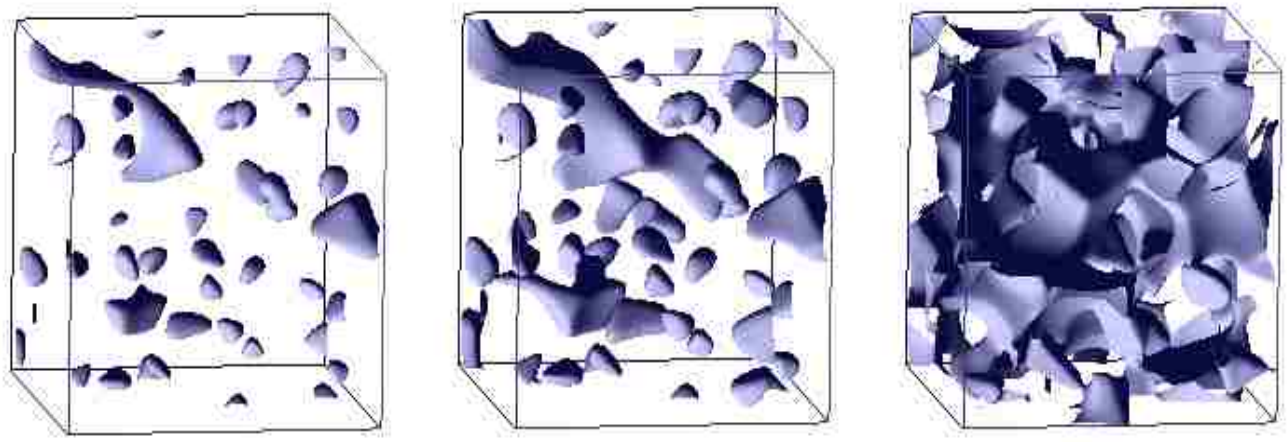


Figure 23: Isosurface for isovalue = 0.02, 0.04, and 0.08 (left to right)[16]

Here, for performance analysis of the AIM and OPIM methods in case of MDV, we consider the charge density data of size 256^3, which represents today's common moderate-sized scalar regular volume data. One dataset requires about 64 MB memory space. A window display size of 1024^2 was considered. We have calculated only T_G and T_R as a function of N on Windows XP machine, which has 3.2 GHz Pentium IV processor, 1 GB RAM, NVIDIA GeForce FX 5200 graphics GPU with 128 MB graphics memory.

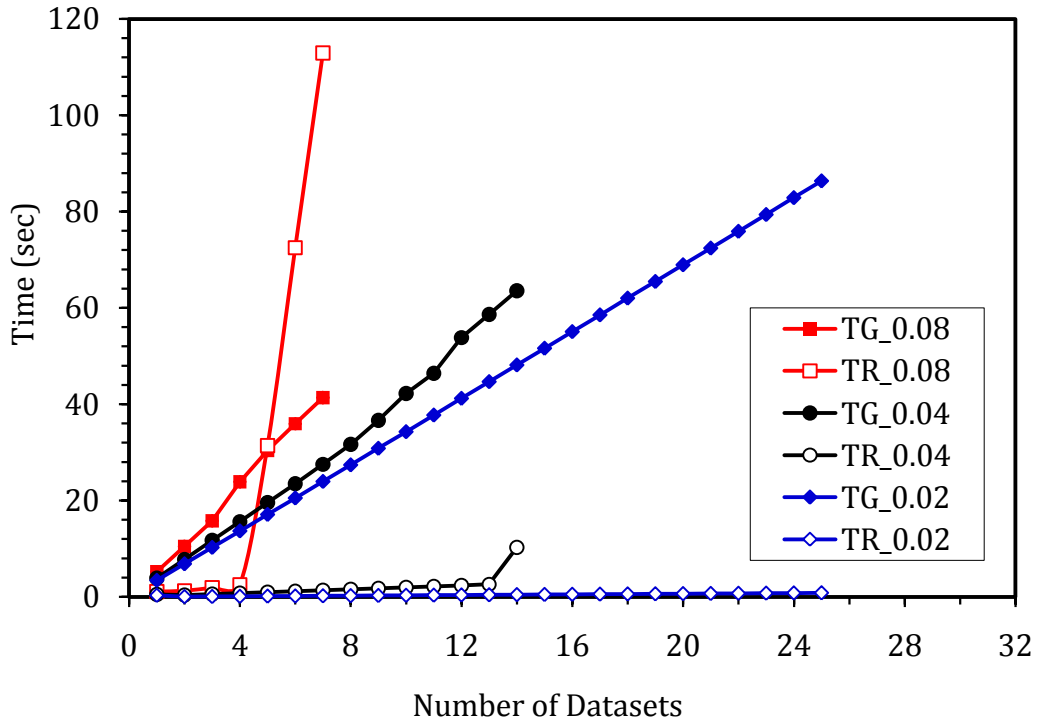


Figure 24: AIM polygon generation (T_G) and rendering (T_R) times for isovalue = 0.02, 0.04 and 0.08 [16]

With the AIM method for isovalue = 0.04, with increasing N, T_G increases linearly up to 8 datasets and thereafter relatively more rapidly [Figure 24]. The maximum number (N_{MAX}) of datasets that can be visualized is 14. On the other hand, T_R increases linearly and smoothly with N up to 13 datasets, followed by a rapid increase. The shape of the *time-N* curve and the value of N_{MAX} are sensitive to the structure and the size of the isosurface. For

isovalue = 0.02, a higher N_{MAX} of 25 is obtained with both T_G and T_R varying linearly with N. On the other hand, for *isovalue* = 0.08, the times are relatively large and N_{MAX} = 7. There are clearly two regions with small and large time-N slopes for T_G and T_R. A relative rapid increase in the high-N region can be associated to the limited memory, which results in the swapping between the main and virtual memory.

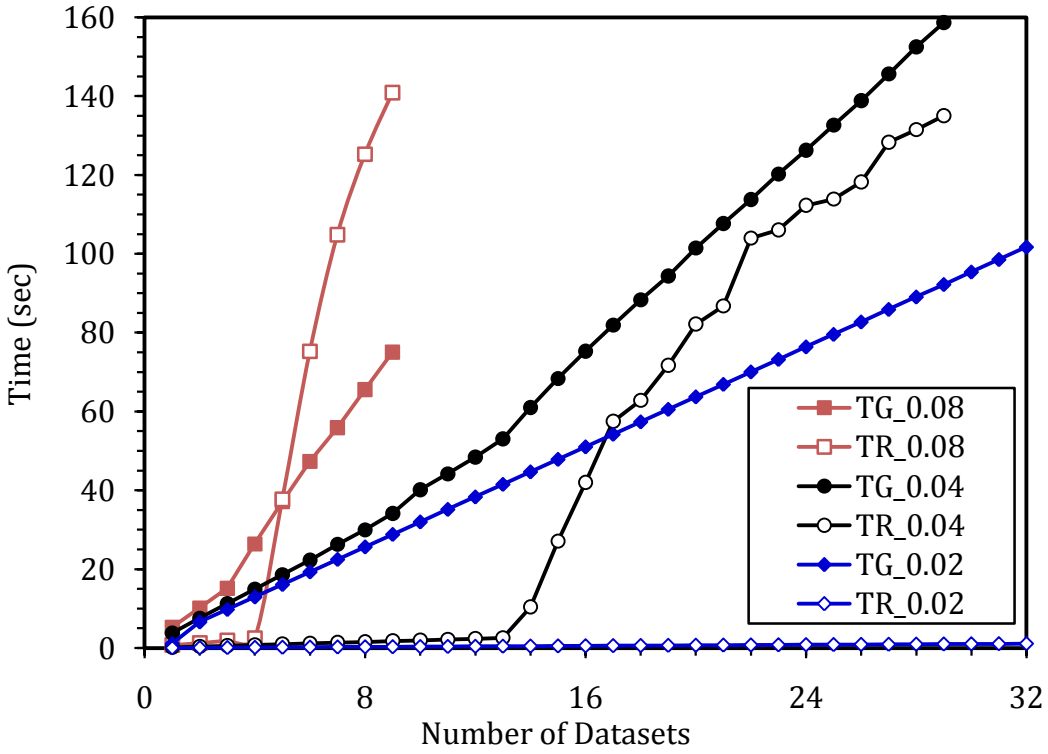


Figure 25: OPIM polygon generation (T_G) and rendering (T_R) times for isovalue = 0.02, 0.04, 0.08 [16]

With the OPIM method for *isovalue* = 0.04, both T_G and T_R show a non-uniform variation with N [Figure 25]. In both cases, the *time-N* slope shows that a clear transition from the low-N region to the high-N region occurs just after N = 13. The rapid increase in the high-N region can be associated with the limited video memory for the polygon data, which causes substantial swapping between the graphics card and main memory in addition to the swapping between the main and secondary memory. No transition occurs for *isovalue* =

0.02 for which N_{MAX} = 64 whereas the transition shifts to a smaller *N* for *isovalue* = 0.08 for which N_{MAX} = 9. Compared to the AIM method, which provides a high flexibility, the OPIM method improves N_{MAX} so the latter is more useful if a larger number of sets of data need to be visualized simultaneously. However, every time *isovalue* is changed, all volume data need to be reloaded.

4.4.4 Performance Analysis (Points vs. Polygons)

All-In Memory Method

With the AIM method using polygons, for *isovalue* = 0.04, with increasing *N*, T_G increases linearly up to 8 datasets and thereafter much more rapidly [Figure 26]. The maximum number (N_{MAX}) of datasets that can be visualized is 19. On the other hand, T_R increases linearly with *N* up to 13 datasets followed by a rapid increase. Thus, there are clearly two regions, one with small and the other with large *time-N* slopes. The shape of the *time-N* curve and the value of N_{MAX} are sensitive to the size of the isosurface. For *isovalue* = 0.02, a higher N_{MAX} of 42 is obtained with a non-linear increase in T_G appearing at 19 datasets while a linear increase in T_R throughout. On the other hand, for *isovalue* = 0.08, the times are larger, N_{MAX} takes a smaller value of 8, and the non-linear trend begins at a smaller number of 4 datasets. The AIM method performs better with points than polygons by giving smaller time values and pushing the nonlinearity region and the value of N_{MAX} towards a higher *N*. For instance, T_R and T_G for *isovalue* = 0.04 and *N* = 8 are, respectively, 0.28 and 20.8 seconds with points, compared to 1.59 and 34.5 seconds with polygons. Unlike polygons, with points only T_G starts to show the nonlinearity at 15 datasets and continues up to N_{MAX} = 34. Similar improvements are observed for *isovalue* = 0.02 and 0.08 for which N_{MAX} = 50 and 18, respectively. The value of N_{MAX} is determined by the system's available virtual memory space for the application, as discussed earlier. The observed non-linear behavior is associated to the system's limited physical memory, which results in substantial swapping between the main and secondary memory once the application's requirement exceeds the available physical memory. However, it also depends on the memory access pattern. For instance, the absence of non-linearity in T_R for *isovalue* = 0.02 and 0.04 means

that the amount of the geometric data can still fit into the physical memory for the repeated rendering.

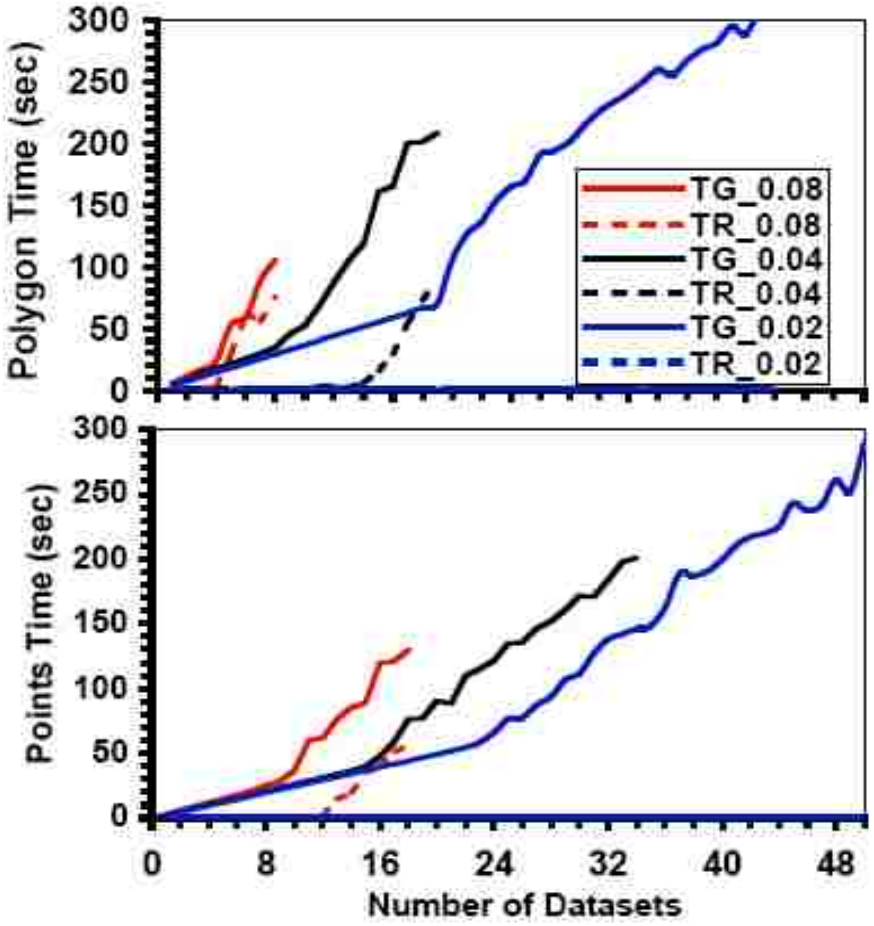


Figure 26: AIM scheme

Only-Polygons- or Points-In-Memory Method

For isovalue = 0.04, both T_R and T_G show a nonlinear variation with N (Figure 27): The time-N slope shows a transition from the low-N region to the high-N region around 14 and 38 datasets for polygons and points, respectively. The rapid increase in the high-N region can be associated with the limited video and main memory for the geometric data, which causes the swapping between the main and secondary memory in addition to swapping

between the graphics card and main memory. The values of NMAX are 30 and, at least, 64 for polygons and points, respectively. No transition occurs for isovalue = 0.02 for which NMAX is beyond 64. The transition shifts to a smaller N for isovalue = 0.08 for which NMAX is 9 and 27 with polygons and points, respectively.

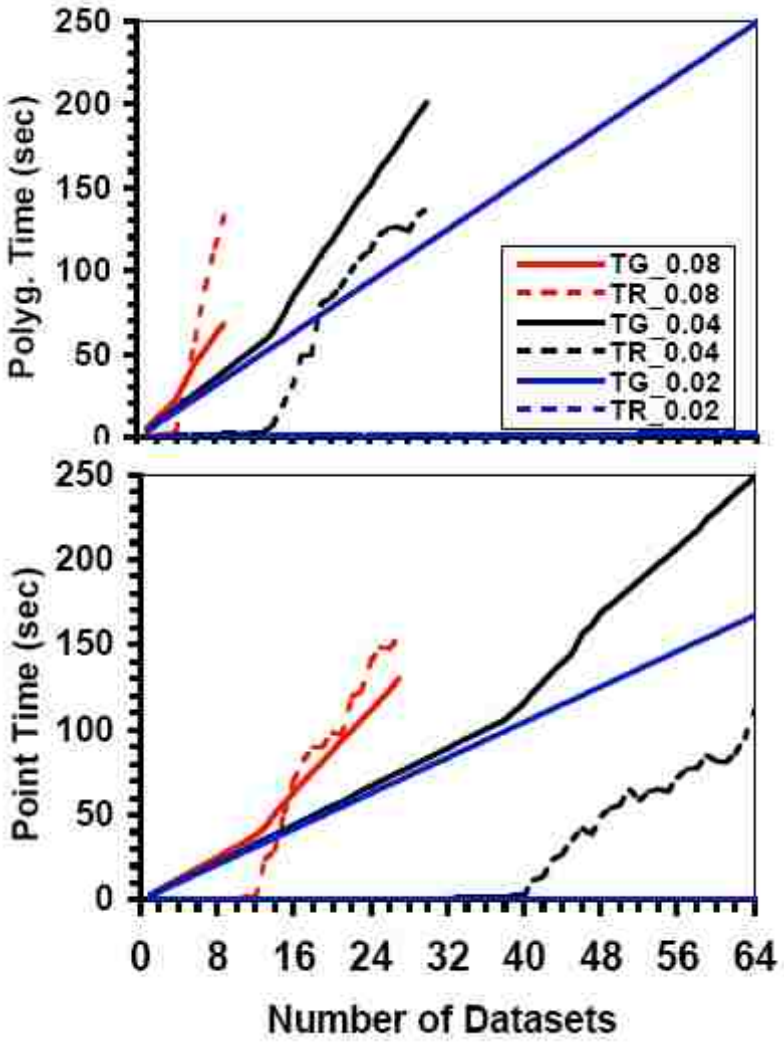


Figure 27: OPIM scheme

From the results shown in Figure 26 and Figure 27, we can see the advantages of using the points over the polygons. Both the results were obtained for same datasets, only the rendering primitive is varied. The points allow more number of datasets to be visualized

simultaneously and better interactivity for same number of datasets for two cases. This is true for both the AIM and OPIM.

4.5 Improving Isosurface Extraction for MDV

4.5.1 Data Coherency

MDV as defined earlier is the visualization of multiple datasets for an in-depth understanding of a given problem under consideration. Since these datasets are related to each other, it is safe to assume that there will be some similarities in the structure of the isosurfaces generated and therefore the differences are confined to some specific regions. Data coherency technique exploits the relationship between the datasets under consideration to identify the regions that will make similar contributions to the corresponding isosurfaces for datasets. One or more dataset are considered as the reference dataset(s). Other datasets (non-reference) use the polygon structure obtained for these reference dataset(s) to approximate their isosurface. For the similar regions in the isosurface of datasets under consideration only one set of polygons is generated.

Figure 28 shows the approximation of the electronic density isosurface of defective magnesium-silicate crystal using the perfect crystal data as a reference dataset. The density distortion due to a defect site in non-reference datasets is localized so the isosurfaces in a defective crystal are different from those of the perfect crystal primarily in the immediate neighbourhood of the defect site. In Figure 28 (a) is the perfect crystal used as the reference dataset. (b) and (c), are Mg and migrating Mg defects, respectively. The lighter areas in (b) and (c) correspond to the approximated isosurface portions while dark areas correspond to the isosurface portions that are directly derived from the data and differ from the corresponding regions in the reference data.

In normal MDV scenario, we extract the isosurfaces corresponding to a given isovalue for all the datasets under consideration independent of each other. Thus, the similar regions between a dataset and the reference dataset, which can be approximated from the reference dataset, are also generated. Generating the complete isosurface gives an actual

isosurface for each dataset, but it results in large memory requirement. Data coherency is defined as a measure of similarity among different datasets. Our data coherency model exploits the similarity in the datasets to extract the isosurface only once for all the similar regions between the reference and non-reference datasets. The generated isosurface for a non-reference dataset is composed of two types of areas; the first type of areas directly extracted from the regions of data that differ from the reference volume and other areas approximated from the reference volume via data coherency. Since, considerable amount of isosurface in the non-reference dataset is similar to the reference dataset; data coherency lowers the memory requirement. It also reduces the time required to extract the isosurface.

Data Coherency Algorithm:

The data coherency algorithm described here achieves the computational and storage benefits by reducing the total number of polygons representing the isosurfaces from multiple datasets by avoiding direct processing of significant large portions of each non-reference volume. The Marching Cubes algorithm visits each voxel in the volume under consideration to find the geometry corresponding to the isosurface under question. Data coherency can be applied in two ways; a) at the level of the geometry (triangles) or b) at the level of voxels. Comparison at the level of geometry is computationally very expensive, as triangles need to be extracted for each data volume for the comparison.

Comparison at the level of voxels seems to be an optimal approach from both the computational and storage points of view. We can assume that similarity at the voxel level means the similarity in the topology of the geometry of the isosurfaces. It is worth mentioning that if the datasets are not of the same size, voxel-based comparison does not work and topological comparison is required. Finding the coherency between the given volume and the reference volume is computationally expensive. The MC algorithm needs to visit each cell to decide whether the geometry needs to be extracted or not. In order to overcome the problem with voxel-by-voxel comparison, our data coherency approach uses an octree data structure so that comparison can be performed at coarser level of blocks representing a large number of voxels.

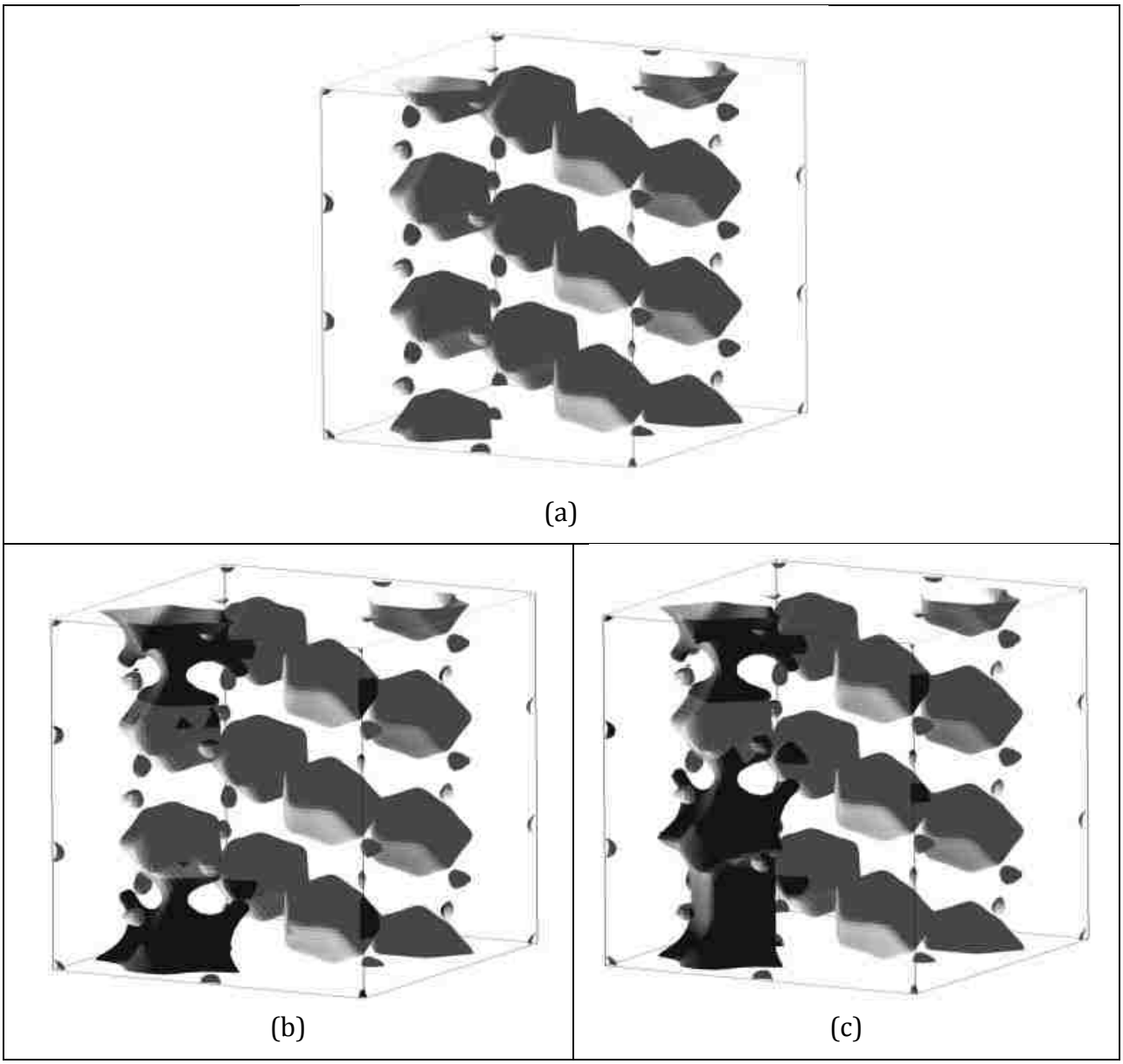


Figure 28: Visualization of electronic charge density using the data coherency technique. (a) Perfect crystal (reference data); (b) and (c) Non-reference data. Light region from reference dataset; dark region is directly extracted.

Octree-Based Data Coherency:

Hierarchical subdivision of the volume data using octree [73, 74] has previously been used with Marching Cubes algorithm to speed up the search process for cells contributing to the isosurface. Every node in the octree data structure stores the index and the minimum and –

maximum data value among all the cells below it. The children nodes also store the index and the minimum-maximum for the cells below it and the process is repeated until we reach at the level of cell. To optimize the memory usage we prune the tree at the level where block size is 8^3 [73]. Considering the dataset of size 256^3, the root node is at level 0. The subdivision in the octree is done until the node represents the block size is of 8^3, these nodes are at level 5.

We maintain a single octree for multiple datasets under consideration. Since all datasets are assumed to be of the same size, each node in the octree for MDV scenario will have a single value of index corresponding to all datasets; however, the minimum-maximum value pair will be different for different dataset. To implement the data coherency technique, the nodes at some user-defined level in the octree are treated as special nodes, referred as *dc_level* nodes. For each dataset, every *dc_level* node stores an additional user defined value (*V*) such as the average or variance of scalar data taking into account all cells under that node. All *dc_level* nodes are checked for data coherency condition (*dc_condition*). The *dc_condition* is of the form *V <= tolerance*, where tolerance tells the percentage of error allowed. The check for *dc_condition* determines whether the triangles for the cells under *dc_level* node need to be extracted or not. If not, the polygon data for the corresponding node in the reference data volume can be used.

In order to keep track of the *dc_condition*, we maintain a bit vector for each dataset under consideration. The number of bits in each bit vector is equal to the number *dc_level* nodes. This bit vector is used during the rendering process. For the reference dataset, all the bits in the bit vector are unset and the polygons are generated from all nodes. For all other datasets, the *dc_condition* is checked for all *dc_level* nodes. The bit in the vector corresponding to the *dc_level* node is set or unset depending upon whether the *dc_condition* is satisfied or not. For a dataset, if the values stored in a *dc_level* node do not satisfy the *dc_condition*, then the geometry is generated for the portion of the dataset represented by this node. The triangles corresponding to the *dc_level* node are tagged together in order to retrieve them efficiently without traversing the complete list of triangles. The number of *dc_level* nodes labels the list of triangles. During rendering, the bit vector corresponding to dataset is checked, and if the value in the bit vector is set then the corresponding set of

triangles are retrieved from the reference set otherwise triangles generated directly for that portion of the dataset are rendered. This process is repeated for all the datasets in the tree.

