IMPORTANCE-DRIVEN ALGORITHMS FOR SCIENTIFIC VISUALIZATION

DISSERTATION

Presented in Partial Fulfillment of the Requirements for

the Degree Doctor of Philosophy in the

Graduate School of The Ohio State University

By

Udeepta Dutta Bordoloi, M.S.

* * * * *

The Ohio State University

2005

Dissertation Committee:

Dr. Han-Wei Shen, Adviser

Dr. Roger Crawfis

Dr. Raghu Machiraju

Approved by

Adviser

Graduate Program in Computer Science and Engineering

ABSTRACT

Much progress has been made in the field of visualization over the past few years; but in many situations, it is still possible that the available visualization resources are overwhelmed by the amount of input data. The bottleneck may be the available computational power, storage capacity or available manpower, or a combination of these. In such situations, it is necessary to adapt the algorithms so that they can be run efficiently with less computation, with less space requirements, and with less time and effort from the human user.

In this thesis, we present three algorithms that work towards reducing the resource constraints while maintaining the integrity of the visualizations. They are bound by a common underlying theme that all data elements are not equal in the particular visualization context– some are more important than others. We use certain data properties to create "importance" measures for the data. These measures allow us to control the distribution of resources – computational, storage or human – to different portions of the data.

We present a space efficient algorithm for speeding up isosurface extraction. Even though there exist algorithms that can achieve optimal search performance to identify isosurface cells, they prove impractical for large datasets due to a high storage overhead. With the dual goals of achieving fast isosurface extraction and

simultaneously reducing the space requirement, we introduce an algorithm based on transform coding.

We present a view selection method using a viewpoint goodness measure based on the formulation of entropy from information theory. It can be used as a guide which suggests good viewpoints for further exploration. We generate a view space partitioning, and select one representative view for each partition. Together, this set of views encapsulates the most important and distinct views of the data.

We present an interactive global visualization technique for dense vector fields using levels of detail. It combines an error-controlled hierarchical approach and hardware acceleration to produce high resolution visualizations at interactive rates. Users can control the trade-off between computation time and image quality, producing visualizations amenable for situations ranging from high frame-rate previewing to accurate analysis.

Dedicated to ma-deuta, mom and dad.

ACKNOWLEDGMENTS

I am grateful for the insightful advice and generous support that I received from my adviser, Dr. Han-Wei Shen. I have been truly lucky to have had the opportunity to work with him. He has been a great teacher and a good friend, in times good and bad.

I thank Dr. Han-Wei Shen, the Ohio State University, NASA and the American tax-payer for the financial support I received.

I appreciate the help and effort of the members of my dissertation committee- Dr. Roger Crawfis and Dr. Raghu Machiraju. Many fruitful hours were spent in discussions with Dr. David Kao, Dr. Jennifer Dungan and Dr. Alex Pang. And a word of thanks also goes to my colleagues for the many illuminating discussions that resulted over lunches and dinners, not to mention the emails. In addition, I would like to specifically mention Guo-Shi Li and Antonio Garcia for their generous help with some well written code.

I have had the privilege of studying under some exceptional teachers during my long education. Most of what you taught has escaped my memory, but I remember the excitement and still carry the inspiration.

And to my dearest friends, I say thank you, for the trust that I could place on you. I can only hope that I have been able to repay in kind. Folks like you do not come by often, and I am deeply grateful.

I have been blessed with unqualified love and support from my family– ma, deuta, Bedi, Suodi. My late grandparents, my uncles and aunts– thank you for caring. Truly, there is no joy like being at home with family.

Finally, I appreciate the fabulous hand that was dealt to me. It is my sincere hope that, someday, we will get a better handle on this madness, and will be able to make the game more equitable for everyone. Peace.

"And they soothed him and they said over and over, the elder son and the second son, "Rest assured, our father, rest assured. The land is not to be sold."

But over the old man's head they looked at each other and smiled."

— from The Good Earth, by Pearl S. Buck.

VITA

December 31, 1975	Born - Namrup, Assam, India.
1997	B.E. Instrumentation Engineering, University of Delhi.
1999	M.S. Electrical Engineering, Washington University, St.Louis.
1999-present	Graduate Assistant, The Ohio State University.

PUBLICATIONS

Research Publications

U.D. Bordoloi and H.-W. Shen. Automatic view selection for volume rendering. Technical Report:OSU–CISRC–3/05–TR16, 2005.

U.D. Bordoloi, D.L. Kao, and H.-W. Shen. Visualization techniques for spatial probability density function data. *Data Science Journal*, 3:153–162, 2004.

H.-W. Shen, G.-S. Li, and U.D. Bordoloi. Interactive visualization of three-dimensional vector fields with flexible appearance control. *IEEE Transactions of Visualization and Computer Graphics*, 10(4):434–445, 2004.

U.D. Bordoloi, D.L. Kao, and H.-W. Shen. Visualization and exploration of spatial probability density functions: A clustering based approach. In *Proceedings of SPIE & IS&T Conference on Visualization and Data Analysis*, pages 57–64, 2004.

U.D. Bordoloi and H.-W. Shen. Space efficient fast isosurface extraction for large datasets. In *Proceedings of Visualization '03*, pages 201–208. IEEE Computer Society Press, 2003.

G.-S. Li, U.D. Bordoloi, and H.-W. Shen. Chameleon: An interactive texture-based rendering framework for visualizing three-dimensional vector fields. In *Proceedings of Visualization '03*, pages 241–248. IEEE Computer Society Press, 2003.

U.D. Bordoloi, D.L. Kao, and H.-W. Shen. Interactive visualization of probability density functions using clustering and textures: A case study. Technical Report:OSU–CISRC–5/03–TR23, 2003.

U.D. Bordoloi and H.-W. Shen. Hardware accelerated interactive vector field visualization: A level of detail approach. *Computer Graphics Forum*, 21(3):605–614, 2002.

U.D. Bordoloi, D.L. Kao, and H.-W. Shen. Understanding time-varying map data using spatio-temporal clustering. *Eos Transactions AGU*, 83(47):Fall Meet. Suppl., Abstract NG12A–1018, 2002.

U.D. Bordoloi and H.-W. Shen. Hierarchical lic for vector field visualization. In *Proceedings of NSF/DoE Lake Tahoe Workshop on Hierarchical Approximation and Geometrical Methods for Scientific Visualization*, 2000.

FIELDS OF STUDY

Major Field: Computer Science and Engineering.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

Scientific visualization techniques are increasingly being used in a diverse array of fields. In many such domains, the data acquisition technology has improved over the past few years, which has enabled scientists to collect and record data at higher spatial and temporal resolutions. In addition, the capacities of data storage options are increasing at a rapid rate, and the storage costs are becoming lower. As a consequence, in many visualization applications, the datasets that the users want to visualize have grown larger and larger. And the trend is apt to continue at the same pace, if not faster.

Even with the predicted advances in silicon technology [54], it is unlikely that computing power and bandwidth will catch up with the data explosion in the foreseeable future. The large data sizes have created new problems for the visualization systems, and exacerbated existing ones. Larger datasets translate to a greater amount of work required to produce the visualization, and hence to slower response time of the computers. In addition, higher resolution displays are becoming more common, and are preferred for large datasets so that more detail

can be visualized. The extra pixels also contribute to slower response time of the visualization system.

The visualization process frequently involves a hit and trial method of parameter tweaking in an effort to create better representations. This works well for highly interactive visualization systems, but for large data and high resolutions, the response time can become uncomfortably large. Many user studies have shown that there is an inverse relationship between human productivity and the response time of the systems [67]. Moreover, longer waiting times are known to significantly increase the anxiety levels of users [30]. For complex tasks, there is evidence that human errors increase when the response takes longer than an optimal time for the given tasks [67][3]. These studies stress the need for maintaining a responsive visualization environment. Large datasets are also responsible for another effect of considerable concern— there is a lot more data for the users to study. Users need a substantially large amount of time to browse through the data; and slow frame rates make the situation worse.

1.1 Challenges

Together, the large dataset sizes and the need for high resolution renderings have posed a lot of challenges to the visualization community. They have resulted in increased workloads for all stages of the visualization pipeline, from the initial data access to the final rendering. Although the specific problems that arise differ from situation to situation, and depend on the visualization method being used, there are some common issues that can be loosely grouped into the categories below:

- **Computation Time**: As the datasets grow in size, the computation required to run any visualization algorithm on them increases. For very large datasets, the time spent waiting before the first result is displayed can become uncomfortably large.

- **Interaction Frame-rates**: Frequently, the users want to move the camera around once the initial results are shown. For the visualization to be effective, the frame-rates need to interactive. Large datasets and high resolution rendering situations necessitate a greater amount of computation, thus reducing the frame-rates that can be achieved during camera interactions.

- **Storage**: Because of the large sizes of the datasets, I/O and data storage have become significant issues. Visualization algorithms often use auxiliary data-structures for faster performance, which can be large for large datasets. A non-trivially large amount of time is spent just on reading and writing the data.

- **Bandwidth**: Large data sizes also create a bottleneck in remote visualization scenarios with a limited bandwidth. Some remote visualization systems solve this problem by not performing all the computations on the server, thus avoiding any data transfer to the client. However, the problem remains for high-resolution displays.

- **Human Effort**: Most of the visualization processes involve (and need) a large amount of user interaction. However, having a human in the loop costs a large amount of manpower, especially for large datasets. Not only are there

more data to explore, but it also takes more time to update each frame as the user browses the data. As a result, the time and effort put in by the user increases drastically with the data and rendering size.

All these issues are directly or indirectly effected by one another. For example, a high storage requirement for an auxiliary data structure will result in a large amount of time spent in I/O, which will drive up the response time of the visualization process. This will in turn force the user to wait for longer periods of time before getting the results back, thus decreasing the utilization of time and increasing the effort. In another situation, if we can achieve even a partial degree of automation in choosing the visualization parameters, the users will not have to tweak the parameters and redo the calculations over and over again, saving them time and freeing up the computational resources for other purposes.

1.2 Strategies

The visualization community has long been concerned with the challenges of visualizing large data effectively. A wide variety of approaches have been proposed. But there is a common theme among most of the approaches— they involve finding one or more properties of the data that can help distinguish portions of the data from the rest. These properties are then used to redistribute the computational and visualization resources. For example, level of detail algorithms use some form of cost function to control the amount of computation spent on different parts of the data. Transfer function design algorithms provide candidate transfer functions that

can be further tweaked by the user, helping focus the human effort on interesting regions in the transfer function space.

With the goal of increasing the interactiveness and effectiveness of the visualization systems, we present three algorithms with a similar underlying theme– *all data elements are not equal*, some are more important than others. For each of our algorithms, we use certain data properties to create "importance" measures for the data. These measures capture the effect of using lesser resources – computational, storage or human – for our algorithms. They are defined either on the spatial domain, or in the value space, or on the viewing angles. Instead of using our resources equally for all portions of the data, we use the "importance" measures as guides for a heterogenous distribution of resources. More resources are devoted to the data regions whose representations will otherwise loose fidelity to a relatively large degree. Portions of the data which are faithfully reproduced even with lesser resources will carry relatively smaller "importance" values. The main ideas behind our measures and associated algorithms are listed below:

1. **Level of Detail**: *By spending less computational resources on the less "important" parts of the data, we can produce results faster and also make them easier to comprehend.*

   When dealing with large datasets, users frequently face the problem of too much information. Instead of presenting all the detail to the viewer (possibly resulting in a cluttered view), Level of Detail (LoD) algorithms selectively reduce the detail in uninteresting regions. This also creates an opportunity

to increase the speed of the visualization technique by reducing the required amount of computation. (Chapter 4).

2. **Compression**: *By using less storage for the less "important" values of the data, we can achieve a higher compression rate for a given reliability.*

   During the visualization of large datasets, a substantial amount of time is spent in I/O. It involves the transfer of the data (either the original dataset or preprocessed data-structures) from disks, or over the network or bus. By compressing the data, both the time spent in this transfer, and the storage required can be reduced. (Chapter 2).

3. **Automation**: *By guiding the user to the more "important" parts of the data, we can save time otherwise spent on searching through the less "important" parts.*

   As mentioned in the previous item, the user can get inundated by all the data that he needs to go through to extract the desired information. Instead of presenting all the data, we can intelligently cull away the uninteresting data. This reduces the amount of work expected of the user during the visualization process, yet at the same time preserves the usefulness of the visualization to a great extent. (Chapter 3).

In the following sections, we give an overview of our research which makes use of the above strategies to increase the effectiveness of the visualization process. We have applied our research to visualization of both scalar and vector data types.

Large datasets for both types pose a problem because the generation of visualizations can be slow enough to discourage interactivity during the visualization process.

1.3 Range-Search using Compressed data-structures

Isosurfacing is one of the most popular methods for visually representing volumetric scalar fields. The size and shape of the surface components give us information about the distribution of the scalar values in the volume. As we have discussed earlier, the effectiveness of isosurface visualization is limited to a large extent by the interactivity of the visualization environment. The scope for interaction lies in two orthogonal components: tweaking the isovalue (isosurface extraction phase), and changing the view parameters (rendering phase). The usefulness of the visualization system is severely restricted if either one of them cannot be changed at interactive speeds.

Due to the increase in the sizes of the datasets, achieving interactive speeds for the isosurface extraction phase has become progressively more and more challenging. Researchers have taken a variety of approaches to expedite the process of isosurface extraction. One group of such methods comprise the isosurface-containing-cell search techniques. In this document, we will use the term *cell* to refer to the smallest volumetric element in a three-dimensional grid. For regular grids, a cell represents the same entity as a voxel. For unstructured grids, a cell may be a tetrahedron, prism, or any other polyhedron. Our method can be used for datasets on either structured or unstructured grids. The $[minimum, maximum]$ range of a cell will be referred to as its *interval*.

These isosurface-containing-cell search algorithms are motivated by the fact that given an isovalue, the volume needs to be searched only for the cells that contain the isosurface. By pre-computing search-friendly data structures, these techniques reduce the time needed to search for those cells at run-time. Some techniques approach the problem as a search in geometric-space. Others, commonly known as value-space methods, search the space of intervals. In chapter 2, we present our research on a new value-space algorithm.

A number of algorithms have been designed based the concept of value-space. These algorithms achieve nearly optimal [50] or optimal [16] speeds for the cell search phase. However, they suffer from one significant disadvantage: the storage requirement for the pre-computed search data structures. With very large datasets (such as the visible human dataset) becoming commonplace, the high storage overhead associated with these search structures is a serious deterrent to their use. In [16], for example, the authors state that the space requirement of Interval trees is four times the number of cells in the dataset. The Interval tree [16] data structure for a 512^3 floating point dataset (512MB) will need more than 2036MB for storage. The large space complexity renders these techniques ([26][63][50][62][16]) practically unusable without out-of-core modifications. Moreover, the algorithms are slowed down considerably because a large amount of time is spent on file I/O.

With the primary objective of fast isosurface extraction, we have proposed a compression-based solution intended to alleviate the above-mentioned problem of bloated search data structures. We have developed a data-structure that compresses the extremal information by exploiting the spatial coherence present in the

data. The coherence implies that, for most of the cells, the cell minimum has a value close to the cell maximum (for example, see figure 2.1). In the span-space, this translates to most of the cells lying close to the $minimum = maximum$ line. From a compression point of view, the region near this line is important—it needs a much higher storage compared to other parts of the value space. If the same compression level is used everywhere in the value space, this region will produce a disproportionately large amount of errors. To put it another way, to achieve the same level of error everywhere in the value space, this region must be subjected to a small compression rate. The (min-max) information of cells is compacted using a form of compression referred to as transform coding. The conventional $[minimum, maximum]$ representation of intervals is transformed to a more compression friendly representation, and then passed through a dataset optimized non-uniform quantization stage. The effect of the transform and the non-uniform quantizer is that different sizes of quantization bins are used in accordance with the importance of the different regions of the value space.

In our search structures, we store, for each cell, the extrema (min-max) values, and the cell identification tag. The isosurface search is carried out directly on the transformed representation, which eliminates a decoding step for reverting the data to their original values. This method can achieve a reduction, of almost four-fold, in the size of the search data structures compared to those used by ISSUE and Interval trees. The compression technique presented provides a storage friendly yet efficient solution for isosurface extraction in large datasets. Our search algorithm can achieve search speeds that are comparable to the existing techniques.

The transform coding method presented is computationally inexpensive, and can be implemented using only additions, subtractions and value comparisons (for sorting). There is a trade-off between storage requirements and search efficiency (see figures 2.6 and 2.7). The trade-off between storage requirements and the speed of the search process can be exploited to suit the available storage resources and the performance demands of the visualization environment. Although our algorithm is most useful for isosurface extraction, the data-structures also provide speedups for volume-rendering situations with isosurface like transfer functions, which are very common in fields like medical visualization. The data-structures and the search can be used in out-of-core implementations without any significant changes.

1.4 Automatic View Selection

Along with isosurface visualization, volume rendering is another popular method for visualizing volumetric data. With the advent of faster hardware and better algorithms, the traditional challenge of speeding up the volume rendering to achieve interactive frame-rates has been overcome for small datasets. But large datasets still pose problems for users who do not have access to supercomputing facilities. In such situations, the adverse effects due to the non-interactive nature of the visualization can be somewhat compensated by other means. We can guide the user to more informative regions of the data, or interesting parts of the visualization parameter space, thus saving time the user would have otherwise spent in a trial-and-error search. The other option would be to show more information on the screen without having a negative effect (e.g., due to occlusion or cluttering). For example, various alternative rendering techniques can be used to provide a more

understandable picture to the user [19][31]. Users can be guided to interesting features, isosurfaces and transfer functions by methods that suggest such candidates [42][72]. In our research, we have proposed a different path to improve the effectiveness of visualization— that of guiding the user to views that convey more information. In chapter 3, we present a novel view selection method for volume rendering. Such interesting viewpoints are helpful both for the purposes of data exploration and data presentation.

In case of complex datasets, it is very difficult to manually find a view that maximizes the visibility of the relevant part of the data and minimizes occlusion. Currently, users can only use subjective judgment to evaluate and compare views. To remedy this situation, our view selection technique introduces a measure to evaluate a view based on the amount of information displayed (and not displayed). It gives the users the ability to objectively compare two different views. The algorithm can be used to generate viewing positions to be used as starting viewpoints for browsing. Such suggested starting camera positions prove very beneficial in rendering situations with non-interactive frame-rates. Because of the time-lag between frames, users do not want to, and should not be made to [67][30][3], search the whole view-space for desirable views. The algorithm can also be used when presenting data in a non-interactive setting. It creates a smart partitioning of the view space, and selects the most representative views from each view group for rendering.

To evaluate and compare viewpoints, we define three viewpoint characteristics associated with each view:

- **View "goodness":** The view-goodness measure tries to capture how closely the voxel visibilities for a given view match a user-input importance function. We define a view to be good if more important voxels in the volume are highly visible, and vice versa. It is maximized when the voxel visibilities are proportional to their importance. When selecting viewpoints, it is desirable that they have high "goodness" scores.

- **View likelihood :** Intuitively, the view likelihood of a given view is the number of other viewpoints on the view sphere which yield a view that is similar (defined by a threshold) to the given view. We define the view similarities in terms of voxel visibilities and importances that are used for view "goodness". A highly likely view is a good candidate for representing the dataset from different views. On the other hand, low likelihood views are interesting because they display information that is not seen from most other viewpoints, and hence is likely to be missed by users during an interactive search.

- **View stability :** View stability of a view denotes the maximal change in view that can occur when the camera position is shifted within a small neighborhood (defined by a threshold). A small change implies a stable view, and a large change would make a view unstable. Unstable views make good starting viewpoints during interactive visualization, because the user can see a large change in view with a small mouse movement.

In the chapter on view selection, we introduce a 'goodness' measure of viewpoints based on the information theory concept of entropy, also called average

information. We propose that good viewpoints are ones which provide higher visibilities to the more important voxels, the importance being judged by the opacities assigned by the transfer function. This interpretation leads us to the formulation of viewpoint information presented in section 3.3.1. We utilize a property of our entropy definition which indicates that when the visibilities are close to their desired values, the viewpoint information is maximized. This measure allows us to compare different viewpoints and suggest the best ones to the user. Given a desired number N of views, our algorithm can be used to find the best N viewpoints over the view space. A GPU-based algorithm is used to find the visibilities at the exact voxel centers of the volume. The most time-consuming part of our algorithm is finding the voxel visibilities, and this shear-warp based algorithm reduces this time to a few minutes. We also use the entropy to find similarity between views, which is then used to create a view space partitioning and find the likelihood and stability of views. Representative views for each partition can be chosen either by taking the highly likely or highly unlikely views. In interactive situations, our method suggests unstable viewpoints, so that a small change in the camera position will yield a large change in view. For time-dependent data, we present a modification of the 'goodness' measure of a viewpoint by taking into account not only the static information but also the change in each time-step. While this algorithm is targeted for volume-rendering applications, it can be also be used with isosurface visualizations.

There have been different approaches to view selection in the case of geometric scenes, including using the entropy function [79]. However, we know of no literature

related to the problem in a volume rendering scenario. Although we use the entropy function, the formulation is quite different from that used in [79]. We use the voxel visibilities and user-defined voxel importances, and we show that our formulation indeed leads to viewpoints that provide higher visibilities to voxels with more important voxels.

1.5 Level of Detail Vector Field Visualization

Slow computation is a problem in the case of vector data too, as the user needs to interact with the visualization system by changing different rendering parameters (for example, texture properties, advection parameters etc.). Global techniques, such as line integral convolution(LIC)[10] and spot noise[77], are able to show the directional information of the field at every pixel, and the only limitation is the resolution of the display. The price for the rich information content of such methods, however, is their high computational cost, which makes interactive exploration difficult.

In Chapter 4, we discuss the problem of interactive visualization of very dense two and three-dimensional vector fields with flexible level of detail controls. Our goal is to achieve interactive frame-rates when rendering large vector fields for large format graphics displays that consist of tens of millions of pixels. Until recently [78], there were no algorithms that could achieve interactive rates in such cases. Furthermore, currently very few global vector field visualization techniques allow the user to freely zoom into the field at various levels of detail. This capability is often needed when the size of the original vector field exceeds the graphics display resolution. In addition, most of the existing global techniques do not allow a flexible

control of the output quality to facilitate either a fast preview or a detailed analysis of the underlying vector field. Finally, if the user wants to change the appearance of the vector field, most existing algorithms have to redo all the calculations from start, thus increasing the response time. A valuable side-effect of our approach is that the streamline advection stage has been decoupled with the texture mapping stage, which enables us to reuse the advection calculations while creating a variety of appearances.

We introduce a level-of-detail based interactive global vector field visualization technique aiming to tackle the above problems. The idea is similar in spirit to PLIC [81], but we use a hardware based rendering together with a quadtree-based hierarchical approximation of the vector dataset. For two-dimensional vector fields, we have proposed a primitive called *streampatch*, which stores the geometry of flow advection. Once the streampatches are extracted, we can control the appearance of the visualization by simply changing the textures and texture mapping parameters. The primitives can be rendered at different levels of detail. We also propose an error metric which allows us to draw *more important* regions of the vector field in finer detail, and use less detail for other regions. It is computed directly as the directional error that would be produced by the simplification at the particular quadtree level. The error is also used as a guide to generating visualizations which attract the users attention to the more important regions. The 2D texture mapping primitive was later extended to 3D *streamtubes* in Li et. al[48], and to time-varying datasets in Shen et. al[66].

The performance goal of our method is set to produce dense LIC-like visualizations with resolutions in the order of a million pixels within a fraction of second. This is accomplished in part by utilizing graphics hardware acceleration, which allows us to increase the output resolution without linearly increasing the computation time. Additionally, our algorithm takes into account both a user-specified error tolerance and image resolution dependent criteria to adaptively select different levels of detail for different regions of the vector field. Moreover, the texture-based nature of our algorithm allows the user to configure various texture properties to control the final appearance of the visualization. This feature provides extra flexibility to represent the directional information, as well as other quantities in various visual forms. Other effects like multi-texturing for adjusting on-screen frequency for curvilinear grids, changing texture frequency to show a scalar variable on the flow field etc. are also presented for this algorithm. We also present hardware based optimizations to reduce redundant rendering of the vector field that is produced by overlapping geometry.

1.6 Organization

In the rest of this document, we present the details of each algorithm and the importance measures used. Chapter 2 presents our research on compression based search data-structures for fast range search. We explain the two steps involved in transform coding– the transformation and the non-linear quantization. Then the preprocessing and search algorithms are introduced, followed by a discussion of the errors created by quantization. Our work on automated view selection for volume rendering is presented in chapter 3. We give the intuition behind our

view-goodness measure, and show examples of its dependence on view. A view-space partitioning scheme is discussed, and an extension to time-varying data is presented. Chapter 4 deals with our work on interactive flow visualization using level of detail techniques and spatial data structures. The error measure used as a basis for the level of detail scheme is presented. We present the geometric primitive and the rendering algorithm that is used to seamlessly integrate regions of different detail. Various computational optimizations and modifications to achieve different visual results are also presented.

"All animals are equal, but some animals are more equal than others."
— from Animal Farm, by George Orwell.

CHAPTER 2

RANGE-SEARCH USING COMPRESSED DATA-STRUCTURES

One approach taken by visualization researchers to speed up isosurface extraction is improving the search speed for isosurface containing cells. Even though there exist algorithms that can achieve optimal search performance to identify isosurface cells, they prove impractical for large datasets due to a high storage overhead. With the dual goals of achieving fast isosurface extraction and simultaneously reducing the space requirement, we propose a fast search algorithm which uses compressed data structures. The algorithm exploits the spatial coherency present in the data, and is based on transform coding to compress the interval information of the cells in a dataset. Compression is achieved by first transforming the cell intervals (minima, maxima) into a form that allows more efficient compaction. It is followed by a dataset optimized non-uniform quantization stage. The compressed data is stored in a data structure that allows fast searches in the compression domain, thus eliminating the need to retrieve the original representation of intervals at run-time. The space requirement of our search data structure is the mandatory cost of storing every cell id once, plus an overhead for quantization

information. The overhead is typically in the order of a few hundredths of the dataset size.