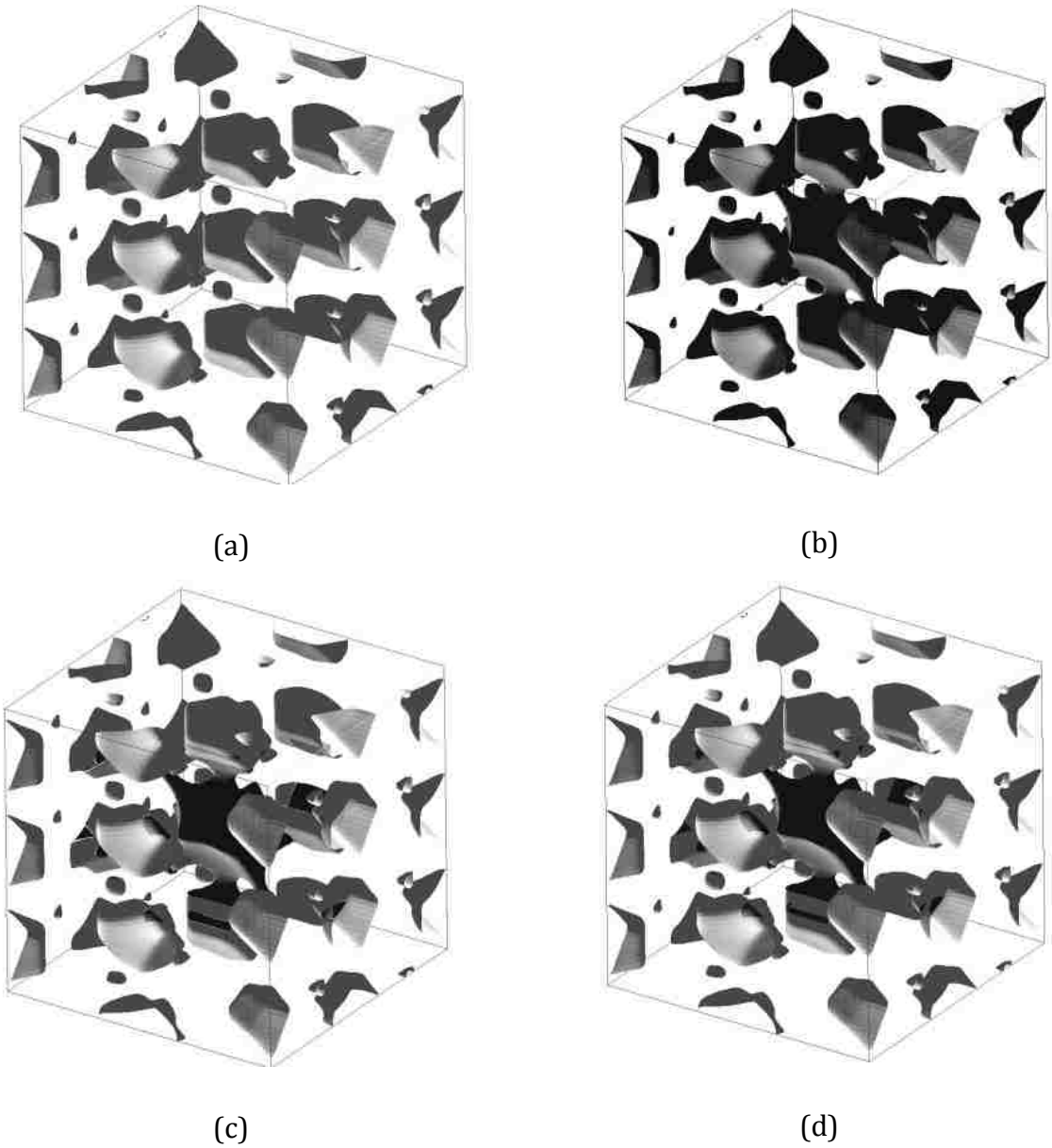


Figure 29: Overlapping for eliminating crack in the dataset. (a) Reference isosurface; (b) Non-reference isosurface; (c) Approximated isosurface with cracks; (d) Approximated isosurface without crack

The performance and effectiveness of the algorithm are determined by two factors; data coherency condition (*dc_condition*) and block size (determined by *dc_level*). The *dc_condition* depend on the tolerance factor and other parameter under consideration used for comparison, which can be one or more of a) min-max pair b) min-max-avg c) min-max-variance. The selected parameter from a non-reference dataset is compared with that from the reference dataset to estimate the difference. If the difference in the parameter value between two datasets is less than the tolerance factor, the isosurface can be approximated by the reference one otherwise a new isosurface is generated. Second factor that determines the performance is the block size, which, in tune, depends on the number of the *dc_level* nodes. In octree, the root node is at level 0, its children are at level 1, and their children are at level 2 and so on.

By block size we mean the portion of the volume of the dataset represented by the *dc_level node*. If *dc_condition* is unsatisfied then all the cells (voxels) under that node are processed for isosurface and the generated triangles are grouped together. Data coherency technique thus exploits the similarity in the datasets to speed up the isosurface extraction process and reduce the storage requirement. However, the extracted isosurfaces may be inaccurate and also suffer from distracting artefacts including cracks.

The crack problem in case of isosurfaces was mentioned [74-77] earlier for meshes with varying resolution. There are several known solutions for patching crack. Shekhar's [74] method fixes cracks by matching the isosurface generated from the higher-resolution cells with the isosurface generated from the lower-resolution cell. Shu's [75] used a polygon of the same shape as the crack to patch cracks in isosurfaces. Weber [76] proposed a solution for crack fixing for cell-centered data. Fang [77] proposed a method to use pyramidal regions for avoiding cracks. Westermann [50] used "triangle fans" to match the isosurface from the lower-resolution cells to the isosurface of the higher-resolution cells.

The goal of the data coherency approach is to minimize the computation. We follow a simple approach to avoid the discontinuities of approximated isosurfaces by minimizing the extra computational overhead. In our approach, cracks are avoided by overlapping the isosurface derived from the reference data and that directly derived from non-reference

data across the boundary regions. The overlapping is done at the nodes defined by the *dc_level.*

Figure 29 uses the overlapping technique to patch the cracks. The number of extra triangles generated is given in Table 1. The ratio of extra triangles to the number of original triangles corresponds to the extra number of cells being processed. This is because these cells have a high potential of having an isosurface as they are at the boundary of the isosurface already present in the reference dataset or in the non-reference dataset. Table 1 also shows the reduction in the number of triangles achieved by data coherency technique. For the 256^3 data considered in this study, if we were not using data coherency technique, the number of triangles generated for non-reference dataset would be approximately 550K. For offset 0 i.e. there is no overlapping, data coherency reduces this number to about 100K.

Table 1: Number of triangles with overlapping

Overlapping Offset	# of triangle in reference dataset	# of triangle in non reference dataset
0	522064	112240
1	735196	157557
2	981887	209602
4	1549793	330863

Offset in Table 1 represents the increase (in terms of cells) in the size of the block (determined by data coherency node level) along each direction for overlapping. Offset of zero means no extra cells (i.e., no overlapping) are considered. The difference in number of triangle in offset 1 and offset 0 for block size of 16^3 cells is 40%, which is approximately equal to the number of the extra voxels being processed. Considering these extra triangles, the given algorithm needs to be modified to optimize storage. Initially, the region along the non-reference dataset (used for direct isosurface extraction) is allowed to grow to patch

the crack. As the number of datasets (*N*) is increased, the cumulative sum of the extra triangles surpasses the number of extra triangle generated if only reference dataset is allowed to grow. For this value of *N*, we only allow the reference dataset to grow, thus optimizing the number of extra triangles that needs to be generated to patch the crack.

Results for Data Coherency

Our analysis involves rendering and polygon generation time for various tolerance factors and various block sizes (determined by *dc_level*). The results were obtained for 256^3 data on Pentium 4 3.2 GHz, 1 GB RAM and 128 MB Geforce FX 5200 graphics card. In our analysis, electron density dataset for the perfect crystal was used as the reference dataset while multiple copies of dataset for a defect crystal were used as the non-reference datasets.

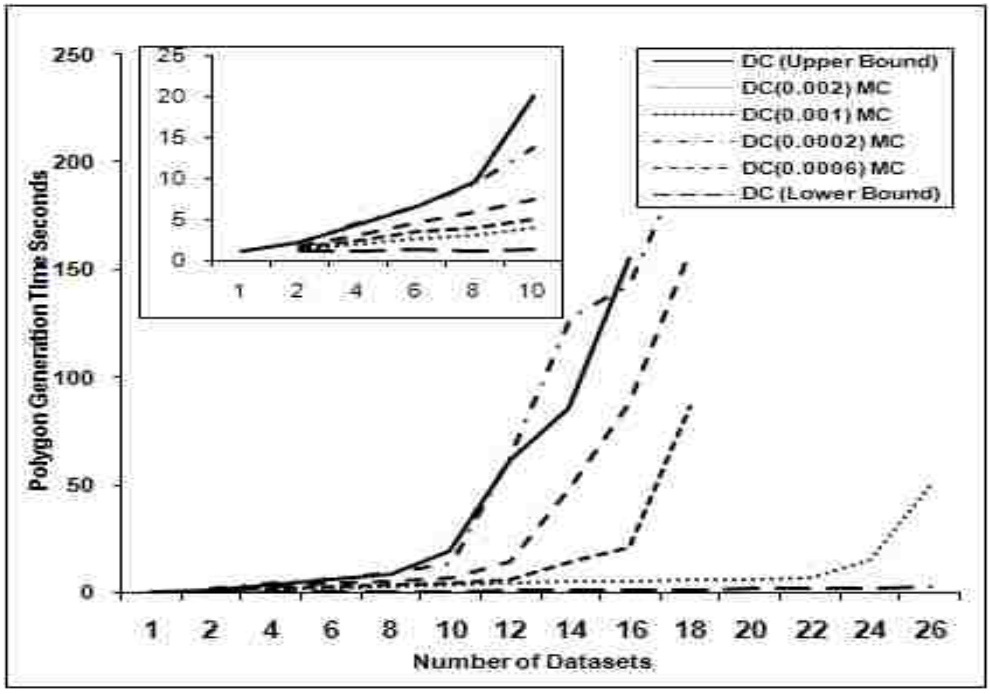


Figure 30: Effect of tolerance factor on polygon generation time with increasing number of datasets

Figure 30 shows the polygon generation time with varying tolerance factors. The inset in Figure 30 gives the detailed view of the lower region up to 10 datasets. There can be two extreme cases, one where there is no data coherence at all and the second where there is complete coherence. In the first case, the *time-N* curve closely resembles with the result obtained using the normal octree MDV. However, in the second case, data coherency plays a major role since all datasets share the same single set of polygon data. These two curves place the lower and upper bounds on the polygon generation time. The time varies with the tolerance factor used to apply data coherency condition. As the tolerance factor is reduced, the generation time increases and the accuracy of the approximated isosurface improves. For the dataset used, the time approximately converges to the upper bound for tolerance factor of 0.0002. Other curves corresponding to 0.0006, 0.001 and 0.002 show the increase in the time with decreasing tolerance factor because more portions of the non-reference datasets need to be processed to obtain the isosurfaces. The number of *dc_level* nodes is equal to 4096 for these results. After a certain number of datasets processed, the generation time rapidly increases due to the limited main memory of the system.

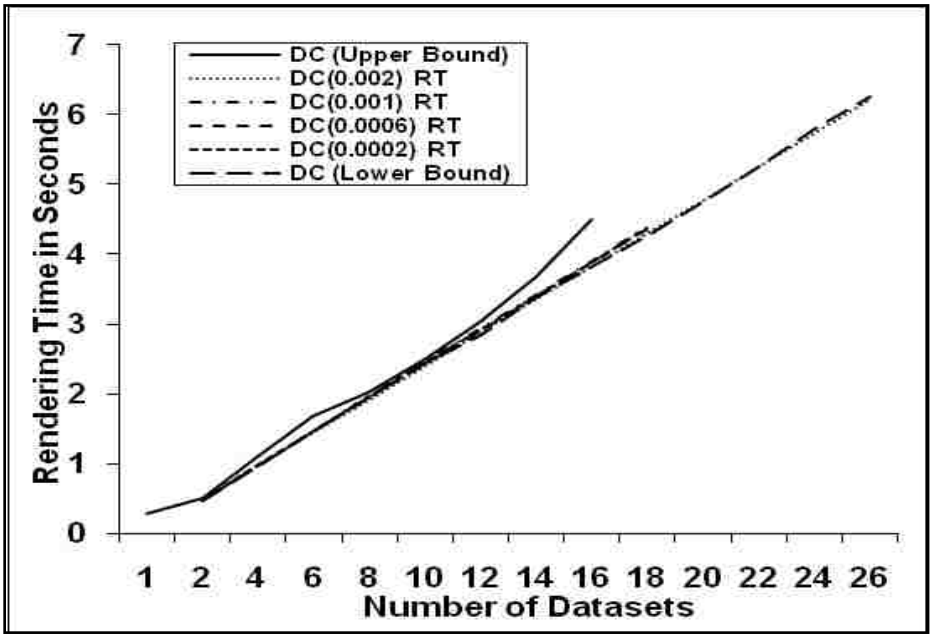


Figure 31: Effect of tolerance factor on polygon rendering time with increasing number of datasets

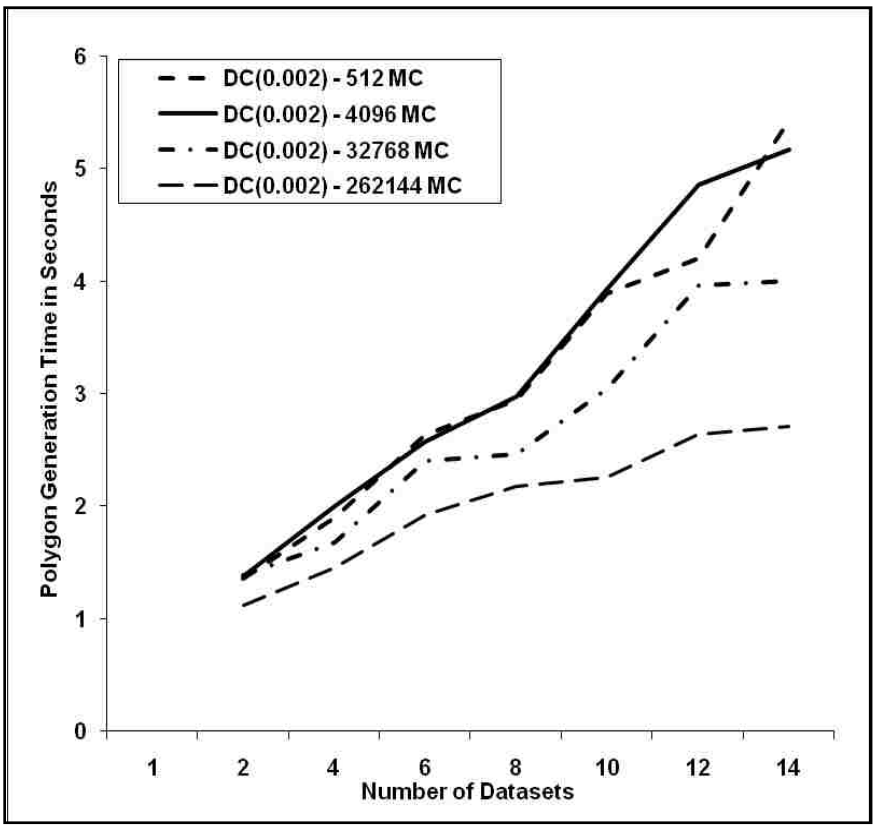


Figure 32: Effect of block size on polygon generation time with increasing number of datasets

Figure 31 shows the rendering time with varying tolerance factor. The rendering times for all levels of coherency are similar to each other and to the result from the normal octree-MDV method. This is expected; since the complete polygonal data (directly extracted plus approximated) are rendered for each dataset. In other words, the polygon data for the matched portions of the isosurfaces are rendered multiple times as needed depending upon the number of datasets, which are compared to the reference dataset. A slight improvement arises because the common polygon data are rendered in multiple viewports at the same time to avoid repeated transfer of the data through graphics bus.

Figure 32 shows the performance analysis with varying block size, which depends on number of *dc_level* nodes. The nodes at level 3, in the case of 256^3 dataset result in 512 *dc_level* nodes, while *dc_level* nodes at levels 4 and 5 result in 4096 and 32768 blocks,

respectively. The block size determines the accuracy of the approximated isosurface and polygon generation time. Smaller the block size, higher is the accuracy of the isosurface. When the level of *dc_level* node is 3, we miss some portions of the isosurfaces. Therefore, the curve corresponding to *dc_level* node at level 3 coincides with the curve on *dc_level* node at level 4. On the other hand, when *dc_level* nodes is at level 5 and 6, the derived isosurface resembles quite closely to the original isosurface and the polygon generation time reduces with increase in the number of blocks. The rendering time shows the same behaviour as explained earlier.

4.5.2 Multi-Resolution

Like the data-coherency, the multi-resolution aims at reducing the memory requirement and providing interactivity during MDV. Multi-resolution [78-80] technique tries to reduce the amount of memory required by reducing the resolution in case of isosurface. The implementation provides flexibility to the user to select the dataset that needs to be visualized at lower resolution or at higher resolution or certain portion of the data at higher resolution and other at lower resolution. Thus inter/intra multi-resolution reduces the memory requirements and thus improves the performance and the interactivity, which are important for the MDV system. Further, multi-resolution scheme allows the user to define the criteria at which the resolution for data needs to be reduced. This criteria can depend upon memory availability or frame rate or data loading time or may be as simple as number of datasets. The multi-resolution technique for MDV can be implemented at the data level or at the polygon level for isosurface extraction in the following three ways:

- **Dynamic Resolution**

  The dynamic resolution approach works by reducing the resolution of all the datasets according to the user-defined criteria. The criterion for reducing the can depend upon the data and the desired results. For example if we want to maintain certain frame rate then the resolution of all the datasets is reduced as soon as frame rates drop below the given threshold. The criteria can also take in to account the

number of datasets or some parameter, specific to the data. All the datasets at any specific time are at same resolution level.

- **Hybrid Resolution**

  Hybrid resolution is similar to the dynamic resolution but it allows the user to selectively reduce the resolution of the datasets while keeping the resolution of the other datasets at some level. In this scenario, the different datasets can be at different resolution level. Thus, during visualization some of the datasets will be at higher resolution and some at lower resolution. The interface allows the user to select the dataset and the resolution of the dataset.

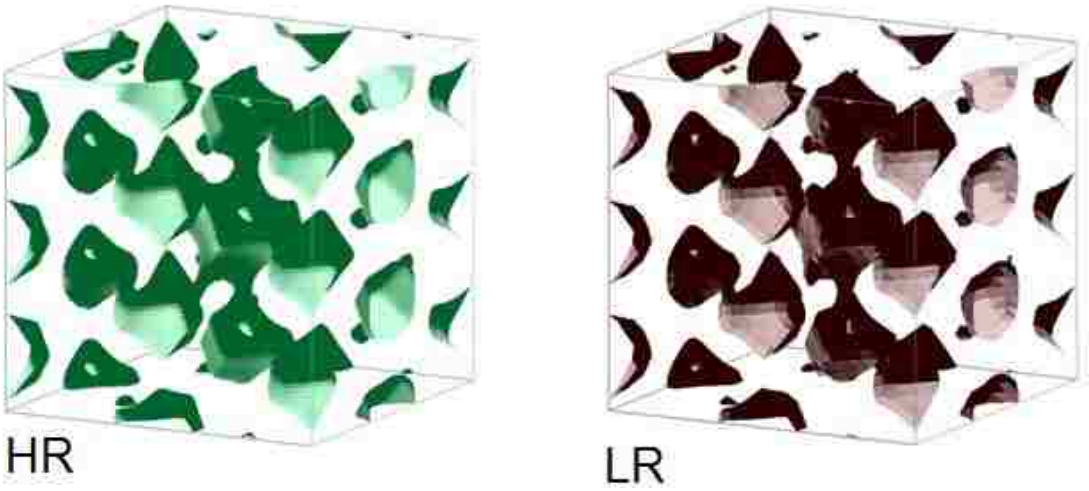


Figure 33: Hybrid resolution approach

- **Mixed Resolution**

  Mixed resolution approach allows the user to select a portion of each dataset at different resolution level. The portion of the dataset and its resolution level can be are user defined parameters and can be altered at run-time. In this case, the user needs to define the bounding box in the data grid and visualization algorithm generates the primitives for the isosurface corresponding to the voxels that lie inside the user defined bounding box at the user-defined level for that region.

Dynamic and hybrid resolution can be considered as a special case of mixed resolution scheme where complete dataset (instead of a portion) is at a user defined resolution.

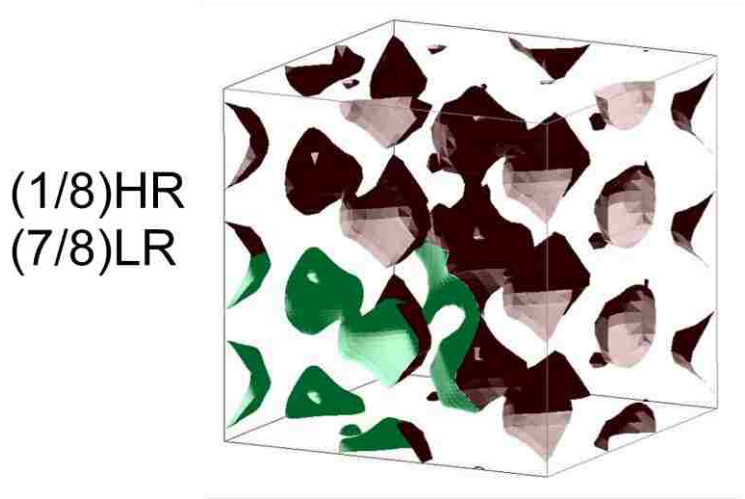


Figure 34: Mixed resolution approach

The multi-resolution allows loading more datasets and improving the responsiveness of the MDV application. To estimate the advantage in terms of the number of datasets that can be gained at lower resolution can be estimated using the following conditions. Assume that the total available memory for the application is M. Now consider that the number of datasets at high resolution is N_{HR} and the number of triangles in one high-resolution (HR) dataset is. T_{HR}. Similarly, for lower resolution, the number of datasets is N_{LR} and the number of triangles in one low-resolution (LR) dataset is T_{LR}.

***CASE 1**:* Reducing the resolution at the geometry level (scaling factor f in each dimension)

$$M = N_{HR} * M_D + N_{HR} * M_{THR} \qquad \text{Equation 29}$$

where, M_D and M_{THR} are the memory for the dataset and triangles per dataset at higher resolution respectively. Similarly for the lower resolution,

$$M = N_{LR} * M_D + N_{LR} * M_{TLR} \quad \text{Equation 30}$$

$$T_{HR} = f^2 * T_{LR} \quad \text{Equation 31([81])}$$

$$\text{Therefore } M_{THR} = f^2 * M_{TLR} \quad \text{Equation 32}$$

$$N_{HR} * M_D + N_{HR} * M_{THR} = N_{LR} * M_D + N_{LR} * \frac{M_{THR}}{f^2} \quad \text{Equation 33}$$

$$N_{LR}/N_{HR} = (M_D + M_{THR})/\left(M_D + \frac{M_{THR}}{f^2}\right) \quad \text{Equation 34}$$

CASE 2*:* Reducing the resolution at the data level (scaling at factor *f* in each dimension)

$$M = N_{HR} * M_{DHR} + N_{HR} * M_{THR} \quad \text{Equation 35}$$

where, M_{DHR} and M_{THR} is the memory for the dataset and triangles per dataset at higher resolution respectively. Similarly for lower resolution,

$$M = N_{LR} * M_{DLR} + N_{LR} * M_{TLR} \quad \text{Equation 36}$$

$$T_{HR} = f^2 * T_{LR} \quad \text{Equation 37([81])}$$

$$\text{Therefore } M_{THR} = f^2 * M_{TLR} \text{ and } M_{DHR} = f^3 * M_{DLR} \quad \text{Equation 38}$$

$$N_{HR} * M_{DHR} + N_{HR} * M_{THR} = N_{LR} * \frac{M_{DHR}}{f^3} + N_{LR} * \frac{M_{THR}}{f^2} \quad \text{Equation 39}$$

$$N_{LR}/N_{HR} = (M_{DHR} + M_{THR})/\left(\frac{M_{DHR}}{f^3} + \frac{M_{THR}}{f^2}\right) \quad \text{Equation 40}$$

Crack Problem

The crack problem in case of isosurfaces has been mentioned [76, 82-85] earlier for meshes with varying resolution. There are several known solutions for patching crack. Shekhar's [83], method fixes cracks by matching the isosurface generated from the higher-resolution cells to the isosurface generated from the lower-resolution cell. Shu's [82] uses

polygon of the same shape as the crack to patches cracks in isosurfaces Weber [76] proposed a solution for crack fixing for cell-centered data. Fang [85] proposed a method to use pyramidal regions for avoiding cracks. Westermann [50] used "triangle fans" to match the isosurface from the lower-resolution cells to the isosurface of the higher-resolution cells.

Dynamic-Resolution (DR) Approach for AIM and OPIM

The AIM and OPIM methods show the general trend that the polygon generation (T_G) and rendering (T_R) increase non-uniformly with increasing *N*. The average slope of the *time-N* curve in the high-*N* region tends to be larger than that in the low-*N* region. An improved MDV method is expected to reduce the time in overall, maintain a linear time-*N* relationship over the entire *N*-regime, and increase N_{MAX}.

Performance Analysis

Three user-defined threshold criteria are used for the resolution change: The first criterion keeps the polygon rendering time below a threshold value so that a lower bound can be placed on the frame-rate. For instance, when $T_R(N+1) \geq 1.0\ sec$, switch to a low-resolution mode for the *N*+1 datasets. The second criterion keeps the polygon generation time below a threshold value such as $T_G(N+1) \geq 25\ sec$. The third criterion avoids the relatively rapid increase in the polygon generation and/or rendering times by examining the *time-N* slope. For instance, when $R = \frac{T(N+1)-T(N)}{T(N)-T(N-1)} \geq 1.25$, switch to a low-resolution mode for the *N*+1 and more datasets. Here, T can be either T_R or T_G.

For the DR-MDV technique, the all three criteria have resulted in two successive transitions (Figure 35): For the first criterion, the transitions occur after 4 and 20 datasets; for the second criterion, they occur after 7 and 45 datasets, and finally for the third criteria they occur at 13 and 55. Thus, the DR-MDV technique with some appropriate user-defined criteria allow a simultaneous of visualization of up to 64 datasets thereby maintaining acceptable polygon generation and rendering times over the entire data range. The calculated *time-N* relationship is shown to be linear for each region with its slope in

different regions decreasing as the resolution of the data is reduced (lines from the left to the right in Figure 35). The DR-approach thus avoids the high-*N* region characterized by a relative rapid increase in time for both AIM and OPIM by dynamically switching to a reduced resolution mode. It is thus expected the same *time-N* relationship holds whether the DR approach is applied in conjunction with AIM or OPIM.