This chapter is organized as follows. In section 2.1, we discuss the previous work in isosurface search. The transform coding based compression method for intervals is explained in section 2.2, followed by the preprocessing and search algorithms in section 2.3. Finally, the results are presented in section 2.4.

2.1 Previous Work

Since Lorensen and Cline[51] proposed the Marching Cubes algorithm for constructing isosurfaces in 1987, a number of techniques have been proposed to speed up the search for isosurface containing cells. Active list[26] by Giles and Haimes, Span Filter[23] by Gallagher, Sweeping Simplices[63] by Shen and Johnson, and Octrees[87] by Wilhelms and Van Gelder are a few of the early methods. The first three are value-space based methods, while the ever popular octree is a geometric-space technique utilizing hierarchical spatial subdivision. Itoh and Koyamada (extrema graphs)[36][37] and Bajaj et al. (seed cell set)[2][76] use isosurface propagation techniques to avoid the need to search all the cells intersected by the isosurface. Propagation in unstructured grids needs adjacency information to be stored, which increases the storage. Below, we mention three value-space algorithms which are most related to the technique proposed in this chapter.

2.1.1 Storage Requirements of Optimal Search Algorithms

In 1996, Livnat et al.[50] introduced the span space representation for intervals in a near optimal algorithm (NOISE). The span space is a two-dimensional space

of intervals with the x-axis and y-axis representing minima and maxima respectively. Each cell can be depicted as a point in the span space with the coordinates $(minimum, maximum)$. The span space is subdivided using a kd-tree, where each node divides the space into two partitions. The subdivision is alternated between a partitioning of the minima-axis and the maxima-axis at even and odd levels. The ISSUE algorithm by Shen et al.[62] employs a lattice-based subdivision of the span-space. Sequential and parallel algorithms are presented for performing a search over the lattice elements. Cignoni et al.[16] proposed an optimal search algorithm using Interval trees. Each node of the tree divides the intervals into three groups: the intervals whose maxima are less than the value of the node, those whose minima are greater than the node value, and the third set which contain the node value in between their extrema. The first group of intervals are passed onto the left child, the second to the right child, and the third group is put into two sorted lists associated with the node. Next, we discuss the storage requirements of NOISE, ISSUE and Interval trees.

Let us assume that there are N cells in the dataset, and the identity of each cell (cell id) is stored as a number that requires c bytes. Also, suppose that each data value requires d bytes. A pointer-less kd-tree, as used in NOISE, stores the information {cell id, $minimum$, $maximum$} once for each cell. The space requirement is thus $(c + 2d)N$. In ISSUE, all the lattice elements (except those intersected by the $minimum = maximum$ line) store two data structures. *Row* is a list of {cell id, $maximum$} pairs sorted by the cell maxima. The *Column* list comprises of {cell id, $minimum$} sorted by cell minima. Each cell in a lattice

element contributes once to both *Row* and *Column* structures. So, the space needed is $(c+d)2N$, plus overhead. Each node of Interval trees stores two sorted lists: $\mathcal{AL}$ and $\mathcal{DR}$. $\mathcal{AL}$ is an ascending list of left extremes, *i.e.*, of {cell id, *minimum*} pairs, and $\mathcal{DR}$ is a descending list of right extremes, *i.e.*, of {cell id, *maximum*} pairs. Ignoring the tree overhead, the space needed is $(c+d)2N$. If cell ids are stored as 4-byte (one word) integers and data values as 4-byte floats, then the space requirement of NOISE, ISSUE, and Interval trees is respectively $3N$, $4N$ and $4N$ words. For a 512^3 floating point dataset (512MB), for instance, $N = 511^3$ and $4N$ words occupy 2036MB.

2.1.2 Existing Low Storage Solutions

In the case of large datasets, which are common nowadays, the high storage requirement severely restricts the usability of these algorithms. This has prompted researchers to propose modifications so that large datasets can be used with these algorithms. Cignoni et al.[16] present a 3D chess-board arrangement for regular grids to reduce the number of cells the interval tree stores. Cells are colored using a chess-board pattern, and only cells having black color are used to construct the interval tree. Chiang and Silva[14] proposed the first out-of core isosurfacing technique in the form of an I/O optimal implementation of the interval tree. Later, Chiang et al.[15] introduced a method to efficiently group individual cells into meta-cells. They construct an interval tree using the meta-cells instead of individual cells. Both the chess-board and the meta-cell techniques lower the space requirement by reducing the number of cells stored in the search data structures. In this chapter, we present an algorithm which achieves the same goal through efficient

space utilization combined with compression of cell $[maximum, minimum]$ information. The compression method used is based on transform coding. If desired, the cell reduction techniques mentioned above (chess-board and/or meta-cell) can be incorporated into our algorithm to further decrease the search structure size. Let the effective number of cells (individual cells, or black cells in the chess-board pattern, or meta-cells) be N. In an uncompressed form, the {cell id, $minimum$, $maximum$} information requires $3N$ words. Using transform coding, we compress the {$minimum$, $maximum$} information to a few hundredths of N words. The total space requirement of our method is thus one and a few hundredths of N words, as opposed to $4N$ words ISSUE[62] and Interval trees[16], and $3N$ words in NOISE[50].

2.1.3 Transform Coding

Transform coding is a well known data compression approach, and has an extensive body of literature. The basic principle utilized by transform coding is that multiple dimensions of vector data are often correlated to a lesser or higher degree. (If the input data is scalar, multiple samples are collected to form a vector.) The redundancy of data values (due to correlation) is exploited for compression by transforming the vector data and then quantizing each scalar dimension. The transformation allows a better compaction of the data compared to the untransformed values. The best compression ratios are achieved if the transformed data dimensions are not statistically correlated. Hotelling [34] presented the first transform to decorrelate discrete data in the method of *principal components*. Karhunen and Loéve derived the analogous transform for continuous functions, which is now

popularly known as the K-L transform [60][27]. One of the most widely used transform coding applications today is the discrete cosine transform (DCT), which is a part of many image and video coding standards, *e.g.*, JPEG, MPEG etc. For a more detailed review of transform coding and quantization, the reader is referred to [60] and [27].

In the ensuing sections, we present our compression based algorithm for fast isosurface extraction.

2.2 Transform coding for intervals

A number of isosurface extraction algorithms have been developed to perform the search for cells in the value space, *i.e.*, the space of $[minimum, maximum]$ intervals of cells. The minima-maxima space, however, is not suitable for compression due to the high statistical dependence between the $minimum$ and $maximum$ values of cells (see figure 2.1). To reduce this dependence, we use a linear transform to transform this space into a new space (which we will refer to as the **UV**-space). This transformation is the first stage of our compression algorithm. Sections 2.2.1 and 2.2.1 discuss this step in greater detail. In the **UV**-space, each cell (or equivalently, each interval) is represented by its u- and v-coordinates. These coordinates are quantized using a dataset distribution optimized non-uniform quantizer. We use a companded quantizer, which simulates the non-uniform quantization process using a uniform quantizer. While choosing the output values of the quantizer, quantization errors are taken into account. This ensures that the isosurface search does not miss any cell that contains the isosurface. The quantization process

is described in section 2.2.3. The compressed information (in the form of uv-coordinates) is then stored in a search friendly data structure, which is presented in section 2.3.1. At run-time, the search algorithm finds the cells for isosurface extraction based on the supplied isovalue. The search process is explained in section 2.3.2.

In the following sections, we give some background on transform coding, followed by details of the transform and quantization phases of our algorithm.

2.2.1 Background

The central theme of transform coding is that the input data is modified, using a reversible transform, to another form which can be better quantized. The quantized data can then be converted back to the original form using the reverse transform. For the following discussion, we represent a multi-dimensional input data sample as the vector $\mathbf{x}$. There are three stages in transform coding:

1. **Transform**: The input data $\mathbf{x}$ is transformed into $\mathbf{y}$ using a reversible transform $\mathbf{A}$, where $\mathbf{y} = \mathbf{A}\mathbf{x}$. The transformation $\mathbf{A}$ is selected such that $\mathbf{y}$ has better compression characteristics than $\mathbf{x}$, *i.e.*, given a fixed distortion, compressing $\mathbf{y}$ yields a smaller output than that of $\mathbf{x}$. Or, given a fixed compression rate, $\mathbf{y}$ has lower distortion compared to $\mathbf{x}$. The best compression results are achieved when the data dimensions are decorrelated. This suggests that the ideal transform for compaction is the method of *principal components*. This step by itself does not result in any compaction of the data, which is achieved by the next two steps.

2. **Quantization**: The transformed data $\mathbf{y}$ is then quantized to a finite number of levels. Each dimension of the data can be quantized independently using different quantization strategies. The number of quantization levels depends on the desired amount of compaction. The statistics of the data $\mathbf{y}$ influence the design of the quantizer. For example, appropriate uniform or non-uniform quantizers can be chosen depending on input data properties and desired output statistics.

3. **Encoding**: The quantized data is then passed through a binary encoding stage (*e.g.*, Huffman or arithmetic coding). This results is the final compressed form of the data.

Constructing a compression scheme thus boils down to three tasks: finding an appropriate transform, designing quantizers based on the desired compression ratio and error limit constraints, and selecting a proper binary encoder. The data decoding process consists of inverting the effects of the first and third stages above. The second stage is lossy, and that information cannot be recovered. The compressed data is passed through a matching binary **decoder**, and then an **inverse transform** $\mathbf{A}^{-1}$ is applied to recover the data in the original form.

For the problem we are concerned with, the input data is a set of two-dimensional points which represent the $[minimum, maximum]$ intervals of cells. In the rest of section 2.2, we propose a suitable transformation for the intervals, and then design a quantization scheme for the two transformed axes. Since our ultimate goal is fast isosurface extraction, as opposed to achieving the best possible compression, do not use any binary encoding stage. Such a stage would necessitate a decoder

during the cell search phase, which would slow it down and defeat the primary purpose of this research. However, if the situation so demands, a binary encoder can be easily applied as the third stage of encoding.

2.2.2 Transform

The best compression rates can be attained if we use a transformation which statistically decorrelates the minima and the maxima [60]. Hence, the ideal choice for a transformation is the method of principal components. However, it is very expensive to compute, specially for large datasets, which makes it a very impractical choice. Instead, we use a simpler transformation based on the following observation: *it is usual for the minima and maxima of the cells to be highly correlated.* That is, cells with higher maxima tend to have higher minima and vice versa. Figure 2.1 shows a histogram plot of the difference between *maximum* and *minimum* values in the visible woman dataset. As can be expected, the vast majority of cells have a very small difference between their maxima and minima.

Consider a two-dimensional space in which the x-axis represents the cell minima and the y-axis represents the maxima (this is the span space in [50]). Since the cells tend to distribute themselves along the $minimum = maximum$ line, the principal component of any dataset will have an orientation close to the $minimum = maximum$ line. So, instead of the exact principal component transformation, we use a transformation to the 45° line. (Note that the transformation can be interpreted as rotation of the coordinate frame). Each interval is represented as a vector

$$\mathbf{x} = \begin{bmatrix} minimum \\ maximum \end{bmatrix} \tag{2.1}$$

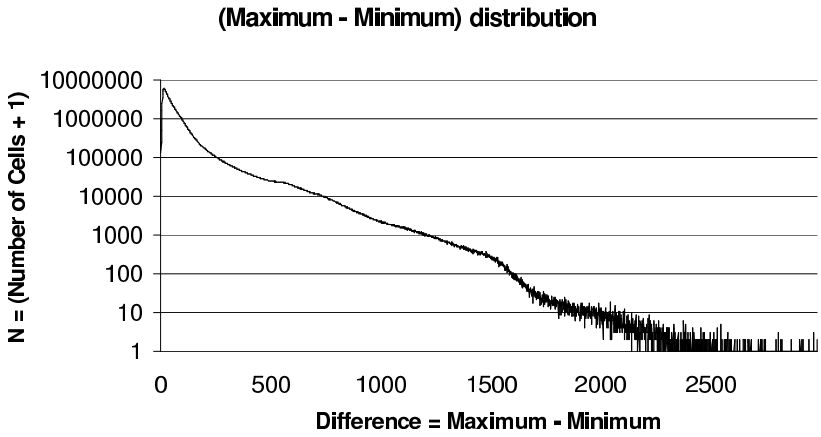


Figure 2.1: A histogram plot of the difference between *maximum* and *minimum* values in the visible woman dataset. The x-axis represents the difference, and the y-axis shows the number of cells which have that difference. The y-axis is shown in a $\log_{10}$ scale. In this dataset, the largest difference is 2978, but 90% of cells have a difference less than 163.

The transformation is given by

$$\mathbf{A} = \begin{bmatrix} cos\theta & sin\theta \\ -sin\theta & cos\theta \end{bmatrix} \tag{2.2}$$

where $\theta = 45°$. After the transformation, each interval is represented by the vector

$$\mathbf{y} = \begin{bmatrix} u \\ v \end{bmatrix} = \mathbf{Ax} \tag{2.3}$$

Or,

$$\begin{bmatrix} u \\ v \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} minimum \\ maximum \end{bmatrix} \tag{2.4}$$

Adding a scaling factor to the transform does not affect the compression results in any way. We reduce the computational expense of the transform by removing the multiplication present in equation (2.4). Defining

$$\mathbf{y} = \mathbf{Bx} = (\sqrt{2}\mathbf{A})\mathbf{x} \tag{2.5}$$

We get

$$u = maximum + minimum \tag{2.6}$$

$$v = maximum - minimum \tag{2.7}$$

Each cell is represented as a point with coordinates (u, v) in the u-v frame, which is obtained by a counter-clockwise rotation of the original *min-max* frame by 45°, followed by a scaling with $\sqrt{2}$ (figure 2.2). We will refer to the two-dimensional space represented by the u-v frame as the **UV**-Space. Alternatively, u can be thought of as twice the mid-point of the interval, and v is the range of the interval. While the minimum and maximum values of an interval have high statistical correlation, the mid-point and range of an interval have a low correlation.

Using the equations (2.6) and (2.7), the interval *minimum* and *maximum* can be expressed as

$$minimum = (u - v)/2 \tag{2.8}$$

$$maximum = (u + v)/2 \tag{2.9}$$

The cells which contain the isosurface satisfy the following condition:

$$minimum < isovalue < maximum \tag{2.10}$$

For simplicity, we have assumed $minimum \neq isovalue \neq maximum$. From equations (2.8), (2.9) and (2.10), we can derive the following condition for a cell which intersects the isosurface

$$v > | u - isovalue \times 2 | \tag{2.11}$$

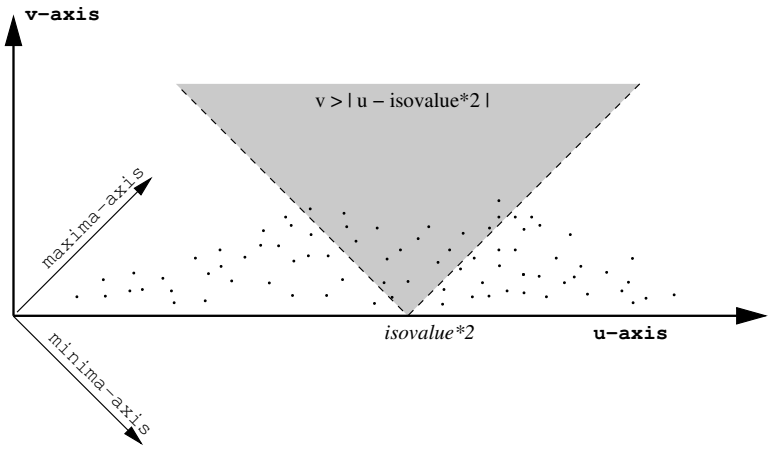


Figure 2.2: **UV**-Space. The **UV**-space is a two-dimensional space of intervals. Each cell is represented as a point with coordinates (u, v) defined by the equations (2.6) and (2.7). The u-v frame is obtained by a counter-clockwise rotation of the original *min-max* frame by 45°, followed by a scaling by $\sqrt{2}$. The isosurface passes through the cells in the shaded region.

In figure 2.2, any cell lying inside the shaded region (defined by equation (2.11)) will be intersected by the isosurface.

In addition to permitting better compression rates, the transformation given by equations (2.6) and (2.7) also has other advantages. First, it requires very little computation in the form of a couple of additive operations. An inverse transformation is not needed: the isosurface test can be done in the transform domain using equation (2.11). Moreover, the transformed space lends itself to a simple and efficient search data structure, which we will present in section 2.3. As will be evident, we will need to store only one sorted list of cells, as opposed to two sorted lists in most algorithms ([26][63][62][16]).

2.2.3 Quantization

Quantization of the **UV**-space is performed in two phases: first, the u-axis is quantized, followed by a quantization of the v-axis. For both axes, we use data distribution optimized non-uniform quantization.

Companded Quantization

After the transformations given by equations (2.6) and (2.7), let the minimum u value for the dataset be u_L, and the maximum u value be u_R. We want to quantize the range $[u_L, u_R]$ into M intervals, where M is input by the user. The design of the quantizer involves deciding the following two sets of values:

- *Decision Boundaries*: The $M+1$ endpoints $\{b_i\}_{i=0}^{M}$ of the M intervals. We already have $b_0 = u_L$, and $b_M = u_R$.
- *Reconstruction Levels*: The M representative values $\{r_i\}_{i=1}^{M}$ for each interval.

The quantizer function, $Q(\cdot)$, is given by

$$Q(u) = r_i \qquad \text{iff} \quad b_{i-1} < u \leq b_i \tag{2.12}$$

Since the distribution of cells in along the u-axis can be (and usually is) non-uniform, we will use a non-uniform quantization strategy. Specifically, we will use an approach called Companded Quantization [27][60], which simulates a *distribution optimized* non-uniform quantizer. A compander has three stages:

1. *Compressor*: The input values (u coordinates of cells) are mapped into another value (say, u') such that the output (u') is uniformly distributed. The regions of the input which have high density are stretched, while regions with

low density are compressed. The mapping conserves the ordering of the input values, *i.e.*, if $u_i < u_j$, then $u'_i < u'_j$. The concept is the same as that used in image equalization.

2. *Uniform Quantizer*: The output of the compressor stage (u') is quantized into M levels using a uniform quantizer. The decision boundaries of this quantizer are $\{b'_i\}_{i=0}^{M}$, and the reconstruction values are $\{r'_i\}_{i=1}^{M}$.

3. *Expander*: The quantized u' values are mapped back to the u-axis using an expander function, which inverts the warping introduced by the compressor function. The compander decision bounds $\{b_i\}_{i=0}^{M}$ are derived from $\{b'_i\}_{i=0}^{M}$, and the reconstruction levels $\{r_i\}_{i=1}^{M}$ are obtained from $\{r'_i\}_{i=1}^{M}$.

Quantization of u-axis

For the first phase, the user specifies the number of quantization intervals, M, of the u-coordinates. We implement the compressor stage by sorting the cells by their u-coordinates. If the u-values of two cells are equal, we break the tie using cell ids. The position of a cell in the sorted sequence is used as its u' value for the uniform quantizer. The first $n_M = N/M$ cells are quantized into the first interval, the next n_M cells in the second interval and so on. The decision boundaries, $\{b'_i\}_{i=0}^{M}$, of the uniform quantizer are the sequence positions of the extreme (the first, and the last) cells of the intervals. The expander stage involves mapping the $\{b'_i\}_{i=0}^{M}$ values to the u-axis using an inverse of the compressor stage. Let the u-value of the jth cell (in the sorted sequence) be u_j, and let $\eta = n_M$. Then the compander

decision boundaries, $\{b_i\}_{i=0}^{M}$, are defined as

$$\begin{array}{rcc} b_0 & = & u_L \\ b_1 & = & (u_\eta + u_{\eta+1})/2 \\ b_2 & = & (u_{2\eta} + u_{2\eta+1})/2 \\ \vdots & \vdots & \vdots \\ b_M & = & u_R \end{array} \tag{2.13}$$

We have assumed that $\{u_{i\cdot\eta} \neq u_{i\cdot\eta+1}\}_{i=1}^{M-1}$. If that does not hold, we take b_i as the average of the u-values of the next two satisfying cells. Figure 2.3 shows the quantization of u-coordinates with $M = 11$. The vertical lines at the decision boundaries $\{b_i\}_{i=0}^{11}$ divide the **UV**-space into $M = 11$ partitions $\{P_i\}_{i=1}^{11}$, which we will refer to as the *U-partitions*.

Unlike usual quantization procedures, the definition of the reconstruction values $\{r_i\}_{i=1}^{M}$ is deferred till run-time. To avoid holes in the isosurface due to quantization errors, we need to incorporate the isovalue into the assignment of $\{r_i\}_{i=1}^{M}$. Consider the cells A and B in the U-partition P_4 in figure 2.3, where the value $u_{iso} = 2 \times isovalue$ lies in U-partition P_7. Both will have the same quantized u-coordinate r_4, which will be used at run-time for the isosurface test in equation (2.11). If cell B fails the test, the resulting isosurface will have a hole in it. To prevent any potential isosurface cell from failing the test, we have to ensure that the right-hand side of the inequality ($v > \mid u - isovalue \times 2 \mid$) does not increase as a result of quantization. Hence, we choose the reconstruction level to be the greatest u-coordinate any cell in partition P_4 can take. This happens to be the right decision boundary of P_4, and so we take r_4 to be equal to b_4. For the same reasons, cells C and D in partition P_{11} are assigned the reconstruction value $r_{11} = b_{10}$. Note that cells A and D will satisfy equation (2.11) and will be sent to the geometry extraction phase, which

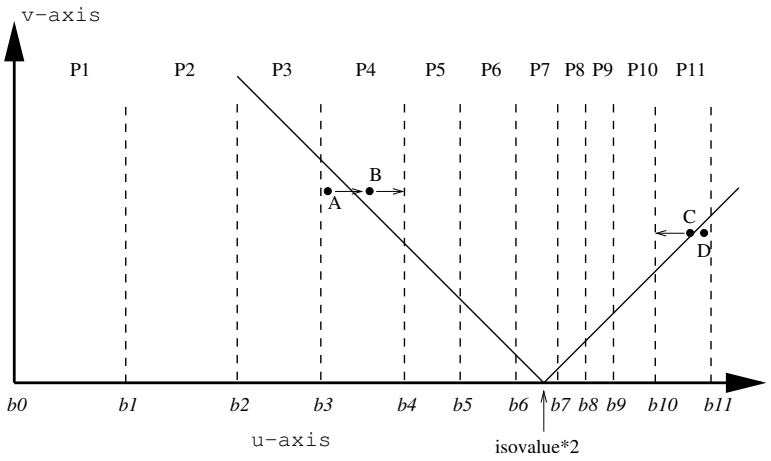


Figure 2.3: Quantization of u-axis. The u-axis is divided into $M = 11$ levels. The *decision boundaries* $\{b_i\}_{i=0}^{11}$ are given by equation (2.13). For the given isovalue, the *reconstruction levels* are given by equation (2.14): $r_1 = b_1$,..., $r_6 = b_6$, $r_8 = b_7$,..., $r_{11} = b_{10}$. r_7 is not used by the search algorithm. The $M = 11$ partitions of the **UV**-space will be referred to as *U-partitions*.

will simply ignore them. For the partition P_7, which contains the value u_{iso}, all the cells are presumed to have passed the test. We define the reconstruction levels in terms of the stored decision boundaries and the given *isovalue* using the following formula: assuming $b_{iso-1} < isovalue \times 2 \leq b_{iso}$

$$\begin{array}{ccc} r_1 & = & b_1 \\ \vdots & \vdots & \vdots \\ r_{iso-1} & = & b_{iso-1} \\ r_{iso} & = & \text{not required} \\ r_{iso+1} & = & b_{iso} \\ \vdots & \vdots & \vdots \\ r_M & = & b_{M-1} \end{array} \tag{2.14}$$

Quantization of v-axis

After the quantization of the u-axis, we proceed to the second phase of our algorithm. We quantize the v-axis in each partition $\{P_i\}_{i=1}^{M}$ of the **UV**-space separately. The quantization strategy is similar to that used for the u-coordinates. The user specifies the number of quantization levels, L, for each U-partition. The following actions are then performed for each partition P_i $(i = 1...M)$. The cells are initially sorted by their v-values, breaking ties by cell ids. The first $n_L = n_M/L$ cells are put in the first interval, the next n_L cells in the second interval and so on. Unlike the quantization stage of u-axis, we do not prevent cells with the same v-values from being put into different intervals. We do so to ensure that each interval contains the same number (n_L) of cells. As a result, we do not have to explicitly store the number of cells in each interval in our data structure.

In the previous discussion on quantizing the u-axis, we argued the need to prevent quantization errors which might result in holes due to isosurface-containing cells being indicated otherwise. In the isosurface test $(v >\mid u - isovalue \times 2 \mid$, equation (2.11)), this translates to the requirement that the v-value should not decrease after quantization. Accordingly, for each interval, the highest v-value of its member cells is used as the reconstruction level for that interval. Let the v-value of the kth cell (in the sorted sequence) in the U-partition P_i be v_{ik}. Then, the reconstruction values, $\{s_{il}\}_{l=1}^{L}$ are given by

$$\begin{array}{ccc} s_{i1} & = & v_{i(n_L)} \\ s_{i2} & = & v_{i(2n_L)} \\ \vdots & \vdots & \vdots \\ s_{iL} & = & v_{i(Ln_L)} \end{array} \tag{2.15}$$

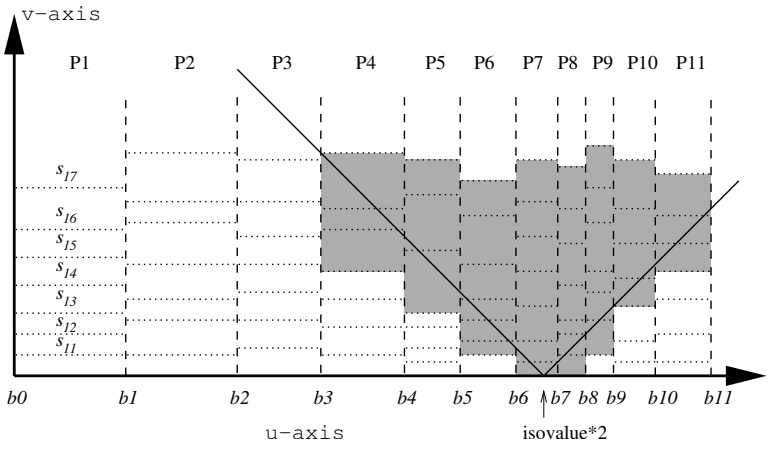


Figure 2.4: Quantization of v-axis. After the u-axis has been quantized using $M = 11$, the v-axis is quantized separately into $L = 7$ levels for each U-partition $\{P_i\}_{i=1}^{11}$. The v-axis *reconstruction levels*, which are also the *decision boundaries*, are shown as the horizontal lines, and the values are calculated from equation (2.15). The $L = 7$ intervals of each U-partition will be referred to as *UV-partitions*.

Figure 2.4 shows the quantization of v-coordinates after the u-axis has been quantized (with $M = 11$, as shown in figure 2.3). Each U-partition has been further divided by $L(= 7)$ horizontal lines, which represent the reconstruction levels (also the decision boundaries) of the v-values within the U-partition. We will call the resulting rectangular regions *UV-partitions*. The UV-partitions which are to the left of the U-partition P_7 are represented by the uv-coordinates of their top-right corners. Similarly, those to the right of P_7 are represented by the uv-values of their top-left corners. For the given isovalue, the shaded UV-partitions pass the isosurface test as their representative corners satisfy equation (2.11).

2.3 Search Algorithm

Following the transform coding steps outlined in the previous section, we construct data structures which store the information of the **UV**-space in a compressed form (section 2.3.1). These can then be used for fast isosurface extraction searches (section 2.3.2).

2.3.1 Data Structures

The preprocessing stage of our algorithm consists of the transformation and quantization steps that have been mentioned in sections 2.2.2 and 2.2.3 respectively. The results of the preprocessing stage are stored in appropriate data structures that enable a fast run time search for isosurface containing cells. The information that needs to be stored is: the user-specified quantization parameters M and L, the reconstruction levels for the u- and v-axes, and the cell ids in each UV-partition. We use the data structures given below to store that information:

1. *U-Array*: The decision boundaries $\{b_i\}_{i=0}^{M}$ for the u-axis, given by equation (2.13). These values are required at run time to derive the reconstruction levels for u-coordinates according to equation (2.14). The storage required is the space for $M + 1$ values.

2. *V-Array*: A two-dimensional array $\{d_{ij}\}_{i=1,j=1}^{M,L}$ with each element storing the v-axis reconstruction levels of the corresponding UV-partition given by equation (2.15). For example, d_{ij} stores the decision boundary of the jth UV-partition of the ith U-partition. This needs a storage of ML values.

3. *ID-Array*: A two-dimensional array $\{A_{ij}\}_{i=1,j=1}^{M,L}$ with each element storing the ids of cells in the corresponding UV-partition. $A(i,j)$ stores the cells within the jth UV-partition of the ith U-partition. The storage needed is that for N cell ids.

The total storage requirement is the space needed for N cell ids and $ML + M + 1$ quantization levels, where $ML + M + 1$ is typically of the order of a hundredth of N. This offers significant space reduction compared to most algorithms ([26][63][62][16]), which store $2N$ cell ids and $2N$ min-max values. During preprocessing, the three arrays are filled simultaneously through the quantization process described in section 2.2.3. To recap, each cell is transformed to uv-coordinates using equations (2.6) and (2.7). They are then sorted by their u-coordinates and the quantization interval endpoints $\{b_i\}_{i=0}^{M}$ derived using equation (2.13). The cells are then grouped into M U-partitions. The cells in each U-partition are now sorted by their v-values. For each U-partition i, the V-array elements (v-axis decision bounds) are filled in according to equation (2.15). Simultaneously, ids of cells in each UV-partition are stored in the ID-Array.

2.3.2 Search

Given an isovalue, the search for isosurface containing cells over the **UV**-space can be decomposed into separate searches over each U-partition. For a given U-partition, the search can be thought of as a search for satisfying UV-partitions (because all the cells within a given UV-partition have the same quantized uv-values). The U-partition is traversed in order of decreasing v-coordinates, beginning with

the topmost UV-partition (the one with highest v-value). The reconstruction values of the UV-partition are read from the U-Array and the V-Array, and tested in equation (2.11). If the UV-partition satisfies the isosurface test, all the cells in the corresponding ID-Array position are selected for geometry extraction, and the search moves to the next UV-partition (the one below). When a UV-partition is reached whose uv-coordinates fail equation (2.11), the traversal for the current U-partition is terminated and the another U-partition is taken up for traversal. The search is complete when all the U-partitions have been individually searched.

Incremental Search

If the isovalue is changed by a small amount from the previous isovalue, it is advantageous to do an incremental update to the results of the previous search. We assume that the previous isovalue search results for each U-partition are stored. For each U-partition, we also need to remember the position of last UV-partition accessed before the traversal was terminated. Let the previous isovalue be iso_p. Without any loss of generality, let us assume that the isovalue has increased to a new value iso_n. Let the corresponding u-axis points be $u_p = 2 \times iso_p$ and $u_n = 2 \times iso_n$ respectively. Then the addition of new cells and removal of cells no longer intersecting the isosurface are handled as follows:

1. *Addition*: New cells will be added to U-partitions that are to the right of $u_{mid} = (u_p + u_n)/2$. For these U-partitions, we start an incremental search from the previous terminating UV-partition. The current traversal is continued till a UV-partition is reached which does not satisfy the isosurface

condition (equation (2.11)). The cells of the newly traversed UV-partitions are added to the isosurface extraction list.

2. *Removal*: For U-partitions to the left of u_{mid}, we will need to potentially remove cells which were selected for isosurfacing for the previous isovalue. Each U-partition is traversed upwards (towards increasing v-values), starting from the terminating UV-partition of the previous traversal. The upward traversal is stopped when a UV-partition is reached which satisfies the isosurface test. The UV-partitions encountered during this reverse traversal no longer contain the isosurface and are removed.

The U-partition which contains u_{mid} can belongs to the addition category if the previous terminating UV-partition satisfies the isovalue. Otherwise, it is in the removal set. As in any incremental update search, this is more beneficial in case of small datasets, for which the intermediate results can be stored in main memory.

Empty Space Culling

Our algorithm can also be used for empty space culling in volume rendering applications. When an isosurface-like transfer function is used, a large proportion of the voxels will be classified as empty space. Instead of doing an exhaustive search, object order algorithms (like splatting and object-order ray-casting[55]) can use a range search algorithm and achieve speedups much like the isosurfacing algorithms.

In this context, the main difference between searching for an isosurface application and a volume rendering one is that transfer functions usually specify non-zero opacities over a range of values, instead of a single value (as in the isovalue). This

fact can be reinterpreted as searching for a range of isovalues in our data structures. Doing such a search is straight-forward using our algorithm. Suppose the non-zero opacities lie in the range $[iso_{left}, iso_{right}]$. Let the points on the U-axis corresponding to these two values be $u_{left} = 2 \times iso_{left}$ and $u_{right} = 2 \times iso_{right}$. Then, all the cells in the U-partitions that fully or partially overlap the range $[u_{left}, u_{right}]$ are assumed to be non-empty. For the U-partitions that are to the left of u_{left}, we do a regular search assuming the isovalue to be iso_{left}. For those to the right of u_{right}, we use iso_{right}.

2.3.3 Errors

The quantization of the **UV**-space introduces errors which may result in false conclusions for some cells in the isosurface test (equation (2.11)). We have designed our quantizer (section 2.2.3) such that the search does not miss any cell that contains the isosurface. Instead, some cells that do not truly intersect the isosurface will satisfy equation (2.11). The errors are the combined effect of u-value quantization error and the v-axis quantization error. Below, we give an empirical discussion on the average effect of the u-axis quantization on the number of such erroneous cells. For this discussion, we first assume that the v-values are not quantized. Later, we will extend the error analysis to include the v-coordinate quantization.

Consider the U-partition P_i in figure 2.5(a), which is to the left of $u_{iso} = 2 \times isovalue$. In other words, $b_i < 2 \times isovalue$. Due to quantization of u-coordinates, all the cells within the shaded triangular region will satisfy the isosurface test, and will constitute the error for this U-partition. Each U-partition that is searched will

contribute a similar group of erroneous cells. It should be noted that if the topmost UV-partition of a U-partition fails the isosurface test, it will not be traversed at all and hence will not contribute any error. For instance, in figure 2.4, the U-partitions P_1, P_2 and P_3 will not have any error since the topmost UV-partitions lie outside the isosurface region. In practice, the dynamic range of u-values is usually much higher than the spread of v-values. As a result, a large number of U-partitions will not be traversed and so will not contribute any error. For this discussion, we assume that on an average, a fraction h of the total number M of U-partitions is traversed. Let the average width of a U-partition be u_{ave}, and the mean concentration of cells be c_{ave}. Then, on an average, each U-partition will contribute $(u_{ave}^2/2)c_{ave}$ false cells. If the u-value limits for the dataset are $[u_L, u_R]$, then the average number of cells which falsely satisfy equation (2.11) is

$$\text{Average U-Error} = hM(\frac{u_{ave}^2}{2} \cdot c_{ave}) = \frac{(u_R - u_L)^2 h c_{ave}}{2M} \tag{2.16}$$

where $n_M (= N/M)$ is the number of cells per U-partition. The expected number of non-isosurface cells satisfying equation (2.11) is inversely related to M.

Next, the additional effect of v-coordinate quantization is considered. In figure 2.5(b), all the cells in the UV-partition V_{ij} have the same uv-coordinates $(b_i, s_{i(j+1)})$, and thus all satisfy equation (2.11). The triangular region contains cells which incorrectly satisfy the isosurface test due to u-axis quantization errors. The error added by v-axis quantization are those cells in the UV-partition V_{ij} whose v-coordinates are less than $v_i = 2 \times isovalue - b_i$. If the total number of cells in the dataset is N, and M and L are the number of quantization levels for u- and v-axes respectively, then each UV-partition has $n_L = N/ML$ cells. On an

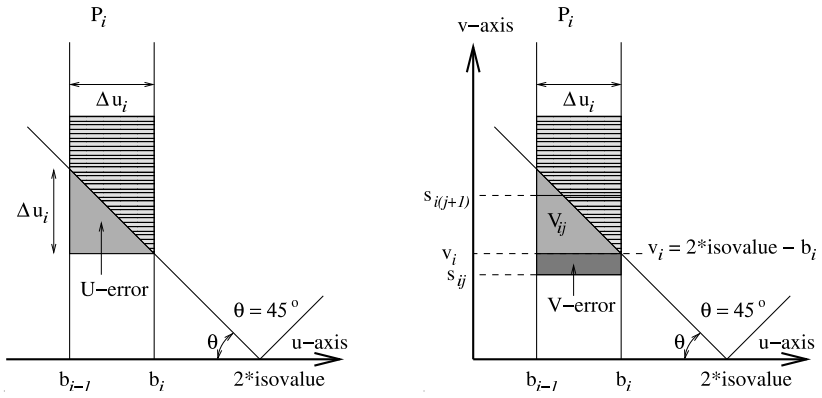


Figure 2.5: Quantization Errors. All the cells in the shaded triangular region in figure (a) satisfy the isosurface test and contribute to the error due to quantization of u-axis. The additional error due to v-axis quantization is shown in figure (b).

average, the total number of erroneous cells due to v-coordinate quantization is

$$\text{Average V-Error} = hM \cdot \frac{n_L}{2} = \frac{hN}{2L} \tag{2.17}$$

The expected error is directly related to the number of cells n_L in each UV-partition. As the value of L is increased, n_L decreases, driving the average v-error down.

2.3.4 Meta-Cells

It is straight-forward to use meta-cells[15] in our data structure. The min-max values of a meta-cell would come from the extremal values of the group of cells it represents. Any preferred technique can be used to combine multiple cells into a meta-cell. In a regular grid dataset, for example, we can represent a $2 \times 2 \times 2$ block of cells by one meta-cell. The minimal (maximal) value among these eight cells

will be the minima (maxima) for the meta-cell. If our search algorithm returns positive for a particular meta-cell, then we have to search for the isosurface in each of the individual cells that is contained within the meta-cell.

Now, we will briefly discuss the errors that can result when merging cells into meta-cells. Suppose a cells A_1 and A_2 have minima min_{A_1} and min_{A_2} respectively. Without any loss of generality, lets assume that $min_{A_1} < min_{A_2}$. If we merge both the cells together into a single meta-cell, then we have to use the smaller minima value min_{A_1} for the meta-cell to ensure correctness. (Similarly, we have to use the larger of the two maxima values as the meta-cell maxima.) The meta-cell will return a false positive for cell A_2 for all isovalues that lie between min_{A_1} and min_{A_2}. Assuming that all isovalues are equally likely, the probability of false positive error due to the merged minima of this meta-cell will be proportional to $(min_{A_2} - min_{A_1})$. In fact, if the values in the particular dataset lie in the range $[min_{Dataset}, max_{Dataset}]$, then the probability of false positive error for cell A_2 is

$$Error = (min_{A_2} - min_{A_1})/(max_{Dataset} - min_{Dataset}) \tag{2.18}$$

A similar effect occurs due to the maxima values. Then, the overall probability of error due to this meta-cell is proportional to

$$\begin{aligned} Error &\propto Distance(cell_{A_1}, cell_{A_2}) && (2.19) \\ &= |min_{A_1} - min_{A_2}| + |max_{A_1} - max_{A_2}| && (2.20) \end{aligned}$$

which is the Manhattan distance between the min-max values of the two cells. This is the probability, due to merging, that either cell A_1 or cell A_2 will return a false positive error.

Lets now suppose that the cell information is merged for the cells $\{A_1, A_2, \ldots, A_k\}$. Then, this particular meta-cell is represented by the smallest minima and the greatest maxima of the set $\{A_1, A_2, \ldots, A_k\}$. We will denote these by min_{meta} and max_{meta}. The error introduced by this meta-cell is

$$Error \propto \sum_{i=1}^{k} Distance(cell_{meta}, cell_i) \tag{2.21}$$

$$= \sum_{i=1}^{k} (|min_{meta} - min_{A_i}| + |max_{meta} - max_{A_i}|) \tag{2.22}$$

While we have assumed that all isovalues are equally probable, that is not always true. We can use domain specific knowledge of datasets to assign different probabilities to different isovalue regions. Equation (2.21) can then be tweaked to include the effect of non-uniform probability in the error. In such cases, terms like $|min_{A_1} - min_{A_2}|$ will be replaced with the area under the probability density function in the range $[min_{A_1}, min_{A_2}]$. Given a user-defined error threshold, min-max information of cells can now be merged into meta-cells. This method of merging cells to create meta-cells can also be used for time-varying datasets.

2.4 Results

In this section, we present results of the compression algorithm on static data. We discuss the effect of the quantization parameters M and L on the size and search efficiency of the search data structures. We then present out-of-core results from our algorithm and also compare the performance with that of the interval tree. We have tested our algorithm on the UNC MR-brain dataset ($256 \times 256 \times 109$ 2-byte integer), a Rayleigh-Taylor hydrodynamic instability dataset (256^3 floating-point)

which we will refer to as Rage256, and the visible woman dataset ($512 \times 512 \times 1728$ 2-byte integer).

2.4.1 Compression and Errors

We have mentioned before that either the meta-cell technique [15] or the chess-board method [16] can be used with our algorithm. For the following discussion, we will denote the number of effective cells (single cells, meta-cells, or black cells in the chess-board pattern) by N. Let M be the number of U-partitions and L the number of UV-partitions per U-partition. The space requirement of our search data structure is the storage for N cell ids (ID-Array) and $ML+M+1$ quantization levels (U-Array and V-Array). Since we are not compressing the cell ids, the space required to store the ID-Array will remain constant for all quantization parameters. We present the compression results as the ratio of the size of the U-Array and V-Array to the space required for storing the min-max values for every cell. Figure 2.6 shows the compression ratios for the MR-brain dataset. Interval trees and ISSUE data structures store $2N$ cell ids and N min-max pairs. Compared to these, the storage required by our search data structure is 37.1% for MR-brain, 27.4% for Rage256, and 33.4% for visible woman dataset for a ($M = 4000, L = 400$) quantization.

Figure 2.7 shows the variation of error with L and M. The error is due to cells which are selected by the search algorithm but do not contain the isosurface. Please note that there is no error in the isosurface itself. The error is defined as the ratio of the erroneous cells to the number of isosurface containing cells. As expected, the error decreases with increase in both L and M. Note that the rate

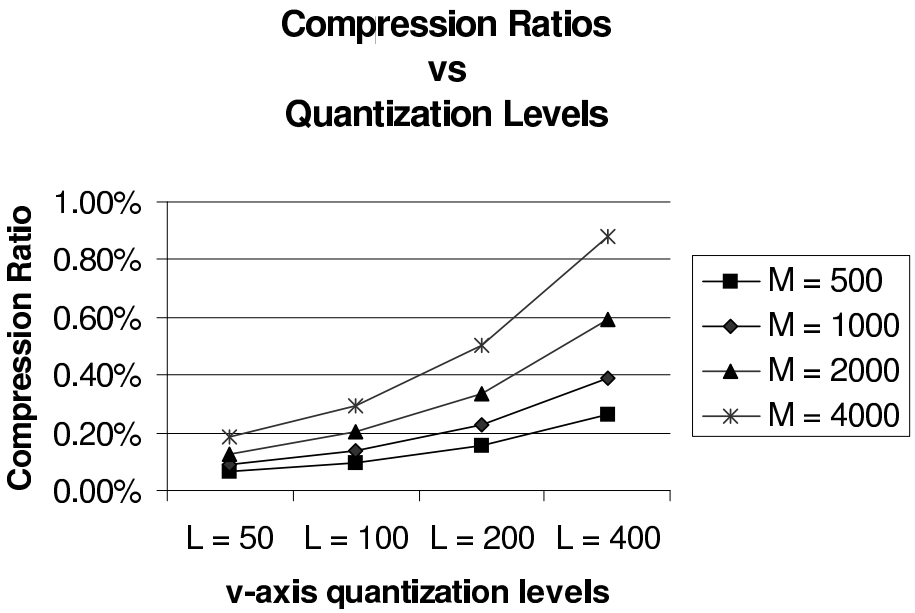


Figure 2.6: Effect of Quantization parameters M and L on data structure size. The size of the quantization data structures (U-Array and V-Array) remain within 1% of the size of the uncompressed min-max information for the MR-brain. The total size of our data structure is approximately 30% of the size of data structures generated by Interval Trees, ISSUE etc.

of decrease falls as L or M get larger. Keeping in mind the trade-off between search and space efficiencies, users can choose an (M,L) combination suitable for their requirements. For instance, the very little difference between performance of the $M = 2000$ and $M = 4000$ graphs may not justify the associated increase in storage space. Table 2.1 gives the preprocessing, search and extraction times for the MR-brain dataset for a subset of quantization parameters from figures 2.6 and 2.7.

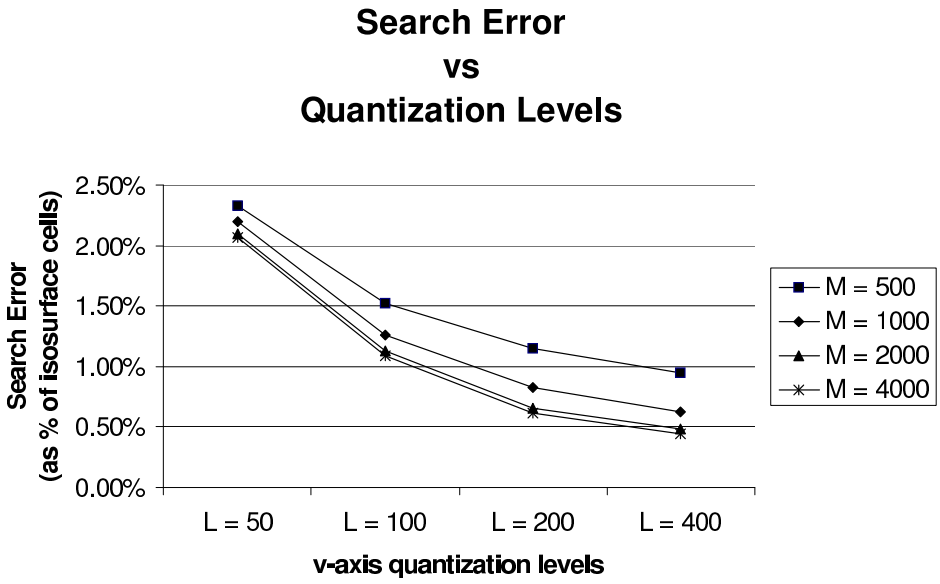


Figure 2.7: Effect of Quantization parameters M and L on search error for the MR-brain (isovalue = 1070.5, number of isosurface cells = 4352196). The number of cells erroneously identified as containing the isosurface grows when M and/or L is decreased.

2.4.2 Performance

In case of large datasets, the search data structures may not fit into main memory and out-of-core techniques have to be implemented. Because we store the min-max information and the cell ids in separate data structures, we do not need to modify our search algorithm for large datasets. Only the U-Array and the V-Array need to be kept in-core. During the search phase, the V-Array is scanned as described in the search algorithm (sec.2.3.2). If the uv-coordinates stored at a V-Array position pass the isosurface test, the corresponding ID-Array entry is read from the disk and the cells passed to the extraction stage. Table 2.2 shows the

search and extraction times for the visible woman dataset. For this experiment, we have used a $2 \times 2 \times 2$ meta-cell for constructing our data structure. The error (number of meta-cells selected due to quantization error) is given as a percentage of the isosurface meta-cells, given in the second column. The data-structure I/O times are included in the extraction times. The compression ratio of min-max information is 3.27% for the data structures used. The size of the search data structure is 34.5% of the size of the ISSUE/Interval-Tree data structures.

Table 2.3 compares the size and performance of our algorithm to an in-core interval tree implementation on a MIPS R10000 Processor. We present results for a floating-point MR-brain dataset and the Rage256 dataset for both methods. The interval tree search performs marginally better than the search using compressed min-max values. The search data structures of our algorithm are smaller by a factor of four or more compared to the interval tree.

"I've seen a dying eye
Run round and round a room
In search of something, as it seemed,
Then cloudier become;"

— from Time and Eternity, by Emily Dickinson.

M,L	Compression	Error	Preprocess	Search	Extract
500, 50	0.18%	2.33%	11.03s	0.04s	7.48s
2000, 200	2.86%	0.66%	11.40s	0.03s	7.44s
4000, 400	11.42%	0.44%	11.50s	0.03s	7.43s

Table 2.1: Search and space efficiency trade-off. Processing times on a 600MHz PIII for different (M,L) combinations are shown (isovalue = 1070.5). The extraction time for a zero search error is 7.41s. The associated compression and errors are shown in figures 2.6 and 2.7.

Isovalue	Cells	Error	Search	Extraction
600.5	2,066,710	4.39%	0.05s	13.0s
1100.5	4,433,023	4.28%	0.12s	27.7s
1400.5	809,193	9.47%	0.04s	6.1s

Table 2.2: Search and extraction times for the visible woman dataset using a compression of 3.27% of the min-max values. The size of the search data structure is 34.5% of the size of the ISSUE/Interval-Tree data structures. $2 \times 2 \times 2$ meta-cells are used while constructing the search data structures. The number of isosurface containing meta-cells are given, along with the error introduced by quantization.

Dataset	Search Method	Search structure size	Search Time
MR-brain	QS (2.5%)	20.8MB	0.18s
MR-brain	I-Tree	93.8MB	0.13s
Rage256	QS (1.3%)	49.0MB	0.09s
Rage256	I-Tree	221.4MB	0.07s

Table 2.3: Comparison of search times for the quantized search (QS) and the interval tree (I-Tree). The compression ratios for the min-max data is given in parentheses.

CHAPTER 3

AUTOMATED VIEW SELECTION

3.1 Introduction

In visualizations of three-dimensional datasets, the insights gained are often dependent on what is seen and what is occluded. The visualization process frequently involves a trial-and-error method of parameter tweaking in an effort to create better representations- that is, better arrangements of visible and occluded regions. This works well for smaller datasets, but for large data, the response times of the visualization system can become uncomfortably large. Larger datasets also translate to a greater amount of work required from the person. While this problem can be tackled by creating more interactive systems [68], we take the approach of reducing the trial-and-error tweaking the user has to go through to create a desirable visualization. Automatic (or semi-automatic) methods for generating transfer functions [52][42] or selecting isosurfaces[72] can be thought of as efforts in this direction. In this paper, we focus on helping the user with one specific component of interaction— view selection. Suggestions for interesting viewpoints can improve both the speed and efficiency of data understanding.