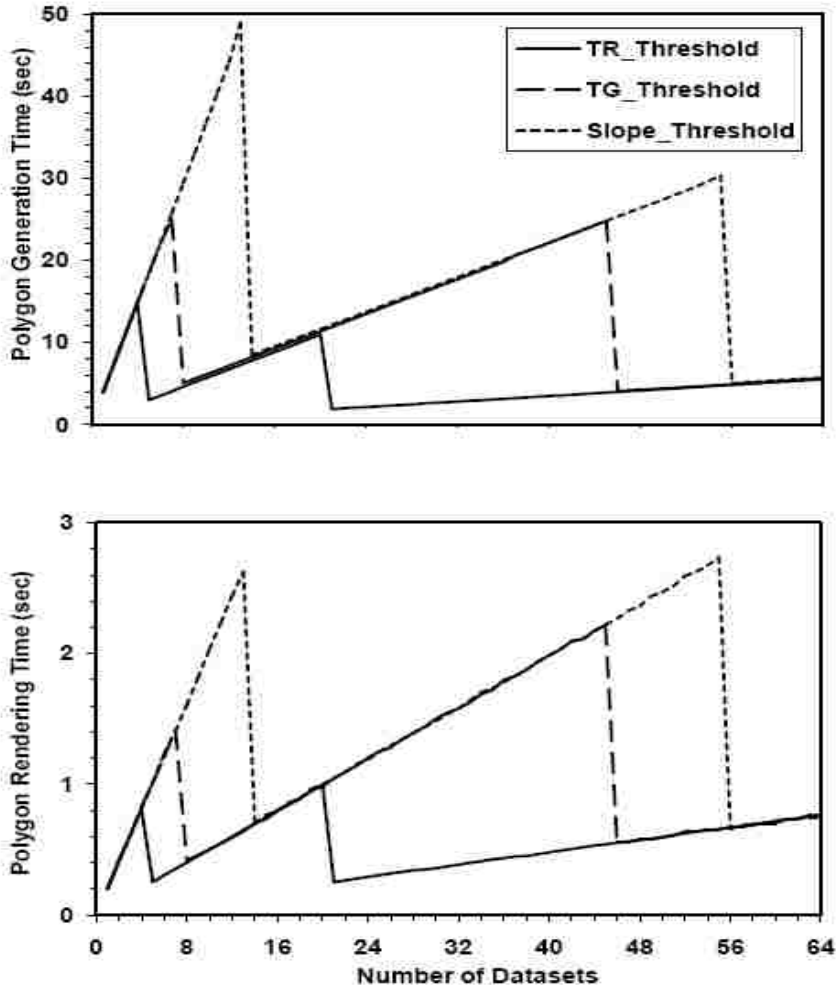


Figure 35: DR-MDV: Polygon generation (top) and rendering (bottom) times for isovalue = 0.04 for three criteria

Both the AIM and OPIM methods show the general trend, which can be described as follows: First, the geometry generation and rendering times increase non-linearly with

increasing *N*. The average slope of the *time-N* curve in the high-*N* region is much larger than that in the low-*N* region. Second, the maximum number of datasets that can be processed concurrently is limited. Third, as *N* increases, both T_G and T_R increase even in the linear low-N region, so as to be no longer acceptable for the desired interactivity. An improvement over these methods is expected to maintain a linear time-*N* relationship over the entire *N*-regime (i.e., to preserve the linear time complexity of the underlying MC and DC algorithms), increase N_{MAX} and improve the generation and rendering times. MDV requires an approach, which reduces the memory usage as well as speeds up the geometry extraction and rendering process. Approaches, which manipulate the resolution at the original and/or extracted geometric data levels, are of particular interest. For instance, the adaptive method was previously used to reconstruct isosurfaces from volume data at arbitrary levels of details to enable real-time navigation through the data while providing user-adjustable resolution levels [50]. On the other hand, the mesh reduction approaches, which was applied to an isosurface either as a post-process after the extraction phase or during the extraction phase itself, tried to reduce the number of polygons [46]. Both the approaches tend to reduce the number of triangles but they do not deal with the volume data.

Figure 36 shows the comparison between point and polygons approach for the dynamic resolution. All three criteria for the DR-AIM have resulted in two successive transitions (Figure 36): With polygons, the first, second and third criteria show these transitions at 5 and 20 datasets, 9 and 36 datasets, and 0 and 40 datasets, respectively. With points, they occur at 5 and 24 datasets for the $T_R(N+1) \geq 0.2$ sec criterion, at 13 and 40 datasets for the $T_G(N+1) \geq 5$ sec, and at 15 datasets for the slope-threshold criterion. Thus, the DR-AIM method allows a simultaneous of visualization of up to 64 datasets thereby maintaining acceptable generation and rendering times over the entire data range. The calculated *time-N* relationship is shown to be linear for each region with its slope in successive regions decreasing as the resolution of the data is reduced (lines from the left to the right in Figure 36). The high-*N* region of the AIM method characterized by a rapid increase in time is thus avoided completely by dynamically switching to a reduced

resolution mode. A similar *time-N* relationship holds for the DR-OPIM method as long as we are in the linear region, and the transitions occur at larger *N* values.

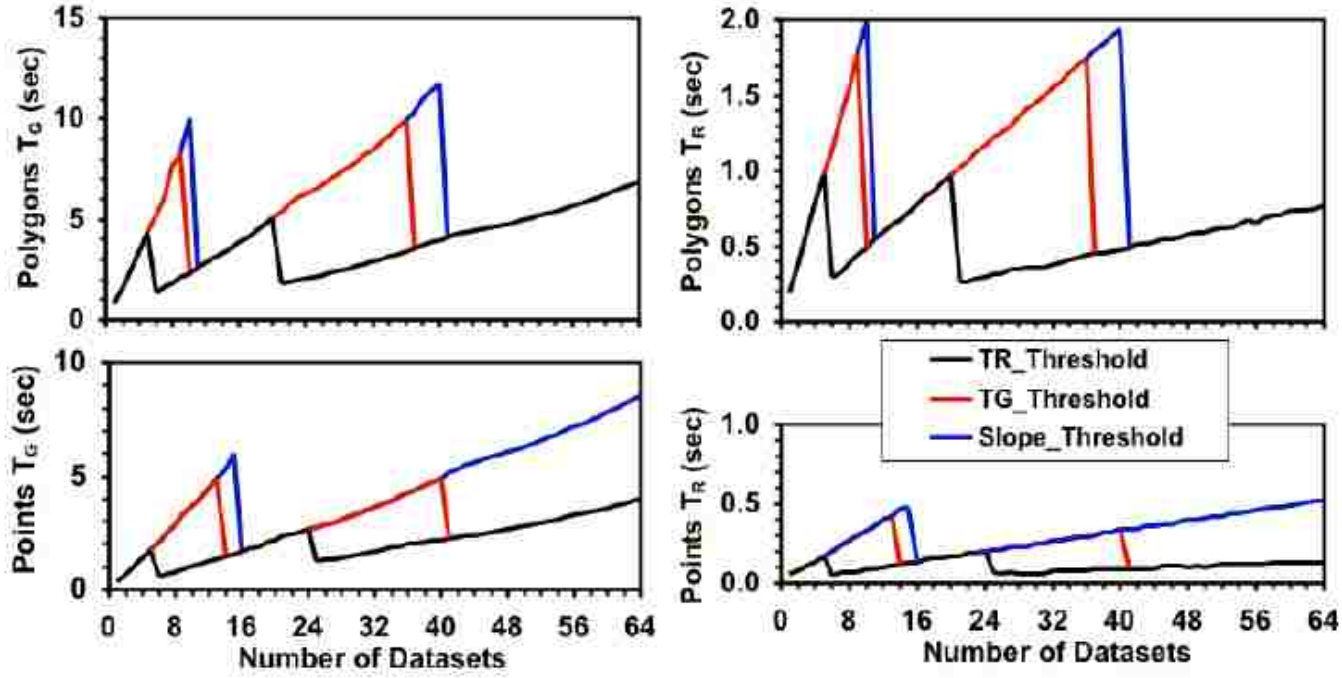


Figure 36: DR-AIM - Point vs. Polygons

Performance Analysis

This section covers the performance analysis for the dynamic, hybrid and mixed mode multiresolution. AIM scheme is followed for all the results. Figure 37 shows the multi-resolution images for 0.01 and 0.04 isovalue for different resolution levels. The number of triangles reduces by approximately one fourth as we lower down the resolution. For HR there are approximately 200K triangles, while in LR case there are approximately 600K triangles. This is analogous to the theoretical calculations. This reduces the memory requirement considerably and allows the visualization of larger number of datasets in single visualization. Figure 38 and Figure 39 shows the isosurface extraction time and rendering time for multi-resolution techniques. For performance analysis, we consider different cases for dynamic, mixed and hybrid resolution.

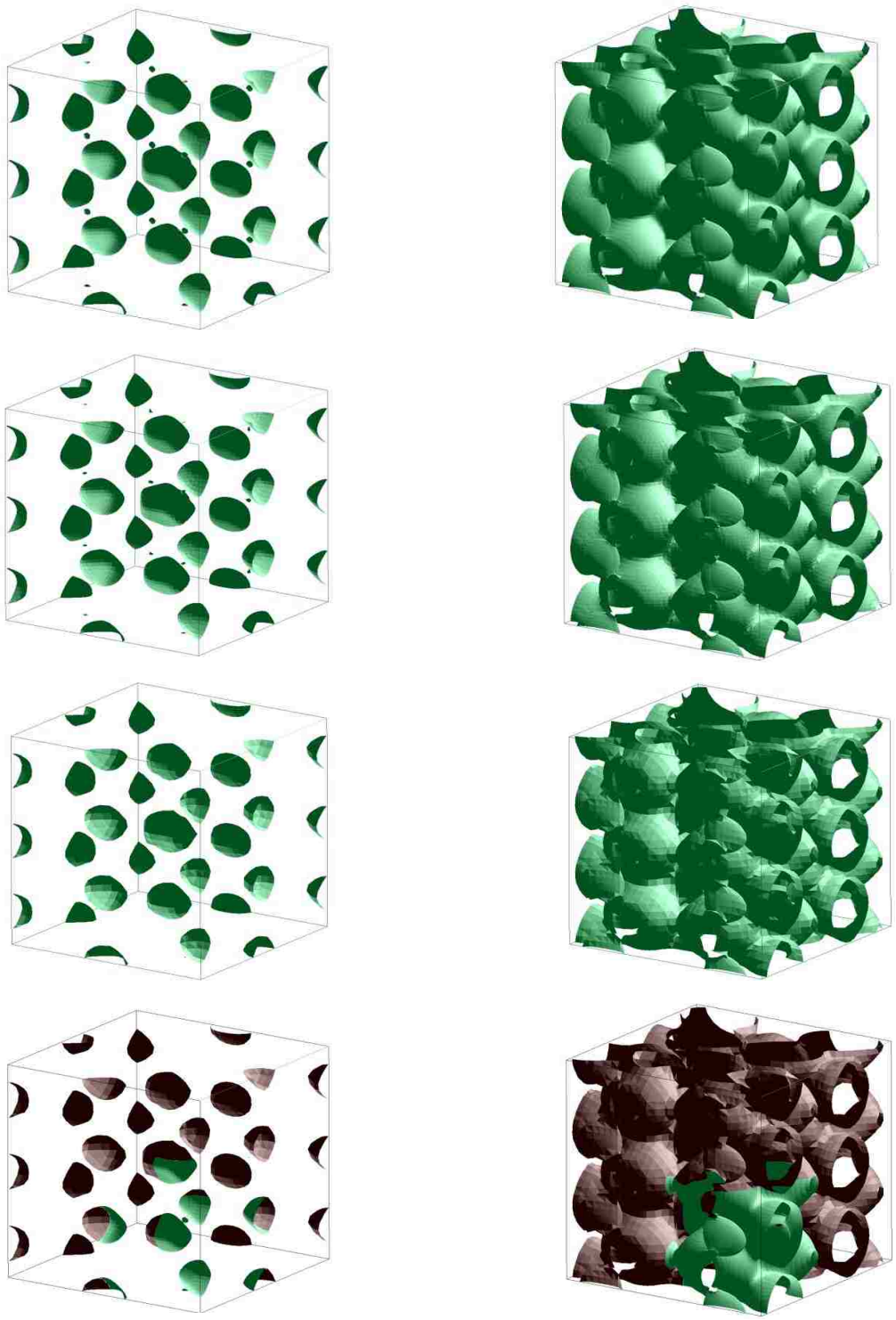


Figure 37: Multi-Resolution for 0.01 (Left) and 0.04 (Right) isovalue. Top row shows isosurfaces at HR, second row shows isosurfaces at LR, third row shows surfaces at LLR and last row shows isosurfaces mixed resolution (HR and LLR)

The HR scenario is when all the datasets are at the highest resolution, while LLR scenario is when all the datasets are lowered by two levels. LR case is for the scenario when all the datasets are lowered in resolution by one level. Further, 1HR + (N-1)LR shows the hybrid resolution where one dataset is at higher resolution and all other datasets are at lower resolution. The mixed resolution approach where part of the dataset is at higher resolution and other part is at lower resolution is represented as (1/8) HR + (7/8)LR and 0.5HR + 0.5 LR. First one represents the case where only 1/8 portion of the data is at higher resolution and remaining portion is at lower resolution. Second represents the case where half of the dataset is at higher resolution and other half is lower resolution. For mixed resolution same criteria applies to all the datasets under consideration. Apart from these we also consider the cases such as (N/2)HR + (N/2)LR and 0.5HR + 0.5LR. There are two extreme scenarios, the lower resolution (LLR) case and higher resolution (HR) scenario is when all the datasets are at higher resolution. All other multi-resolution cases arising from hybrid or mixed fall in this extreme range.

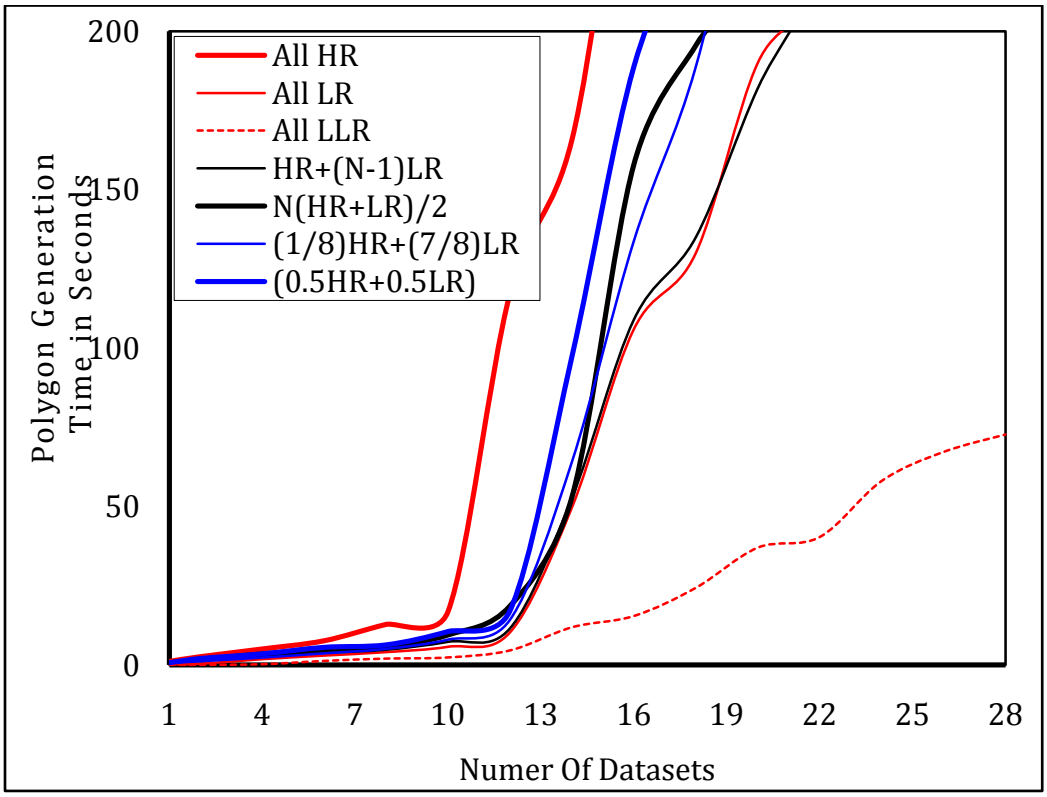


Figure 38: Isosurface generation time for multi resolution

Multi-resolution approach helps in reducing the memory requirements. As we lower down the resolution of the polygons generated, we see the improved interactivity. Also similar to data coherency and other techniques we witness the sudden jump in polygon generation times (Figure 38) datasets. This is due to use of virtual memory. For HR case, the jump is around 9 datasets, while for all other cases the jump is seen around 12 datasets. In case of LLR, the amount of jump is significantly lower as compared to other cases. It is important to note the number of datasets in LLR case is higher as compared to HR case. Rendering times for different cases are shown in Figure 39. Rendering times becomes smaller as we move from higher resolution to lower resolution. This is due to fact that we are rendering lesser number of triangles. This results in better interactivity and lowering in memory requirements.

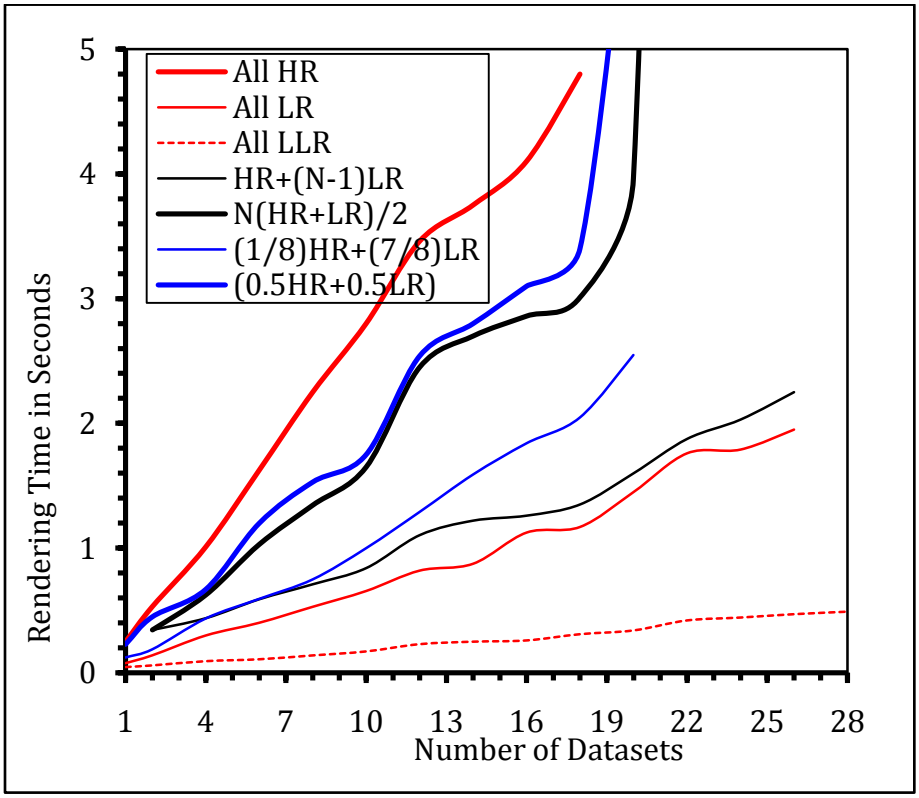


Figure 39: Rendering time for multiresolution

As we can see, there is a linear relation between the polygon generation time and the number of datasets. This linear relation is disrupted due to virtual memory requirement as we visualize more number of datasets. The reduction in the number of triangles helps in improved interactivity of the application.

4.5.3 Quasi-4D Isosurface Extraction

The quasi-4D isosurface extraction like data coherency is only specific to multiple dataset visualization. The key idea behind this approach is to reduce the amount of rendering, while still having the ability of finding the relationships between the similar datasets. In the earlier discussed approaches such as data coherency and multi-resolution, the different datasets are rendered in different viewports. They generate the triangular mesh in XYZ space. Our quasi-4D isosurface extraction method generates the triangular mesh in the XYZN space, where N is the number of datasets under consideration. Therefore, instead of viewing each dataset as a whole at one instant, in the case of quasi-4D isosurface we are viewing a certain number of slices from each dataset at a time. In this approach, the number of dataset is thus treated as the fourth dimension and at particular instant only slices at particular depth (which could be one of the three axes) are be rendered from all the datasets, all in the same viewport. The similar datasets processed this way would produce different structures at different depths, thus giving the relationship between these datasets.

Steps involved in generating the isosurface for this process are:

- **Step 1**: Load the dataset in the three dimensional array.
- **Step 2**: Construct the dataset from the above loaded datasets and generate the octree for fast isosurface extraction.
- **Step 3**: Generate an isosurface for the dataset construct in the step 2.
- **Step 4**: The slices from different datasets are rendered in same viewport.

This approach significantly reduces the amount of data to be rendered, thus improving interactivity. Suppose the number of datasets being visualized in N at particular instant.

Let the dimension of the i^{th} datasets be $d_i \times d_i \times d_i = d_i^3$

In addition, thickness of t^{th} slice is t_{ik} for i^{th} dataset. Total number of slices in i^{th} dataset is n_i. Then the total number of slices to be rendered will be N and total volume (V_k) being rendered at any instant is given by will be given by

$$V_k = \sum_{i=1}^{N}(t_{ik})\ for\ k^{th}\ slice\ . \qquad \text{Equation 41}$$

If all the volumes are being rendered simultaneously, then total volume (V) is given by

$$V = = \sum_{i=1}^{N}\sum_{k=1}^{n_i}(t_{ik}) \qquad \text{Equation 42}$$

From above given equations it can be clearly seen that the amount of volume being rendered is considerably lower when this scheme is used. Secondly, this approach gives the positional difference in the datasets at a particular depth. The key challenge in this approach will be finding the structure at the particular depth value and rendering it.

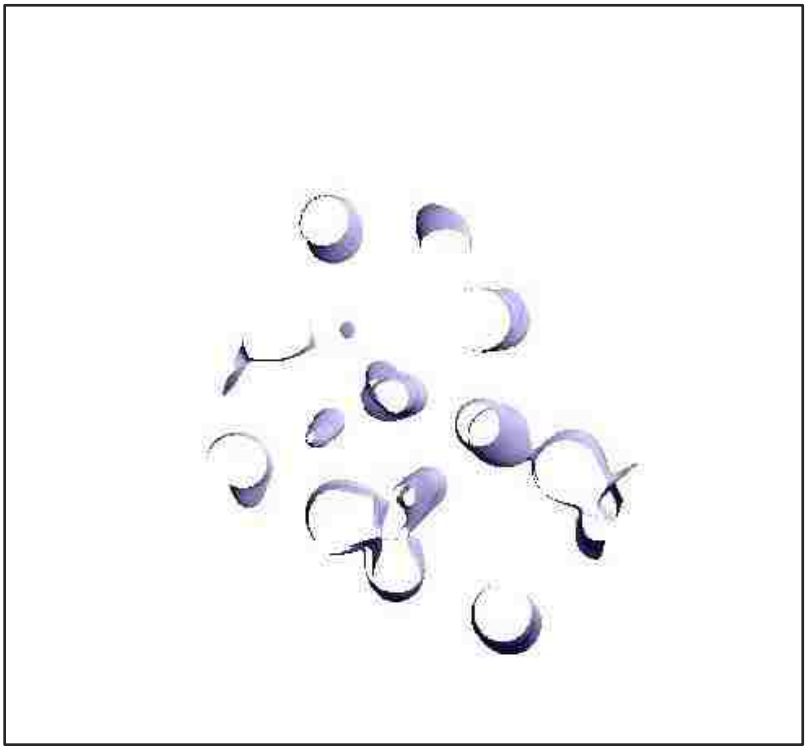
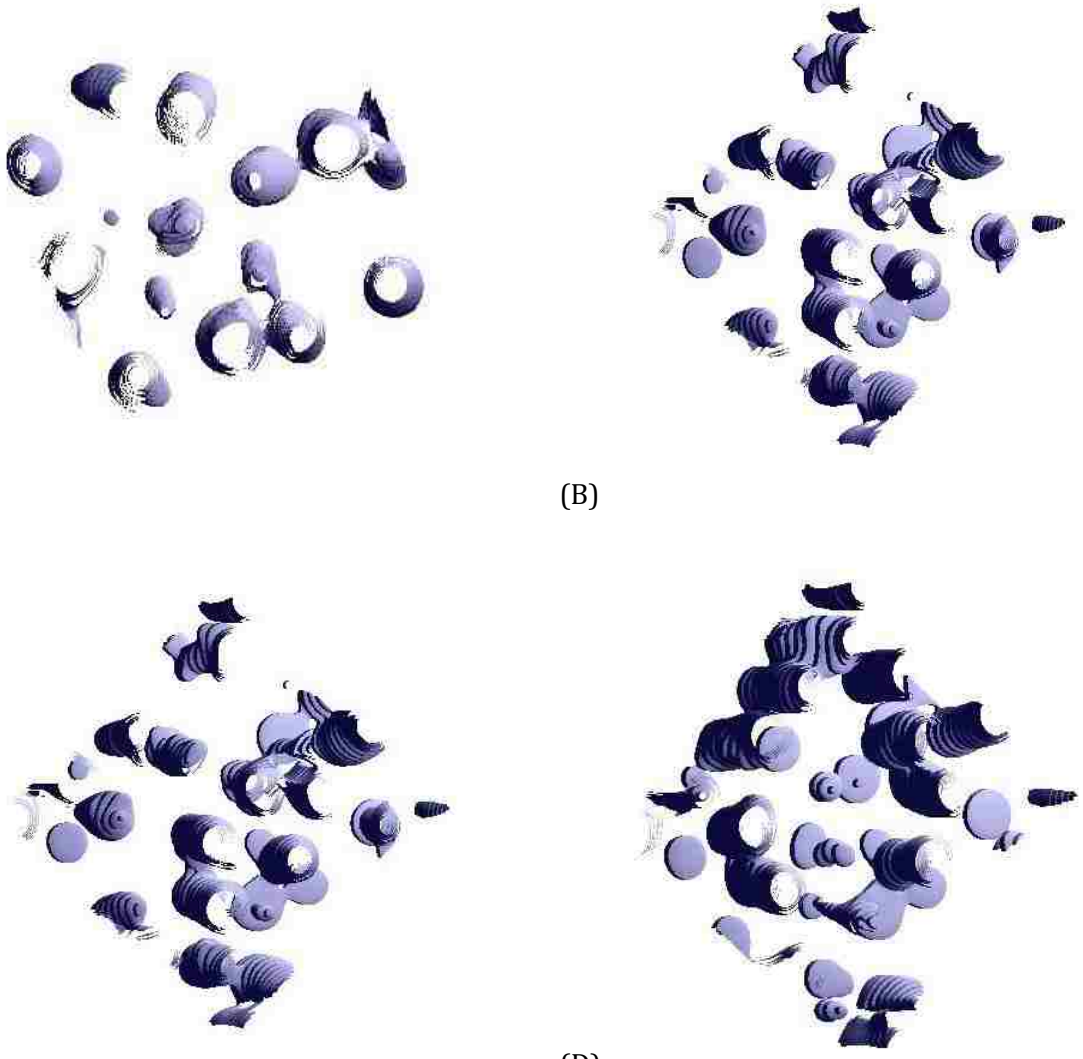


Figure 40: Quasi-4D Isosurface

Variation in Data Generation Process



(A) (B)

(C) (D)

Figure 41: Quasi-4D Isosurface for magnesium silicate liquid (A) Slice 1 at isovalue of 0.5. (B) Slice 15 at isovalue of 0.3 (C) Slice 15 at isovalue of 0.7 (D) – Slice 25 at the isovalue of 0.7

There are two approaches for generating the dataset to be used for quasi-4D isosurface extraction. Both these methodologies differ in the way the datasets is constructed from the original datasets. In the first scheme, the dataset is constructed using only the single slice from each of the original datasets. The slice is replicated multiple times to give the

thickness to the structure. Figure 41 shows the structure of quasi-4D isosurface of hydrous silicate. First slice of each dataset is replicated 10 times to produce the isosurface.