In a typical volume rendering scenario, the user starts with a default viewpoint. After the first image is rendered, he changes the view to look at parts of the dataset that are occluded in the current view. This process continues till he is satisfied. The longer he has to wait for the rendering at the new viewpoint, the less efficient and more frustrated he becomes. The problem is exacerbated in highly non-interactive situations, and it is desirable that he quickly find a satisfactory view. This manual view selection method can be especially tricky and time consuming in the case of volume rendering of a time-varying dataset. The user tries to get a better view based on a few time steps, but it is very difficult for him to imagine how the image will change with the viewpoint for all the time steps in the sequence. Our algorithm makes the manual view selection faster by suggesting good viewpoints to the user. These viewpoints can then be used as a starting point for further exploration. Once the user finishes exploration in the neighborhood of one suggested viewpoint, he can pick another suggested viewpoint to explore.

The viewpoints suggested by our technique can also be used to improve image-based volume rendering algorithms [56][12]. Frequently, IBR methods use the scene properties to create a non-uniform camera placement for the pre-rendered views based on the scene. Our formulation can be used to quantify the change between two volume-rendered views. An adaptive sampling of the view space can be generated by creating more pre-rendered samples in neighborhoods of large view changes and vice-versa. This can help the IBR system achieve better rendering quality with less storage.

This chapter presents a view selection method designed for volume visualization. It can be used to find informative views for a given scene, or to find a minimal set of representative views which capture the entire scene. It becomes particularly useful when the visualization process is non-interactive— for example, when visualizing large datasets or time-varying sequences. We introduce a viewpoint "goodness measure" based on the formulation of entropy from information theory. The measure takes into account the transfer function, the data distribution and the view-dependent visibility of the voxels. Combined with viewpoint properties like view-likelihood and view-stability, this technique can be used as a guide which suggests "interesting" viewpoints for further exploration. Domain knowledge is incorporated into the algorithm via an importance transfer function or importance volume. The view selections can thus be easily configured to obtain behaviors tailored to very specific situations. We generate a view space partitioning, and select one representative view for each partition. Together, this set of views encapsulates the most important and distinct views of the data. Viewpoints in this set can be used as starting points for interactive exploration of the data, thus reducing the human effort in visualization. In fully non-interactive situations, such a set can be used as a representative visualization of the dataset from all directions. We present a hardware based solution to performing the view calculations. This algorithm can also be used with isosurface visualizations.

3.2 Related Work

The idea of comparing different views developed much before computer graphics and visualization matured. As early as 1976, Koenderink and van Doors [44][45]

had studied singularities in 2D projections of smooth bodies. They showed that for most views (called stable views), the topology of the projection does not change for small changes in the viewpoint. The topological changes between viewpoints can be stored in an aspect graph. Each node in the graph represents a region of stable views, and each edge represents a transition from one such region to an adjacent one. These regions form a partitioning of the view space, which is typically a sphere of a fixed radius with the object of interest at its center. The aspect graph (or its dual, the view space partition) defines the minimal number of views required to represent all the topologically different projections of the object. A lot of research has been done since the early papers, mainly in the field of computer vision, which extended the ideas to more complex objects. In the case of volume rendering, a similar topology based partitioning can not be constructed. Instead, we find a visibility based partitioning by comparing visibilities of voxels in neighboring views, and clustering together viewpoints that are similar.

Viewpoint selection has been an active topic of research in many fields. For instance, viewpoint selection solutions have been proposed for the problem of modelling a three-dimensional object from range data [89] and from images [21], and also for object recognition [1]. However, the topic has not been well investigated in the fields of computer graphics and visualization, possibly because applications in these domains have relied heavily on human control. Recently, Vázquez et al. [79][80] have presented an entropy based technique to find good views of polygonal scenes. They define an entropy for any given view, which is derived from the projected area of the faces of the geometric models in the scene. Their motivation

is to achieve a balance between the number of faces visible and their projection areas. The entropy value is maximized when all the faces project to an equal area on the screen. The viewpoint measure presented in this paper is based on the entropy function, but is designed for volumetric data. Each voxel is assigned a visual significance, and the entropy is maximized when the visibilities of the voxels approach the respective significance values. Entropy based methods have been used in a variety of problems, e.g., for calculating scene complexity in radiosity algorithms [20], for object recognition [1] and for aiding light source placement [28].

3.3 Viewpoint Evaluation

The essential goal of this chapter is to have a computer suggest 'good' viewpoint(s) to the user. This naturally leads us to the question: "what is a good viewpoint?", or, "what makes a viewpoint better than another?". The answer will depend greatly on the viewing context and the desired outcome. For example, a photographer will choose the view which best contributes to the chosen mood and visual effect. For this paper, the context is the process of volume rendering, which is being used to obtain visual information from the data. Hence, for our purposes, *a viewpoint is better than another if it conveys more information about the dataset.* In this section, we present a method for quantifying the information contained in a view using properties of the entropy function from information theory.

The information that is transferred from a volumetric dataset to the two-dimensional screen is governed by the optical model which is used for the projection. In this paper, we assume the popular absorption plus emission model [53].

The intensity Y at a pixel D is given by

$$Y(D) = Y_0 T(0) + \int_0^D g(s)T(s)ds \tag{3.1}$$

where, $T(s)$ is the transparency of the material between the pixel D and the point s. We will refer to $T(s)$ as the visibility of the location s. The first term in the equation represents the contribution of the background, Y_0 being its intensity. The second term adds the contributions of all the voxels along the viewing ray passing through D. A voxel at point s has an emission factor of $g(s)$, and its effect on the pixel intensity is scaled by its visibility $T(s)$. If two voxels have the same emission factor, then the one with a higher visibility will contribute more toward the final image.

The emission factors of voxels are usually defined by the users. They set the transfer function to highlight the group of voxels they want to see, and to make the others more transparent. We use this fact to define a *noteworthiness* factor for each voxel (section 3.3.2), which captures, among other things, the importance of the voxel as defined by the transfer function. Based on the preceding discussion, we have the following two (not necessarily disjoint) guidelines for defining a good viewpoint:

1. A viewpoint is good if voxels with high noteworthiness factors have high visibilities.

2. A viewpoint is good if the projection of the volumetric dataset contains a high amount of information.

In the following section, we present the details of our view information function and its properties.

3.3.1 Entropy and View Information

Consider any information source X which outputs a random sequence of symbols taken from the alphabet set $\{a_0, a_1, \ldots, a_{J-1}\}$. Suppose the symbols occur with the probabilities $\mathbf{p} = \{p_0, p_1, \ldots, p_{J-1}\}$. Alternatively, we can think of it as the random variable X which gets the value a_j with probability p_j. The information associated with a single occurrence of a_j is defined in information theory as $I(a_j) = -\log p_j$. The logarithm can be taken with base 2 or e, and the unit of information is bits or nats respectively. In a sequence of length n, the symbol a_j will occur np_j times, and will carry $-np_j \log p_j$ units of information. Then the *average information* of the sequence, also called its entropy, is defined as

$$H(X) \equiv H(\mathbf{p}) = -\sum_{j=0}^{J-1} p_j \cdot \log_2 p_j \qquad \text{bits/symbol} \tag{3.2}$$

with $0 \cdot \log_2 0$ defined as zero [5]. Even though the entropy is frequently expressed as a function of the random variable X, it is actually a function of the probability distribution $\mathbf{p}$ of the variable X. We will use the following two properties of the entropy function in constructing our viewpoint evaluation measure:

1. For a given number of symbols J, the maximum entropy occurs for the distribution $\mathbf{p_{eq}}$, where $\{p_0 = p_1 = \ldots = p_{J-1} = 1/J\}$. (See figure 3.1, which gives an example of the entropy values for a three dimensional distribution.)

2. Entropy is a concave function, which implies that the local maximum at $\mathbf{p_{eq}}$ is also the global maximum. It also implies that as we move away from the

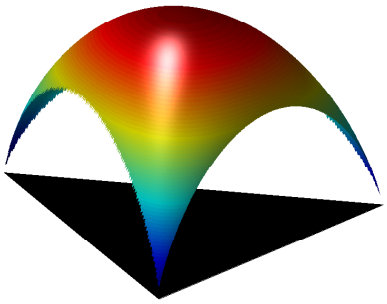


Figure 3.1: Entropy Function for probability vectors of dimension three: $\mathbf{p} = \{p_0, p_1, p_2\}$. The function is defined only over the plane $p_0 + p_1 + p_2 = 1$, within the triangular region specified by $0 \leq p_1, p_2, p_3 \leq 1$. The maximum occurs at the point $p_0 = p_1 = p_2 = 1/3$, and the value falls as we move away from that point in any direction. So, increasing the entropy has the effect of making the probabilities more uniform.

equal distribution $\mathbf{p_{eq}}$, along a straight line in any direction, the value of entropy decreases (or remains the same, but does not increase).

We will use probability distributions associated with views to calculate their entropy (average information). For each voxel j, we define an importance factor W_j. We will call it the *noteworthiness* of the voxel, and it indicates the visual significance of the voxel. (More details about W_j are given in section 3.3.2). Suppose, for a given view V, the visibility of the voxel is $v_j(V)$. We are using the term 'visibility' to denote the transparency of the material between the camera and the voxel. It is typically equivalent to $T(s)$ in equation (3.1). Then, for the view V, we define the *visual probability*, q_j, of the voxel as

$$q_j \equiv q_j(V) = \frac{1}{\sigma} \cdot \frac{v_j(V)}{W_j} \qquad \text{where, } \sigma = \sum_{j=0}^{J-1} \frac{v_j(V)}{W_j} \tag{3.3}$$

where the summation is taken over all voxels in the data. The division by σ is required to make all probabilities add up to unity. Thus, for any view V, we have a visual probability distribution $\mathbf{q} \equiv \{q_0, q_1, \ldots, q_{J-1}\}$, where J is the number of voxels in the dataset. Then, we define the entropy (average information) of the view to be

$$H(V) \equiv H(\mathbf{q}) = -\sum_{j=0}^{J-1} q_j \cdot \log_2 q_j \tag{3.4}$$

The view with the highest entropy is then chosen as the best view. This satisfies the two guidelines presented earlier in section 3.3:

1. The best view has the highest information content (averaged over all voxels).

2. The visual probability distribution of the voxels is the closest (of all the given views) to the equal distribution $\{q_0 = q_1 = \ldots = q_{J-1} = 1/J\}$, which implies that the voxel visibilities are closest to being proportional to their noteworthiness.

To calculate the view entropy, we need to know the voxel visibilities and the noteworthiness factors. Visibilities can be queried through any standard volume rendering technique such as ray casting. The noteworthiness, described in the next section, is view independent, and needs to be calculated only once for a given transfer function.

3.3.2 Noteworthiness

The noteworthiness factor of each voxel denotes the significance of the voxel to the visualization. It should be high for voxels which are desired to be seen, and vice versa. Considering the diverse array of situations volume rendering is

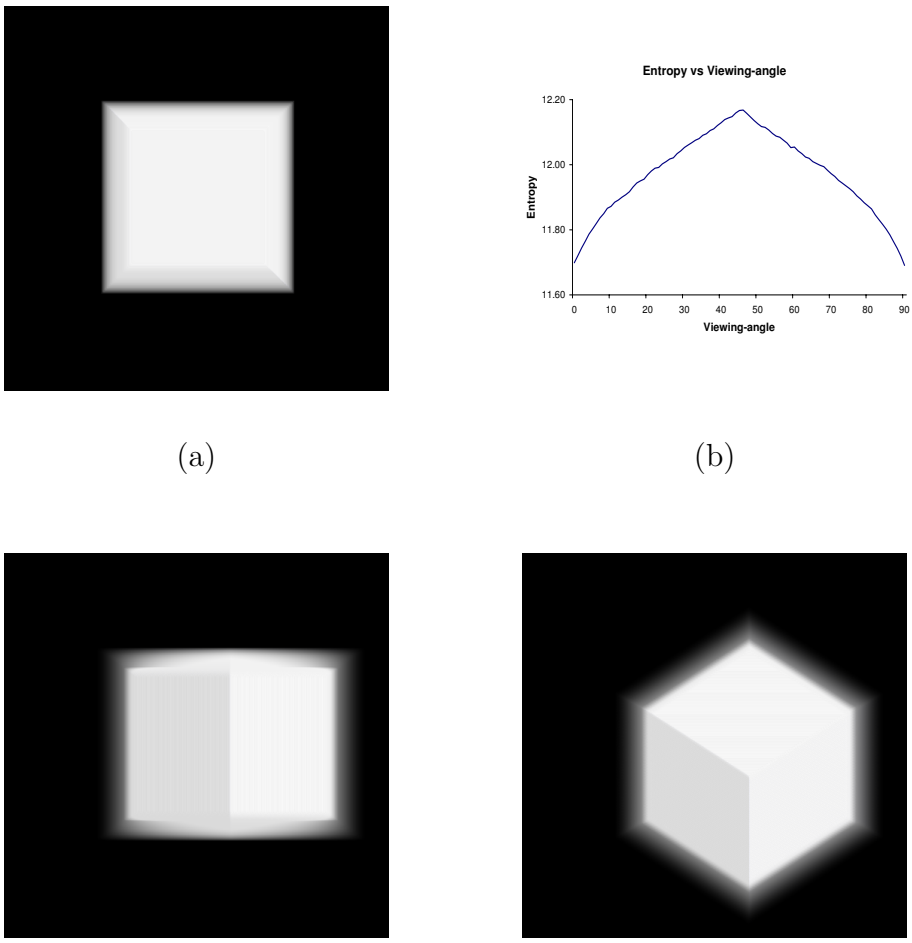
used in, it is practically impossible to give a single definition of noteworthiness that satisfies expectations of all users. Instead, we can rely on the user-specified transfer functions to deliver us a definition which is tailor-made for the particular situation. The opacity of a voxel, as assigned by the transfer function, is part of the emission factor $g(s)$ in equation (3.1), and controls the contribution of the voxel to the final image. We use opacity as one element of the noteworthiness of the voxel. Another consideration is that some voxels are more visually meaningful to the viewer than other voxels. Consider a simple example: suppose the dataset has a small region of yellow voxels and the rest of the voxels are blue. When the yellow region occludes part of the blue region, Gestalt principles [58] suggest that the human mind extrapolates the larger object (called ground) behind the smaller one (called figure). If, on the other hand, the yellow region is occluded, the viewer will have no idea of knowing it even exists. In this case, the visibility of the yellow region is more important than that of a similar number of blue voxels.

Based on these observations, we construct the noteworthiness W_j of the jth voxel as follows. We assign probabilities to voxels in our dataset by constructing a histogram of the data. All the voxels are assigned to bins of the histogram according to their value, and each voxel gets a probability from the frequency of its bin. The information I_j carried by the jth voxel is then $-\log f_j$, where f_j is its probability (bin frequency). Then, W_j for the voxel is $\alpha_j I_j$, where α_j is its opacity. We ignore voxels whose opacities are zero or close to zero. These voxels are not included in the evaluation of equation (3.4).

The answer to what is interesting and what is not is very subjective. Our algorithm can be made to suit the goals of any particular visualization situation just by changing the noteworthiness factors. Domain specific knowledge can be readily incorporated into the framework by adapting the noteworthiness. Irrespective of the method used to specify the interestingness of the voxels, maximizing the entropy serves to give better visibility to the more interesting voxels.

3.3.3 A Simple Example

To demonstrate our concept of view information, we constructed the test dataset shown in figure 3.2. The voxel opacities of the cube dataset increase linearly with distance from the boundary of the cube. Figure 3.2(a) shows a volume rendering of the dataset when the camera is looking directly at one of its faces. Next, we revolve the camera about the vertical axis of the dataset, (or, equivalently, rotate the dataset in the opposite direction about the vertical axis) at 1° increments. The view entropy steadily increases (figure 3.2(b)) as more and more voxels on the side face start becoming visible. It reaches a maximum when the camera has moved by 45°, which is the view that shows the two faces equally (figure 3.2(c)). Further movement of the camera results in greater occlusion of voxels near the first face, and the entropy begins to drop again. Upon evaluating the entropies for all camera positions around the dataset, the view in figure 3.2(d) results in the highest entropy. Clearly, this is one of the more informative views about the cube dataset for a human observer.



(a) (b)

(c) (d)

Figure 3.2: An illustration of the change in view entropy (equation 3.4) with camera position for a test dataset. Figure (a) shows the initial position of the camera. Figure (b) shows the behavior of entropy as the camera revolves around the dataset (around the vertical axis in the figure) at 1° increments. The entropy increases steadily and reaches a maximum for a movement of 45° (figure (c)), and then begins to decrease again. The maximum entropy for the whole view space is obtained for the view in figure (d).

3.4 Finding the Good View

The view selection proceeds as follows. The dataset is centered at the origin, and the camera is restricted to be at a suitable fixed distance from the origin. This spherical set of all possible camera positions defines the view sphere, and represents all the view directions. The view space is then sampled by placing the camera at sample points on this sphere. We create a uniform triangular tessellation of the sphere and place the viewpoints at the triangle centroids. The camera position and the origin specify the eye and the center points for the modelview transformation. Since the roll of the camera does not affect the visibilities, the up vector can be arbitrarily chosen.

Next, the voxel visibilities are calculated for each sample view position. Our technique is not dependent on any particular volume rendering method, and both software and hardware renderers can be used by modifying them to output voxel visibilities. (Please note that the transparency or voxel visibility as given in equation (3.1) is numerically the same as the accumulated opacity subtracted from unity.) Most renderers, however, do not perform the opacity calculations exactly at the voxel centers. Ray-casters accumulate opacities along the rays, and texture based renderers accumulate opacities at frame-buffer pixel locations, neither of which are necessarily aligned with voxel centers. We use the GPU to calculate the visibilities at the exact voxel centers by rendering the volume slices in a front-to-back manner using a modified shear-warp algorithm. We give a brief description of our implementation below.

3.4.1 Hardware Implementation

The object-aligned slicing direction is taken along the axis which is most perpendicular to the viewing plane. The slices are rendered perpendicular to the screen, with a relative displacement given by the shear factors[46]. A floating point p-buffer with the same resolution as the volume slices is used. 32-bit floating point precision is used for both the p-buffer and the textures that store visibility. The first slice has no occlusion, so all the voxels in this slice have their visibilities set to unity. We loop through the rest of the slices in a front-to-back order, calculating the visibilities of one slice in each loop. The input in each loop iteration is the data and the visibilities of the previous slice. The following actions are performed during each iteration:

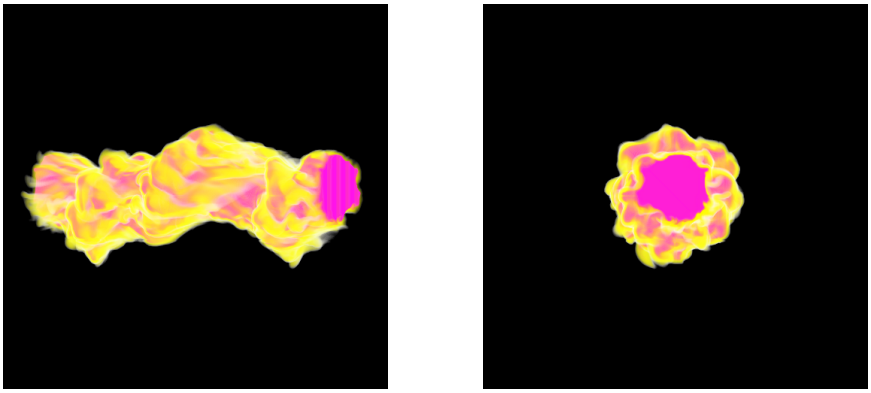
The frame-buffer (p-buffer) is cleared, and the camera is set such that the current slice aligns perfectly with the frame-buffer. Then, the previous slice is rendered with a relative shear, and with two sets of multi-texture coordinates- one corresponding to the data texture, and the other to its visibility (which is also stored as a texture). A fragment program looks up the data opacity according to the transfer function, and combines it with the slice visibility texture. The frame-buffer now contains the visibilities of the current slice, which is read back for further processing. Render to texture techniques are used to speed up the texture initialization for the next loop iteration, in which the visibilities of the current texture will be used. Once the visibilities for all the slices are retrieved, the entropy for the given view direction is calculated by using the visibilities and the noteworthiness factor. Voxels with opacities close to zero (defined by a threshold)

are classified as empty space and are not used in the evaluation of equation (3.4). This reduces the computational and memory requirements for the entropy and also the similarity calculations (section 3.5.1).

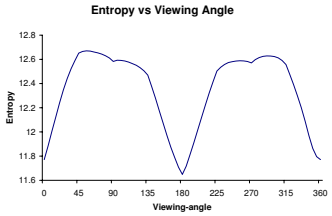
Figure 3.3 shows a time-step of a 256-cube shockwave dataset. The camera was rotated about the vertical axis in a complete circle, with the dataset centered at the origin. Entropy was evaluated at 5° increments. The first figure shows the view at 55° rotation which was the view with the highest entropy. Figure(b) shows the worst view, which occurred at 180°. Figure 3.4 shows a $128 \times 128 \times 80$ tooth dataset. The view sphere was sampled at 128 points. Figures (a) and (b) have the highest view entropy values. Figures (c) and (d) have the lowest entropy, and not surprisingly, are highly occluded views. It is notable that the viewpoints for (c) and (d) are not very far apart, and that (a) and (b) show much of the same voxels. This shows that if the user wants a few (say, N) good views from the algorithm, returning the N highest entropy views might not be the best option. Instead we can try to find a set of good views whose view samples are well distributed over the view sphere. The next section presents such a solution.

3.5 View Space Partitioning

The goodness measure presented in the previous section can be used as a yard-stick to measure the information captured by different volume rendering views and select the best view. But the calculation of goodness considers information from only the given view, and ignores the information that might be contained in other views. In particular, neighboring viewpoints tend to have similar visibilities, and comparing a viewpoint with its neighbors can provide additional properties of the

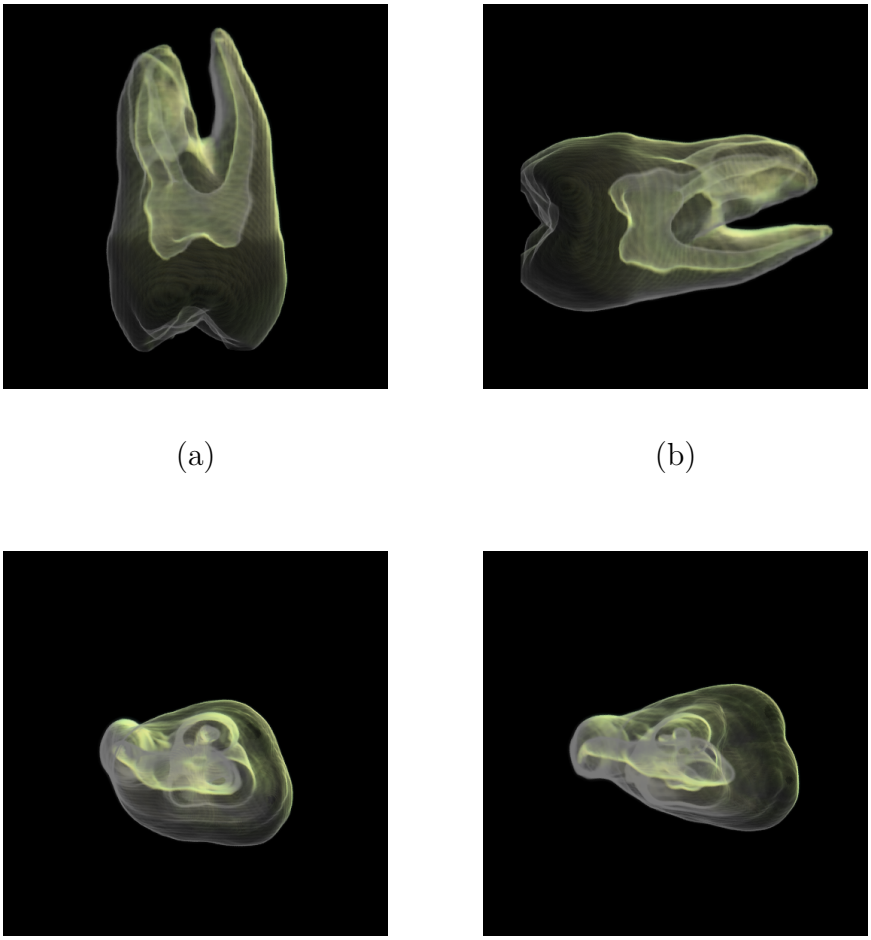


(a) (b)



(c)

Figure 3.3: The figure shows a time-step of a 256-cube shockwave dataset. The camera was rotated about the vertical axis in a complete circle, with the dataset centered at the origin. Entropy was evaluated at 5° increments. Figure (a) shows the view at a 55° rotation which was the view with the highest entropy. Figure(b) shows the worst view, which occurred at 180°. Figure (c) plots the change of entropy with change in angle.



(a) (b)

(c) (d)

Figure 3.4: The two highest entropy views for the tooth dataset are shown in (a) and (b), and the two worst ones in (c) and (d).

viewpoint. Also, for most datasets, a single view does not give enough information to the user. The user will almost certainly want to look at the dataset from another angle. Instead of a single view, it is desirable to present to the user a set of views such that, together, all the views in the set provide a complete visual description of the dataset. This can also be thought of as a solution to the best N views problem: given a positive number N, we want to find the best N views which together give the best visual representation of the dataset.

We propose to find the N views by partitioning the view sphere into N disjoint partitions, and selecting a representative view for each partition. A similar partitioning is defined by aspect graphs [44][45], where each node (aspect) of the graph represents a set of stable views. Each set shows the same group of features on the surface of the object. However, the aspect graph creation methods deal mostly with algebraic and polygonal models and their topology, and cannot be applied in a straightforward manner to volume rendering. Instead, we compute the partitioning by grouping similar viewpoints together.

3.5.1 View Similarity

To find the (dis)similarity of viewpoints, we use the visual probability distributions associated with each viewpoint (section 3.3.1). Popular measures for computing the dissimilarity between two distributions $\mathbf{p}$ and $\mathbf{p}'$ are the relative entropy (also known as the Kullback-Leibler (KL) distance), and its symmetric form (known as divergence) which is a true metric [5]. (Please note that some

texts refer to the KL distance as divergence instead.):

$$D(\mathbf{p}\|\mathbf{p}') = \sum_{j=0}^{J-1} p_j \log \frac{p_j}{p'_j} \tag{3.5}$$

$$D_s(\mathbf{p},\mathbf{p}') = D(\mathbf{p}\|\mathbf{p}') + D(\mathbf{p}'\|\mathbf{p}) \tag{3.6}$$

Although these measures have some nice properties, there are some issues with these measures that make them less than ideal. If $p'_j = 0$ and $p_j \neq 0$ for any j, then $D(\mathbf{p}\|\mathbf{p}')$ is undefined. In our case, any voxel which is fully occluded (zero visibility) will get a visual probability q_j of zero (equation (3.3)). If it is visible in one view but occluded in the other, we cannot evaluate equation 3.5 for these views. Also, $D(\mathbf{p}\|\mathbf{p}')$ and $D_s(\mathbf{p},\mathbf{p}')$ do not offer any nice upper-bounds. To overcome these problems, we instead use the Jensen-Shannon divergence measure [49]:

$$JS(\mathbf{p},\mathbf{p}') = JS(\mathbf{p}',\mathbf{p}) = K(\mathbf{p},\mathbf{p}') + K(\mathbf{p}',\mathbf{p}) \tag{3.7}$$

$$\text{where, } K(\mathbf{p},\mathbf{p}') = D(\mathbf{p}\|(\frac{1}{2}\mathbf{p} + \frac{1}{2}\mathbf{p}')) \tag{3.8}$$

The distance between two views V_1 and V_2, with distributions $\mathbf{q}_1$ and $\mathbf{q}_2$, is then defined as $JS(\mathbf{q}_1,\mathbf{q}_2)$. This measure does not have the zero visual probability problem, since the denominator of the log term is zero iff the numerator is zero. It is also nicely bounded by $0 < JS(\mathbf{q}_1,\mathbf{q}_2) < 2$. Moreover, it can be expressed in terms of entropy [49], which allows us to reuse the view information calculations given in equation (3.4):

$$JS(\mathbf{q}_1,\mathbf{q}_2) = 2H(\frac{1}{2}\mathbf{q}_1 + \frac{1}{2}\mathbf{q}_2) - H(\mathbf{q}_1) - H(\mathbf{q}_2) \tag{3.9}$$

3.5.2 View Likelihood and Stability

We can now use the definition of view-distance given by equation 3.9 to define two additional characteristics of viewpoints- view likelihood and view-stability. View likelihood of a view V is defined as the probability of finding another view anywhere on the view-sphere whose view-distance to V is less than a threshold. In our scenario, it is given as the number of view samples on the view sphere that are within the threshold of V. If a view has a (relatively) high likelihood, it implies that the object or dataset projects a similar image for a (relatively) large number of views. On the other hand, a view with low likelihood provides information that is unique to a few views. This property is indirectly taken into consideration when we partition the set of all the view samples (that is, the view sphere). Large partitions have views with high likelihoods.

Sometimes it is not the view itself but the change in view that provides important information. If the view is changed from one viewpoint to another very similar view, the user does not see much new information. But, if the rendering changes by a large amount, the user sees not just the new information in the visualization but also derives knowledge from the change that has occurred. Occlusion is one of the most important depth cues that is available to the user when visualizing three-dimensional renderings on a two-dimensional surface. A large change in occlusion implies a large change in visibilities, which results in a large JS distance between two viewpoints. This concept is captured by view-stability, which is another view property that can be used to select viewpoints during interaction. It is defined as the maximal change that occurs when the viewpoint is moved anywhere within a

given radius from its original position. The greater the change, the more unstable a viewpoint is. We calculate the (un)stability as the maximum view-distance between a view sample and its neighboring view samples in the triangular tessellation of the view-sphere. The 180° viewpoint in figure 3.3(b) is an unstable viewpoint for this particular viewpoint sequence.

3.5.3 Partitioning

Once the visual probability functions (q) and their entropies (H) are calculated as described in section 3.3, we use the JS-divergence to find the (dis)similarities between all pairs of view samples. We then cluster the samples to create a disjoint partition of view sphere. The number of desired clusters can be specified by the user. Each partition represents a set of similar views, i.e., these views show the voxels at similar visibilities. If desired, the JS-measure can be weighted using the physical distance between the view samples to yield tight regional clusters.

The best (highest entropy) views within each partition are selected as representatives of the cluster and displayed to the user. Together, this set of images give a good visualization of the dataset from many different viewpoints. Sometimes, it might happen that the selected representatives of two neighboring partitions lie on the common boundary and next to each other. If the view distance between two selected view samples is less than a threshold, we use a greedy approach and select the next best sample.

We have used the clustering package CLUTO [41] to perform clustering. It is designed to work with both low and high dimensional datasets. A direct clustering method was used, as opposed to partitioning or agglomeration algorithms. Ten

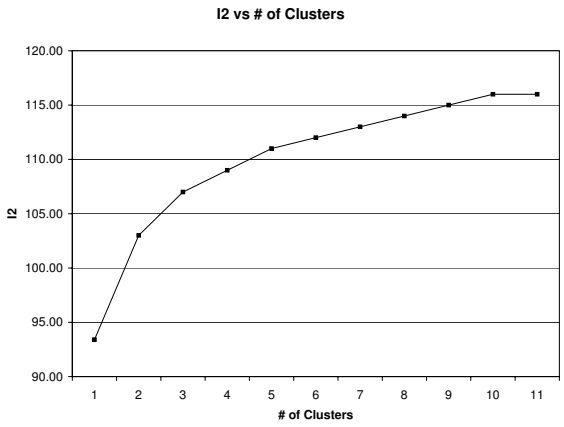


Figure 3.5: Plot of $I2$ vs number of clusters.

different clustering trials were performed, each using different seed points, and the best clustering was selected. We used the $I2$ criterion function [91]. $I2$ captures the cumulative similarities between the cluster members and the centroids of the clusters. Maximizing $I2$ thus maximizes intra-cluster similarity. If k is the total number of clusters, S_i is the set of objects assigned to the i th cluster, n_i is the number of objects in the i th cluster, v and u represent two objects, and $sim(v, u)$ is the similarity between two objects, then $I2$ can also be found using the following formula:

$$I2 = \sum_{i=1}^{k} \sqrt{\sum_{v,u \epsilon S_i} sim(v, u)} \tag{3.10}$$

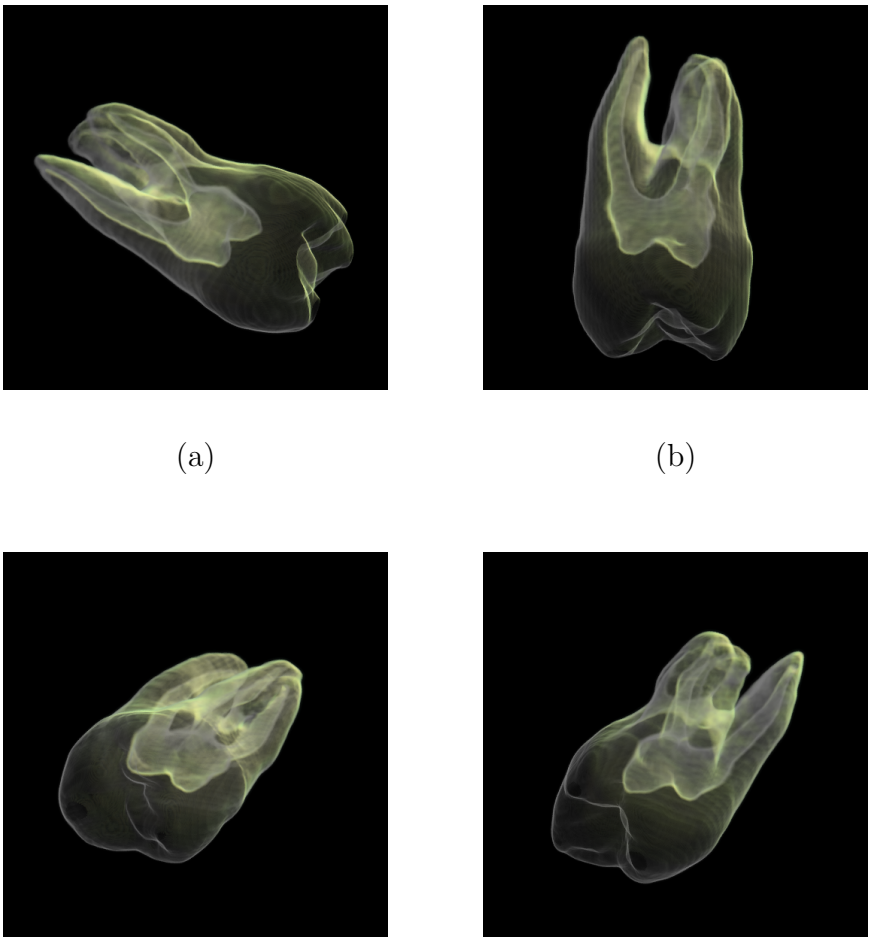
Figure 3.5 shows the values of $I2$ clustering performed on the tooth dataset. It can be seen that the slope of the graph decreases when we move from 5 clusters

to 6 clusters. Figure 3.6 shows the results of a 5-way partitioning of the view space for the dataset. 128 view samples were used with a JS-divergence measure. The largest partition contains 39 samples, while the smallest one has 18. The representative views from four of the partitions are shown. The view for the fifth partition is figure 3.4(a). Figures 3.4(a) and 3.4(b) both lie in the same partition. In fact, the top ten high entropy viewpoints fall in the same partition. They show the same voxels and capture similar information. Although they are good views when considered individually, they contain a lot of redundant information as a group. This illustrates the need for selecting representative views from different partitions.

3.6 Time Varying Data

Suggestion of good views becomes all the more useful in the case of time dependent data. The time required to compute a volume rendering animation of the dataset grows with the number of time steps. In an interactive setting, this creates a large lag between a viewpoint update and the completion of all the frames. Moreover, it takes more tries by the user to find the desired viewpoint because the data changes with time, and the user has to consider not only the current time step but also the previous and future ones. The user's job is made harder by cases where an interesting view in a few time steps turns out to be a dull view in the rest.

In section 3.3, we discussed the notion of a good view and presented a measure of view information for a volume dataset. For time-dependent data, using equation (3.4) separately for each time-step is not the desired solution— it can yield



(a) (b)

(c) (d)

Figure 3.6: Representative views for a 5-way partitioning of the view-sphere for the tooth dataset. The view for the fifth partition is figure 3.4(a).

viewpoints in adjacent time-steps that are far from each other, thus resulting in abrupt jumps of the camera during the animation. A natural solution is to constrain the camera, but it still does not guarantee the most informative viewpoint. For instance, it can result in a viewpoint which has a high information value for each individual time-step, but does not show any time-varying changes. It is contrary to what is expected from an animation— it should show both the data at each time-step, and also the changes occurring from one frame to the next. In the next section, we present an alternate version of viewpoint information tailored to capture the view information present in one time-step, taking into account the information present in the previous step.

3.6.1 View Information

Consider two random variables X and Y with probability distributions $\mathbf{r}$ and $\mathbf{s}$ respectively. If X and Y are related (not independent), then an observation of X gives us some information about Y. As a result, the information carried by Y, conditional on observing X, becomes $H(Y|X) \equiv H(\mathbf{s}|\mathbf{r})$. Then the information carried together by X and Y is $H(X,Y) = H(Y,X) = H(X) + H(Y|X)$, as opposed to $H(X)+H(Y)$. We will use this concept to create a modified viewpoint goodness measure for time dependent data.

Suppose there are n time-steps $\{t_1, t_2, \ldots, t_n\}$ in the dataset. For a given view V, we denote the entropy for time-step t_i as $H(V, t_i) \equiv H_V(t_i)$. The view entropy for all the time-steps together is $H_V(t_1, t_2, \ldots, t_n)$. We will assume a Markov sequence model for the data, i.e., the data in any time-step t_i is dependent on the data of the time-step t_{i-1}, but independent of older time-steps. Then the

information measure for the view, for all the time-steps taken together, is given by equation (3.12). (3.11) is a standard relation [5], and (3.12) follows from the independence assumption.

$$\begin{aligned} H(V) &= H_V(t_1, t_2, \ldots, t_n) \\ &= H(t_1) + H(t_2|t_1) + \ldots + H(t_n|t_1, \ldots, t_{n-1}) \qquad (3.11) \\ &= H(t_1) + H(t_2|t_1) + \ldots + H(t_n|t_{n-1}) \qquad (3.12) \end{aligned}$$

The conditional entropies will be defined following the same principles outlined in section 3.3. We consider a view to be good when the visibilities of the voxels are in proportion to their noteworthiness. But in the time-varying case, the significance of a voxel is derived not only from its opacity, but also from the change in its opacity from the previous time-step. For the time-step t_i, we then define the noteworthiness factor of the jth voxel as $W_j(t_i|t_{i-1}) =$

$$\{k \cdot |\alpha_j(t_i) - \alpha_j(t_{i-1})| + (1-k) \cdot \alpha_j(t_i)\} \cdot I_j(t_i) \qquad (3.13)$$

where, $0 < k < 1$ is used to weight the effects of voxel opacities and the change in their opacities. A high value of k will highlight the changes in the dataset. Suppose the visibility of the voxel for the view V is $v_j(V, t_i)$. Then, the conditional visual probability, $q_j(t_i|t_{i-1})$, of the voxel is

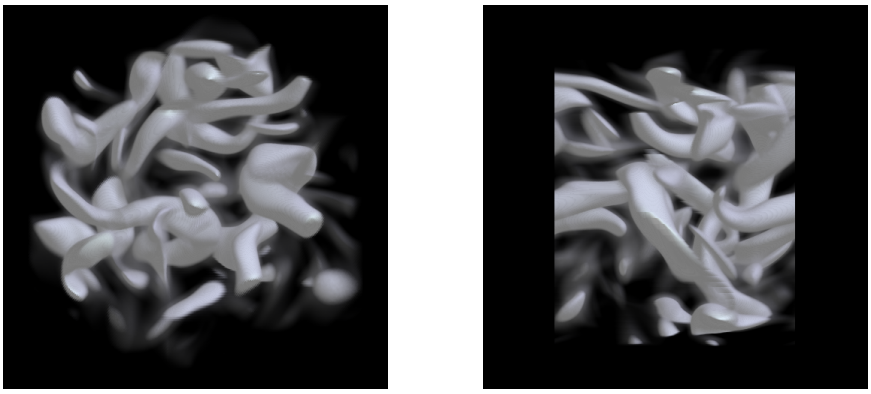
$$q_j(t_i|t_{i-1}) \equiv q_j(V, t_i|t_{i-1}) = \frac{1}{\sigma} \cdot \frac{v_j(V)}{W_j(t_i|t_{i-1})} \qquad (3.14)$$

where, σ is the normalizing factor as in equation (3.3). The entropy of the view V is then calculated using equations (3.12) and (3.14). Voxels with both low opacities and small changes (as defined by thresholds) are ignored for these calculations.

3.7 Results and Discussion

We have implemented our technique using a hardware-based visibility calculation which uses the shear-warp method of rendering. 128 sample views were used for each dataset. The camera positions were obtained by a regular triangular tessellation of a sphere with the dataset place at its center. View selection results for the $128 \times 128 \times 80$ tooth dataset have been shown in figure 3.4. Figure 3.6 shows the results of a 5-way view space partitioning for the dataset using the JS divergence measure. The partitioning helps to avoid selection of a set of good views which happen to be similar to each other. Even though we have not considered the physical distance between the viewpoints during partitioning, it forces the selected viewpoints to be well distributed over the view sphere. Figure 3.7 shows view evaluation results for a 128-cube vortex dataset. Both high and low quality views are shown for comparison.

For time-varying data, we used the view information measure presented in section 3.6. A sequence of 14 time-steps of the 128-cube vortex data was used. The entropy for each view was summed over all the time-steps, as given by equation (3.12). The conditional entropy for each time-step was calculated with $k = 0.9$ in equation (3.13). A high value of k gives more weight to the voxels which are changing their values with time compared to high opacity voxels which remain relatively unaltered. Figure 3.8(a) shows the view with the best cumulative entropy for the time-series. Although the summed entropy gives a good overall view for the whole time-series, there might be other views which are better for particular segments of the time-series. Figure 3.8(b) plots the conditional entropies ($H(t_n|t_{n-1})$) for four



(a) (b)

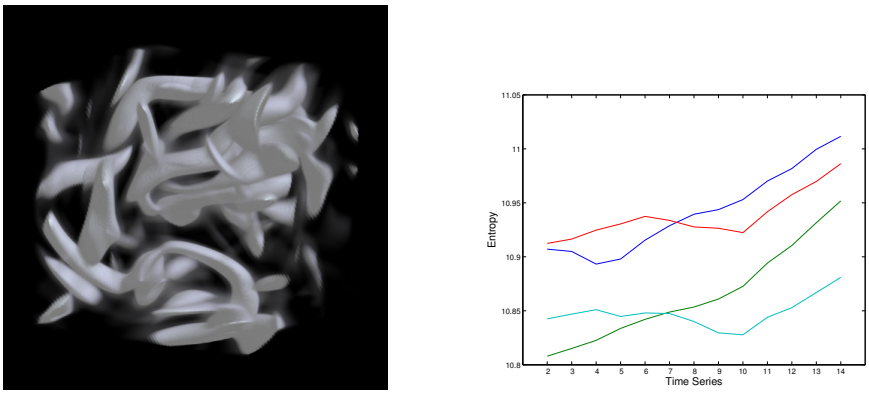
Figure 3.7: View Evaluation results for a 128-cube vortex dataset. Figure (a) shows the recommended view with a high entropy value, (b) shows a bad view for comparison.

selected views of the vortex dataset. The best overall view (figure 3.8(a)), which is represented by the blue curve (highest curve on the right boundary), is not the best choice for the first half of the series. For long time sequences, it might be beneficial to consider different good views for different segments of time.

We calculated the time-varying view entropy over fifty time-steps of the 256-cube shockwave dataset with 128 view samples. Figure 3.9 shows four time-steps from a viewpoint which had a good entropy using the time-varying criteria, and figure 3.10 shows the corresponding time-steps for a viewpoint which resulted in a bad score.

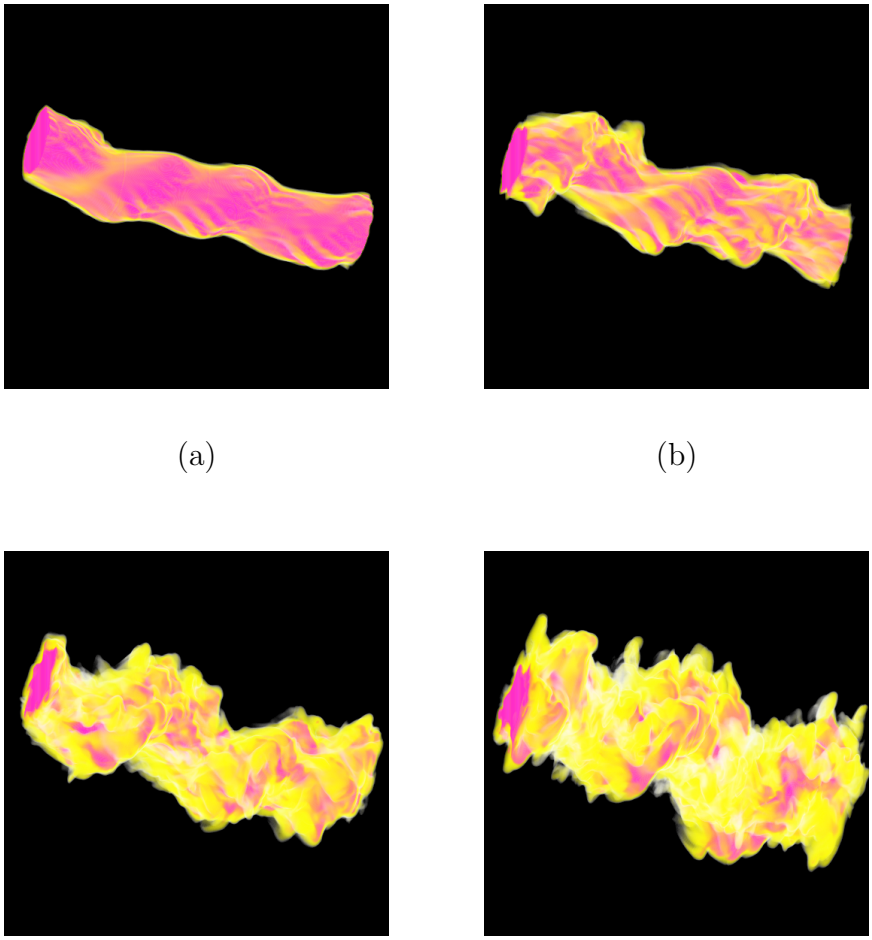
"Out through the fields and the woods
And over the walls I have wended;
I have climbed the hills of view
And looked at the world, and descended;"

— from Reluctance, by Robert Frost.



(a) (b)

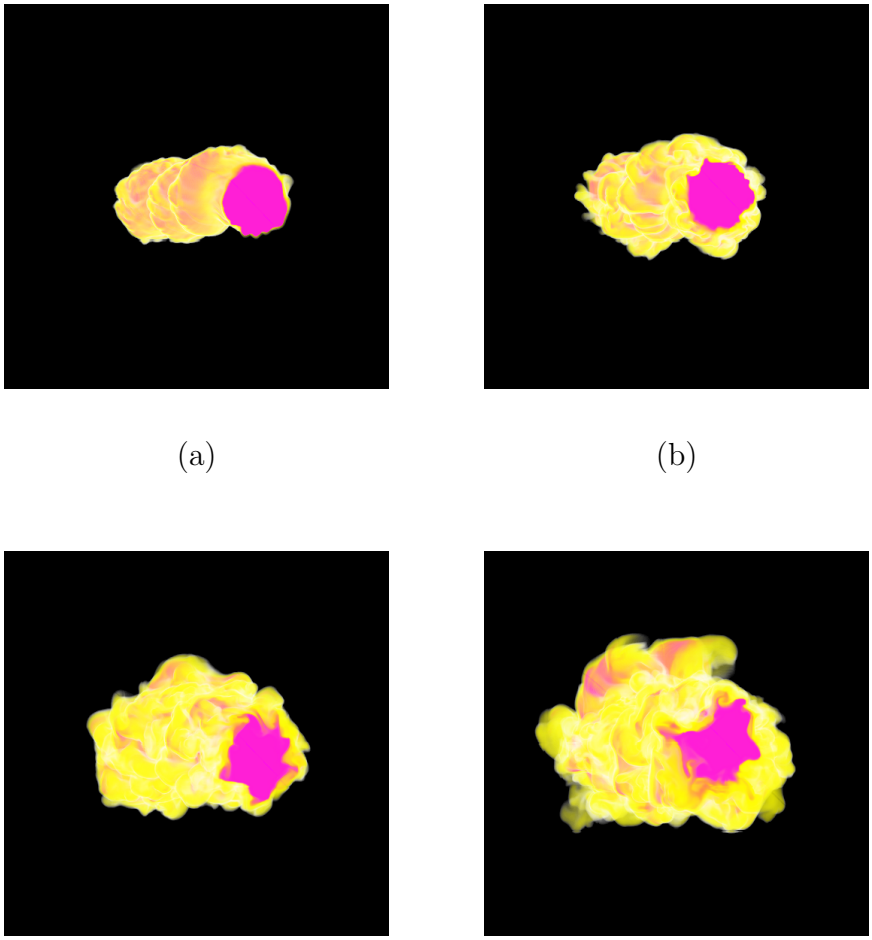
Figure 3.8: View Evaluation for the time-varying vortex dataset. (a) The best overall view for 14 time-steps. (b) The conditional entropies of four selected views for each of the 14 time-steps. The view in (a) is represented by the blue plot (highest curve, top-right corner).



(a) (b)

(c) (d)

Figure 3.9: View entropy results over 50 time-steps of the 256-cube shockwave dataset. Time steps 1, 16, 31 and 46 for a good view.



(a) (b)

(c) (d)

Figure 3.10: Low entropy viewpoint for 50 time-steps of the 256-cube shockwave dataset. Time steps 1, 16, 31 and 46 are shown.

CHAPTER 4

LEVEL OF DETAIL FLOW VISUALIZATION

Visualization techniques for vector fields can be classified into local techniques and global ones. Examples of local techniques include particle traces, streamlines, pathlines, and streaklines, which are primarily used for interactive data exploration. Global techniques such as line integral convolution(LIC)[10] and spot noise[77], on the other hand, are effective in providing global views of very dense vector fields. These techniques are classified as global techniques because directional information is displayed at every point of the field, and the only limitation is the pixel resolution. The price for the rich information content of the global methods, however, is their high computational cost, which makes interactive exploration difficult. These global methods allow users to interact not only by changing the camera, but also by changing texture properties, advection parameters etc. For example, users might want to zoom in/out, change the noise frequency, or modify the advection methods. But every time a parameter is changed, the visualization has to be re-created again from scratch. For high resolution renderings and large datasets, the visualizations cannot be updated at interactive rates.