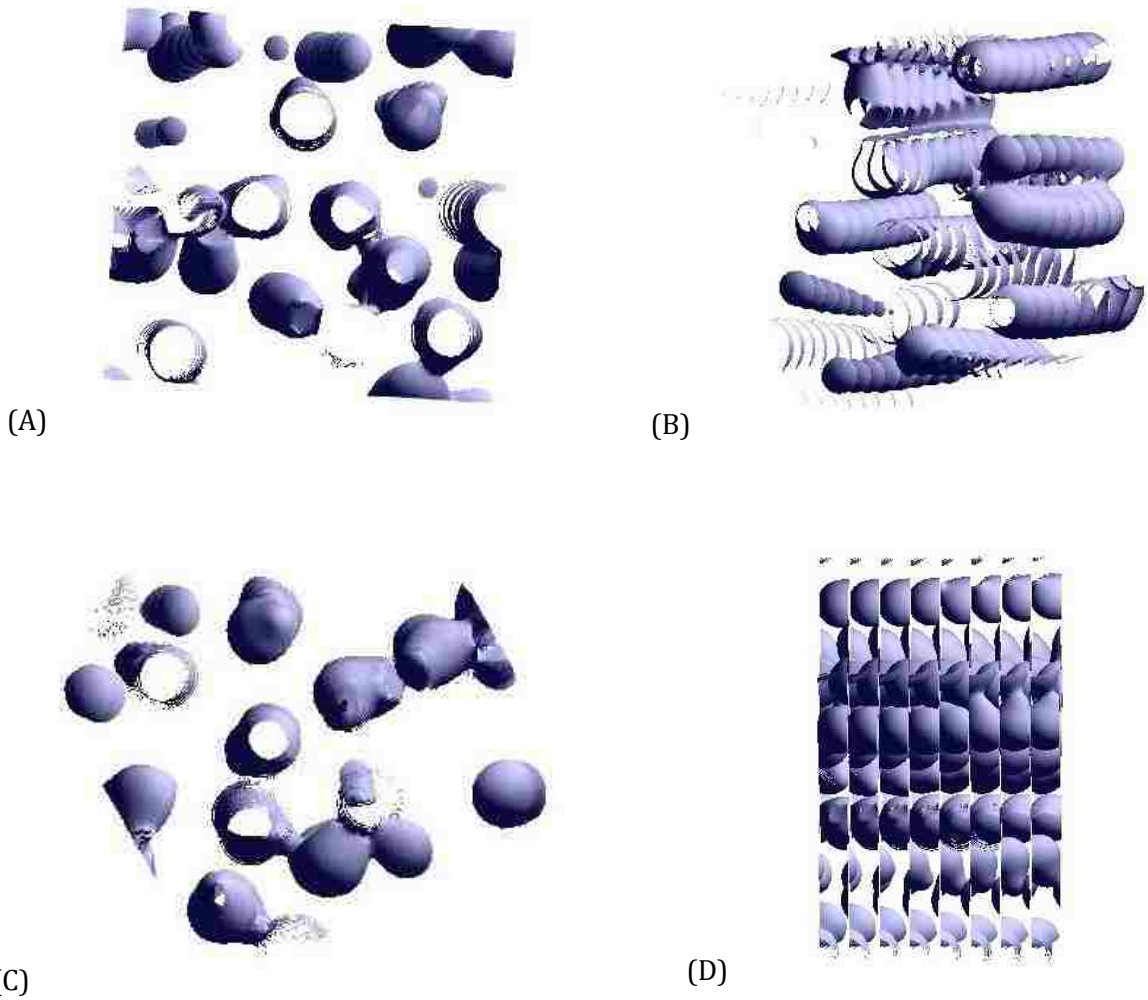


Figure 42: Quasi-4D Isosurface for magnesium silicate liquid. Slice 1-17 for each dataset at isovalue of 0.3 from different views. (D) Discontinuities in the dataset due to slices at different level are adjacent to each other.

In the second scheme (Figure 42), the multiple slices are considered from each dataset. This results in the discontinuities in the structure because in the resulting dataset slices at different level are adjacent to each other.

5 TEXTURE-BASED RENDERING FOR MDV

5.1 Introduction

Textures are simply rectangular arrays of data—for example, color data, luminance data, or color and alpha data. The individual values in a texture array are often called texels. A texture is usually thought of as being two-dimensional, like images, but it can also be one-dimensional or three-dimensional. The data describing a texture may consist of one, two, three, or four elements per texel and may represent an (R, G, B, A) quadruple, a modulation constant, or a depth component.

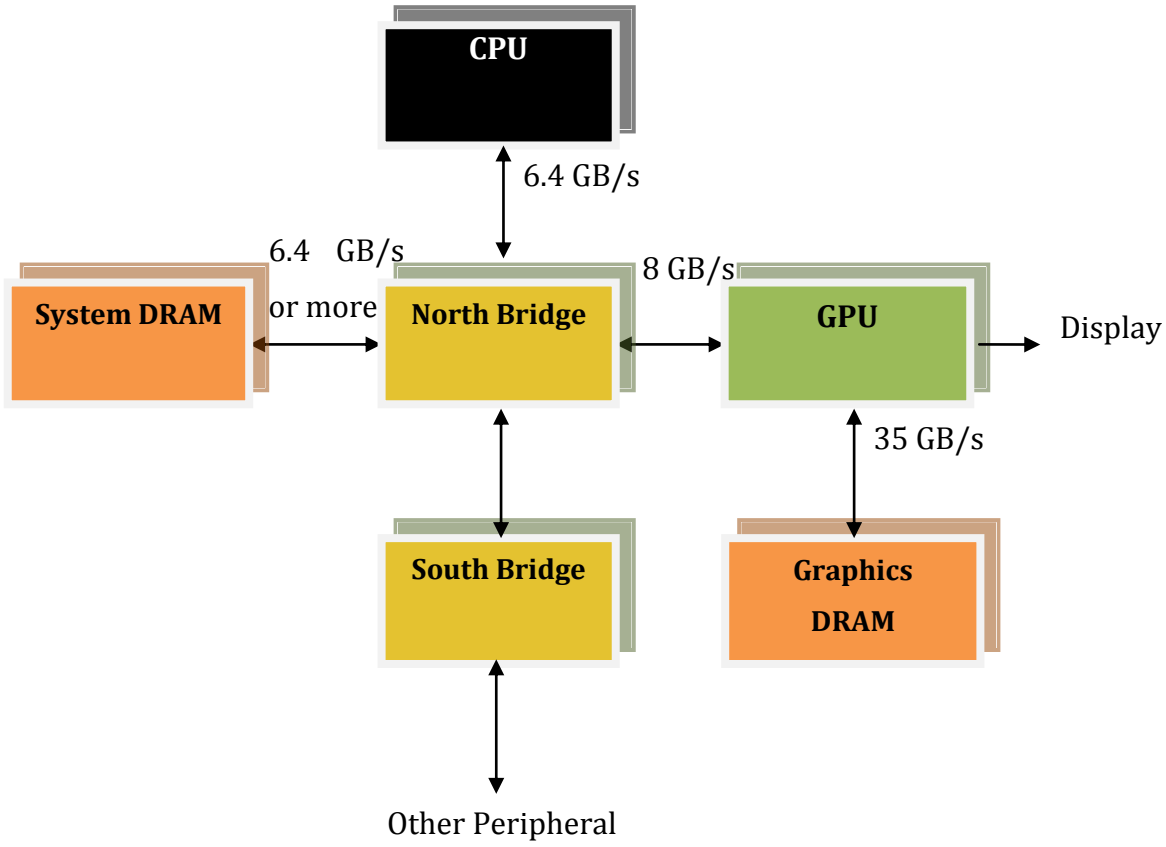


Figure 43 : Memory bandwidth in different parts of computer system [86]

Texture mapping is a process of adding the details or texture to the surface. Capabilities of modern programmable graphical processor units (GPU's) enhance the performance and the applicability of texture mapping approach via vertex and fragment shaders. Figure 43 show the memory bandwidth between different components of the computer system. The vast differences between the GPU's memory interface bandwidth and bandwidth in other parts of the system results in improved performance. There is a very high bandwidth (35 GB/s) available internally on the GPU. Algorithms that run on the GPU can therefore take advantage of this bandwidth to achieve dramatic performance improvements [86]. Apart from the more bandwidth, GPU provides multithreading environment by the means of multiple vertex and fragment units.

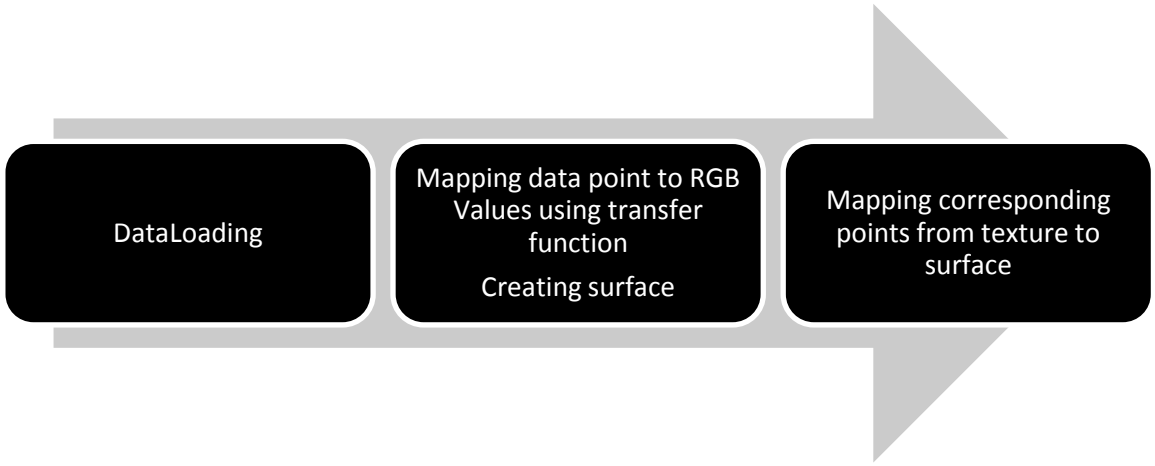


Figure 44: Texture mapping process

The vertex processors in the GPU allow the application of the vertex program to each vertex in the object being rendered and performing any vertex related operations. Onward NVIDIA 6800 series GPU the vertex program are capable of accessing texture data. Thus, they are connected to the texture cache apart from the vertex cache. The vertices after the vertex program are grouped into primitives. After this step the cull/clip/ block performs per-primitive operation. After this, the rasterization step removes the occluded pixel. The remaining pixels are passed to the fragment program for further processing. The fragment program works on the number of pixels simultaneously and thus hides the latency of

texture fetching. Thus by employing the fragment and vertex program, we can achieve better performance.

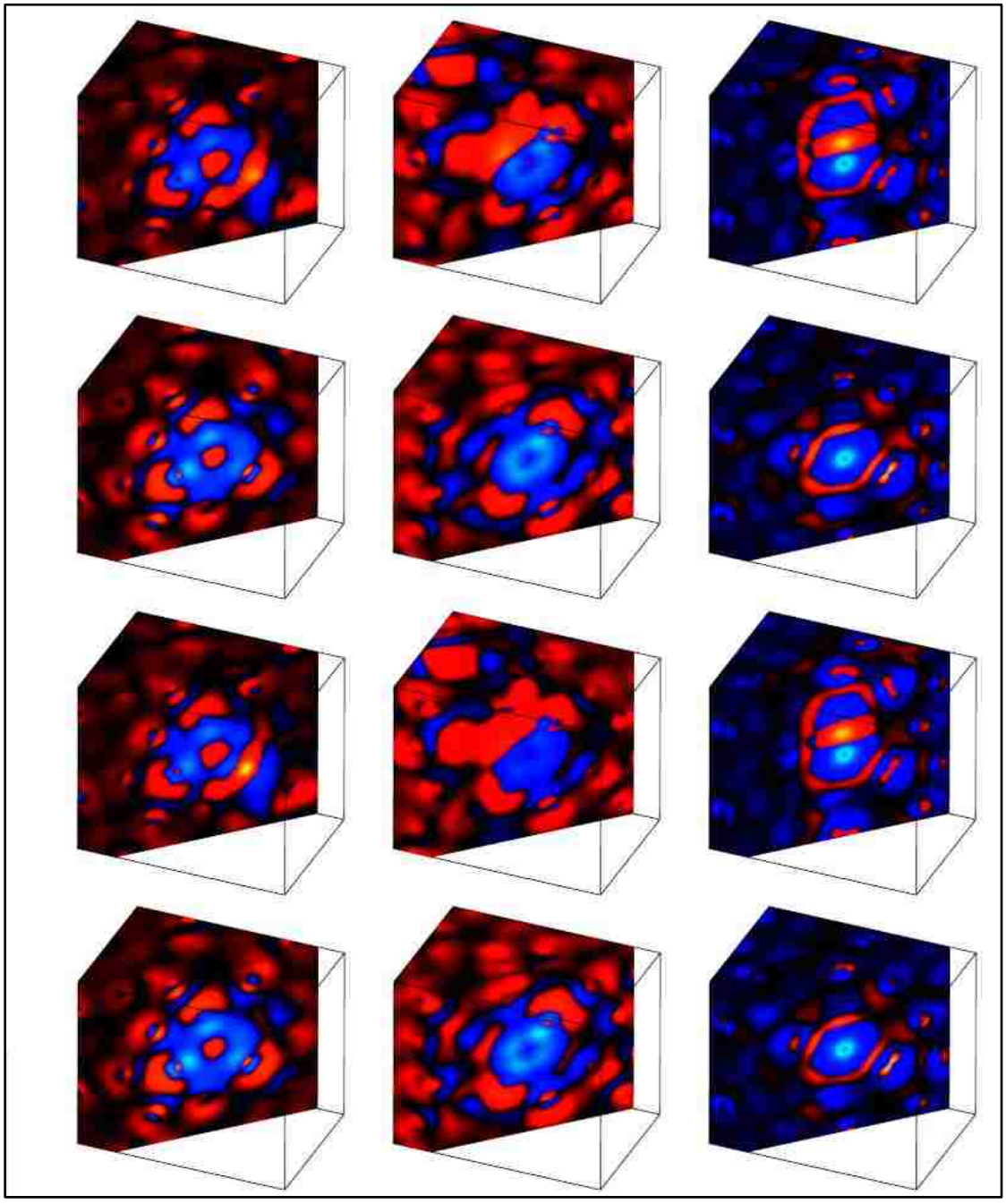


Figure 45: Visualization of electron density difference induced by the Mg (left), Si (centre) and O (right) vacancies in the 60-site $MgSiO_3$ system. First and second rows show final configuration (after atomic structural optimization) for migrating ion and fixed vacancy respectively. Third and fourth rows show the corresponding initial configuration.

5.2 Texture Mapping

Figure 44 shows the steps involved in the texture mapping process. In correspondence to the framework steps, the visualization process involves mapping of data points to the RGB values. It also involves calculation of the visible surface points during rendering and clipping to which the texture is mapped. The rendering thread is responsible for generation of rendering of the textures, i.e. mapping the texture to the surface points (polygons).

5.3 Transfer Function

Visualization of the scalar volume data would require the transfer function for mapping the scalar value to the RGB color space. Choice of transfer function depends upon the data being visualized. In this section, we describe the transfer function (Figure 46) used for visualizing electronic charge density data. For isosurface extraction process, transfer function is not required since we are extracting the isosurface corresponding to single isovalue. R, G and B value ranges from [0, 1]. OpenGL provides the option to specify the color format while generating the texture. For most of the calculations, we are using RGBA format, except for texture based isosurface extraction. For the isosurface extraction, we use floating-point texture, which holds the scalar values corresponding to the voxels, rather than the RGBA components. The RGBA format specifies 8 bits for each of the red, green, blue and opacity component for each voxel. These values are normalized to [0, 1]. Figure 46 shows how RGB mapping is applied for the scalar data.

For electronic charge density data, we need to highlight the positive and negative charge differences. In addition, we need to highlight the regions containing small differences. We have three different colors red, green and blue colors used to represent four difference value regions as shown in Figure 46 using the following equations:

$$T: R \rightarrow R^3 \qquad \text{Equation 43}$$

$$T_R = \begin{cases} R/R_{CUT1} & 0 < R \leq R_{CUT1} \\ 1 & R > R_{CUT1} \\ 0 & Otherwise \end{cases} \qquad \text{Equation 44}$$

$$T_G = \begin{cases} (R - R_{CUT1})/(R_{MAX} - R_{CUT1}) & R \geq R_{CUT1} \\ (R - R_{CUT2})/(R_{MIN} - R_{CUT2}) & R \leq R_{CUT2} \\ 0 & R_{CUT2} < R < R_{CUT1} \end{cases} \quad \text{Equation 45}$$

$$T_B = \begin{cases} R/R_{CUT2} & 0 > R \geq R_{CUT2} \\ 1 & R < R_{CUT2} \\ 0 & Otherwise \end{cases} \quad \text{Equation 46}$$

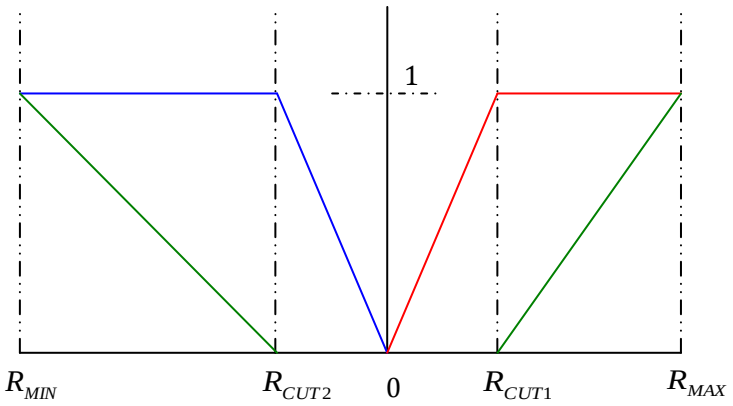


Figure 46: Transfer function for the scalar volume data. Red, green and blue curve represents the transfer function T_R, T_G and T_B respectively. R_{MIN} and R_{MAX} are the maximum and minimum value in the dataset.

Here R is the scalar value. By changing R_{CUT1} and R_{CUT2}, we can highlight different region in the dataset. We have used linear functions for RGB values, which can be changed to other forms (e.g., exponential) depending upon the dataset and the user requirements. The transfer function can be modified at the per-fragment level using the fragment shaders.

5.4 Volume Rendering

Volume rendering using texture is a three-step process

- Applying Transfer Function
- Texture Generation

- Texture Mapping with Volume Shading

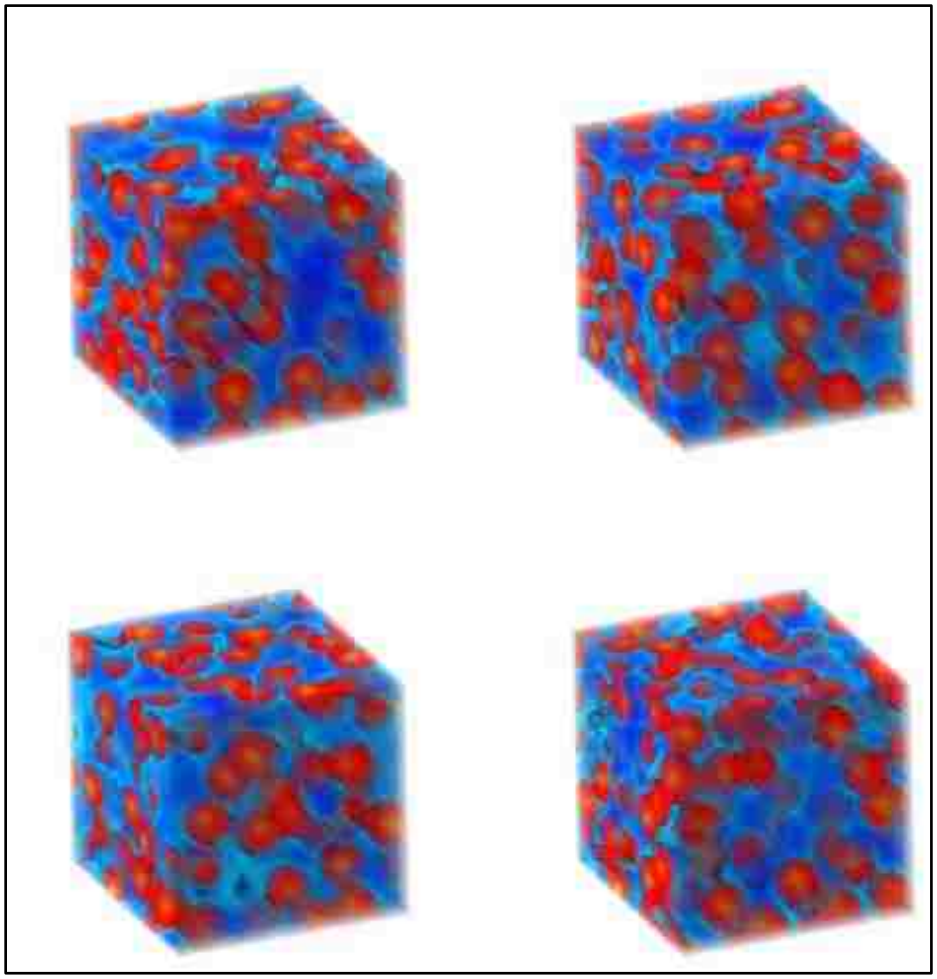


Figure 47: Volume Rendering

After loading the scalar data in the data loading thread, the transfer function is applied before the generation of the textures. Transfer function maps the scalar value to the RGBA component of the texture. In modern GPU, transfer function can be computed on the fly. In this case, we have to pack the scalar value in the texture. Thus, the texture will be a floating-point texture holding the data values instead of RGBA texture in the former case. After the transfer function is applied to 2D or 3D textures, the texture are generated using OpenGL commands (glTextureGen, glTexImage2D and glTexImage3D respectively). The generated texture is bound to the texture memory for volume rendering as described in section 2.5.2.

Alpha Blending

To improve upon the volume rendering, we use volume shading and alpha blending. Volume shading is volume rendering with lighting. Details about volume shading are presented in section 5.10. Alpha blending is the process of compositing color value of fragment being processed with the pixel already stored in the frame buffer [65]. During the process of blending, the color values of the incoming fragment are blended with the color components of the already stored pixel. The source and destination colors are blended according to the following equation

$$\left(R_S S_r + R_D D_r, G_S S_g + G_D D_g, B_S S_b + B_D D_b, A_S S_a + A_D D_a\right) \qquad \text{Equation 47}$$

Where (S_r, S_g, S_b, S_a) be the source-blending factor and (D_r, D_g, D_b, D_a) be the destination blending factor for red, green, blue and alpha components. The (R_S, G_S, B_S, A_S) and (R_D, G_D, B_D, A_D) are color at source and destination pixel. OpenGL specifies the way to specify certain predefined blending factor. This can also be done using the shading language in the fragment program. The user can specify the blending factor.

5.5 External 3D Surface Rendering

In this texture-based MDV scheme, we improve the interactivity by reducing the amount of texture mapping. The basic idea is to restrict the rendering of data to the external (visible) surfaces of the volume instead of performing complete or nearly complete 3D volume texture mapping. One place where such surface rendering makes sense is the visualization of the volume data by clipping. In the case of clipping whether it uses a simple clip plane or more complex 3D clip geometry. For instance, one can view scalar data on a cross-section of the volume with a cutting plane. One defines a regular grid on the clip plane, calculates data values on this grid by interpolation of the original data, and uses an appropriate color-map to make the data visible. If this is the case and we are using 3D texture-based rendering, then there is no need to have texture mapped on all slices to cover the complete volume.

As compared to regular texture mapping, surface-mapping technique performs the mapping on the external visible surface. The 3D surface rendering of the original volume can be done by simply extracting and mapping textures on the six surfaces of the cube (front and back, left and right, and top and bottom square faces), as shown in Figure 48. In effect, the problem of 3D volume texture mapping is reduced to the problem of *3D surface texture mapping*, which renders textured data on only those six surfaces. Each square surface can be represented by two edge-sharing triangles, so that 12 triangles thus needed are used to implement the clipping operation, which is described in the next Section. The 3D surface rendering approach works only with the 3D texture because it allows us to extract an arbitrary textured-slice without requiring re-sampling of the volume data. The polygon on which we want to map a texture can intersect the volume at any location and orientation. Texture mapping then requires the vertices of this polygon, which are passed as texture coordinates

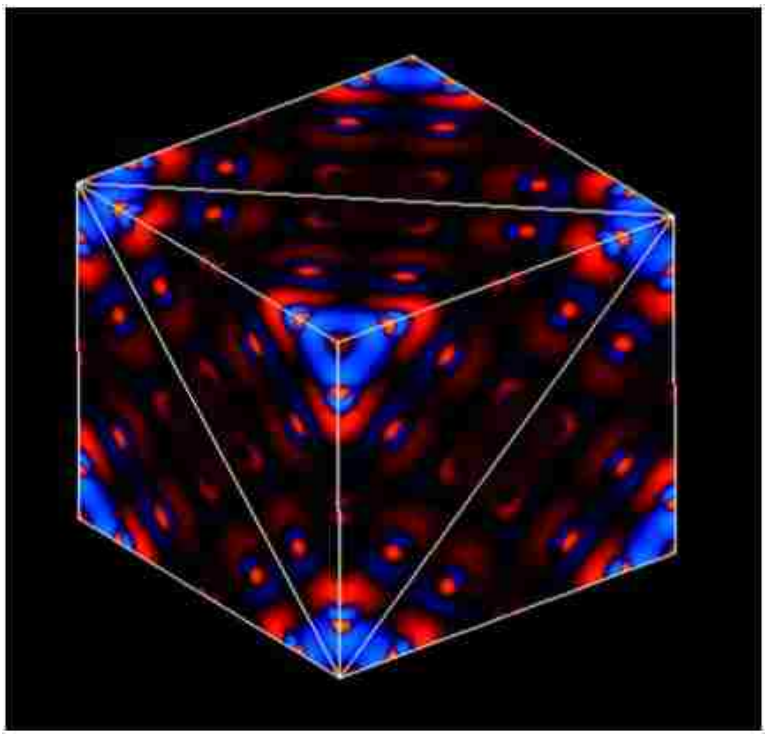


Figure 48: Surface rendering using texture mapping

5.6 Clipping

The purpose of clipping is to explore the hidden details in the dataset. The clipping can be either, volume probing, where fragments lying inside the clipped geometry are kept or volume clipping, where fragments lying outside the clipping geometry are kept. We now adopt an approach by exploiting the texture-mapping hardware and general clipping to support a fast visualization of multiple sets of volumetric scalar data. Similar approaches were previously used to visualize a single dataset with interactive planar clipping and volume clipping via per-fragment operations supported by graphics hardware [10, 14, 19]. They involve rendering of all texels of the 3D textures passing a given clip test, for instance, on average the half of the total number of 256^3 texels are rendered. In other words, the texture-based volume rendering (with or without clipping) uses large number of textured slices, which becomes bottleneck as the number of datasets increase.

5.6.1 Clipping with External Surface Rendering

This section describes how clipping can be combined with 3D surface texture mapping. During the clipping process, we find surfaces of the clipping object clipping the volume and then bound the intersecting surfaces of the volume in the form of simple polygons. These visible polygons determine the new set of surfaces of the volume and the texture mapping is performed only on these polygons. Thus, complete volume is never rendered only the surfaces defined by a set of visible clipping polygons (single or multiple) are rendered. Each clipping polygon is tessellated in terms of triangles. During initial rendering, the six planar surfaces of simulation box, each represented in terms of two adjacent triangles are rendered. During clipping process the intersections of these 12 triangles with a given clip object are calculated. If a triangle intersects, it is divided into two polygons that lie on either side of the clip plane; and one of them is discarded depending upon clipping. Intersection points are used to define new polygons to map textures. Every time the clip plane is adjusted in 3D space, intersections of the original 12 triangles are determined to define polygons [23], which bound new visible surfaces. The clipping method is described below. A plane in 3D space is given by, $Ax + By + Cz + D = 0$. Given three points in space

(x1, y1, z1), (x2, y2, z2) and (x3, y3, z3), the four coefficients, which define the equation of the plane passing through these points, are given by the following determinants.

$$A = \begin{vmatrix} 1 & y1 & z1 \\ 1 & y2 & z2 \\ 1 & y3 & z3 \end{vmatrix} \quad B = \begin{vmatrix} x1 & 1 & z1 \\ x2 & 1 & z2 \\ x3 & 1 & z3 \end{vmatrix} \quad C = \begin{vmatrix} x1 & y1 & 1 \\ x2 & y2 & 1 \\ x3 & y3 & 1 \end{vmatrix} \quad D = -\begin{vmatrix} x1 & y1 & z1 \\ x2 & y2 & z2 \\ x3 & y3 & z3 \end{vmatrix}$$

Equation 48

The clipping side of the plane is taken to be that one containing the normal. For a vertex (Q_x, *Qy, Qz*), the expression

$$side(Q) = AQ_x + BQ_y + CQ_Z + D \qquad \text{Equation 49}$$

can be used to determine which side of the plane the vertex lies on.

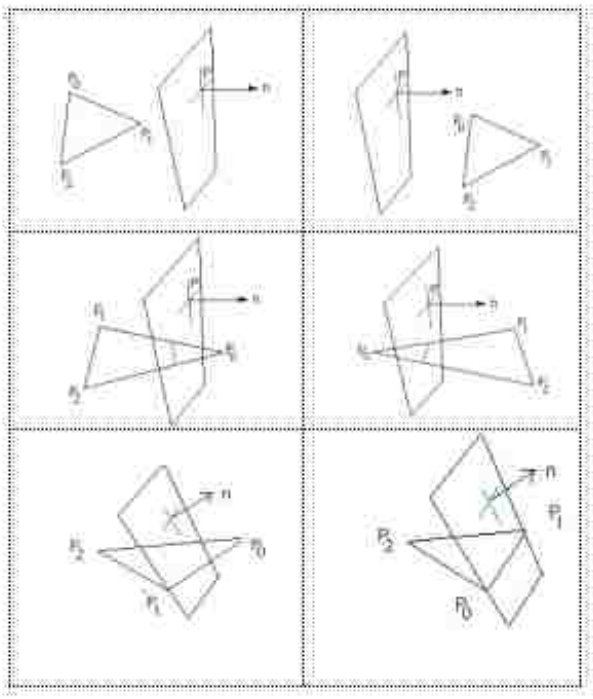


Figure 49: Intersection of triangle and plane

If it is positive the point lies on the same side as the normal, if negative the point lies on the other side, or if it is zero the point lies on the plane. After determining whether an edge intersects the clip plane, it is necessary to calculate the intersection point that will become a vertex of the clipped facet. Let the edge be between two points P_0 and P_1, the equation of the points along the line segment

$$P = P_0 + u(P_1 - P_0) \qquad \text{Equation 50}$$

Where u lies between 0 and 1. Substituting this into the above expression for the plane

$$A\big(P_{0x} + u(P_{1x} - P_{0x})\big) + B\left(P_{0y} + u\big(P_{1y} - P_{0y}\big)\right) + C\big(P_{0z} + u(P_{1z} - P_{0z})\big) + D = 0 \qquad \text{Equation 51}$$

Solving for u

$$u = -side(P_0)/(side(P_1) - side(P_0)) \qquad \text{Equation 52}$$

Substituting this into the equation of the line gives the actual intersection point. There are six different cases to consider, they are illustrated in Figure 49. The top two cases are when all the points are on either side of the clipping plane. Next two cases on the left are when there is only one vertex on the clipping side of the plane, the case on the right occurs when there are two vertices on the clipping side of the plane. The bottom left case occurs when one of the vertices is on the plane and bottom right one when two vertices fall on the clip plane.

A planar clipping is demonstrated in Figure 49. Every time a clip plane changes (rotates or translates in space), new polygons for texture mapping are generated. We have implemented a box-clipping object, which is represented as a set of six clipping planes. Visible surfaces are obtained by repeating the same process, which is used in the case of a single plane.

Once these surfaces are determined, textures are mapped on them. The clip box can be rotated, translated and scaled. Figure 50 shows the outer and inner box clipping. Outer box clipping means removing portion of 3D object that lie outside the clipping box while the

inner box clipping is just the reverse of outer box clipping where the portion outside the box is retained and portion inside the box is removed. For instance, the simulated charge density that considered here is confined in a cubic volume.

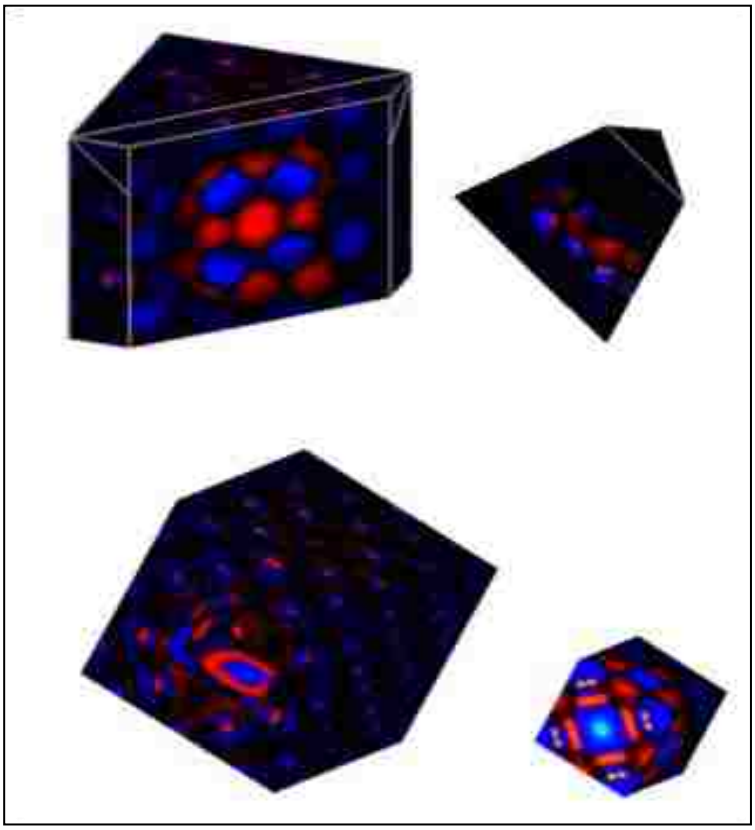


Figure 50: Planar and box clipping

5.6.2 Volume Clipping

Voxelized or Volume clipping technique [Wei03] utilizes two textures, volume object and clipping object. The 3D texture of the clipping object uses binary representation. The alpha value of the clipping texture is set to 0 or 1 depending upon whether the voxel lies inside the clipping geometry or not. Depending upon the alpha value of the clipping texture GPU assigns the alpha value to the volume texture. The fragments lying outside the clipping texture are assigned value zero. These fragments are discarded using the alpha test for

volume probing. For volume clipping, the fragments lying outside are assigned the value (1-alpha). The clip object can be moved around by mapping the texture coordinates of the clipping texture although changing the shape of the clipping object requires the loading of the new 3D texture. Since the method utilizes the binary representation there is a visible transition between the voxels having value 0 and 1. To overcome this artifact induced by the use of the binary textures, we utilize the distance map similar to the one given by Lorenson [Lor93]. Evaluation of F(x, y, z) at any point produces a value < 0, = 0, or > 0, where F(x, y, z) gives the signed Euclidean distance of the voxel from the surface of the clipping object.

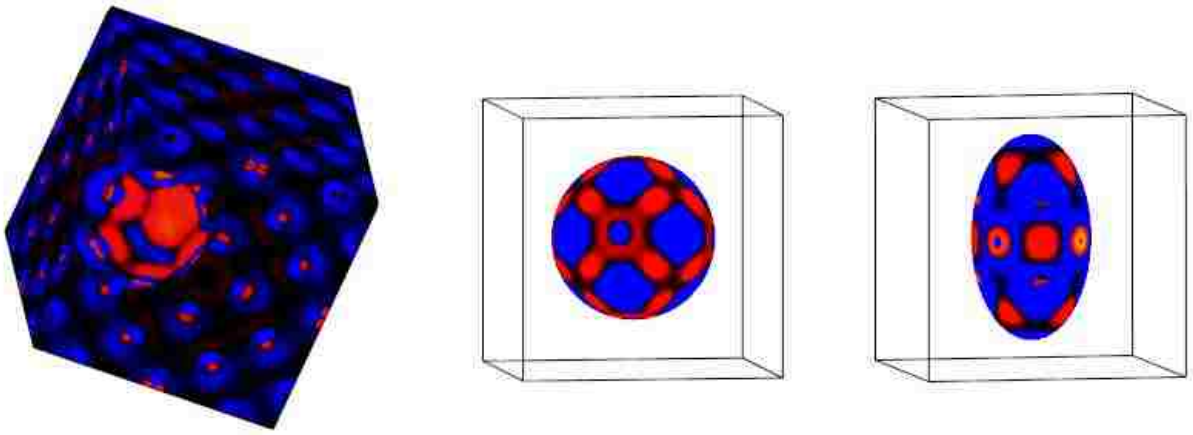


Figure 51: Volume clipping using textures

Points that lie inside the clipping object are assigned negative values and points outside the clipping object are assigned the positive values. The points on the surface of the clipping object have a distance zero. If RGBA 3D textures are utilized, these distances will be clamped to the range (0, 1). The clamping floating-point textures retain the floating-point value stored in them as it is, thus removing any round of error due to clamping in RGBA textures. For different clipping geometries there needs to be a different clipping 3D texture. The same 3D clipping object of a specific clipping geometry can be used for the more than

one number of dataset in case of MDV. A new 3D texture needs to be loaded if different clipping geometries are used with different datasets. Figure 51 shows the clipping based on the spherical clipping object.

5.7 Hardware Isosurface Extraction

This section presents an isosurface technique, which avoids an intermediate generation of polygonal geometry for the isosurface. Our approach is similar to one proposed by Westermann [47]. Since we can exploit the trilinear interpolation of 3D textures, isosurface generation using 3D textures is thus equivalent to picking up those voxels through which isosurface passes. Each element in the 3D texture is assigned the scalar value as its alpha component. Then the modified alpha test:

$$(R_{max}) > isovalue > (R_{min}) \qquad \text{Equation 53}$$

is applied to the resulting 3D texture to find the voxels containing the isosurface. Here (R_{MAX}) and (R_{MIN}) would give the maximum and minimum value respectively for the corresponding voxel, which is used to test whether the isosurface corresponding to the particular isovalue passes through the voxel or not. Modern programmable GPUs can be utilized to perform this task efficiently without the loss of performance. Since the values in the 3D texture are clamped to the region [0, 1], there is a possibility that this approach might not pick up all the areas for the resulting isosurface or the isosurface may not be continuous.

To overcome this problem we use the floating-point 3D textures as described above. For isosurface generation we can use 16 or 32 bit floating point 3D textures. Most of the modern GPU's provide trilinear interpolation for 16-bit floating-point 3D textures. This functionality has not yet been implemented for the 32 bit floating point textures. Therefore, for using 32 bit floating point 3D textures for the isosurface extraction, trilinear interpolation needs to be implemented at the GPU level.

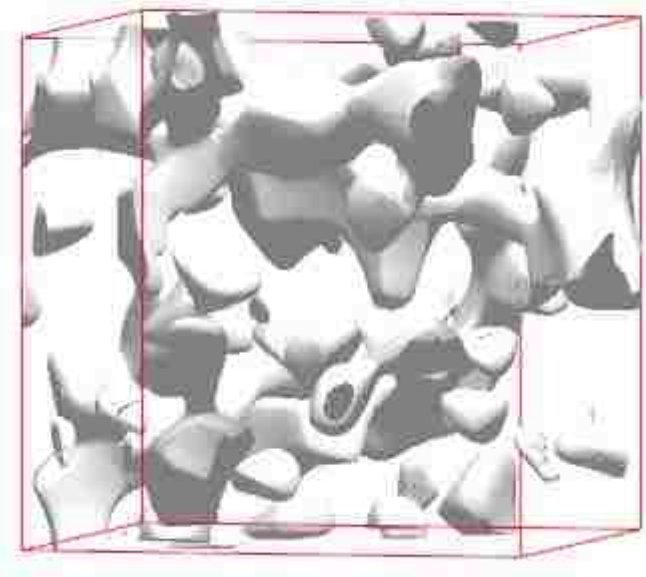


Figure 52: Isosurface extraction using textures

5.8 Hardware Based Isosurface Difference

For comparative visualization of the two isosurfaces, depth information can be utilized provided two isosurface are rendered in the same orientation. The depth information from the isosurface is extracted and stored in a 2D depth texture from both the isosurfaces. The difference in the isosurface would result in two cases; either the depth field of both the isosurface at each texel would be equal or it will be more for one isosurface than the other. If the voxel satisfies the first case than the isosurface at that point are same. In second case, isosurface with lesser depth field will be occluding the voxel with more depth field. This process can be utilized to find the difference in the isosurfaces of related datasets. This process would be useful in the scenario where visually two isosurface seems similar when projected of 2D screen but differ in third dimension.

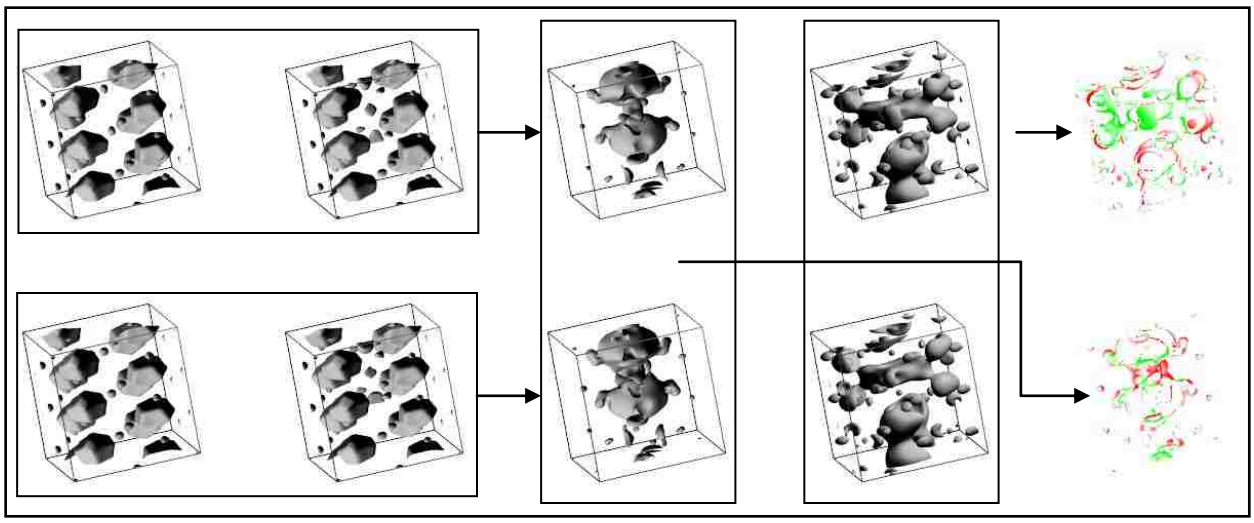
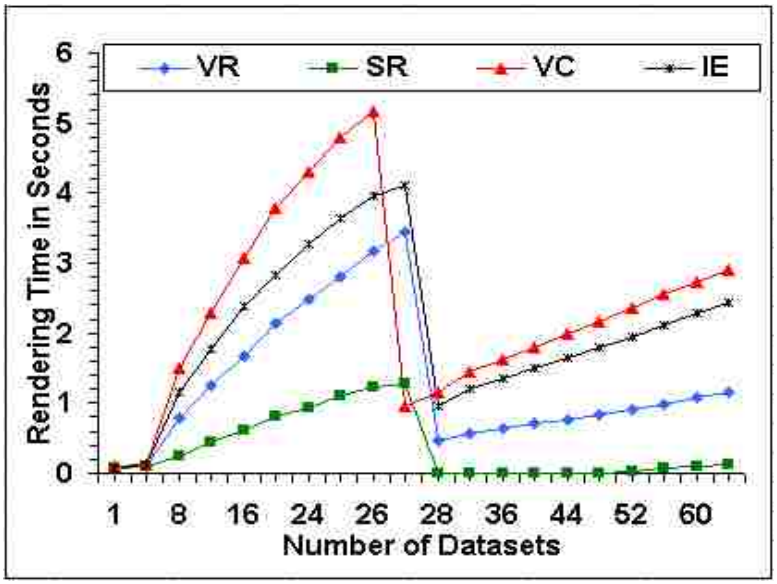


Figure 53: Isosurface difference using GPU for $MgSiO_3$ data. Arrows show the difference in the isosurface in the box. First difference is based on data and second difference is based on depth using GPU.

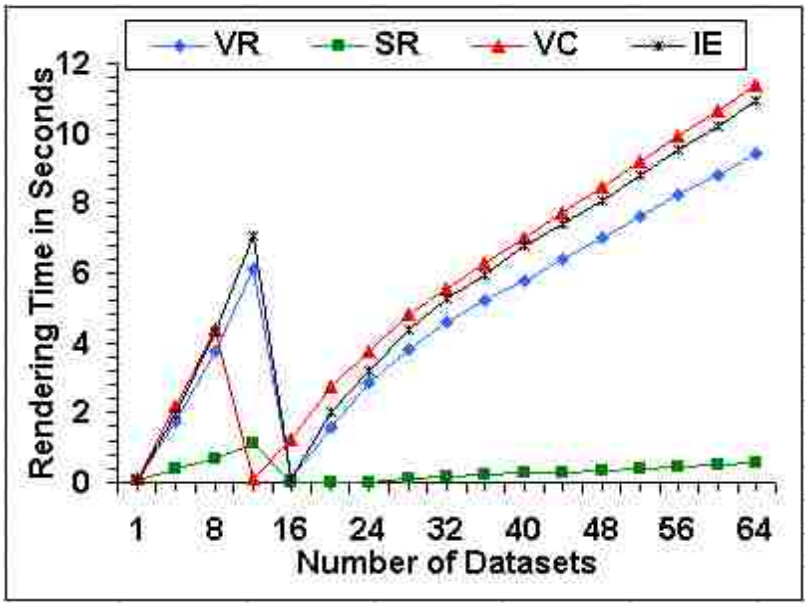
5.9 DR Based Texture Mapping

Using the DR approach, we define the threshold criteria to maintain the linear behavior of rendering time with increasing number of datasets (N). Similar to the dynamic resolution approach for the isosurface the idea is to maintain the interactive behavior of the system. Here, we have considered the rendering time of the textures.

The criteria for switching to lower resolution can be user defined. It can be based on rendering time, level of interactivity (number of frame rates) desired by the user, number of datasets or any other user defined criterion. The criterion avoids the relatively rapid increase in the rendering times by examining the *time-N* slope. For instance, when $R = \frac{T(N+1)-T(N)}{T(N)-T(N-1)} \geq 1.5$, we switch to a lower resolution mode for the *N+1* and more datasets. Here R, defines the slope, and T_N defines the rendering time in seconds for N number of datasets. The above criteria, reduces the resolution when the second transition occurs. The first transition does not satisfy the above criteria and is ignored. The condition could be modified to capture the first transition and reducing the resolution. When shading is disabled the resolution is decreased around 27 datasets (Figure 54a).



(a)



(b)

Figure 54: (a) Rendering time for various techniques without shading with DR. (b) Rendering time for various techniques with shading with DR. (VR- volume rendering, SR – surface rendering, VC- voxelized clipping, IE- isosurface extraction)

After this point, we see the linear behavior until 64 datasets. Similarly, when shading is enabled, transition to the lower resolution occurs around 12th dataset (Figure 54b). For voxelized clipping the transition to lower resolution occurs little early due to the use of

extra texture. Reducing the resolution also frees up memory and thus allowing more number of datasets to be loaded. Using the different criteria linear *time-N* slope can be obtained for all the three regions of the curve shown in Figure 55 and Figure 56.

5.10 Volume Shading

Illuminating the region of the interest in the data further facilitates the understanding. Volume shading adds the illumination term to the volume rendering .A number of techniques for combining 3D texture volume rendering with lighting and shading have been proposed [47, 59, 60]. In [59] the sum of precomputed ambient and reflected light is stored in texture volume and standard 3D texture rendering is performed. In [Hau96] the voxel gradient and volume density are stored together as 3D texture. Due to recent advances of commodity graphics hardware, texture-based rendering is able to achieve acceptable frame-rates with high image quality [32, 47, 56-58, 60]. There are number of shading models developed for volume shading namely, phong, quadratic and gouraud [59, 87-92]. They differ in the quality of the shading and computation required. By default OpenGL implements the phong model [92]. We use this model for the isosurface extracted (software visualization technique) and for the texture based rendering (hardware visualization technique) by implementing it using GPU. According to the phong model [92], light at any given point is combination of diffuse, specular, and ambient component. These three components are added to determine the illumination or color of a point or polygon. The equation for the phong model are given as below

$$I_{DIFFUSE} = I_S R_{DIFFUSE} \left(\vec{L} \cdot \vec{N}\right) \qquad \text{Equation 54}$$

$$I_{SPECULAR} = I_S R_{SPECULAR} \left(\overrightarrow{L_R} \cdot \overrightarrow{N_V}\right)^f \qquad \text{Equation 55}$$

$$I_{AMBIENT} = I_A R_{AMBIENT} \qquad \text{Equation 56}$$

$$I = I_{DIFFUSE} + I_{SPECULAR} + I_{AMBIENT} \qquad \text{Equation 57}$$

Table 2: Description of terms in lighting equation

$I_{AMBIENT}$	Ambient Intensity of Light
$I_{SPECULAR}$	Specular Intensity of Light
$I_{DIFFUSE}$	Diffuse Intensity of Light
$R_{AMBIENT}$	Ambient Reflectivity of Surface
$R_{SPECULAR}$	Specular Reflectivity of Surface
$R_{AMBIENT}$	Diffuse Reflectivity of Surface
I_S	Intensity of Light Source
I_A	Global Ambient Light Intensity
$\vec{N}$	Surface Normal (normalized)
$\overrightarrow{N_V}$	Vector form Surface to Viewer (normalized)
$\vec{L}$	Vector form Surface to Light Source(normalized)
$\overrightarrow{L_R}$	Vector in direction of Light Reflection (normalized)
I	Final intensity of Light
f	Specular Exponent

Table 1 gives the description of the terms used in the lighting equation. Calculating the surface normal is done using the gradient. For each 3D texture, a gradient texture corresponding to the scalar dataset is created which hold the normal for each voxel depending upon the scalar values. Therefore, for each 3D texture being visualized there is a gradient texture. Both these are passed to the GPU and lighting calculations are performed. This technique requires an additional texture per dataset, thus reducing the number of

datasets that can be visualized without shading to half. Since we already have a pre-computed clipping texture, applying to the isosurface or the clipped volume is straightforward. The gradient vector is generated based on the scalar values corresponding to the voxel and are stored in a gradient texture. This texture is used along with the RGBA texture to provide the shading in case of textures.

Estimating Gradient On the Fly

When using texture mapping for rendering volume data, no gradient estimation is supported in hardware. In order to implement the shading, normally, the surface normal (gradient) of the density is pre-computed and stored in the texture memory. The gradient and the volume data are loaded into texture with RGBA format. It consumes lots of the limited memory in graphics hardware, i.e., for the 512^3 volume, at least 512MB 3D texture memory is needed to store the data which is not available in most customer graphics hardware. To decrease the memory consumption is to estimate the gradient on the fly. To balance the quality and the interactivity, the central difference based gradient estimation [93] is used. For a scalar field s, the gradient at voxel (x, y, z) is calculated as follows:

$$g(x,y,z) = \nabla s(x_i, y_j, z_k) = \frac{1}{2}\left(s(x_{i+1}, y_j, z_k) - s(x_{i-1}, y_j, z_k)\right) + \frac{1}{2}\left(s(x_i, y_{j+1}, z_k) - sxi,yj-1,zk + 12sxi,yj,zk+1-sxi,yj,zk-1\right) \quad \text{Equation 58}$$

The six neighborhoods are fetched using dependent texture in the graphics hardware.

Calculating Lighting Per Pixel

In traditional texture based rendering, when slicing the volume, the gradient at the slicing polygon is trilinearly interpolated and then used to calculate the lighting. The tri-linear interpolation introduces the not normalized gradient, which is used for the lighting calculation. The hardware lighting is calculated on each vertex, and then it is interpolated to get the lighting contribution on the pixels. These result in shading artifacts and greatly

degrade the image quality. In current PC graphics hardware, the fragment program is executed after the geometry processing and on the rasterization stage, which is pixel based processing. After the gradient is estimated on the fly, it is normalized using the arithmetic operations of the fragment program. The implemented lighting model is computed as

$$I = I_d C(\vec{n}.\vec{l}) + I_s C(\vec{n}.\vec{h})^{16} \qquad \text{Equation 59}$$

,where $\vec{n}$ denotes the normalized gradient, $\vec{l}$ the diffuse light source direction, $\vec{h}$ the half-way vector, and C the color of the ray segment fetched from the transfer function. All the lighting is calculated at real time, the light source direction can be changed freely, so the dynamic lighting is supported without any performance loss. The most costly part of the lighting calculation is the exponentiation at the specular term. However, it is sufficient to approximate the exponentiation with a function that evokes a similar visual impression. Schlick [9] proposed to use the following function to approximate the exponentiation.

$$x^n = \frac{x}{n - nx + x} \qquad \text{Equation 60}$$

Quadratic shading has been proposed as a technique giving better results than Gouraud shading [88], but which is substantially faster than Phong shading [92]. Quadratic interpolation could be setup by fitting a second order surface to six points. Both the diffuse and specular light must be computed at these points and the best quality is obtained if these are interpolated separately. Furthermore, the power in the specular computation must be computed per pixel. Besides this computation, quadratic shading, including both diffuse and specular light, can be performed using four additions per pixel instead of four additions [87] and a reciprocal square root for Phong shading. The main drawback using quadratic shading has been the rather complex rasterization setup.