We have developed a level-of-detail based global vector field visualization technique aiming to tackle the above problems. Our algorithm uses a primitive called *streampatch*, which stores the geometry of flow advection. These primitives can be rendered at different levels of detail, thus allowing a trade-off between accuracy and rendering speed. An error metric is used to find the appropriate detail level. Our framework decouples the advection and the rendering stages, and as a consequence increases user interactivity by reusing the advection calculations and giving the user greater flexibility in choosing input textures. The error can also be used as a guide for generating visualizations which attract the users attention to the more important regions.

This chapter is organized as follows. In section 4.1, we mention the previous research in flow visualization. We then present the level-of-detail algorithm for 2D steady state flow (section 4.2). We then discuss the level of detail selection process (section 4.3). The hardware acceleration issues are presented next (section 4.4), followed by the results and examples of use of LoD in 2D fields(section 4.5).

4.1 Flow Visualization

4.1.1 Texture Based Methods

Texture based methods are the most popular techniques for visualization of dense vector fields. Spot noise, proposed by Van Wijk[77], convolved a random texture along straight lines parallel to the vector field. Another popular technique is the Line Integral Convolution (LIC). Originally developed by Cabral and Leedom[10], it uses a white noise texture and a vector field as its input, and results in an output image which is of the same dimensions as the vector field. Stalling and

Hege[69] introduced an optimized version by exploiting coherency along streamlines. Their method, called 'Fast LIC', uses cubic Hermite-interpolation of the advected streamlines, and optionally uses a directional gradient filter to increase image contrast. Forssell[22] applied the LIC algorithm to curvilinear grids. Okada and Kao[57] used post-filtering to increase image contrast and highlight flow features. Forssell[22] and Shen et. al[65] extended the technique to unsteady flow fields. Verma et. al[81] developed an algorithm called 'Pseudo LIC' (PLIC) which uses texture mapping of streamlines to produce LIC-like images of a vector field. They start with a regular grid overlaying the vector field grid, but they compute streamlines only over grid points uniformly sub-sampled from the original grid. Jobard and Lefer[40] applied texture mapping techniques to create animations of arbitrary density for unsteady flow.

4.1.2 Level of Detail

Level-of-detail algorithms have been applied in various forms to almost all areas of visualization, including flow visualization[25],[73]. Cabral and Leedom[11] used an adaptive quad-subdivision meshing scheme in which the quads are recursively subdivided if the integral of the local vector field curvature is greater than a given threshold. We use a similar subdivision for our level-of-detail approach. Depending on the error-threshold, our algorithm can produce visualizations spanning the whole range from high-fidelity images to preview-quality (high frame-rate) images. Being resolution independent, it allows the user to freely zoom in and out of the vector field at interactive rates. Unlike many variations of LIC which require post-processing steps like equalization, a second pass of LIC, or high-pass filtering[57],

our method does not need any extra steps. Moreover, changing textures and/or the texture-mapping parameters can allow us to produce a wide range of static representations and animations.

4.2 Level of Detail Overview

In this section we present an interactive algorithm for global visualization of dense vector fields. The interactivity is achieved by level-of-detail computations and hardware acceleration. Level-of-detail approximations make it possible to save varying amounts of processing time in different regions based on the local complexity of the underlying vector field, thus providing a flexible run-time user-controlled trade-off between quality and execution time. Hardware acceleration allows us to compute dense LIC-like textures more efficiently than line integral convolution. Use of graphics hardware makes it possible to display the vector field at very high resolutions while maintaining the high texture frequency and low computation times.

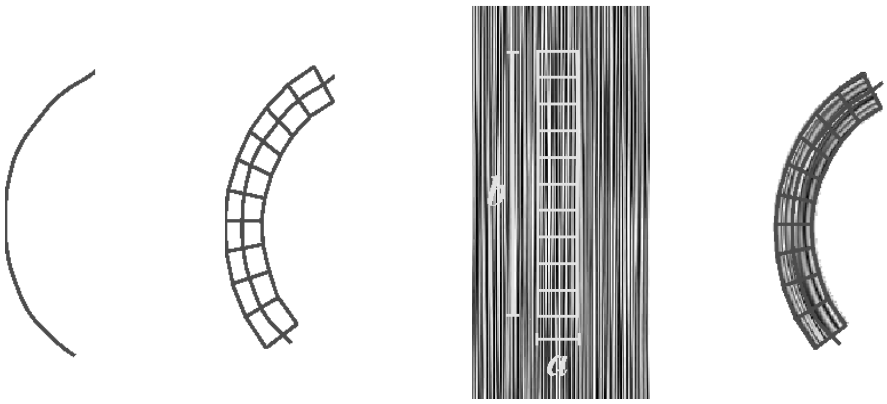
To perform level-of-detail estimation, we define an error measure over the vector field domain (section 4.3.1). As a preprocessing step, we then construct a branch-on-need (BONO)[87] quadtree which serves as a hierarchical data structure for the error measure. The error associated with a node of the quadtree represents the error when only one representative streamline is computed for all the points within the entire region corresponding to the node. At run time, the quadtree is traversed and the error measure stored in each node is compared against a user-specified tolerance. Using the level-of-detail traversal we are able to selectively reduce the number of streamlines required to generate the flow textures. In section 4.3.2,

we discuss how the quadtree traversal is controlled based on the resolution of the display.

Hardware accelerated texture mapping is used to generate a dense image from the scattered streamlines output by the traversal phase of the algorithm. During the above mentioned quadtree traversal, quad blocks of different levels corresponding to different spatial sizes are generated. For each region, a streamline is originated from its center. A quadrilateral strip, with a width equal to the diagonal of the region, is constructed following the streamline. This ensures that the entire region is covered. Henceforth we will refer to this quadrilateral strip as a 'stream-patch' and to the streamline as the 'medial streamline'. The stream-patch is then texture mapped with precomputed LIC images of a straight vector field. The texture coordinates for the quad-strip are derived by constructing a corresponding quad-strip at a random position in texture space. Figure 4.1 shows the construction and texture mapping of a stream-patch, and details are presented in section 4.4.1. The stream-patches for different regions are blended together (section 4.4.2). Each stream-patch extends beyond the originating region, covering many regions lying on its path. If a region has already been drawn over by one or more adjacent regions' stream-patches, it is no longer necessary to render the stream-patch for the region. In section 4.4.3, we discuss the use of the stencil buffer in graphics hardware to skip such regions.

4.2.1 Extension to 3D algorithm

Li et. al[48] extended the 2D primitives presented in the previous sections to 3D. They use *streamtubes*, a 3D structure corresponding to streampatches in 2D.



(a) (b) (c) (d)

Figure 4.1: Construction of the stream-patch: (a) the medial streamline, (b) the quadrilateral strip constructed on the streamline, (c) texture coordinates corresponding to the vertices of the quad-strip, and the texture parameters a, b, (d) the quad-strip after it has been texture mapped.

The streamtubes are voxelized, and each voxel stores the texture coordinates to be used in texture mapping. By dynamically changing the input texture, they are able to achieve flexible appearance control. The volume output by voxelization of the streamtubes is rendered on the fly using hardware. Since the current hardware can support, at a maximum, only 256-cube volumes, the level-of-detail algorithm cannot offer substantial savings in computation time for this algorithm.

In the following sections, we first explain the level-of-detail selection criteria, and then the hardware acceleration features of the algorithm.

4.3 Level-of-Detail Selection

The level-of-detail selection process involves two distinct phases: (1) construction of a quadtree (for level-of-detail errors) as a preprocessing step, and (2) resolution dependent traversal of the quadtree at run-time with user-specified thresholds. Below, we elaborate on each of these stages.

4.3.1 Error Measures

An 'ideal' error metric for a level-of-detail representation should give a measure of how (in)correct the vector field approximation will be compared to the original field data. The textures produced by the algorithm provide information through unquantifiable visual stimulus. Therefore it is difficult to formally define the 'correctness' of the visualization produced. The fidelity of illustration of the vector field should be measured not from the values of the pixels of the image produced, but from the visual effect that the texture pattern has on the user. We use an error measure which tries to capture the difference between the texture direction at a point, which is the approximated vector direction, and the actual vector field direction at the point.

Consider the texture pattern at a point which is not on the medial streamline, but falls within the stream-patch. For each quad of the quad-strip, the texture direction is parallel to the medial-streamline segment within that quad. So, within each quad, we are approximating the vector field as a field parallel to the medial-streamline. The error can thus be quantified by the angular difference between the directions of the medial streamline and the actual vector field at the sample point.

Since each quadtree node originates a stream-patch that will travel outside the node boundary, the error measure associated with a particular quadtree node should consider all the points that are within the footprint of the stream-patch. To do this, for each quad in the quad-strip, we find the angular difference between the directions of the medial streamline segment and each of the vector field's grid points within the quad. The error for a stream-patch originated from a quadtree node is then defined as the maximum angular difference for the grid points across all quads of that stream-patch. Since the error would depend on the length of the stream-patch (which is user-configurable), we take a conservative approach and calculate the errors assuming large values of length. Note that since taking the maximum angular deviation as the error always keeps the error below the user-specified tolerances, it can be very sensitive to noise. For noisy data, taking a weighted average might prove helpful. The level-of-detail approximation errors for a particular level are computed by constructing stream-patches for all the quadtree nodes for that level and then computing the errors for each stream-patch. The error values for the vector field in figure 4.7 are shown in figure 4.2, and those for the vector field in figure 4.8 are in figure 4.3. The error values are normalized to the range [0.0,1.0] to make them user friendly.

It can be seen from the images in figure 4.2 that in any particular level, the regions around the critical points (the three vortices and two saddle points) have the highest error values. The critical points do not need any special handling as they would be represented by stream-patches of the finest level-of-detail allowed by the display resolution (section 4.3.2). Because of high curvature around critical

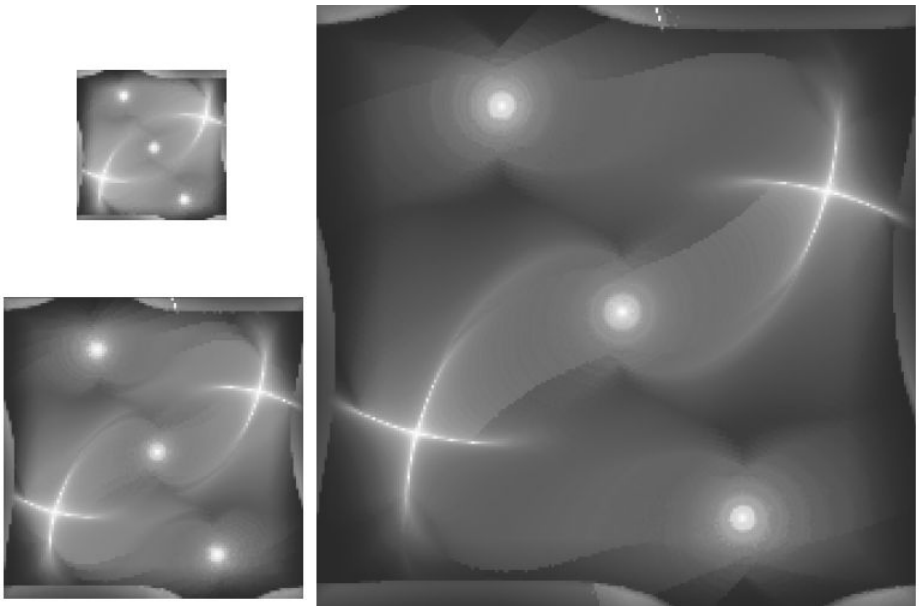


Figure 4.2: Multi-level error for the vector field shown in figure 4.7. The images shown correspond to the following three levels of the error quadtree: 16x16 (top-left), 8x8 (bottom-left) and 4x4 (right) square regions. The error value range is mapped to [0.0,1.0], with 0.0 being the darkest and 1.0 the brightest.

points, use of thicker stream-patches would have resulted in artifacts due to self-intersections. The errors gradually fall off as we move away from the critical points, and hence would result in progressively coarser level-of-detail stream-patches. Also, errors for any particular region increase across levels. So, a high error threshold will permit a coarse level-of-detail approximation. Similarly, in figure 4.3, the regions near the vortices have the highest errors. Since no run-time parameters are required for the error calculations, this error quadtree needs to be generated only once for the entire life of the dataset. At run-time, it can be read in along with the dataset.

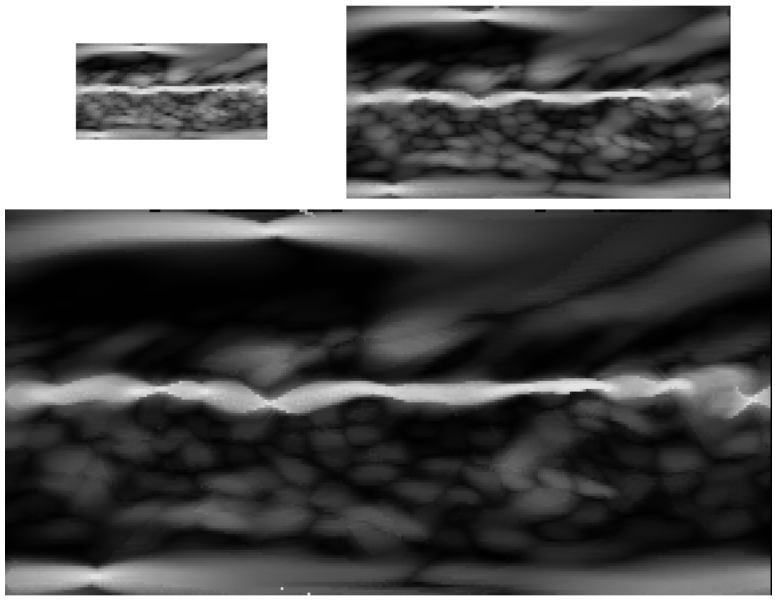


Figure 4.3: Multi-level error for the vector field shown in figure 4.8. The images shown correspond to the levels 8x8 (top-left), 4x4 (top-right) and 2x2 (bottom) of the error quadtree. The error value range is mapped to [0.0,1.0], with 0.0 being the darkest and 1.0 the brightest.

4.3.2 Resolution dependent level-of-detail selection

In this section we describe the run-time aspects of the level-of-detail selection phase, which requires the user to input a threshold for acceptable error. We shall call the ratio of the vector field resolution to the display resolution the *resolution ratio*, k. Consider, for example, an $x_v \times y_v$ vector field dataset, and suppose our visualization window resolution is $x_w \times y_w$. Let

$$x_v = k \times x_w; \quad y_v = k \times y_w \tag{4.1}$$

Note that in an interactive setting, the value of k changes if the user zooms in/out. If the display window has a resolution not smaller than that of the vector field, i.e,

$k \leq 1$, the quadtree is traversed in a depth first manner, and stream-patches are rendered for quad blocks satisfying the error condition. In cases the display resolution is smaller, i.e., $k > 1$, the quadtree traversal is performed using resolution dependent tests in addition to the error threshold test. The resolution dependent controls in traversal are motivated by two goals: (1)we want to limit the quadtree traversal to the minimum block size which occupies more than one pixel on the display, and (2)we want to avoid a potential popping effect caused by a changing k which causes the above mentioned minimum block size to go up (or down) by one level.

For the discussion below, let us assume that at a particular instant $m > k > \frac{m}{2}$, where $\frac{m}{2} = 2^i,\ i = \{1, 2, ...\}$. Since $\frac{m}{2} \times \frac{m}{2}$ blocks occupy a display area less than the size of one pixel, we limit the quadtree traversal to the $m \times m$ block level. Now, if the user zooms in, k becomes progressively smaller, and at some point of time we will have $k = \frac{m}{2}$. At this instant, the minimum displayable block size becomes $\frac{m}{2}$. A lot of $m \times m$ blocks (those that do not satisfy the error threshold) will be eligible to change the level-of-detail to $\frac{m}{2} \times \frac{m}{2}$. If allowed to do so, it will result in a popping effect. To avoid this, we allow only a very small number of $m \times m$ blocks to change their level-of-detail to $\frac{m}{2} \times \frac{m}{2}$. As the user keeps zooming in, k continues to decrease, and we gradually allow more and more blocks to change their level-of-detail. If the user continues to zoom in, by the time k becomes equal to $\frac{m}{4}$, all the $m \times m$ blocks will have changed into $\frac{m}{2} \times \frac{m}{2}$ blocks. This gradual change in the level-of-detail is achieved by modifying the error threshold test for $m \times m$ blocks. The user supplied error threshold is scaled to a high value when

$k = \frac{m}{2}$. As k decreases, we decrease the scaled threshold, such that by the time $k = \frac{m}{4}$, the threshold reaches its original user supplied value.

4.4 Hardware Acceleration

After the quadtree traversal phase has resolved the levels-of-detail for different parts of the vector field, stream-patches are constructed for each region corresponding to its level-of-detail. They are then texture mapped and rendered using graphics hardware (section 4.4.1). The different sized stream-patches need to be blended together to construct a smooth image (section 4.4.2). An OpenGL stencil buffer optimization is used to further reduce the number of medial streamlines computed (section 4.4.3).

4.4.1 Resolution Independence

For each quadtree node that the traversal phase returns, a stream-patch is constructed using the medial streamline for that node. The texture coordinates for the stream-patch are derived by constructing a corresponding quad-strip at a random position in the texture space (figure 4.1(c)). The parameters a (width) and b (height) of the corresponding texture space quad-strip determine the frequency of the texture on the texture mapped stream-patch.

Since we are essentially rendering textured polygons, the output image can easily be rendered at any resolution. When the window size is the same as the vector field size, that is, the resolution ratio k is unity (equation (4.1)), the stream-patches have a width equal to the diagonal of the quadtree nodes they correspond to. When k is changed (e.g., when the user interactively zooms in/out, or when

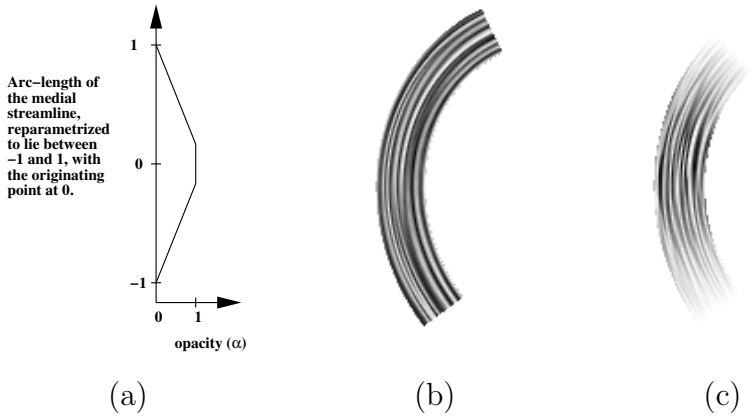
the window size is changed), the width of the stream-patches at each level-of-detail is modulated by $\frac{1}{k}$. Simultaneously, a and b need to be changed to reflect the change in k. Otherwise, when we zoom out, the texture shrinks leading to severe aliasing (figure 4.9(b)). Note that we cannot use anti-aliasing techniques like mipmaps as the texture will get blurred and all directional information will be lost. Similarly, for zoom ins, the texture is stretched, and we lose the granularity (figure 4.10(b)). For high zoom ins, or for high resolution large format displays, the 'constant texture frequency' feature of our algorithm proves very useful. When the display resolution is finer than the vector field resolution ($k < 1$), we are left with sparsely distributed streamlines. But due to the high texture frequency, the final image gives the perception of dense streamlines, which can be considered to be interpolated from the sparse original streamlines.

4.4.2 Blending Stream-patches

To ensure that the final image shows no noticeable transition between adjacent simplified regions, a smooth blending of neighboring stream-patches needs to be performed. A uniform blending (averaging) will result in two undesirable properties. Firstly, the resultant image will loose contrast. If many stream-patches are rendered over a pixel, its value tends to the middle of the gray scale range. This is specially unsuitable for our algorithm, as the loss of contrast will be non-uniform across the image due to the non-uniform nature of the level-of-detail decomposition of the vector field. Secondly, the correctness of the final image (up to the user-supplied threshold) will be compromised. This happens because a

stream-patch corresponding to a coarser level-of-detail will have a non-zero effect on its neighboring regions, some of which might correspond to finer level-of-details. To prevent coarser level-of-detail stream-patches from affecting the pixel values of nearby finer level-of-detail regions, we use a coarser-level-of-detail to finer-level-of-detail rendering order, combined with the opacity function shown in figure 4.4. The OpenGL blending function used is glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA). Because we use an opacity value of unity at the central part of the stream-patch, the pixels covered by this part are completely overwritten with texture values of this patch, thus erasing previous values due to coarser level-of-detail stream-patches. Moreover, this opacity function results in a uniform contrast across the image, even though the number of stream-patches drawn and blended over different parts of the image varies a lot. To reduce aliasing patterns, we jitter the advection length of medial streamlines in either direction and adjust the opacity function accordingly.

To ensure a smooth transition between adjacent patches, we also vary the stream-patch opacities in the direction perpendicular to the medial-streamline using a similar opacity function. This is done at a cost of increased rendering time since we add one quad strip on each side of the stream-patch so that the opacities can be varied laterally. In our implementation, the user can turn the lateral opacity variation off for high-frame rate preview quality requirements.



(a) (b) (c)

Figure 4.4: Opacity function: (a) the value of α over the length of the stream-patch, (b) the patch without blending, and (c) the stream-patch after blending.

4.4.3 Reducing Streamline Redundancy

Because stream-patches continue beyond their originating regions, each stream-patch would cover many pixels, albeit with different opacities. From our experiments, we found that a pixel goes to an opacity value of unity with the first few stream-patches that are rendered over it. Thus, if a pixel has been drawn over by a few stream-patches, then we do not need to texture map any more stream-patches for this pixel. We use this idea to reduce the number of stream-patches that need to be rendered, and hence the number of medial streamlines that need to be computed. However, among the many stream-patches that may have been rendered over this pixel, only those which are of the same or finer level of detail as this pixel's level-of-detail should be counted.

We avoid doing this 'minimum number of renderings' test in software by using the OpenGL Stencil buffer to keep track of how many stream-patches have been

drawn over a pixel. When GL_STENCIL_TEST is enabled, the stencil buffer comparison is performed for each pixel being rendered to. The stencil test is configured using the following OpenGL functions:

- glStencilFunc(GL_ALWAYS, 0, 0): Specifies the comparison function used for the stencil test. For our purpose, every pixel needs to pass the stencil test (GL_ALWAYS).

- glStencilOp(GL_KEEP, GL_INCR, GL_INCR): Sets the actions on the stencil buffer for the following three scenarios: the stencil test fails, stencil test passes but depth test fails, and both tests pass. In our case, the stencil test always passes, so we increment (GL_INCR) the value of the stencil buffer at the pixel being tested by one.

Before starting to compute the stream-patch for a region, we read the stencil buffer values for all the pixels corresponding to that region. If all the pixels have been rendered to a minimum number of times, we skip the region. If not, we render the stream-patch and the stencil buffer values of all the pixels which this patch covers are updated by OpenGL. While going through our coarser-to-finer level drawing order (as mentioned in section 4.4.2), we clear the stencil buffer each time we finish one level. Otherwise, a finer level-of-detail region which has been drawn over by coarser level-of-detail stream-patches might be skipped because each pixel in the region has already satisfied the threshold of minimum number of renderings. This will violate the error-criteria for the region.

The stencil buffer read operation is an expensive one in terms of time. If done for every stream-patch, it would take so much time that we would be better off

not using it. For our implementation, we read the stencil-buffer once every few hundred stream-patches. The values are reused till we read in the buffer once again.

4.5 Results and Discussion

We present the performance and various visual results of our algorithm implemented in C++ using FLTK for the GUI. The timings are taken on a 1.7 GHz Pentium with an nVidia Quadro video card. The results show that the image quality remains reasonable even when error thresholds are increased to achieve high speedups.

Moreover, various aspects of the visualizations are interactively configurable, as shown by the different visuals presented. We describe how the algorithm can be configured to achieve different effects– interactive zoom-in or zoom-out, embedding scalar information, animation, unsteady flow, multiple textures etc.

4.5.1 Range of Image Quality and Speed

We present results for a simulated dataset of vortices (and saddles) with dimensions of 1000x1000, and for a real 573x288 dataset of ocean winds. The error calculation times for the vortices dataset was 148 seconds, and the ocean dataset required 20 seconds. For each dataset, two images are shown: one with tight error limits, and the other with relaxed bounds. For comparison, the images produced by FastLIC are shown in figures 4.5,4.6. A fourth order adaptive Runge-Kutte integration is used with the same parameters for both FastLIC and our algorithm.

dataset	*FastLIC*	*LOD*(*low error*)	*LOD*(*high error*)
Vortices	12.32*s*	0.81*s*(15.2)	0.39*s*(31.6)
Ocean	1.9*s*	0.32*s*(5.9)	0.16*s*(11.9)

Table 4.1: Timing results for the 1000x1000 Vortices dataset and the 573x288 ocean winds dataset. The timings reported are for FastLIC, and our algorithm for two level-of-detail error thresholds. The speed-ups for the level-of-detail algorithm with respect to FastLIC are shown in parentheses. No post-processing has been done in any of these runs. The images for these runs are in figures 4.5, 4.6, 4.7 and 4.8.

The medial streamlines in our algorithm were advected to the same length as the convolution length used for FastLIC.