5.11 Performance Analysis

We consider all the cases, volume rendering with alpha blending, surface rendering with clipping, voxelized clipping and isosurface extraction for the performance analysis. Our analysis involves the rendering time for using these techniques in context of MDV. Figure

55 shows the rendering time as a function of number (N) of datasets, which are visualized concurrently. First, we consider the case of volume shading disabled. With volume rendering, we get the frame rates of 15.6 fps for the single dataset. The rendering time increases with the increasing number of datasets and we can see the non-linear behavior after 5 datasets [Figure 55]. In case of box clipping using surface rendering technique, rendering time is smaller as compared to the volume rendering. This is due to less number of texture mappings involved in the process. In surface rendering case also, we see the non-linear behavior with the increasing number of datasets, but rendering time is better, compared to the volume rendering. Even for 10 datasets, the frame rate is above 3, which is desirable for multiple datasets visualization. In case of voxelized clipping, we see similar behavior. The rendering time increases considerably after 4 datasets. After this, there is a similar trend of increasing rendering time with increasing N as in the case of volume rendering. Finally, for isosurface extraction the rendering time is similar to that for volume rendering. We can see a sudden jump in rendering time after 5 datasets and then there is a rapid increase in the rendering time. Isosurface can be changed in real time, as this technique does not generate the polygonal representation of the isosurface.

Considering the rendering time (fps) for all the texture based techniques [Figure 55], we observe two transitions. First transition is around 5 datasets and second one is around 25 datasets. These transitions clearly show the effect of the data swapping in the memory. The first transition is the results of swapping occurring between the main memory and texture memory. Second transition occurs due to swapping between the main memory and the virtual memory. Since the second transition is due to virtual memory, the rendering time increases much more rapidly after the second transition as compared to the increase after the first transition. In case of box clipping, such a high frame rate can be attributed to the fact that the number of slices being mapped is considerably less than the number of slices mapped in the case of actual volume rendering. With voxelized clipping approach, the frame rates are comparatively smaller due to the extra textures required for clipping. The first transition in rendering time occurs little early as compared to other techniques because of the extra texture.

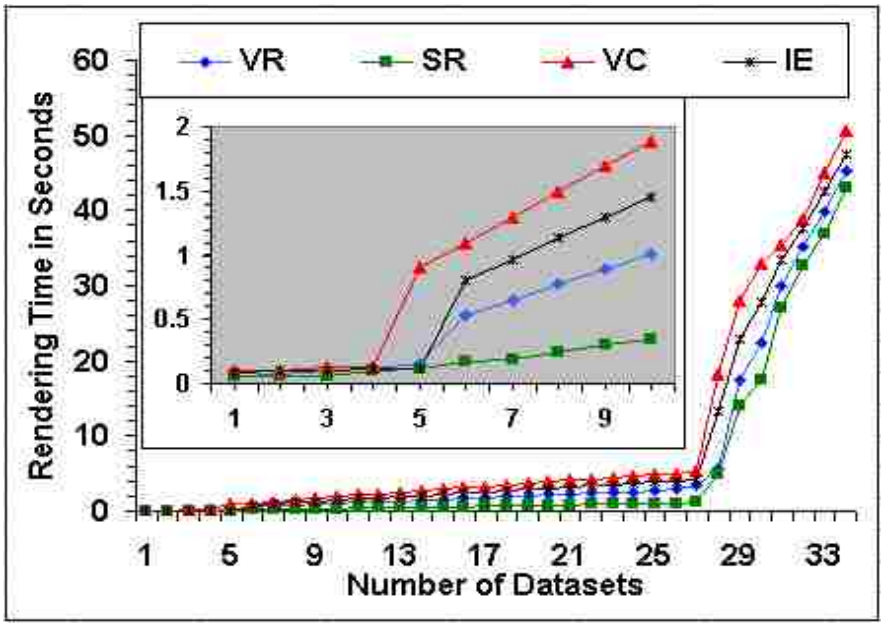


Figure 55: Rendering time for various techniques without volume shading. The inset shows the first transition in the low N regime. (VR- volume rendering, SR – surface rendering, VC- voxelized clipping, IE- isosurface extraction)

When shading is enabled, the maximum number of datasets that can be visualized together is reduced roughly to half. This is due to the requirement of one gradient texture per dataset. Figure 56 shows the rendering time for the above techniques when the shading is enabled. Although a decrease in rendering time with shading is primarily due to extra gradient texture, it is also related to the lighting calculation done by GPU for every visible fragment. The number of datasets corresponding to the transitions decreases and rendering time increases [Figure 56]. There is also the decrease in the total number of datasets that can be visualized together. The first transition occurs after 2 datasets, since while we are displaying 2 textures with shading; we are effectively dealing with 4 textures in the memory. The second transition occurs at 13 dataset in contrast to 27 datasets when shading is disabled. The frame rates are also affected in a similar way. For 4 datasets, volume rendering frame rate is around 9 fps with shading while it is 0.67 fps with shading. Similar behavior can be seen when MDV for 9 or 16 datasets is done. For 9 datasets with

voxelized clipping the frame rates are 0.47 and 0.10 with and without shading respectively. Since the maximum number of datasets that can be handled together is only 17.

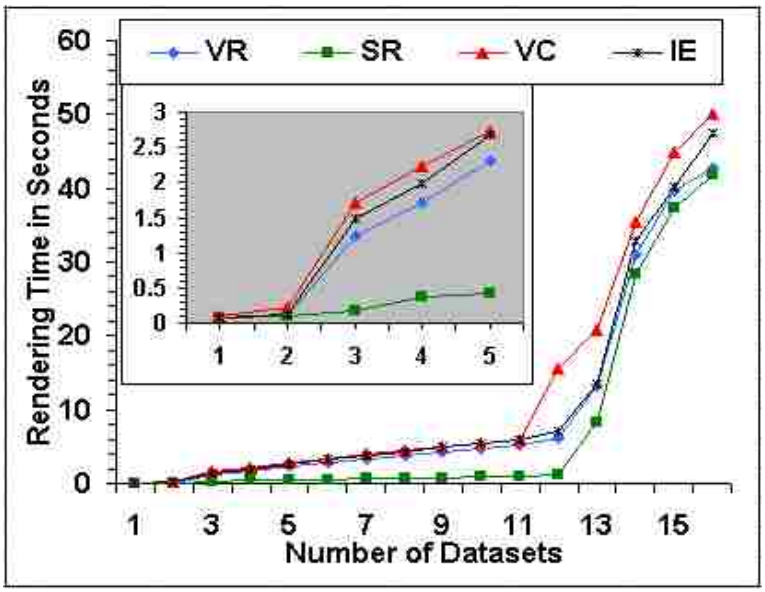


Figure 56: Rendering time for various techniques with shading. The inset shows the first transition in the low *N* regime. (VR- volume rendering, SR – surface rendering, VC- voxelized clipping, IE- isosurface extraction)

6 APPLICATIONS: VISUALIZATION OF ELECTRON DENSITY DATA

We have applied the MDV technique to visualize multiple sets of scalar volume data produced by computer simulations of real materials [20]. The data used in this study represent the electron charge density distributions in two important materials, MgO (magnesium oxide) and $MgSiO_3$ (magnesium silicate). These materials are thought to be the most abundant constituents of Earth's interior (the lower mantle regime representing the depth range of ~400 km to ~2900 km). A given material system can be viewed as composed of atoms and electrons. While atoms occupy well-defined positions in a crystalline material, electrons (primarily, valence electrons) are distributed over extended regions within the crystal. Simulation normally uses a finite size box (called a supercell) containing a couple of hundreds of atoms and about the four times number of electrons with the periodic boundary conditions. Simulating a real material system is solving a many-particle problem, where particles are interacting atoms and interacting electrons. The solution to this problem means solving numerically the fundamental equations of quantum mechanics. Our data sets represent electron distributions for perfect crystal with all atoms at their well-defined positions as well as defective crystals with an isolated vacancy (one atom missing). The vacancy can be of different types, e.g., Mg, Si or O vacancy in magnesium silicate. Furthermore, it can be in a neutral state or charged state (positively or negatively charged). Vacancies are important for ionic diffusion since an ion can hop from one vacancy to another. A vacancy defect or a migrating ion in a crystal distorts both the atomic and electronic structures, more profoundly, in its immediate neighborhood. It is very interesting to understand how these structures compare among different defects. Simulations are also performed over a wide pressure range covering the Earth's deep interior. Multiple data sets of interest might correspond to different samples, pressures, or temperatures. Moreover, in a dynamical system like liquid the atomic and electronic structures vary with time so multiple datasets correspond to different time snapshots. MDV is expected to be useful in gaining insight into the electronic structures (how electronic charge is distributed in three-dimensional space) of defective MgO and $MgSiO_3$ crystals by

visualizing multiple sets of relevant data simultaneously. We present some case studies of applications in the subsequent sections.

6.1 Simulation Data

Electronic charge density data from materials simulations are defined on the regular grid. Simulations were performed using the parallel code known as PWscf (Plane-Wave Self-Consistent Field), which is based on density functional theory describing electron-electron interactions [20, 94, 95]. A supercell containing 216 atoms, which is a 3x3x3 (Figure 57) replication of the 8-atom conventional cubic cell was used in the case of MgO. The crystal structure of MgO a face centered cubic. The grid size in case of MgO is 144 x 144 x 144 and for liquid case, it is 121 x 121 x 121. In the case of $MgSiO_3$ (in its perovskite and post perovskite phases), the supercells consisting of 80 and 60 atoms were used, respectively. For both systems, to the point defects (vacancies) and migrating ions were simulated. For instance, a defective supercell contains only one charged defect of given type at a time. A cation vacancy was created by removing an Mg (or Si) ion leaving two (or four) valence electrons whereas an anion vacancy was formed by removing an O ion together with eight valence electrons. Thus, the net charges of the Mg, Si and O vacancies are –2e, -4e and 2e, respectively. The grid size in the case of $MgSiO_3$ is 120 x 90 x 96 in perovskite and 90 x 96 x 64 in case of post perovskite.

The simulation model generates the atomic structure and electronic structure corresponding to the vacancy defects and the migrating ion defect at varying parameters (pressure, volume and temperature) which are determined while conducting the simulation. To understand the behavior of the system, different datasets (generated with different configuration parameters) are visualized simultaneously. The user might want to look the data at different temperature or at different pressures or at different time steps between initial and final configuration. Apart from these, the user might be interested in the looking at the vacancy defect in the system. Point defects such as cation and anion vacancies have been area of research for long time. The detailed knowledge about energetic of the system and structural properties is key in understanding the how these defects affect

various physical properties of the system. Our goal is to visualize these massive datasets to gain insight into the electronic structures at varying parameter (temperature, size, pressure) and their response to the presence of defects in the system. For visualization of these datasets, we apply the MDV technique to the dataset.

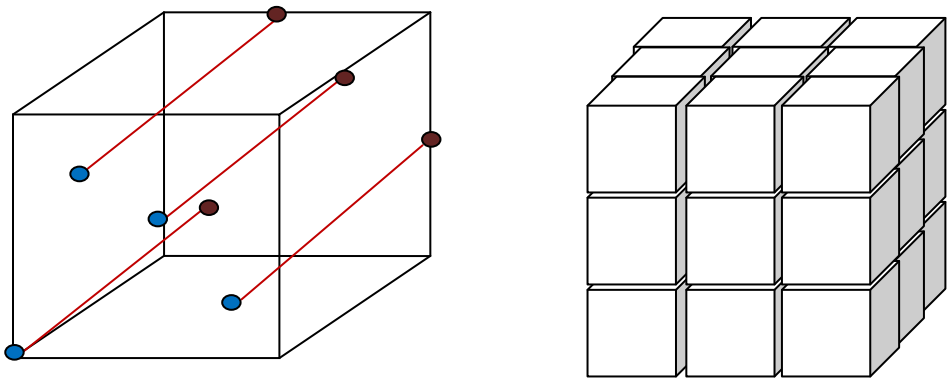


Figure 57: MgO super cell (red circles indicate the positions of the oxygen atoms and blue indicate the positions of the magnesium atoms

6.2 Simulation Data Visualization

We present visualization results for the electronic charge density data for MgO and $MgSiO_3$.

6.2.1 MgO Data

In MgO, Mg atoms give up their valence electrons, which along with those from O atoms are primarily localized near O ions thereby making the oxide highly ionic in the nature. Charges associated with the Mg and O vacancy defects are thus of opposite sign. These charged defects are expected to induce a strong distortion of the valence electron clouds on neighboring ions. The calculated defect-induced electronic structures are explored in details by visualizing the differences in 3D valence electron density between defective and perfect systems. However, looking at such difference makes sense only for the unrelaxed

systems, which are considered here. In relaxed vacancy systems, the charge distribution is also strongly influenced by ionic displacements. Figure 58 shows the atomic structure distortion due to point defects (Mg and O vacancies shown by green spheres). Each atom is displayed as a sphere whose size is proportional to the magnitude of the displacement. The orientation of the line starting from the surface of the sphere represents the direction of the displacement.

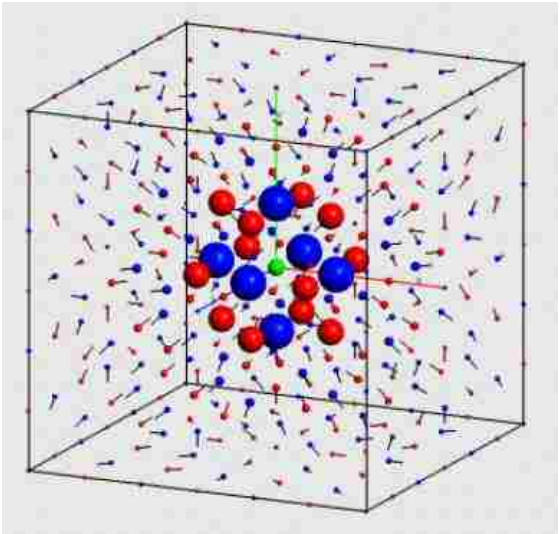


Figure 35 (a)

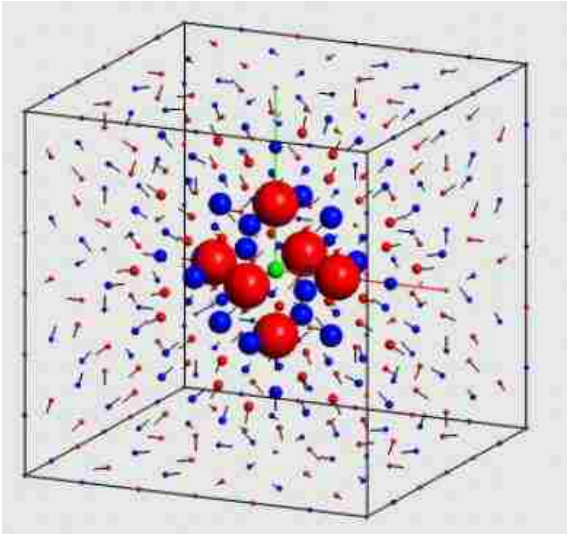


Figure 35 (b)

Figure 58: Visualization of atomic displacements induced by a) the Mg^{2+} vacancy and b) the O^{2-} vacancy in the 216-site system. The size of the sphere and the orientation of the line, respectively, represent the magnitude and direction of the displacement of a given atom (Mg or O) relative to its position in the perfect crystal. Note that some minimum size is given to each sphere to make it visible on display. The Mg and O atoms are displayed by red and blue sphere, respectively, whereas the vacancy site is displayed by green sphere. In Figure a) for the Mg^{2+} vacancy, the nearest (N) neighboring atoms (the first shell) are the largest blue spheres, the next-nearest (NN) neighboring atoms (the second shell) are the largest red spheres, the next-next-nearest (NNN) neighboring atoms (the third shell) are the second-largest blue spheres and so on. [96]

Visualization of the fully relaxed vacancy systems reveals that relaxations of atoms of different types at different distances from the defect site differ in the magnitude as well in the direction. The spheres close to the defect site are relatively large implying that the most relaxation occurs in the immediate vicinity of the defect site. Figure 59 (top) show the visualization snapshot of the charge density differences (defective – perfect) for both point defects (vacancies). Two color-mapping scales are used. A fine-level scale uses the red and blue colors to represent the positive and negative differences with magnitude up to 0.002 (in units of $Å^{-3}$) whereas a coarse-level scale adds green color component to red and blue colors to map the positive and negative differences with magnitudes higher than 0.002.

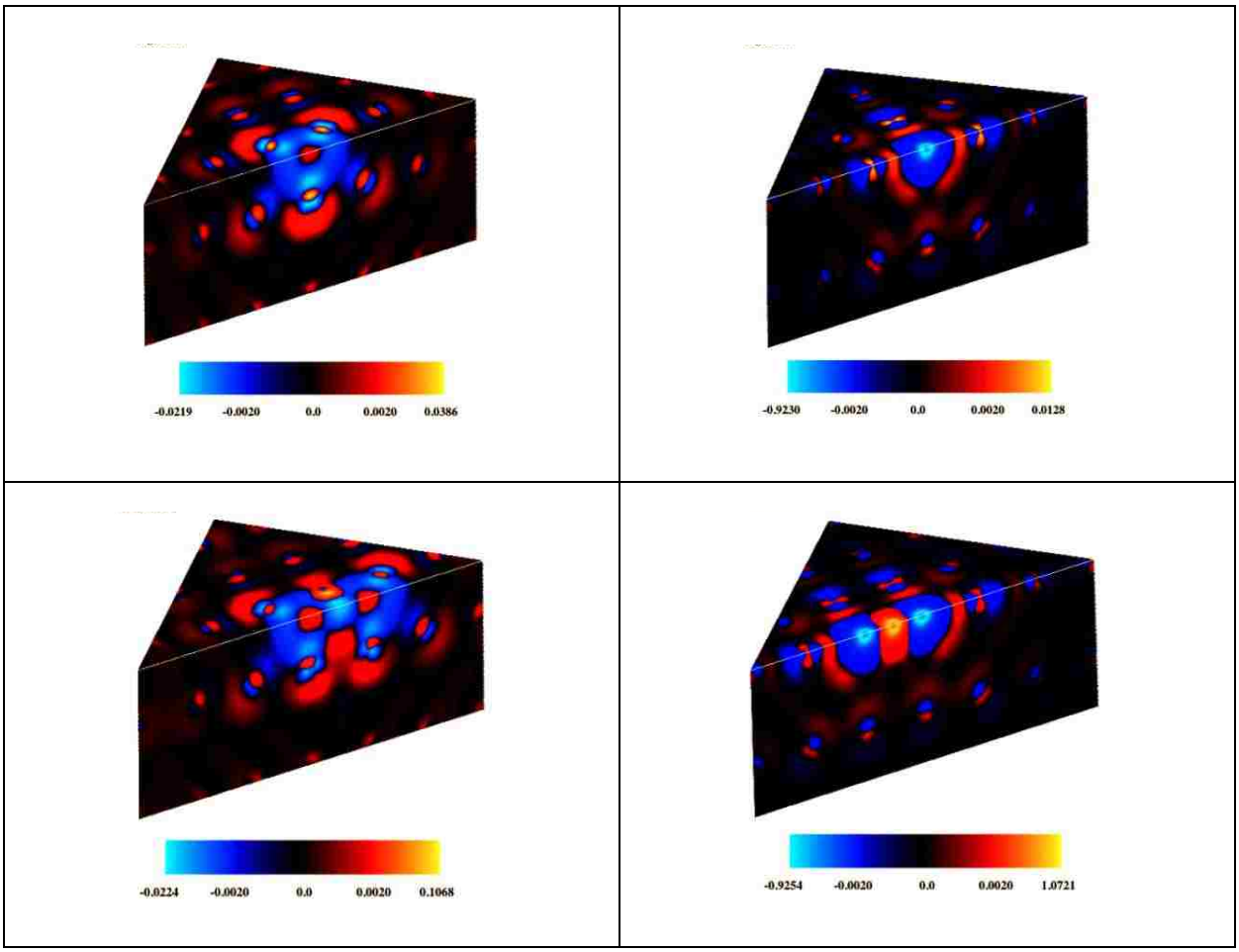


Figure 59: Top: Mg and O vacancy defect in MgO. Bottom: migrating defect in MgO for migrating Mg and O ion

Thus the black-to-red-to yellow range maps the regions where electrons are deposited (moved in) and the black-to-blue-to-cyan range maps the regions where electrons are depleted (moved away). Substantial distortions occur in the electron clouds of neighboring ions of each defect. For the Mg vacancy, the valence electrons on neighboring O ions repelled by the negatively charged defect as displayed by blue regions surrounding the O sites and red regions towards Mg sites. Nearly the same behavior (with much smaller degree of distortion) is seen around the O sites further away from the defect. Also, note that the mostly blue regions surrounding the defect site indicate depletion of electrons. For the positively charged O vacancy, the electrons from the neighboring O ions are pulled towards the defect site; more along (110) directions than along (100) directions. Note that the mostly red regions surround the defect site (which is the blue region). The oxygen p orbital pointing towards the vacancy is directly involved in the charge redistribution in both cases; the effects are reflected in blue regions, which are mostly aligned towards the defect site.

The difference valence electron density (defective – perfect) is also visualized for a migrating ion when it is held at the saddle point, midway between initial and final vacancy sites for both Mg and O ions (Figure 59). The electronic structures of the neighboring ions remain essentially unchanged, compared to those for non-migrating defects, shown in Figure 59. The structure near the migrating ion clearly represents the existence of two half-vacancies separated by the ion itself. The defect-induced electronic structures in general are found to remain qualitatively unaffected by pressure although the sizes and intensities of red and blue regions vary to some extent.

We have also visualized the density data for MgO liquid. Figure 60 shows the electronic structures of liquid MgO at different temperatures. Since it is liquid, we can clearly see there is no organized structure. For simulation purposes, the MgO solid was first melted at very high temperatures (10000 Kelvin). Then the liquid was slowly quenched to a desired lower temperature. The simulations were carried out at different temperatures above the melting point of MgO, which is 2000 kelvins. Visualization snapshot highlights three different regions of density distribution. The red regions represent the high concentration of electrons defining the positions of the O atoms in the supercell and the blue region is the area of low electron charge density.

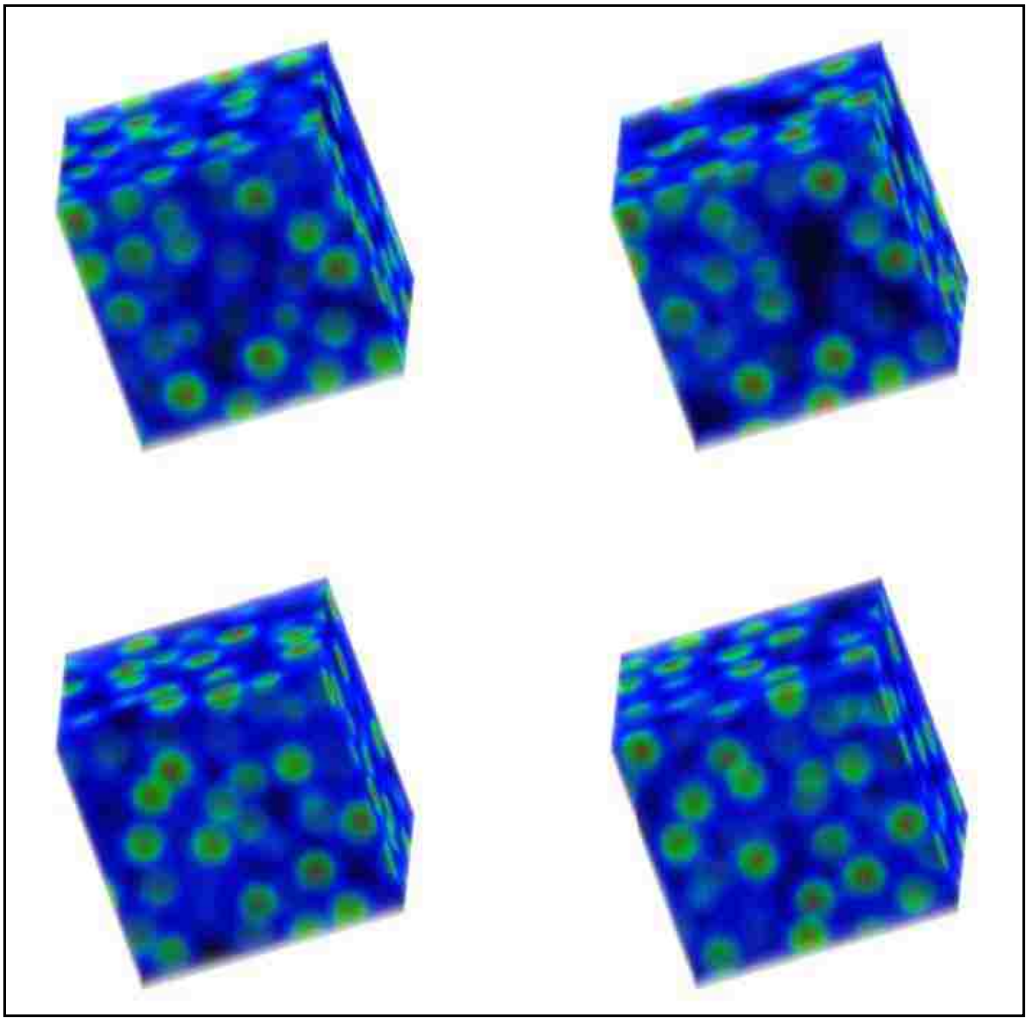


Figure 60: Texture based MDV off our sets of electronic charge density distributions of liquid MgO. The color and opacity values for each pixel are based on the density value associated with that pixel: a multiscale RGB color mapping is used. B represents values from 0 to 0.05, G is added to represent values up to 0.4 and then R is increased and both B and G are decreased for higher values

Figure 61 shows the isosurface at solid MgO at different compression levels. It shows the distribution of the electronic charge density changes at the spatial location at different compression level. The isosurface shows the distribution of electronic charge density distribution around O atom. It is important to note how the distribution of the charge changes at different compression level. The empty region between the isosurface tells location of the Mg ion in the structure.

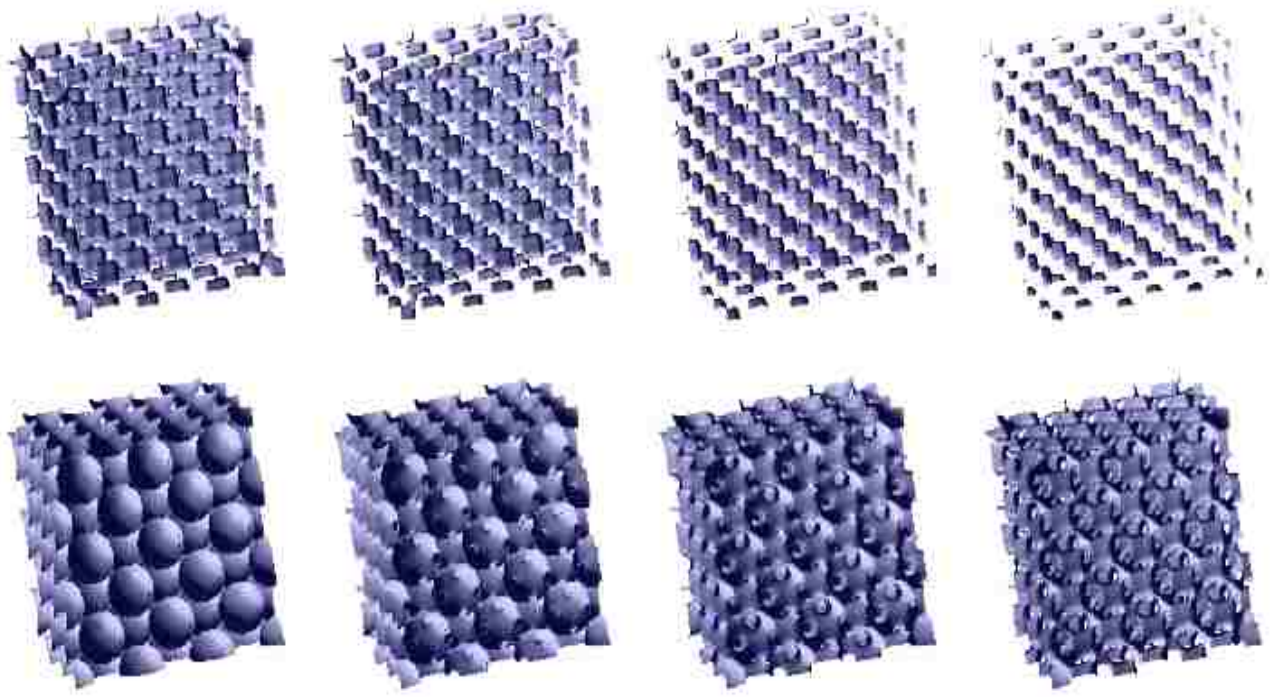


Figure 61: Isosurfaces for eight sets of the electronic charge density data for solid MgO at different compressions (compression increasing from the lower left to the upper right). The structures represent the charge distribution around o ion sites.