Figures 4.7 show the results of our level-of-detail algorithm for the vortices dataset rendered for an 1000x1000 display window. Figure 4.7(a) was generated using a low error threshold in 0.81 seconds, while figure 4.7(b) was produced using a high error threshold in 0.39 seconds. Compared to FastLIC, we achieve speed-ups of 15-30 depending on the error threshold for level-of-detail selection. Figures 4.8(a) and (b) are outputs for the ocean dataset, rendered for a display window of same dimensions. A low error threshold was used for figure 4.8(a); it was relaxed for figure 4.8(b). The times taken were 0.32 and 0.16 seconds respectively, for speed-ups of 5.9-11.9 compared to FastLIC. There is no visible difference in image quality in either dataset for the low error thresholds. For the higher thresholds (figure 4.7(b) and 4.8(b)), the differences are very minute and not readily noticeable.

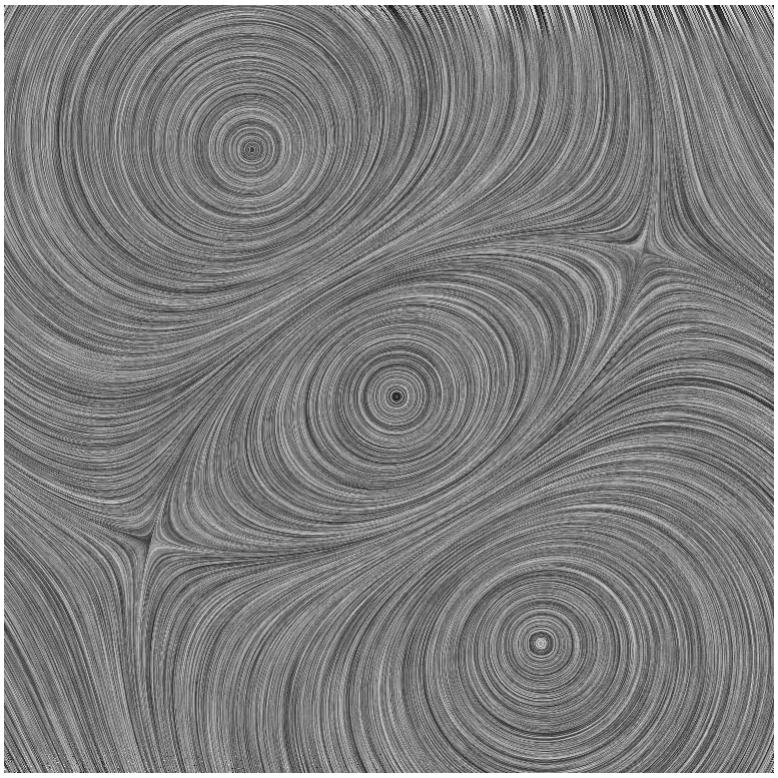


Figure 4.5: LIC image of the 1000x1000 vortices dataset in 12.32s, using convolution length of 60

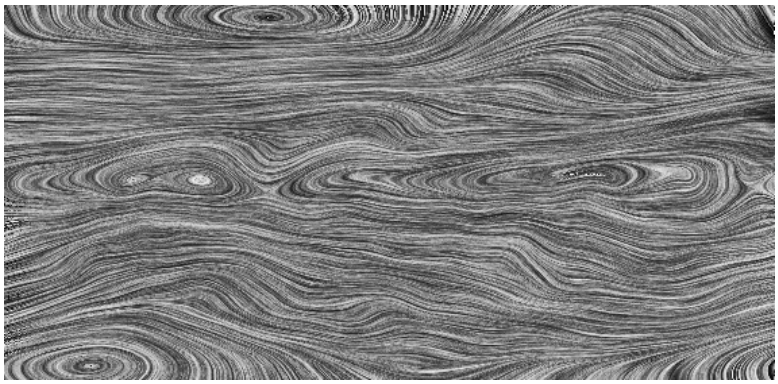
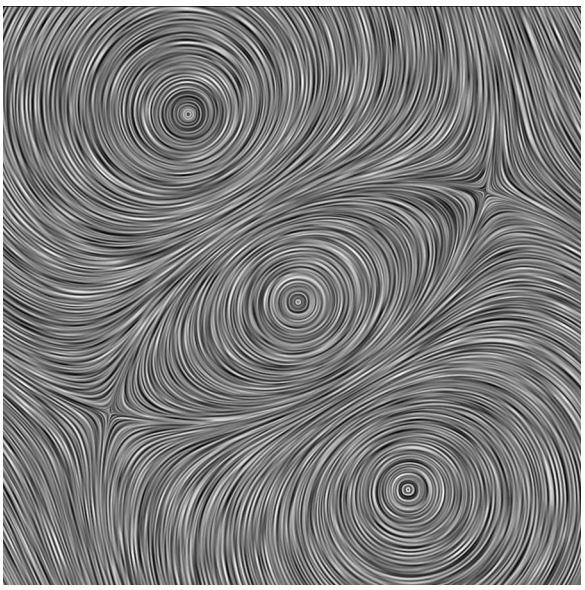
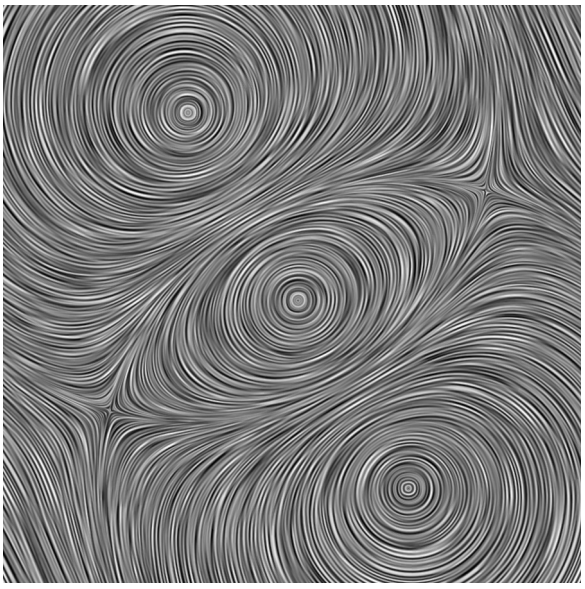


Figure 4.6: LIC image of the 573x288 ocean wind dataset in 1.9s, using convolution length 30.

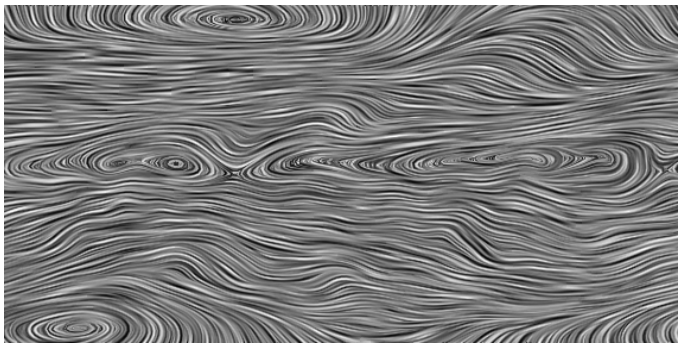


(a)

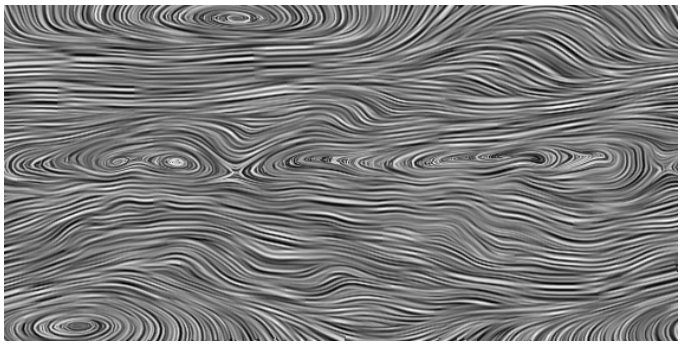


(b)

Figure 4.7: Vortices dataset using: (a)low error threshold in 0.81s, (b)high error threshold in 0.39s. The image quality remains good even for high error thresholds.



(a)



(b)

Figure 4.8: Ocean dataset using: (a)low error threshold in 0.32s, (b)high error threshold in 0.19s. The image quality remains good even for high error thresholds.

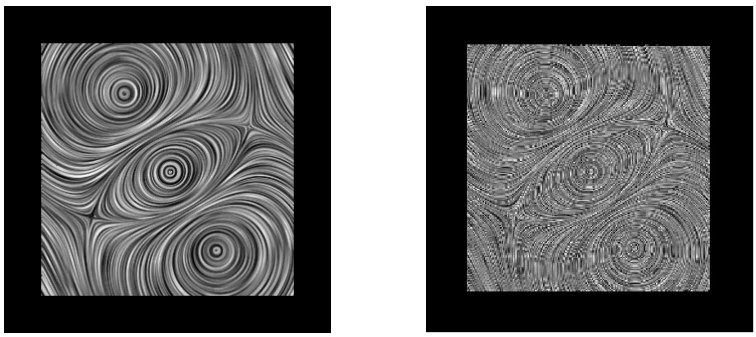
4.5.2 Interactive Exploration

The level-of-detail features allow users to visualize the vector field at a wide range of resolutions. The algorithm automatically adjusts the display parameters to render an unaliased image of a large dataset for display on a small screen. Figures 4.9 (a) and (b) show the results for visualizations on low resolution displays, with and without resolution adjusted parameters. At the other extreme, the vector data can be rendered at very high resolutions when displayed on large format graphics displays, or when viewed at high zoom factors. The hardware texture mapping allows us to change resolutions at interactive frame rates, thus allowing the user to freely zoom into and out of the dataset. Figure 4.10(a) shows the central vortex of the vortices dataset at a magnification factor of 20x. The texture mapping parameters are changed dynamically, so that the texture does not get stretched. For comparison, figure 4.10(b) shows the same rendering when the texture parameters are not adjusted.

4.5.3 Scalar Variable Information

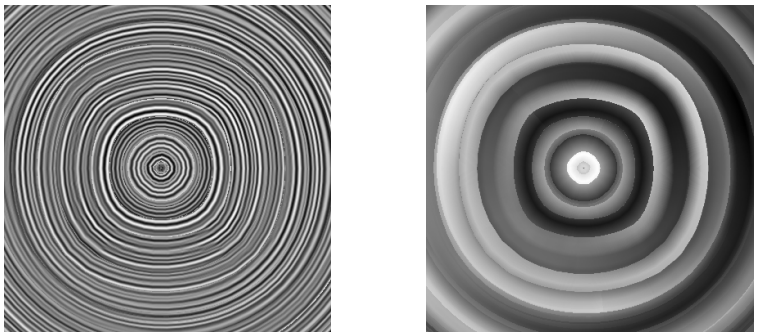
Information about a scalar variable defined on the vector field can be superimposed on the directional representation of the vector field. One technique for that is to modulate the texture frequency of the final image based on the local values of the scalar. Kiu and Banks[43] applied this scheme to LIC images by using noise textures of different frequencies. We follow a different approach of using multiple textures to achieve the same goal.

We start with an ordered set of precomputed textures, in which some texture property varies monotonically from one extreme of the set to the other. For the



(a) (b)

Figure 4.9: Zoom out: The vortices dataset rendered for a display window one-fourth its size: (a)The texture coordinates are adjusted to prevent aliasing, and the finest displayable level-of-detail is adjusted to match display resolution. (b)aliasing results if the image is scaled down without adjusting the texture parameters.



(a) (b)

Figure 4.10: Zoom in: The central vortex of the dataset in figure 4.7 is shown at a magnification factor of 20x: (a)The texture coordinates are adjusted automatically to compensate for the magnification factor, thus maintaining the original texture frequency, (b)the texture loses granularity if the coordinates are not adjusted.

example presented in figure 4.11, the property is the length of the LIC directional pattern. That is, textures of short LIC patterns (produced by small convolution lengths) are at one end of the set and those with long patterns (large convolution lengths) are at the other. For each quad in a stream-patch, different textures are selected as the source for texture mapping based on the value of the scalar, much like mipmaps are used depending on screen area. However, use of different textures in adjacent quads of the quad-strip destroys the directional continuity of the texture mapped strip. As a way around this, in a process analogous to blending two adjacent levels of mipmaps, each quad is rendered twice using textures adjacent to each other in the ordered set of textures. The opacities of the two textures rendered are weighted so that the resultant texture smoothly varies as the scalar value changes. The net effect is a texture property which varies smoothly relative to a scalar value. The detail of the scalar representation in a particular image is limited to the level-of-detail approximation that is used for generating the image. In some situations it may be desired to control the error in the scalar value illustration. Then the quadtree of errors would need to be constructed using both the angular error (section 4.3.1) and the error in the scalar value. Figure 4.11 is generated using the multi-texturing method to show the magnitude of the velocity of the vortices dataset. The short patterns indicate lower velocity, whereas the LIC pattern is long in areas of high velocity.

We can also apply the multiple texture technique to preserve constant texture frequency for vector fields on grids which are not regular. Using the method presented by Forsell[22], a vector field representation is generated on a regular

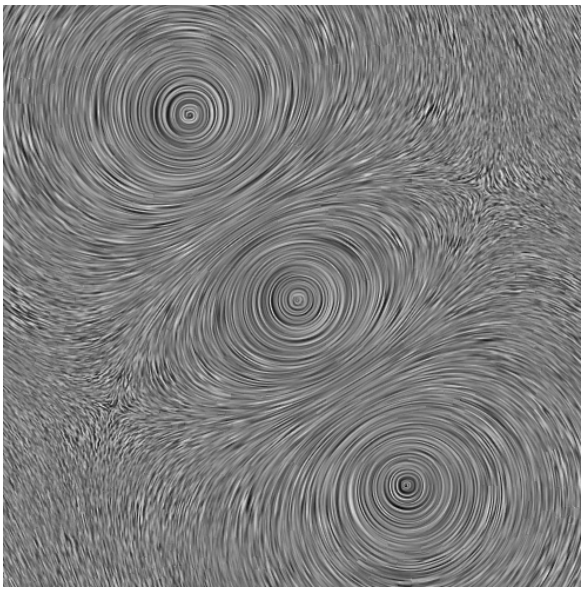
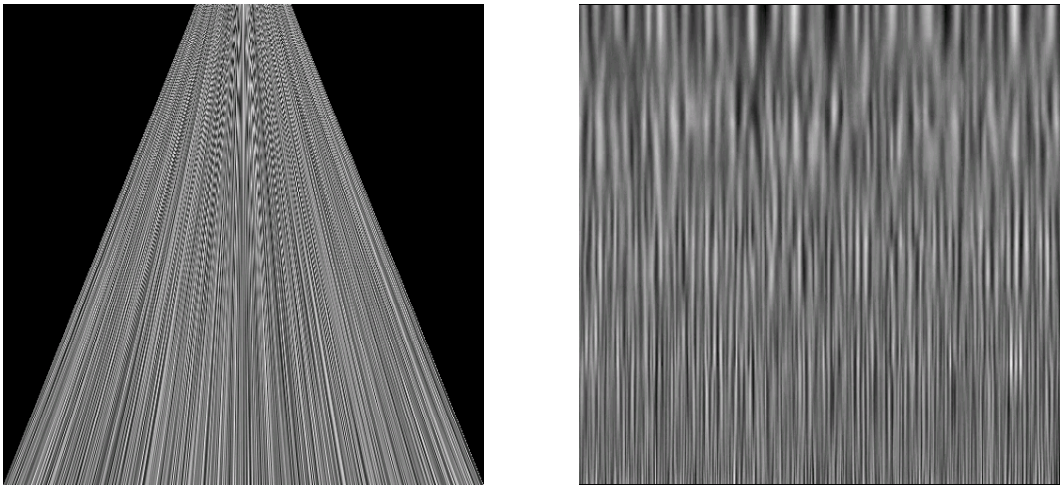
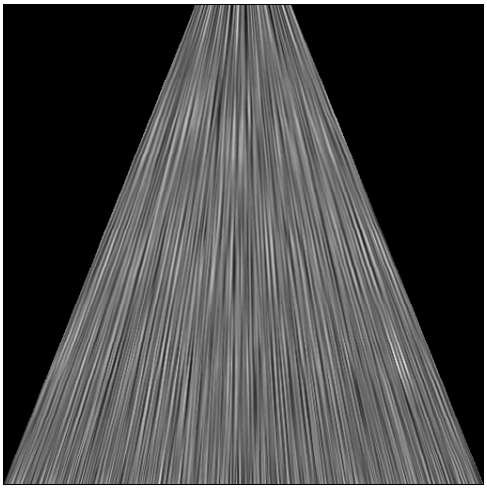


Figure 4.11: Illustration of use of multiple textures to show a scalar variable on the vector field, velocity in this example. The velocity is high in parts around the vortices, and is low at the lower left and upper right corners.

grid. This image is then texture mapped onto the actual surface in physical space. During this mapping, the regular grid cells used for computation are transformed to different sizes in physical space, which can result in the stretching or pinching of the texture. To prevent this, we use textures of varying frequencies for each quad subject to the area the quad occupies in physical space. Figure 4.12(a) shows the result when a constant frequency texture is mapped on a grid in which the cells grow progressively smaller from bottom to top, resulting in aliasing. Figure 4.12(b) is the image generated by our algorithm using multiple frequency textures subject to the grid cell area. The texture frequency decreases from bottom to top, so that when it is mapped to the grid (figure 4.12(c)), there is no aliasing.



(a) (b)



(c)

Figure 4.12: Multiple texture rendering to show antialiasing for a grid which has small cells at the top and large ones at the bottom: (a)constant frequency image mapped to the grid, showing aliasing, (b)multi-frequency texture generated using multiple textures, (c)the texture in (b) mapped to the grid, without aliasing.

4.5.4 Streamline Textures

The level-of-detail technique can be applied with a variety of textures to get diverse visualizations. If we use a sparse texture, it gives the output an appearance of streamline representation. This is a popular method of flow visualization [74],[38], [83].

Figures 4.13(a-c) are generated using a stroke-like texture to give the image a hand drawn feeling. The stroke in the texture is oriented by making the tail wider than the head. This adds directional information to the image. Unlike previous textures, this texture has an alpha component which is non-zero only over the oriented stroke. Figure 4.13(a) is generated by constraining the level-of-detail approximation to a single level-of-detail. This creates an uniform distribution of streamlines. The stencil buffer option (section 4.4.3) is turned on to limit the streamlines from crowding one another. Figures 4.13(b-c) have been rendered using the level-of-detail approximations. Since the level-of-detail is finer near the saddle points and the vortices, more streamlines are drawn near these parts. Due to the relative lack of streamlines in the parts of the vector field which have low errors, the more complex parts of the vector field stand out to the viewer.

4.5.5 Animation

Animation of the vector field images is achieved by translating the texture coordinates of each stream-patch from one frame to another. The only additional constraint is that a cyclic texture be used for texture mapping the stream-patches. For example, the texture in figure 4.1(c) needs to be cyclic along the vertical direction. For each time step, the image is rendered by vertically shifting (downwards)

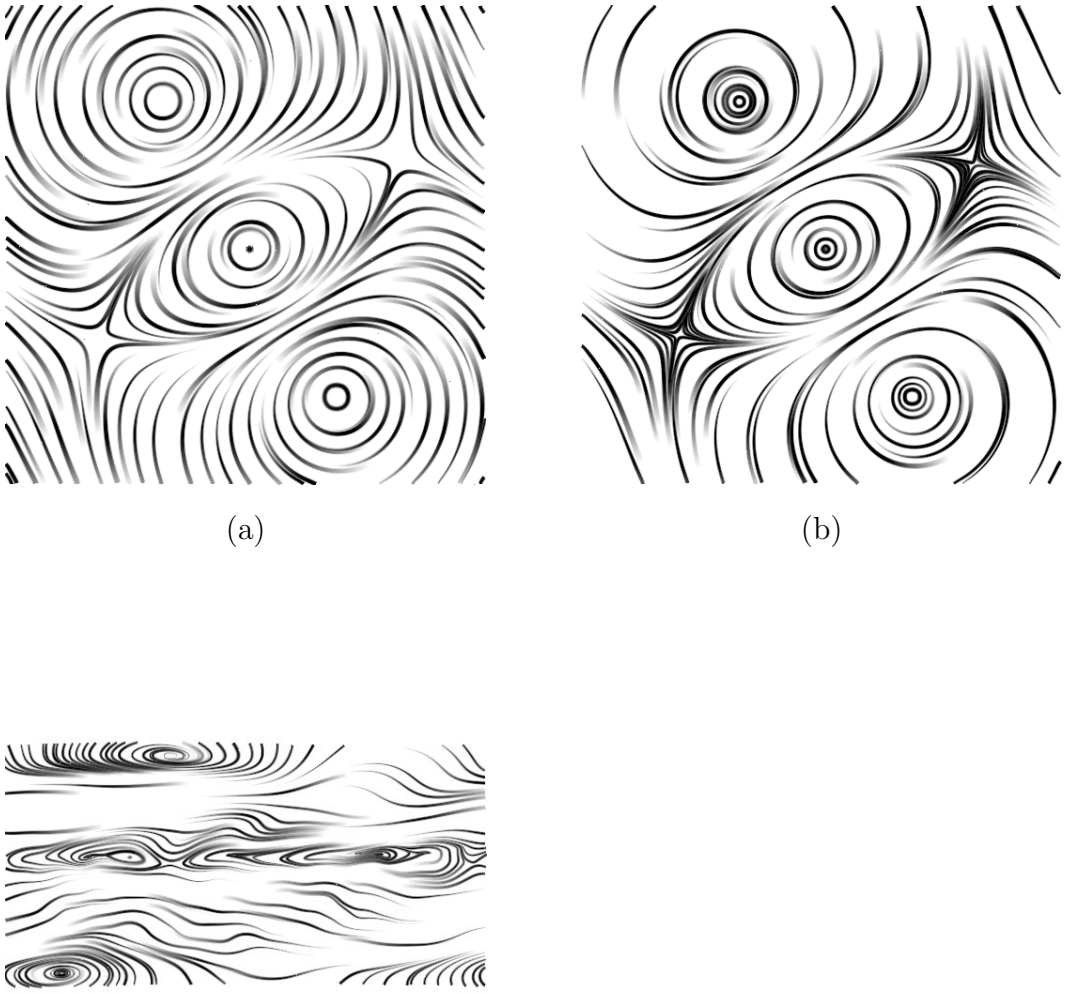


Figure 4.13: Stroke-like textures used to generate streamline representation: (a) Uniform placement of streamlines using constant level-of-detail, (b),(c) streamlines generated using the error quadtree traversal. The level-of-detail algorithm generates a non-uniform distribution of streamlines, giving more detail to higher error regions and vice versa.

the texture coordinates compared to the previous frame. Since the texture is cyclic, if the texture coordinates move past the bottom edge of the texture, they reappear at the top edge. To show different speeds, the texture coordinates for each stream-patch are moved by an amount proportional to the velocity of the seed-point of the patch.

4.5.6 Unsteady Flow

The main benefit of the 2D geometry primitives (and their 3D extension in [48]) is the decoupling of the advection stage and the rendering stage. Since time-varying flow datasets require a lot of calculations in the advection stage, these algorithms are particularly suited for unsteady flow applications.