6.2.2 $MgSiO_3$ Data

In $MgSiO_3$, Mg and Si atoms give up their valence electrons, which along with those from the O atoms are primarily localized near the O ions. The charges associated with the cationic (Mg and Si) and anionic (O) vacancy defects are of opposite signs. Similar to the MgO system these vacancy defects induce a strong distortion. Figure 62 show the atomic displacement in case of $MgSiO_3$ post perovskite. The Figure 62 shows the distortion due to vacancy defect for $MgSiO_3$. Small black spheres show the defect sites. The size and direction associated with the sphere shows the amount of distortion due to defect similar to the MgO structure. In case of Si defect, the distortion is more as compared to the Mg defect, because of 4 valence electron in case of Si as compared to two valence electron in Mg.

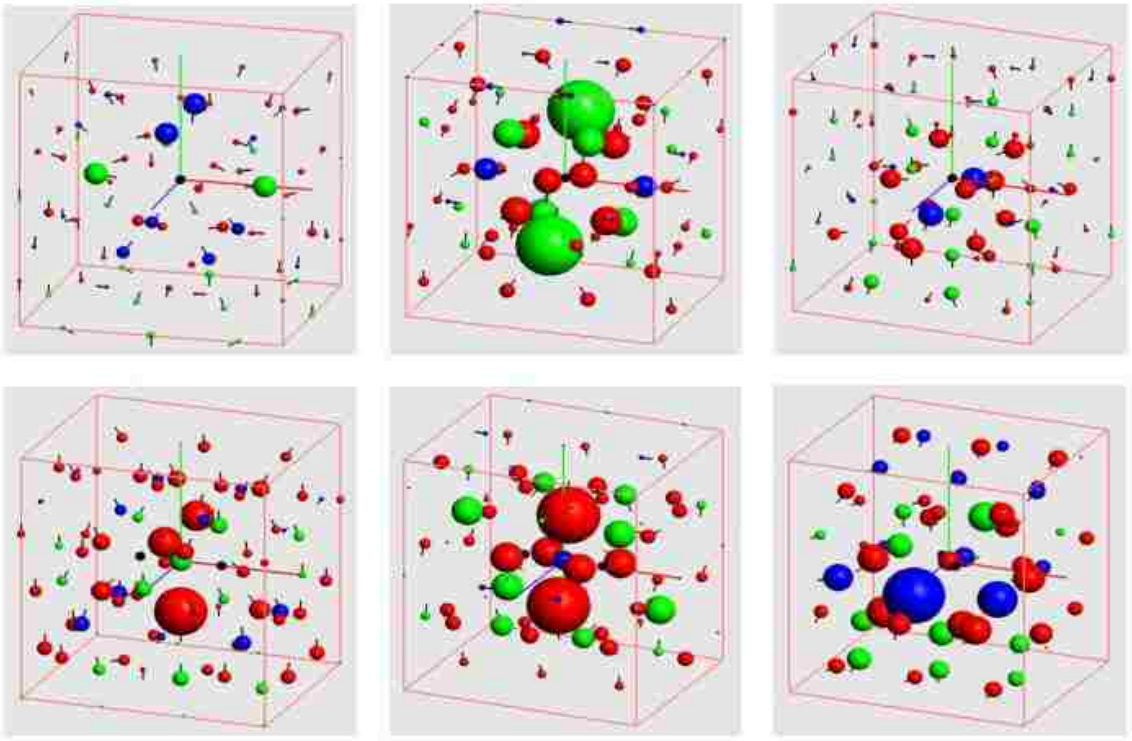


Figure 62: Visualization of atomic displacements. Upper row: Mg (left), Si (center), and O (right) vacancies in the 60-site post-perovskite system at 120 GPa. Green, blue and red spheres represent Mg, Si and O atoms, respectively. A black sphere at the center of the supercell indicates the defect (vacant site) site. Lower row: Mg (left), Si (center) and O (right) migrating ions located at the center of the supercell. Adjacent vacancy sites are indicated by two black spheres located on each side of the center along the line of migration.

Figure 63 shows the electronic charge density distribution around defect sites corresponding to the atomic distortion shown in Figure 62. Since valence electrons from Mg, Si and of nearby O atoms are primarily localized near O ions. Defect-induced distortions are mostly manifested around O atoms for both vacancy and migrating ion. The defect-induced structures show similar pattern as MgO in case of vacancy defect and migrating ion defect.

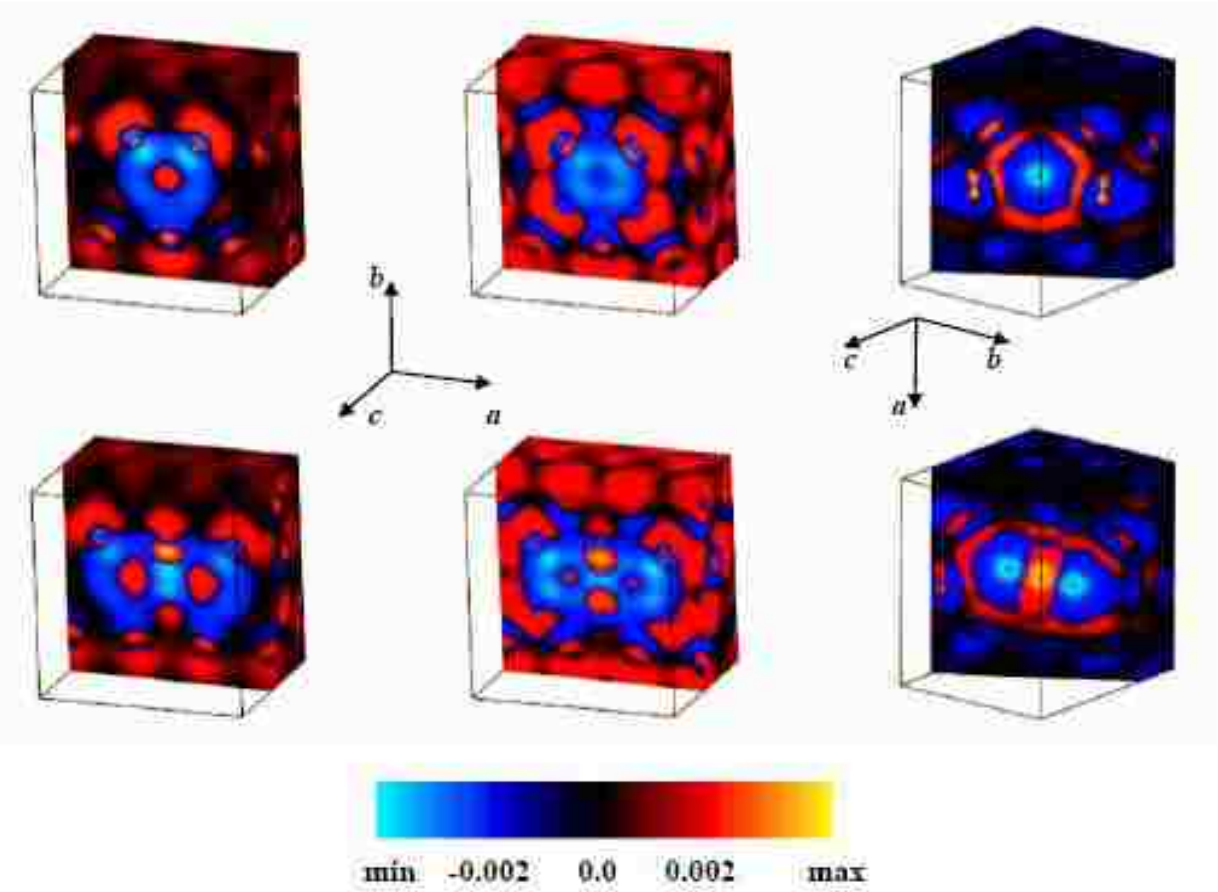


Figure 63: Visualization of electron density difference (in units of $Å^{-3}$) in $MgSiO_3$ post-perovskite. Upper three images are for vacancies and lower three images are for migration: Mg (left), Si (center) or O (right). In each case, the vertical surface represents the clipping plane passing through the defect side or migrating ion site, which is located at the center of the image. Note that the plane also contains the migration direction.

Figure 45 shows the electronic charge density difference distribution in the initial and final configurations for fixed vacancy and migrating ion. Note that the initial and final configurations are not very different with respect to each other. This is important because the defect-induced electronic structural distortions are not affected by ionic relaxations in the supercell.

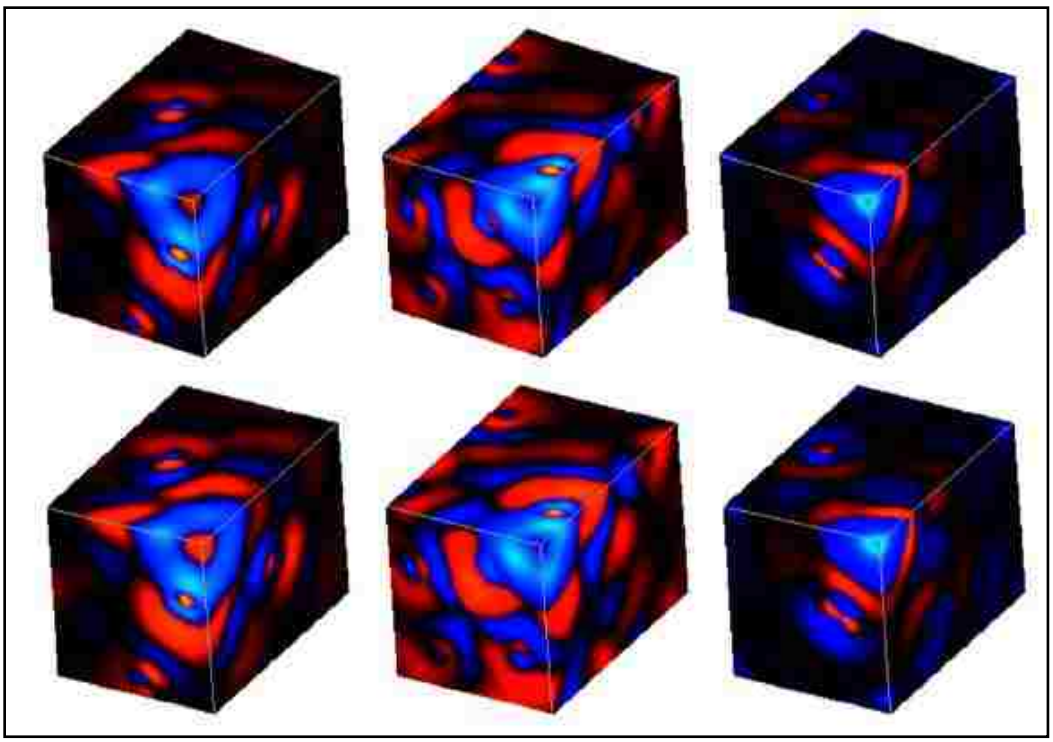


Figure 64 Visualization of electron density difference (in units of $Å^{-3}$) induced by the Mg^{2+} (left), Si^{4+} (centre) and O^{2-} (right) vacancies in the 160-site system at zero pressure (upper three) and 150 GPa (lower three). A box clipping is used to show the interior of the volume by removing the exterior portion of the volume data. In each image, three visible surfaces represent the planes perpendicular to three axes intersecting at defect site, which is the front upper right corner.

Figure 64 shows the charge density differences (defective–perfect) for all three point defects in $MgSiO_3$ perovskite at 0 and 150 GPa. We have applied box clipping with surface texture rendering in this case. The black-to-red-to-yellow range maps the regions where electrons are deposited (moved in) and the black-to-blue-to-cyan range maps the regions where electrons are depleted (moved away). Substantial distortions can be seen in the electron clouds of neighbouring ions of each defect. The Mg and Si vacancy result in the valence electrons from the O ion to be repelled from the defect site. This is displayed as blue regions surrounding the O sites and the red regions towards the Mg sites. Figure 65 shows the $MgSiO_3$ before and after optimization. The lower layer shows the isosurface before optimization. Upper layer shows the structure after optimization.

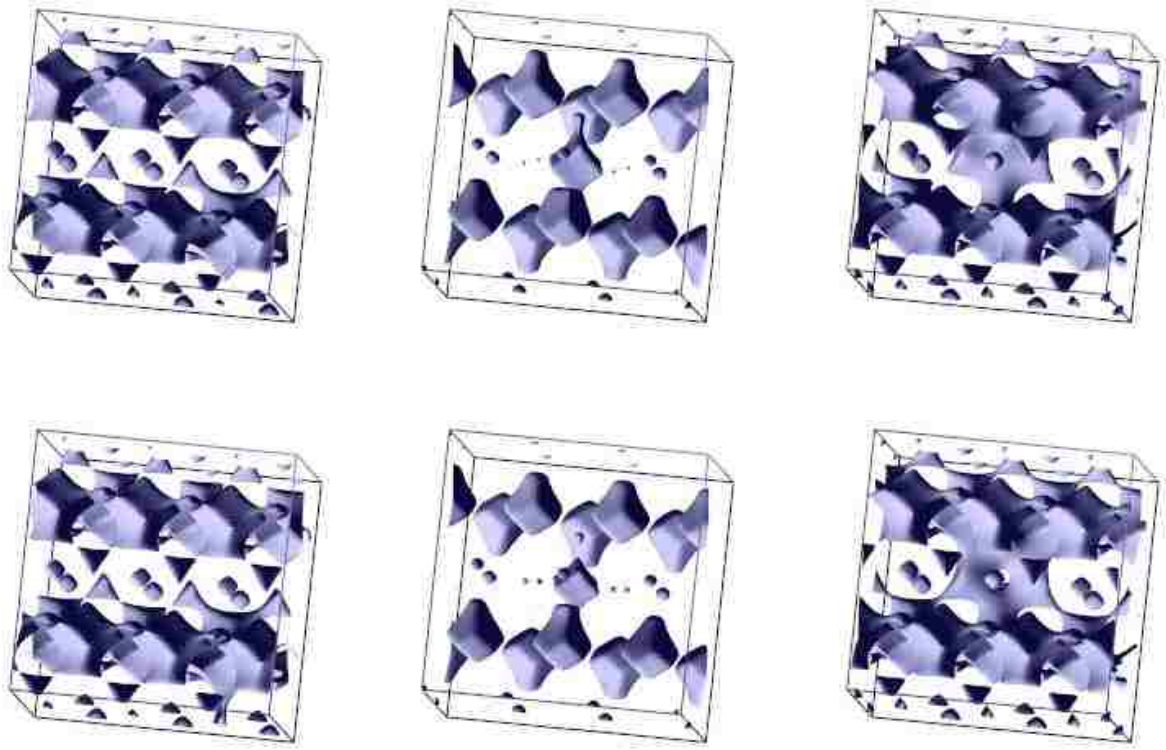


Figure 65: Isosurface Extraction of MgSiO3 showing Mg, Si and O vacancy from left to right with isovalue 0.021. Top layer shows the isosurface after the optimization and lower layer shows the isosurface before the optimization

7 CONCLUSIONS AND FUTURE DIRECTIONS

7.1 Conclusions

In this PhD thesis, we have introduced multiple dataset visualization (MDV), which is simultaneous processing and rendering of two or more datasets. This work shows that MDV is an effective approach for doing comparative analysis and finding correlations among datasets. The major bottlenecks in MDV are memory requirement and interactivity, primarily, because of relatively large amounts of data to be processed at the same time. We have proposed a simple, innovative framework for multiple dataset visualization. The framework consists of three major components, namely, data loading/preparation, visualization and rendering. In our implementation of this framework, each major component runs in at least one independent thread (and in future extensions, potentially an ensemble of threads). The user input determines the interaction between different components. The MDV engine parses the user input and sends it to one of the components for handling the user request. The analysis module presented in the framework, is responsible for providing simple analysis technique such as isosurface difference. In the future, it can be extended to perform more complex techniques.

The rendering component of the framework utilizes spreadsheet-based approach with off-screen rendering and texture based rendering. It supports different display options (normal, comparative and analysis). The off-screen rendering also allows manipulation of a portion of the display. It optimizes the use of GPU and releases it for handling graphics intensive operations. For the system used for performance analysis, off-screen rendering allowed better utilization of the available bandwidth between north-bridge and GPU.

For the visualization component, we have proposed techniques such as data coherency, multi-resolution and texture-based rendering approaches to address MDV issues such as storage, computation and interactivity. Our all-in-memory (AIM) approach for MDV isosurface extraction keeps multiple datasets under consideration and the corresponding geometric data in the memory. When the available main memory is completely used, adding new datasets results in rapid performance degradation. Some of the data format

such as HDF5 [97] could be used in future to alleviate the memory issues to certain constraint by allowing efficient down-sampling and paging. The only-polygons- or points-in-memory (OPIM) approach reduces the memory usage by keeping the scalar data in the memory only until the geometric data are extracted. This allows us to process relatively large number of datasets. The adaptive data coherency and multiresolution schemes extend the isosurface approach to address storage and computation issues. The data coherency scheme exploits the similarities among datasets to approximate the isosurfaces from similar regions of datasets (non-reference) using the already generated polygon data for one or more reference dataset(s). Only those non-reference data regions blocks (i.e., octree nodes) which differ from the reference blocks are directly processed. Our intra/inter-dataset multiresolution scheme processes the selected portions of each data volume or the selected entire data volumes at high (original) resolution and renders the rest data at varying levels of reduced resolution. It includes dynamic, hybrid and mixed multiresolution. The texture-based approaches adopted for MDV include surface rendering coupled with clipping, isosurface extraction and volume rendering. Performance measurements were carried out by considering up to 64 set of scalar volume data with size of 256^3 for a normal desktop environments. With appropriate user-defined threshold criteria for these techniques, we show that interactive frame rates are achievable for moderate number of dataset in desktop environments.

The main objective of this work is to provide a simplified framework for the MDV process. It primarily differs from the earlier works [10, 17-19, 25, 44, 98] in conceptualizing MDV as an important area in visualization with some specific issues that need to be addressed. We have provided in-depth analysis of various visualizing techniques (isosurface and textured based) for MDV.

The proposed framework is expected to serve a foundation upon which future extensions can be made towards supporting visualization of a relatively large number of datasets. We have successfully applied our MDV system to understand important differences in the electron density distributions of materials at different temperature and pressure conditions.

7.2 Future Directions

There is still a lot to be done for full development of MDV. Below are some directions for future work:

Extending MDV to Parallel Environment

We have so far focused on the desktop environment with the goal of providing interactive MDV in a resource limited environment. With an increasing trend in grid and cloud computing and easy availability of parallel resources, there is a need of adopting MDV in parallel environment. Some challenges with parallel implementation are load distribution and composite display. In our sample implementation (explained below), we allocate one core for each dataset. Since the size of the isosurface mesh can vary significantly, dividing by the number of dataset to the available resources might not be the ideal solution and may cause load-balancing problem. Another issue is providing single display – bringing the output from each node to single display and to provide interactive user experience could be challenging. The off-screen rendering approach using the FBO in our rendering module can be used.

Extending Visualization and Rendering Techniques

The present framework has been developed around isosurface and texture rendering techniques. It would be interesting to study other visualization techniques from MDV perspective. It is important to identify problems associated with the visualization techniques and find solutions. Some of the techniques developed here such as data coherency and multiresolution can be extended to other visualization techniques. For rendering, we use off screen buffer so that only portion of the display that has changed could be updated without re-rendering all the datasets. With the rapid improvement in graphics card (such as availability of geometry shaders) there is a scope to improve the rendering mechanism. In addition, with recent work in the area of cell processor that makes the ray tracing process 200 times fast, there is a possibility to provide images that are more realistic.

Performance Improvement

Our major focus in this work has been on finding ways to improve the performance and making MDV interactive. With the availability of powerful graphics card, there is a possibility where the newly available features could be utilized. In addition, study of other techniques such as compression or wavelet techniques from MDV perspective could identify some of areas of the improvement. There is also a possibility to include the other techniques such as compression [68, 99-101] for floating point and geometric data. Wavelets could provide the performance improvement.

Application Areas

In this work, we have successfully used MDV to visualize electron density data produced by materials simulations. Nevertheless, MDV can be applied to visualize other scalar data such as MRI, confocal, seismic tomography and computational fluid dynamics. Tools such as iVici, VisCose, EcoLens and RIVA are good examples of visualization scalar data in different domains. In addition, MDV framework can be extended to address MDV related issues in different domains. This will allow the researchers in the other domains to focus on domain specific research rather than spending time to address MDV related issues.

REFERENCES

[1] C. D. Hansen and C. R. Johnson, *The Visualization Handbook*: Elsevier Academic Press, 2005.

[2] "http://eagereyes.org/," 2009.

[3] Y. Livnat, S. G. Parker, and C. R. Johnson, "Fast Isosurface Extraction Methods for Large Imaging Data Sets," *Handbook of Medical Image Processing, Academic Press,* 2000.

[4] W. E. Lorensen and H. E. Cline, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," *ACM Computer Graphics,* vol. 21, 1987.

[5] M. Levoy, "Efficient Ray Tracing of Volume Data," *ACM Transactions on Computer Graphics,* vol. 8, pp. 245-261, 1990.

[6] L. Westover, "Splatting: A Parallel, Feed-Forward Volume Rendering Algorithm." vol. PhD Dissertation: Univ. of North Carolina at Chapel Hill, 1991.

[7] P. Lacroute and M. Levoy, "Fast Volume Rendering Using Shear-Wrap Factorization of the Viewing Transformation," *ACM Siggraph, Computer Graphics Proceedings,* 1994.

[8] B. Cabral, N. Cam, and J. Foran, "Accelerated volume rendering and tomographic reconstruction using texture-mapping hardware.," *IEEE Proceedings of the 1994 symposium on Volume visualization,* pp. 91-98, 1994.

[9] M. MeiBner, J. Huang, D. Bartz, K. Mueller, and R. Carwfis, "A Practical Evaluation of Popular Volume Rendering Algorithms," *IEEE Symposium on Volume Visualization,* 2000.

[10] Woodring, "High Dimensional Direct Rendering for Time-Varying Volumetric Data," in *Department of Computer Science*: The Ohio State University, 2003.

[11] G. Khanduja and B. B. Karki, "A Systematic Approach to Multiple Datasets Visualization of Scalar Volume Data," in *International Conference on Computer Graphics Theory and Applications, (GRAPP'06)*, 2006.

[12] J. Woodring and H.-W. Shen, "Multi-variate, Time-varying and Comparative Visualization with Contextual Cues," *IEEE Transaction on Visualization and Computer Graphics,* vol. 12, 2006.

[13] A. J. Hanson and P. A. Heng, "Four-Dimensional Views of 3D Scalar Fields," *Proceedings of IEEE Visualization,* pp. 84-91, 1992.

[14] A. J. Hanson and R. A. Cross, "Interactive Visualization Methods for Four Dimensions," *Proceedings of IEEE Visualization,* pp. 196-203, 1993.

[15] C. Bajaj, C. Pascucci, G. Rabbiolo, and D. Schikore, "Hypervolume Visualization: A Challenge in Simplicity," *Proceedings of IEEE Symposium on Volume Visualization,* pp. 95-102, 1998.

[16] G. Khanduja and B. B. Karki, "Multiple Datasets Visualization with Isosurface Extraction," *International Conference on Visualization, Imaging, and Image Processing,* 2006.

[17] M. Levoy, "Spreadsheet of Images," *Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH,* pp. 139-146, 1994.

[18] E. H.-h. Chi, P. Barry, J. Riedl, and J. Konstan, "A Spreadsheet Approach to Information Visualization," *ACM Symposium on User Interface Software and Technology,* pp. 79-80, 1997.

[19] T. J. Jankun-Kelly and K.-L. Ma, "A Spreadsheet Interface for Visualization Exploration," in *Proceedings of the 11th IEEE Visualization 2000 Conference (VIS 2000)*, 2000.

[20] Codes, "Electronic calculation packages: PWScf (www.pwscf.org) and VASP (cms.mpi.univie.ac.at/vasp).", 2005.

[21] http://commfaculty.fullerton.edu/lester/writings/letters.html

[22] "http://score.rims.k12.ca.us/activity/worth/," 2009.

[23] D. A. Aoyamaa, J.-T. T. Hsiaoa, A. F. Cárdenas, and R. K. Pon, "TimeLine and visualization of multiple-data sets and the visualization querying challenge," Journal of Visual Languages & Computing, vol. 18, pp. 1-21, 2007.

[24] L. Bavoil, S. P. Callahan, P. J. Crossno, J. Freire, C. E. Scheidegger, C. T. Silva, and H. T. Vo, "VisTrails: Enabling Interactive Multiple-View Visualizations " *IEEE Visualization,* 2005.

[25] E. Brodsky and W. Block, "Interactive Visualization of Time-Resolved Contrast-Enhanced Magnetic Resonance Angiography (CE-MRA)," *Proceedings of the 14th IEEE Visualization Conference (VIS'03),* 2003 IEEE.

[26] G. Wallace, M. Hibbs, M. Dunham, R. Sealfon, and O. L. Troyanskaya, K., "Scalable, Dynamic Analysis and Visualization for Genomic Datasets," *Parallel and Distributed Processing Symposium, 2007,* pp. 1 - 6, March 2007.

[27] A. F. Hasler, K. Palaniappan, M. Manyin, and J. Dodge, "A high performance interactive image spreadsheet (IISS)," *Computers in Physics,* vol. 8, pp. 325-342, 1994.

[28] D. Voorhies, D. Kirk, and O. Lathrop, "Virtual Graphics," *SIGGRAPH,* 1988.

[29] "www.opengl.org," 2007.

[30] "msdn.microsoft.com/en-us/directx/default.aspx," 2007.

[31] A. F. Möbius, *Der barycentrische Calcül*, 1827.

[32] B. Wilson, K.-L. Ma, and P. S. McCormick, "A Hardware-Assisted Hybrid Rendering Technique for Interactive Volume Visualization," *Proceedings of Volume Visualization and Graphics Symposium,* 2002.

[33] H. A. V. Martinez, "Texture Caching," 2006.

[34] A. Moerschell and J. D. Owens, "Distributed Texture Memory in a Multi GPU Environment," in *Graphics Hardware*, 2006.

[35] K. Potter, A. Gooch, B. Gooch, P. Willemsen, J. Kniss, R. Riesenfeld, and P. Shirley, "Resolution Independent NPR-Style 3D Line Textures," *Computer Graphics Forum Journal compilation © 2009 The Eurographics Association and Blackwell Publishing,* vol. 28, pp. 56 - 66, 2009.

[36] S. D. Roth, "Ray Casting for Modeling Solids," *Computer Graphics and Image Processing,* pp. 109-144, 1982.

[37] M. Levoy, "Display of Surfaces from Volume Data," *IEEE,* 1988.

[38] T. Ochotta, S. Hiller, and D. Saupe, "Single-pass high-quality splatting," Konstanzer Schriften in Mathematik und Informatik, University of Konstanz, Germany 2006.

[39] M. Levoy, "Display of Surfaces from Volume Data," *IEEE CG&A,* 1988.

[40] E. E. Catmull, "A Subdivision Algorithm for Computer Display of Curved Surfaces," in *Department of Computer Science*. vol. PhD Salt Lake CIty: Utah University, 1974, p. 83.

[41] P. Kipfer and R. Westermann, "GPU Construction and Transparent Rendering of Iso-Surfaces," *Vision, Modeeling and Visualization,* 2005.

[42] S. Parker, P. Shirley, Y. Livnat, C. hansen, and P.-P. Sloan, "Interactive Ray Tracing for Isosurface Rendering," *IEEE Proc. Visualization,* pp. 233-238, 1998.

[43] J. Georgii and R. Westermann, "A Generic and Scalable Pipeline for GPU Tetrahedral Grid Rendering," *IEEE Transactions on Visualization and Computer Graphics,* vol. 12, 2006.

[44] J. P. Schulze and A. S. Forsberg, "User-friendly volume data set exploration in the Cave," 2004.

[45] H. E. Cline, W. R. Lorensen, S. Ludge, C. R. Crawford, and B. C. Teeter, "Two Algorithms for the Three-Dimensional Reconstruction of Tomographs," *Medical Physics,* vol. 15, pp. 320--327, 1988.

[46] W. J. Schroeder, J. A. Zarge, and W. E. Lorensen, "Decimation of triangle meshes," *Computer Graphics (Siggraph '92 Conf Proceeding),* pp. 65-70, 1992.

[47] R. Westermann and T. Ertl, "Efficiently Using Graphics Hardware in Volume Rendering Applications," *Siggraph,* 1998.

[48] M. Hadwiger, C. Sigg, H. Scharsach, B. Katja, and M. Gross, "Real-Time Ray-Casting and Advanced Shading of Discrete Isosurfaces," *Eurographics,* 2005.

[49] J. Wilhelms and A. V. Gelder, "Octree for Faster Isosurface Generation Extended Abstract," *ACM,* 1990.

[50] R. Westermann, L. Kobbelt, and T. Ertl, "Real-time Exploration of Regular Volume Data by Adaptive Reconstruction of Iso-Surfaces," *The Visual Computer,* vol. 15, pp. 100-111, 1999.

[51] G. M. Nielson and B. Hamann, "The Asymtotic Decider: Resolving the Ambiguity in Marching Cubes," *IEEE,* 1991.

[52] S. V. Matveyev, "Approximation of Isosurface in the Marching Cube: Ambiguity Problem," *IEEE,* pp. 288-292, 1994.

[53] G. Johansson and H. Carr, "Accelarating Marching Cube with Graphics Hardware," 2006.

[54] A. Huang, H.-M. Liu, C.-W. Lee, C.-Y. Yang, and Y.-M. Tsang, "On Concise 3-D Simple Point Characterizations: A Marching Cubes Paradigm," *IEEE Transactions on Medical Imaging,* vol. 28, pp. 43-51, Jan. 2009.

[55] T. S. Newman and H. Yia, "A survey of the marching cubes algorithm," *Elsevier,* 2006.

[56] T. J. Cullip and U. Neumann, "Accelerating volume reconstruction with 3d texture hardware.," University of North Carolina, Chapel Hill, N. C. TR93-027, 1993.

[57] O. Wilson, A. VanGelder, and J. Wilhelms, "Direct Volume Rendering via 3D Textures," 1994.

[58] M. Weiler, R. Westermann, C. Hansen, K. Zimmerman, and T. Ertl, "Level-Of-Detail Volume Rendering via 3D Textures," *Volvis,* 2000.

[59] A. V. Gelder and K. Kim, "Direct Volume Rendering with Shading via Three-Dimensional Textures," *Proc. ACM/IEEE Symposium on Volume Visualization,* 1996.

[60] D. Weiskopf, K. Engel, and T. Ertl, "Interactive Clipping Technique for Texture-Based Volume Visualization and Volume Shading," *IEEE Transactions on Visualization and Computer Graphics,* 2003.

[61] G. Khanduja and B. B. Karki, "Visualization of 3D Scientific Datasets Based on Interactive Clipping," *WSCG,* 2005.

[62] D. Weiskopf, K. Engel, and T. Ertl, "Volume Clipping via Per-Fragment Operations in Texture-Based Volume Visualization," 2002.

[63] Ferguson, "FOCUS User Manual, Release 1.2.," Visual Kinematics, Inc., Mountain View, CA, USA 1992.

[64] P. S. Heckbert, "Survey of Texture Mapping," *IEEE Computer Graphics and Applications,* pp. 56-67, 1986.

[65] M. Woo, J. Neider, T. Davis, and D. Shreiner, *OpenGL Programming Guide*, 4 ed.: Addison-Wesley Publishing Company, 1999.

[66] "http://whatis.techtarget.com/definition/," 2008.

[67] C. R. Salama and A. Kolb, "A Vertex Program for Efficient Box-Plane Clipping," in *VMV*, 2005.

[68] U. D. Bordoloi and H.-W. Shen, "Space Efficient Fast Isosurface Extraction for Large Datasets," *IEEE Visualization,* 2003.

[69] H. Samet and R. E. Webber, "Hierarchical Data Structures and Algorithms for Computer Graphics Part 1," *IEEE Computer Graphics and Applications,* 1988.

[70] H. Samet and R. E. Webber, "Hierarchical Data Structures and Algorithms for Computer Graphics Part 2," *IEEE Computer Graphics and Applications,* 1988.

[71] G. Zachmann and E. Langetepe, "Geometric Data Structures for Computer Graphics," *Siggraph Tutorial,* 2003.

[72] "http://en.wikipedia.org/wiki/Octree," 2008.

[73] J. Wilhelms and A. V. Gelder, "Octrees for faster isosurface generation," *ACM Transactions on Graphics (TOG),* vol. 11, pp. 201 - 227 1992.

[74] R. Shekhar, E. Fayyad, R. Yagel, and J. F. Cornhill, "Octree-based decimation of marching cubes surfaces," *IEEE Visualization, Proceedings of the 7th conference on Visualization '96,* 1996.

[75] R. Shu, C. Zhou, and M. S., "Adaptive Marching Cubes," *Visual Computer,* vol. Vol. 11, No. 4,, pp. 202-217, 1995.

[76] G. H. Weber, O. Kreylos, T. J. Ligocki, J. M. Shalf, H. Hagen, B. Hamann, and K. I. Joy, "Extraction of Crack-free Isosurfaces from Adaptive Mesh Refinement Data," *Approximation and Geometrical Methods for Scientific Visualization,* pp. 19-40, 2001.

[77] D. C. Fang, G. H. Weber, H. R. Childs, E. S. Brugger, B. Hamann, and K. I. Joy, "Extracting Geometrically Continuous Isosurfaces from Adaptive Mesh Refinement Data," *Proceedings of 2004 Hawaii International Conference on Computer Sciences,* pp. 216-224, 2004.

[78] I. Boada, I. Navazo, and R. Scopigno, "Multiresolution Volume Visualization with a texture-based octree," *Visual Computer,* 2001.

[79] Q. Cui, M. O. Ward, E. A. Rundersteiner, and J. Yang, "Measuring Data Abstraction Quality in Multiresolution Visualization," *IEEE Information Visualization,* 2006.

[80] S. Guthe, M. Wand, J. Gonser, and W. Straer, "Interactive rendering of large volume data sets," *In Proceedings of IEEE Visualization'02,* pp. 53-59, 2002.

[81] U. Labsik, K. Hormann, M. Meister, and G. Greiner, "Hierarchical iso-Surface Extraction," *Journal of Computing and Information Science and Engineering,* vol. Volume 2, Number 4, , December 2002.

[82] R. Shu, C. Zhou, and M. S., "Adaptive Marching Cubes," *Visual Computer,* vol. Vol. 11, No. 4, pp. , pp. 202-217, 1995.

[83] R. Shekhar, E. Fayyad, R. Yagel, and J. F. Cornhill, "Octree-based decimation of marching cubes surfaces," *IEEE Visualization, Proceedings of the 7th conference on Visualization '96,* p. 9, 1996.

[84] G. M. Nielson, D. Holiday, and T. Roxborough, "Cracking the cracking problem with coons patches. ," *IEEE Visualization, Proceedings of the conference on Visualization '99,* pp. 285-290, 1999.

[85] D. C. Fang, G. H. Weber, H. R. Childs, E. S. Brugger, B. Hamann, and K. I. Joy, "Extracting Geometrically Continuous Isosurfaces from Adaptive Mesh Refinement Data," *Proceedings of 2004 Hawaii International Conference on Computer Sciences,* pp. 216--224, 2004.

[86] E. Kilgariff and R. Fernando, "The GeForce 6 Series GPU Architecture," in *GPU Gems 2*, M. Pharr and R. Fernando, Eds.: Addison-Wesley, 2005.

[87] T. Barrera, A. Hast, and E. Bengtsson, "Fast Near Phong-Quality Software Shading," in *WSCG*, 2006.

[88] H. Gouraud, "Continuous Shading of Curved Surfaces," *IEEE Transactions on Computers* 1971.

[89] J. Kniss, S. Premoze, C. Hansen, P. Shirley, and A. McPherson, "A Model for Volume Lighting and Modeling," *IEEE Transactions on Visualization and Computer Graphics,* 2003.

[90] S. Bergner, T. Moller, M. Tory, and M. S. Drew, "A Practical Approch to Spectral Volume Rendering," *IEEE Transactions on Visualization and Computer Graphics,* vol. 11, 2005.

[91] A. A. Mohamed, L. Szirmay-Kalos, T. Horvath, and T. Foris, "Quadratic Shading And Its Hardware Implementation," *Machine Graphics and Vision,* vol. 9, 2001.

[92] B. T. Phong, "Illumination for Computer Generated Pictures," *ACM,* 1975.

[93] J. Zheng and H. Ji, "High Quality Per-Pixel Shading in Texture-Based Volume Rendering," *IEEE,* 2005.

[94] S. Baroni, P. Giannozzi, and A. Testa, "Green's-function approach to linear response in solid.," *Physical Reviews Letters,* vol. 58, pp. 1861-1864., 1987.

[95] P. Giannozzi, S. de Gironcoli, P. Pavone, and S. Baroni, "Ab initio calculations of phonon dispersions in semiconductors.," *Physical Review B .* vol. 43, p. 12, 1991.

[96] B. B. Karki and G. Khanduja, "Vacancy Defects in MgO at High Pressure," *American Minerologist,* 2006.

[97] www.hdfgroup.org/HDF5/.

[98] R. M. Crutcher, M. P. Baker, H. Baxter, J. Pixton, and H. Ravlin, "Astronomical data analysis software and systems V," in *ASP Conference Series*, 1996.

[99] S. DiVerdi, N. Candussi, and T. Hollerer, "Real-time Rendering with Wavelet Compressed Multi-Dimensional Datasets on the GPU," University of California at Santa Barbara 2005.

[100] T.-c. Chiueh, C.-k. Yang, T. He, H. Pfister, and A. Kaufman, "Integrated Volume Compression and Visualization," *IEEE Visualization,* 1997.

[101] M. Isenburg, P. Lindstorm, and J. Snoeyink, "Lossless Compression of predicted floating point geometry," *Elsevier, Computer-Aided Design,* pp. 869-877, 2005.

[102] P. Pinnamaneni, S. Saladi, and J. Meyer, "3-D Haar Wavelet Transformation And Texture-Based 3D Reconstruction of Biomedical Data Sets," *In Proceedings of the International Association of Science and Technology for Development,* 2001.

[103] J. Wang, T. T. Wong, P. A. Heng, and C. S. Leung, "Discrete wavelet transform on gpu.," *In: ACM Workshop on General Purpose Computing on Graphics Processors,* 2002.

[104] A. Garcia and H.-W. Shen, "GPU-based 3D Wavelet Reconstruction with Tileboarding," *The Visual Computer,* vol. 21, p. 9, 2005.

[105] F. Payan and M. Antonini, "Mean Square Error Approximation for Wavelet-Based Semiregular Mesh Compression," *IEEE Transactions on Visualization and Computer Graphics,* vol. 12, 2006.

APPENDIX A. PSEUDO CODE

A. 1. Framework

Function *Framework*

Input: *List_1; List_2*

Output: *MDV*.

1. *Initiate three threads Data Loading Thread, Visualization Thread and Rendering Thread.*
All Threads are in Suspended Thread.
List_1 contains the list of datasets to be loaded.

2. a) **while** *(List_1 contains dataset marked as loaded is false)*
{
Invoke Data Loading Thread to load the datasets.
Mark dataset as loaded is true.
Mark dataset as visualized is false.
Invoke visualization thread.
}
Suspend the data loading thread.

b) **while** *(List_1 contains datasets with visualized as false)*
{
Visualize the datasets in List_1.
Add rendering primitives for dataset to List_2.
Mark dataset as Visualized is true.
Mark dataset as rendered is false.
Invoke Rendering Thread.
}
Suspend the Visualization Thread

c) **while** *(List_2 contains datasets with rendered as false)*
{
Rendering the datasets as textures
}
Suspend the rendering thread

3. *Depending on user interaction loaded and rendered properties of the datasets can change and will invoke the corresponding threads.*

A. 2. AIM Method

Function *AIM Method*

Input: *Datasets*

Output: *Visualization of the Datasets*

1. ***If (datasets are available)***
   Load the dataset.

2. *Generate the primitives using the visualization technique.*
   For Isosurface – generate the primitives corresponding to isovalue.
   For Texture – generate the 3D texture.

3. *Render the primitives generated in* ***Step 2***
   For Isosurface – Render the primitives.
   For Texture – Render the texture.

4. ***If(more datasets are available)***
   Goto Step 1.

5. ***User Interaction***
   For Isosurface - Change in isovalue – ***Go to Step 2.***
   For Texture Clipping – ***Go to Step 3.***

A. 3. OPIM Method

Function *OPIM Method*

Input: *Datasets*

Output: *Visualization of the Datasets*

1. ***If (datasets are available)***
   Load the dataset.

2. *Generate the primitives using the visualization technique.*
   Delete the Datasets
   For Isosurface – generate the primitives corresponding to isovalue.
   For Texture – generate the 3D texture.

3. *Render the primitives generated in* ***Step 2***
   For Isosurface – Render the primitives.
   For Texture – Render the texture.

4. *If more datasets are available goto* ***Step 1.***

5. ***User Interaction***
   For Isosurface - Change in isovalue – ***Goto Step 1.***
   For Texture Clipping – ***Goto Step 3.***

A. 4. Rendering Module

Function *Rendering Module*

Input: *Rendering Primitives/Texture, texValidated*

Output: *Spreadsheet based visualization using frame buffer object.*

1. ***For each dataset***

    a. *texValidated = false.*
    b. *Generate the frame buffer object.*
    c. *Bind the texture to the frame buffer object.*
    d. *Render the primitive to the frame buffer, it gets saved in the 2D texture bound to it.*
    e. *Set the texValidated = true for this dataset.*

2. *The number of 2D textures is equal to the datasets.*

3. ***If*** *(texValidated = true)*
    - *Render the 2D textures by mapping texture coordinates with the viewport coordinates.*

4. ***User Interaction on selective datasets (set texValidated = false for the selected datasets.)***

    Transformation
    Changing Isovalue for selected dataset
    Texture Clipping

    - *All interaction requires re-rendering the selected datasets. Other datasets can just be rendered by mapping the 3D textures corresponding to them.*

A. 5. Data Coherency

Function *DataCoherency*

Input: *Octree, dc_level, dc_condition, isovalue, dataset*

Output: *bit vector and polygons.*

1. *Node = Octree.root*

2. ***If*** *(dataset is reference dataset)*

3. ***if*** *(Node. Min <= isovalue <= Node.Max)*
if *(Node.level is equal to dc_level)*
Process child nodes
Generate polygons and group them
Else if *(Node has children)*
For each children of Node goto ***Step 3***

4. ***if*** *(dataset is non-reference dataset)*

5. ***if*** *(Node. Min <= isovalue <= Node.Max)*
if *(Node.level is equal to dc_level)*
Check dc_condition
if *(dc_condition is true)*
Set bit corresponding to this node in Bit vector for this dataset
Don't process any child nodes
Node = next node in dc_level
Goto ***Step 5***
Else
Unset bit corresponding to this node in Bit vector for this dataset
Process children nodes and Generate polygons and group them
Node = next node in dc_level
Goto ***Step 5***
Else if *(Node has children)*
For each children of Node goto ***Step 5***

A. 6. Quasi-4D Isosurface Extraction

Function *Quasi-4D Isosurface Extraction*

Input: *Datasets*

Output: *Quasi-4D Isosurface.*

1. *Load all the datasets.*

2. *Create the dataset from the datasets loaded in Step 1 using either of the following ways*

    a. *Case 1: Consider the single slice at same depth from each dataset.*
    b. *Case 2: Consider the set of same multiple slices from each dataset.*

3. *Generate the isosurface from the new dataset.*

4. *Renderer the datasets on framebuffer object.*

5. *Render the texture generated in Step 4*

6. *User Interaction*
    a. *For changing isovalue – Goto Step 3*
    b. *For other slices – Goto Step 2.*

A. 7. Multi-Resolution

Function *Multi-Resolution*

Input: *Dataset, ResolutionLevel, Bounding-Grid*

Output: *isosurface*

1. *Load the dataset*

2. *For each dataset*
   Case

      Dynamic Resolution
      Hybrid Resolution
         Generate the isosurface at the defined ResolutionLevel

      Mixed Resolution
         Create a virtual grid for the bounding box

         Define ResolutionLevelBB.
         Generate the isosurface
            for regions outside the virtual grid at ResolutionLevel
            for regions inside the virtual grid at ResolutionLevelBB

3. *Render the primitives generated in Step 2*

4. ***User Interaction***
   Changing Isovalue – Goto Step 2
   Changing Resolution Level – Goto Step 2

A. 8. Clipping

Function *Planar/Box Clipping*

Input: *Equation of a Clipping Plane* $AQ_x + BQ_y + CQ_z + D = 0$ *; Volume represented as a set of triangles (*$\boldsymbol{P_0}, \boldsymbol{P_1}, \boldsymbol{P_2}$)

Output: *Clipped Coordinates of the volume*

1. $side(Q) = A(Q_x) + B(Q_y) + C(Q_z) + D$

2. P_0 *and* P_1 *two points of the edge*
$$P = P_0 + u\,(P_1 - P_0)$$

3. *Calculate* $u = -\ side(P_0)/\ (side(P_1) - side(P_0))$
   a. *if denominator is 0 then the normal to the plane is perpendicular to the line. Thus the line is either parallel to the plane and there are no solutions or the line is on the plane in which case are infinite solutions*
   b. u *should be between 0 and 1.*

4. *If (2a and 2b)*
   Generate P using the value of u from Step 3.

5. *Repeat Step 2 to Step 4 for all the edges of all the triangles.*

6. *Map the texture to the newly generated intersection points.*

7. *User Interaction*
   Any transformation of the clipping plane would require repeating Step 1 to Step 5

Note: For box clipping we will have multiple clipping planes

Function *Voxelized Clipping*

Input: *Volume as 3D Texture and Clipping Object (Binary Texture)*

Output: *Voxelized Clipped Object.*

1. ***For each*** *texel*

   If *(texel in clipping object is 1)*
   Include the texel from the 3D Texture
   Apply shading and blending
   Else
   Do not include the texel from volume in final display

A. 9. Hardware Isosurface Extraction

Function *Hardware Isosurface Extraction*

Input: *Dataset, isovalue , delta*

Output: *Isosurface*

1. *Load all the dataset.*

2. *Generate the 3D texture from the loaded dataset. Texture will hold the data values and the normal instead of RGB values*

3. ***For each*** *texel*
   If *(isovalue - delta < texel[value] < isovalue + delta)*
   Include the texel
   Assign RGB value to texel
   Shading using the normal values
   Else
   Do not include the texel

4. *Render the fragments.*

5. *User Interaction*
   For changing isovalue – Goto Step 3

APPENDIX B. PARALLEL IMPLEMENTATION OF THE FRAMEWORK

We have tried sample implementation of the MDV framework for texture-and isosurface-based approaches in the parallel environment to study the possibility of extending the MDV framework. In particular, we have implemented 3D wavelet decomposition and reconstruction of the data using C and MPI.

We have utilized wavelet-based method [99, 102-105] for multiresolution technique. For single dimensional data, wavelets can be defined as a set of basis functions (orthogonal and biorthogonal) which decompose the original signal in to two parts, namely, the low frequency components (L) and the high frequency components (H). For multi-dimensional data, decomposition is done for each of the dimension successively. For three-dimensional dataset, we will have eight components for different permutation of high and low frequency components. For basis, we use the Haar function. The values for Haar (inverse) transform can be calculated in the following way:

$$X_{2i} = \frac{(H_i + L_i)}{\sqrt{2}}$$
$$X_{2i+1} = \frac{(L_i - H_i)}{\sqrt{2}}$$

The equations can be easily derived for the Haar forward transform:

$$L_i = \frac{(X_{2i} + X_{2i+1})}{\sqrt{2}}$$
$$H_i = \frac{(X_{2i} - X_{2i+1})}{\sqrt{2}}$$

We have implemented both forward and inverse 1D, 2D and 3D wavelet transforms. In the case of parallel implementation, we first brick the data and then apply wavelet transform on each of these bricks parallel. Finally, the transformed data is merged together.

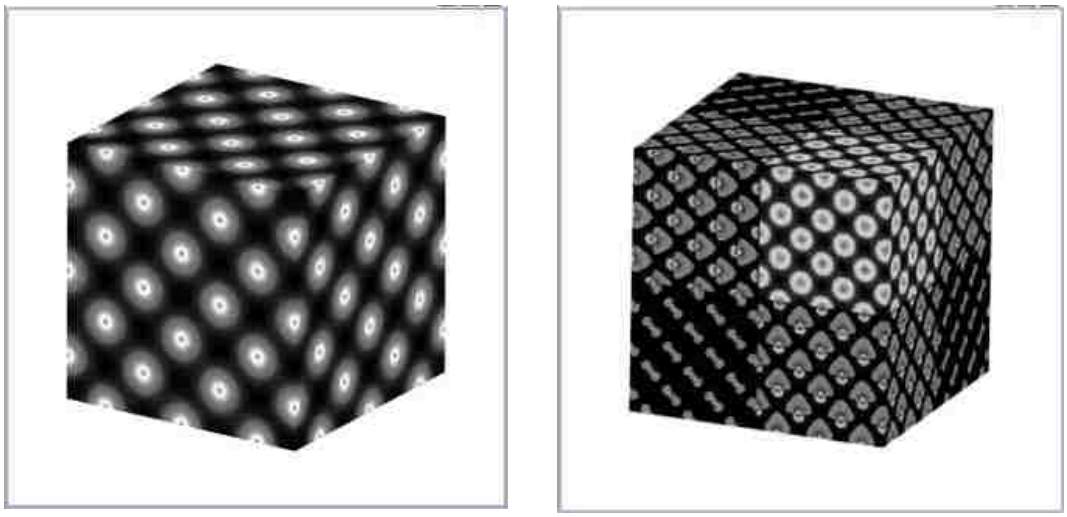
Some constraints that need to be addressed are, for instance, the forward and inverse wavelet transforms should not modify the data, i.e., one should be able to reconstruct the data perfectly. Another constraint with the Haar transform is that the dataset should be of

the dimension of $2^x * 2^y * 2^z$. If any dimension is not of the order of two, then padding is done in that dimension, either by zeros, or by replication of data. If any of the dimensions is one in the above case, then the transform is reduced to 2D wavelet transform. Similarly, if two dimensions are one then it is equivalent to 1D wavelet transform. For this project to simplify things we consider x = y = z. After applying the wavelet transform on the original data, the resulting transformed data is visualized to check the validity of the transform.

Figure 66 shows the 3D wavelet transform (Haar basis function) of the dataset on the MgO solid charge density.

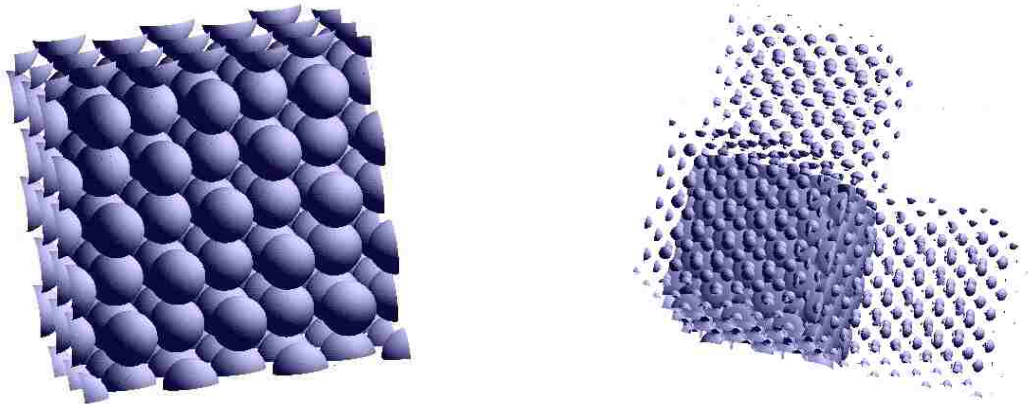
Figure 66(a) shows the original data using 3D texture. Volume rendering is performed using 3D texture surface mapping technique.

Figure 66(b) shows the same data after wavelet transformation. Figure 67 shows the visualization of MgO data using isosurface technique. Figure 70 (a) shows the isosurface from the original data. Figure 70 (b) shows the isosurface for same data after wavelet transform.



(a) (b)

Figure 66: 3D Wavelet transform with application to texture based volume rendering for solid MgO data



(a) (b)

Figure 67: 3D Wavelet transform used for Isosurface generation on the solid MgO dataset. (a) Original isosurface (b) Isosurface after wavelet transform.

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 with honors. He joined Louisiana State University, Baton Rouge, Louisiana to pursue his master's degree in systems science in August 2002. He was awarded Master of Science in Systems Science in May 2005. In Fall 2004, he enrolled in the doctorate program at Louisiana State University, Baton Rouge, Louisiana. He worked as a research assistant to Dr. Bijaya B. Karki while working toward the doctoral degree. His research interest are in the area multiple datasets visualization, parallel computing, scientific visualization and computer graphics. Since September 2008, he is working at ALK Technologies at Princeton, New Jersey. He is a candidate for the degree of Doctor of Philosophy in computer science to be awarded at the commencement of May 2009.