We have added unsteady flow visualization capability to three-dimensional flow visualization algorithm by Li et. al[48]. As mentioned in section 4.2.1, the algorithm creates 3D geometry primitives from the flow streamlines, and stores them in a volumetric form. Each voxel stores the texture coordinates inherited from the geometry (streamtube) passing through the voxel. The volume is rendered using hardware to generate the final image. The main difference between the processing of steady state and unsteady flows comes from the fact that pathlines can intersect with themselves or each other, while streamlines do not. As a result, for time-varying datasets, there can be voxels in the volume that intersect pathlines more than once, and thus need to store more than one set of texture coordinates. We will explain this situation with the help of an example, shown in figure 4.14(a). The pathlines starting from both A and B pass through the voxel shown. The pathline with coordinates (u_1, v_1) passes through the voxel at time $t = 3$, while

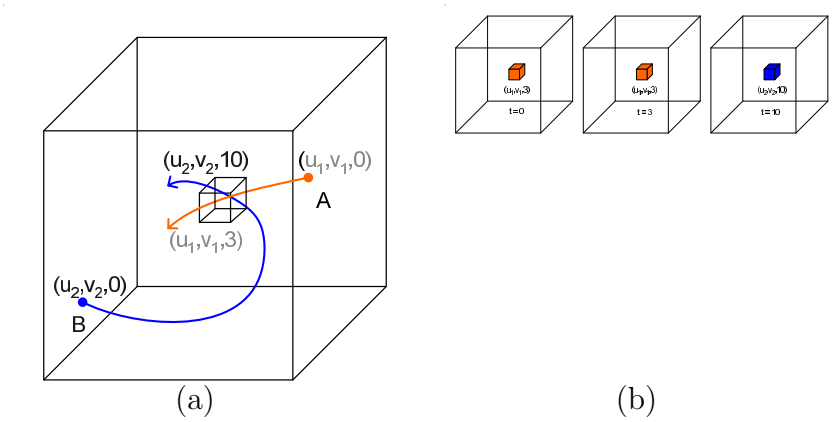


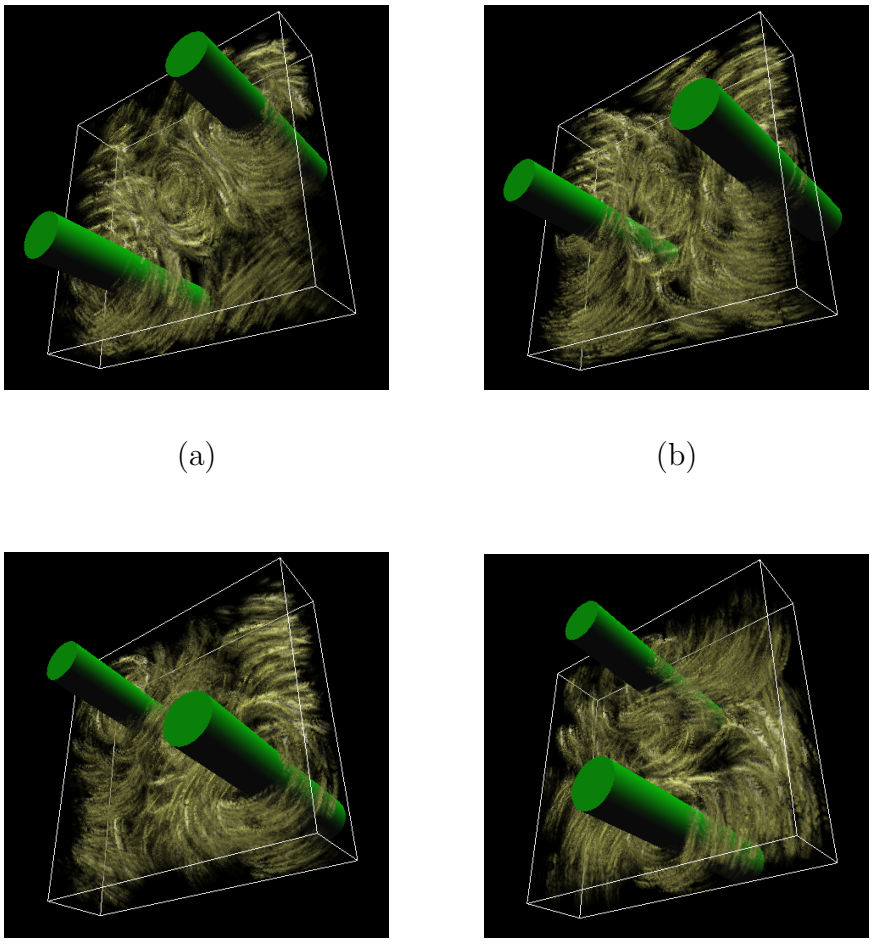
Figure 4.14: Example of voxel update for unsteady flow. (a) Two pathlines pass through the voxel shown, at $t = 3$, and $t = 10$. (b) The texture coordinates corresponding to the pathlines should be written to the voxel *before* the voxel is rendered using those values. (u_1,v_1,3) is written when initially creating the volume, and (u_2,v_2,10) is written after rendering the frame for $t = 3$, but before rendering $t = 10$.

the pathline (u_2, v_2) intersects the voxel at time $t = 10$. Thus the voxel needs to store both (u_1, v_1, 3) and (u_2, v_2, 10). For a correct rendering, the voxel should contain the first coordinates at the time step $t = 3$, and then switch to the second one when the time step equals to 10. To achieve this, the time-varying algorithm will need to perform interactive updates of these coordinates during rendering. Initially, this voxel contains the set with the smallest time-stamp, i.e., (u_1, v_1, 3). At $t = 3$, the voxel is rendered with these values. This set is not needed after $t = 3$, and the voxel should contain the newer coordinates (u_2, v_2, 10) while rendering the frame for $t = 10$. So, the voxel is updated with the second trace tuple information after rendering $t = 3$, but before $t = 10$.

The issue of multiple texture coordinates in a single voxel is handled in the following manner: in a preprocessing stage, pathlines are advected and the voxel contention information (i.e., which voxel to replace at what time-step) is collected. A book-keeping operation is performed to organize the voxels that will intersect the pathlines multiple times. Since those voxels require run-time updates during rendering, if neighboring voxels need to be updated at the same time step, then it is more efficient to update all of them in one go instead of using multiple texture writes for each individual voxel. To do this, the book-keeping operation stores each group of such neighboring voxels into a single *update volume*. Each such volume is generated and stored separately, and there can be multiple update volumes for each time step. After the preprocessing stages complete, the data is visualized by rendering the current volume for each time-step, and dynamically updating the volume with update volumes for every new time-step. Figure 4.5.6 shows four snapshots from an animation of the time-varying algorithm.

"Two roads diverged in a yellow wood,
And sorry I could not travel both
And be one traveler, long I stood
And looked down one as far as I could
To where it bent in the undergrowth;"

— from The Road Not Taken, by Robert Frost.



(a) (b)

(c) (d)

Figure 4.15: Four snapshots from an animation of the vortices data set using the time-varying Chameleon algorithm. The images were generated using a line bundle texture with lighting.

CHAPTER 5

CONCLUSION

In this thesis, we have introduced three algorithms that use importance measures defined on the data to redistribute our resources– compute power, storage, and manpower. These help improve the utility of the visualization techniques in difficult situations, such as when large datasets are used. We outline our contributions below.

5.1 Range-Search using Compressed data-structures

We have presented a data structure for speeding up isosurface extraction using transform coding techniques. We use the spatial coherence present in the data to transform the $[minimum, maximum]$ information of cells into a more compression friendly representation. This information is then quantized in a non-uniform manner, with more storage dedicated to regions of the span-space that have cells with a small difference between their maximum and minimum values. In this scenario, the importance exists in the span-space, where the denser regions (that is, regions with more cells) are given more quantization levels.

Our contributions are as follows:

- We have presented a data structure for speeding up isosurface search. Significant reduction is achieved in terms of the space requirement of the search structures, without compromising the search speed. We achieve compression levels of about 22%. These search structures do not require high precomputation costs, and the implementation is very simple. The only computations involved are additions and subtractions, and sorting.

- We have also presented a search algorithm suitable for large datasets. We achieve search times comparable to the interval trees (see table 2.3). The search algorithm does not need to be modified for out-of-core implementations. We also discuss an incremental search algorithm using our data structure.

- We have presented an analysis of the errors involved in quantization, and have showed the trade-off between search errors and data-structure size as a function of the number of quantization levels.

- The search algorithm can work with both structured and unstructured grid datasets, and can readily incorporate existing methods for reducing the search data-structure size (like using meta-cells and the checker-board patterns).

5.2 Automatic View Selection

We have presented a measure for finding the goodness of a view for volume rendering. We have used the properties of the entropy function to satisfy the intuition that good views show the important voxels more prominently. The user

sets the noteworthiness of the voxels by specifying the transfer function. Our algorithm can be used both as an aid for human interaction in not-so-interactive systems, and also as an oracle to present multiple good views in less interactive contexts. Furthermore, view sampling methods such as IBR can use the sample similarity information to create a better distribution of samples.

Our contributions are as follows:

- We have presented measures for finding different view properties such as view goodness, view stability and view likelihood. We use voxel importances and the view-dependent visibilities in an entropy function form to define the goodness of a view.

- The users are allowed to define the voxel noteworthiness (the voxel importance). Our algorithm can hence easily incorporate domain knowledge. The users are not required to have any visualization expertise to use this method.

- We use the JS-metric to compare the probability vectors for two viewpoints. This is used to compare and contrast two viewpoints using the view stability and view likelihood definitions.

- A view space partitioning scheme is introduced. We partition the viewpoints into different clusters, and select a representative from each cluster. Such a set of representative views captures most of what can be seen of the dataset from all angles, and thus can prove useful in non-interactive situations.

- A GPU-based algorithm for visibility calculation is presented. It is based on the shear-warp algorithm. It speeds up the most time-consuming part our

view selection method, and now the view selection can be done in a matter of few minutes.

- We also present a modification to our entropy definition to incorporate time-varying data.

- Our visibility calculation method is independent of the rotation, translation and scaling parameters of the camera, as long as the camera viewing direction is maintained. In the future, we hope to utilize this fact to relax the camera position from the current restriction of having it lie on view sphere.

5.3 Level of Detail Vector Field Visualization

Using a level-of-detail framework, we have reduced the computation times for creating a dense visualization of vector fields. We have presented an error measure which can discriminate between regions of the vector field based on their susceptibility to errors introduced by simplification. This measure is then used to focus the computational resources on regions which produce greater errors. Coupled with hardware acceleration, the algorithm generates high quality visualizations at interactive rates for large datasets and large displays. The advection and the rendering stages are decoupled, which allows the user to change the textures and other display properties interactively. The resolution independence and user-controlled image quality features make this algorithm extremely useful for vector data exploration.

Our contributions are as follows:

- We have presented a simplification method that can be used to reduce computations required to create a dense visualization of the vector field. Streamlines are used to warp a geometric object, which is the texture mapped to create the appropriate visualization.

- We have presented an error measure which can discriminate between regions of the vector field. It controls the amount of simplification in different regions of the dataset by limiting the errors produced. The error cut-off is input by the user, thus allowing a trade-off between representation accuracy and rendering speed which is controlled by the users.

- We present hardware based techniques to limit redundancy due to overlap of the rendered geometry, and speeding up the process even more.

- By using texture mapped geometry to create the visualization, we decouple the advection and rendering parts of the algorithm. Once the advection is done, we can interactively change the input texture without redoing the advection. This gives us a great deal of flexibility in terms of controlling the final appearance of the visualization. The textures can also be adjusted dynamically to adjust of zoom-in or zoom-out situations, or to reduce antialiasing in curvilinear grids. We have also presented results showing use of multi-texturing, and of alternative textures.

We hope that our research will help, in a small way at least, our users get insights from their data. We also hope that this research will contribute to new ideas and visualization methods.

"The lights begin to twinkle from the rocks:
The long day wanes: the slow moon climbs: the deep
Moans round with many voices. Come, my friends,
'Tis not too late to seek a newer world."

— from Ulysses, by Alfred Lord Tennyson.

BIBLIOGRAPHY

[1] T. Arbel and F. Ferrie. Viewpoint selection by navigation through entropy maps. In *Proc. of the 7th IEEE International Conf. on Computer Vision (ICCV-99)*, volume I, pages 248–254. IEEE, 1999.

[2] C. L. Bajaj, V. Pascucci, and D. R. Schikore. Fast isocontouring for improved interactivity. In *1996 Symposium for Volume Visualization*, pages 39–46. IEEE Computer Society Press, Los Alamitos, CA, Oct. 1996.

[3] Raymond E. Barber and Jr. Henry C. Lucas. System response time operator productivity, and job satisfaction. *Comm. of the ACM*, 26(11):972–986, 1983.

[4] P. Bhaniramka, R. Wenger, and R. Crawfis. Isosurfacing in higher dimensions. In *Proceedings of Visualization '00*, pages 267–273. IEEE Computer Society Press, Los Alamitos, CA, 2000.

[5] Richard E. Blahut. *Principles and practice of information theory.* Addison-Wesley Publ. Co., 1987.

[6] U.D. Bordoloi and H.-W. Shen. Hierarchical lic for vector field visualization. In *Proceedings of NSF/DoE Lake Tahoe Workshop on Hierarchical Approximation and Geometrical Methods for Scientific Visualization*, 2000.

[7] U.D. Bordoloi and H.-W. Shen. Hardware accelerated interactive vector field visualization: A level of detail approach. *Computer Graphics Forum*, 21(3):605–614, 2002.

[8] U.D. Bordoloi and H.-W. Shen. Space efficient fast isosurface extraction for large datasets. In *Proceedings of Visualization '03*, pages 201–208. IEEE Computer Society Press, 2003.

[9] U.D. Bordoloi and H.-W. Shen. Automatic view selection for volume rendering. Technical Report:OSU–CISRC–3/05–TR16, 2005.

[10] B. Cabral and C. Leedom. Imaging vector fields using line integral convolution. In *Proceedings of SIGGRAPH 93*, pages 263–270. ACM SIGGRAPH, 1993.

[11] B. Cabral and C. Leedom. Highly parallel vector visualization using line integral convolution. In *Proceedings of Seventh Siam Conference On Parallel Processing for Scientific Computing*, pages 803–807, 1995.

[12] B. Chen, A. Kaufman, and Q. Tang. Image-based rendering of surfaces from volume data. In *Proc. of IEEE Workshop on Volume Graphics*. IEEE, 2001.

[13] Y.-J. Chiang. Out-of-core isosurface extraction of time-varying fields over irregular grids. In *Proceedings of Visualization '03*, pages 217–224, 2003.

[14] Y.-J. Chiang and C. T. Silva. I/O optimal isosurface extraction. In *Proceedings of Visualization '97*, pages 293–300, 1997.

[15] Y.-J. Chiang, C. T. Silva, and W. J. Schroeder. Interactive out-of-core isosurface extraction. In *Proceedings of Visualization '98*, pages 293–300, 1998.

[16] P. Cignoni, P. Marino, E. Montani, E. Puppo, and R. Scopigno. Speeding up isosurface extraction using interval trees. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):158–170, 1997.

[17] M. Cox and D. Ellsworth. Application-controlled demand paging for out-of-core visualization. In *Proceedings of Visualization '97*, pages 235–244. IEEE Computer Society Press, Los Alamitos, CA, 1997.

[18] David E DeMarle, Steven Parker, Mark Hartner, Christiaan Gribble, and Charles Hansen. Distributed interactive ray tracing for large volume visualization. In *Proc. of IEEE Symposium on Parallel and Large-Data Visualization and Graphics '03*, pages 505–512, 2003.

[19] David Ebert and Penny Rheingans. Volume illustration: Non-photorealistic rendering of volume models. In *Proceedings IEEE Visualization 2000*. IEEE, 2000.

[20] M. Feixas, E. Acebo, P. Bekaert, and M. Sbert. An information theory framework for the analysis of scene complexity. *Computer Graphics Forum (Eurographics'99 Proc.)*, 18(3):95–106, 1999.

[21] S. Fleishman, D. Cohen-Or, and D. Lischinski. Automatic camera placement for image-based modeling. *Computer Graphics Forum*, 19(2):101–110, 2000.

[22] L.K. Forssell and S.D. Cohen. Using line integral convolution for flow visualization: Curvilinear grids, variable-speed animation, and unsteady flows. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):133–141, 1995.

[23] R. S. Gallagher. Span filter: An optimization scheme for volume visualization of large finite element models. In *Proceedings of Visualization '91*, pages 68–75, 1991.

[24] J. Gao, J. Huang, H.-W. Shen, and J.A. Kohl. Visibility culling using plenoptic opacity functions for large volume visualization. In *Proc. of IEEE Visualization '03*, pages 341–348, 2003.

[25] H. Garcke, T. Preußer, M. Rumpf, A. Telea, U. Weikard, and J. van Wijk. A continuous clustering method for vector fields. In *Proceedings of Visualization '00*, pages 351–358. IEEE Computer Society Press, 2000.

[26] M. Giles and R. Haimes. Advanced interactive visualization for CFD. *Computing Systems in Engineering*, 1(1):51–62, 1990.

[27] R. M. Gray and D. L. Neuhoff. Quantization. *IEEE Transactions on Information Theory*, 44(6):2325–2383, 1998.

[28] Stefan Gumhold. Maximum entropy light source placement. In *Proc. of IEEE Visualization '02*, pages 215–222, 2002.

[29] Stefan Guthe, Michael Wand, Julius Gonser, and Wolfgang Straßer. Interactive rendering of large volume data sets. In *Proc. of IEEE Visualization '02*, pages 53–60, 2002.

[30] Jan L. Guynes. Impact of system response time on state anxiety. *Comm. of the ACM*, 31(3):342–347, 1988.

[31] Markus Hadwiger, Christoph Berger, and Helwig Hauser. High-quality two-level volume rendering of segmented data sets on consumer graphics hardware. In *Proc. of IEEE Visualization '03*, 2003.

[32] B. Heckel, G. Weber, B Hamann, and K. Joy. Construction of vector field hierarchies. In *Proceedings of Visualization '99*, pages 19–25. IEEE Computer Society Press, 1999.

[33] Matthias Hopf and Thomas Ertl. Hierarchical splatting of scattered data. In *Proc. of IEEE Visualization '03*, pages 433–440, 2003.

[34] H. Hotelling. Analysis of a complex of statistical variables into principal components. *Journal of Educational Psycology*, 24:417–441,498–520, 1933.

[35] V. Interrante and C. Grosch. Strategies for effectively visualizing 3d flow with volume lic. In *Proceedings of Visualization '97*, pages 421–424. IEEE Computer Society Press, 1997.

[36] T. Itoh and K. Koyamada. Automatic isosurface propagation using an extrema graph and sorted boundary cell lists. *IEEE Transactions on Visualization and Computer Graphics*, 1(4):319–327, 1995.

[37] T. Itoh, Y. Yamaguchi, and K. Koyamada. Volume thinning for automatic isosurface propagation. In *Proceedings of Visualization '96*, pages 303–310, 1996.

[38] B. Jobard and W. Lefer. Creating evenly-spaced streamlines of arbitrary density. In *Proceedings of the eight Eurographics Workshop on visualization in scientific computing*, pages 57–66, 1997.

[39] B. Jobard and W. Lefer. The motion map: Efficient computation of steady flow animations. In *Proceedings of Visualization '97*, pages 323–328. IEEE Computer Society Press, 1997.

[40] B. Jobard and W. Lefer. Unsteady flow visualization by animating evenly-spaced streamlines. *Computer Graphics Forum (Proceedings of Eurographics 2000)*, 19(3), 2000.

[41] G. Karypis. Software package for clustering high-dimensional datasets. In *http://www-users.cs.umn.edu/ karypis/cluto/*, 2003.

[42] G. Kindlmann and J. W. Durkin. Semi-automatic generation of transfer functions for direct volume rendering. In *Proc. of IEEE Symposium on Volume Visualization '98*, pages 79–86, 1998.

[43] M.-H. Kiu and D. Banks. Multi-frequency noise for LIC. In *Proceedings of Visualization '96*, pages 121–126. IEEE Computer Society Press, 1996.

[44] J. J. Koenderink and A. J. van Doorn. The singularities of the visual mapping. *Biological Cybernetics*, 24:51–59, 1976.

[45] J. J. Koenderink and A. J. van Doorn. The internal representation of solid shape with respect to vision. *Biological Cybernetics*, 32:211–216, 1979.

[46] Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *Proc. of SIGGRAPH 1994*, pages 451–458. ACM, 1994.

[47] Tommer Leyvand, Olga Sorkine, and Daniel Cohen-Or. Ray space factorization for from-region visibility. In *Proc. of SIGGRAPH 2003*. ACM, 2003.

[48] G.-S. Li, U.D. Bordoloi, and H.-W. Shen. Chameleon: An interactive texture-based rendering framework for visualizing three-dimensional vector fields. In *Proceedings of Visualization '03*, pages 241–248. IEEE Computer Society Press, 2003.

[49] Jianhua Lin. Divergence measures based on the shannon entropy. *IEEE Trans. on Information Theory*, 37(1):145–151, January 1991.

[50] Y. Livnat, H.-W. Shen, and C. R. Johnson. A near optimal isosurface extraction algorithm using the span space. *IEEE Transactions on Visualization and Computer Graphics*, 2(1):201–227, March 1996.

[51] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics*, 21(4):163–169, July 1987.

[52] J. Marks, B. Andalman, P.A. Beardsley, W. Freeman, S. Gibson, J. Hodgins, T. Kang, B. Mirtich, H. Pfister, W. Ruml, K. Ryall, J. Seims, and S. Shieber. Design galleries: A general approach to setting parameters for computer graphics and animation. In *Proc. of SIGGRAPH 1997*. ACM, 1997.

[53] Nelson Max. Optical models for direct volume rendering. *IEEE Trans. on Visualization and Computer Graphics*, 1(2):99–108, June 1995.

[54] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[55] B. Mora, J.-P. Jessel, and R. Caubet. A new object-order ray-casting algorithm. In *Proceedings of Visualization '02*, Washington, DC, USA, 2002. IEEE Computer Society.

[56] K. Mueller, N. Shareef, J. Huang, and R. Crawfis. Ibr-assisted volume rendering. In *Late Breaking Hot Topics, IEEE Visualization Conf.* IEEE, 1999.

[57] A. Okada and D. L. Kao. Enhanced line integral convolution with flow feature detection. In *Proceedings of IS&T/SPIE Electronic Imaging '97*, pages 206–217, 1997.

[58] L. Pessoa, E. Thompson, and A. Noë. Finding out about filling-in: A guide to perceptual completion for visual science and the philosophy of perception. *Behavioral and Brain Sciences*, 21(6):723–748, 1998.

[59] GE Medical Systems Press Release. *http://www.gehealthcare.com/company/pressroom/releases/pr_release_6600.html*. GE, 2002.

[60] K. Sayood. *Introduction to Data Compression*. Morgan Kaufmann Publishers, Inc., 2000.

[61] H.-W. Shen. Isosurface extraction in time-varying fields using a temporal hierarchical index tree. In *Proceedings of Visualization '98*, pages 159–166, 1998.

[62] H.-W. Shen, C. D. Hansen, Y. Livnat, and C. R. Johnson. Isosurfacing in span space with utmost efficiency (ISSUE). In *Proceedings of Visualization '96*, pages 287–294, 1996.

[63] H.-W. Shen and C. R. Johnson. Sweeping simplices: A fast isosurface extraction algorithm for unstructured grids. In *Proceedings of Visualization '95*, pages 143–151, 1995.

[64] H.-W. Shen, C.R. Johnson, and K.-L. Ma. Visualizing vector fields using line integral convolution and dye advection. In *Proceedings of 1996 Symposium on Volume Visualization*, pages 63–70. IEEE Computer Society Press, 1996.

[65] H.-W. Shen and D.L Kao. A new line integral convolution algorithm for visualizing time-varying flow fields. *IEEE Transactions on Visualization and Computer Graphics*, 4(2), 1998.

[66] H.-W. Shen, G.-S. Li, and U.D. Bordoloi. Interactive visualization of three-dimensional vector fields with flexible appearance control. *IEEE Transactions of Visualization and Computer Graphics*, 10(4):434–445, 2004.

[67] Ben Shneiderman. Response time and display rate in human performance with computers. *ACM Computing Surveys*, 16(3):265–285, 1984.

[68] C. Silva, D. Bartz, P. Lindstrom, J. Klosowski, and W. Shroeder. High-performance visualization of large and complex scientific datasets. Tutorial M9, Super Computing 2002, Baltimore, 2002.

[69] D. Stalling and H.-C. Hege. Fast and resolution independent line integral convolution. In *Proceedings of SIGGRAPH '95*, pages 249–256. ACM SIGGRAPH, 1995.

[70] P. Sutton, C. Hansen, H.-W. Shen, and D. Schikore. A case study of isosurface extraction algorithm performance. In *Proceedings of Joint EUROGRAPHICS-IEEE TCCG Symposium on Visualization*, 2000.

[71] A. Telea and J. van Wijk. Simplified representation of vector fields. In *Proceedings of Visualization '99*, pages 35–42. IEEE Computer Society Press, 1999.

[72] Shivaraj Tenginakai, Jinho Lee, and Raghu Machiraju. Salient iso-surface detection with model-independent statistical signatures. In *IEEE Visualization 2001*, pages 231–238, 2001.

[73] X. Tricoche, G. Scheuermann, and H. Hagen. Continuous topology simplification of planar vector fields. In *Proceedings of Visualization '01*, pages 159–166. IEEE Computer Society Press, 2001.

[74] G. Turk and D. Banks. Image-guided streamline placement. In *Proceedings of SIGGRAPH '96*, pages 453–460. ACM SIGGRAPH, 1996.

[75] F.-Y. Tzeng, E.B. Lum, and K.-L. Ma. A novel interface for higher-dimensional classification of volume data. In *Proc. of IEEE Visualization '03*, pages 87–94, 2003.

[76] M. van Kreveld, R. van Oostrum, C. L. Bajaj, D. R. Schikore, and V. Pascucci. Contour trees and small seed sets for isosurface traversal. In *Proceedings of 13th ACM Symposium on Comp. Geom.*, pages 212–219, 1997.

[77] J. van Wijk. Spot noise: Texture synthesis for data visualization. *Computer Graphics*, 25(4):309–318, 1991.

[78] J. J. van Wijk. Image based flow visualization. In *Proceedings of SIGGRAPH 2002*, Computer Graphics Proceedings, Annual Conference Series, pages 745–754. ACM, ACM Press / ACM SIGGRAPH, 2002.

[79] P. P. Vázquez, M. Feixas, M. Sbert, and W. Heidrich. Viewpoint selection using viewpoint entropy. In *Proc. of Vision, Modelling, and Visualization '01*, pages 273–280, 2001.

[80] P. P. Vázquez, M. Feixas, M. Sbert, and W. Heidrich. Automatic view selection using viewpoint entropy and its application to image-based modeling. *Computer Graphics Forum*, 22(4):689–700, 2003.

[81] V. Verma, D. Kao, and A. Pang. Plic: Bridging the gap between streamlines and lic. In *Proceedings of Visualization '99*, pages 341–348. IEEE Computer Society Press, 1999.

[82] V. Verma, D. Kao, and A. Pang. A flow-guided streamline seeding strategy. In *Proceedings of Visualization '00*, pages 163–170. IEEE Computer Society Press, 2000.

[83] R. Wegenkittl and E. Gröller. Fast oriented line integral convolution for vector field visualization via the internet. In *Proceedings of Visualization '97*, pages 309–316, 1997.

[84] C. Weigle and D. Banks. Complex-valued contour meshing. In *Proceedings of Visualization '96*, pages 173–180. IEEE Computer Society Press, Los Alamitos, CA, 1996.

[85] D. Weinshall and M. Werman. On view likelihood and stability. *IEEE Transactions on Pattern Analysis and Machince Intelligence*, 19(2):97–108, 1997.

[86] D. Weinshall, M. Werman, and N. Tishby. Stability and likelihood of views of three dimensional objects. In *Proceedings of Third European Conference of Computer Vision*, 1994.

[87] J. Wilhelm and A. Van Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11(3):201–227, July 1992.

[88] J. Wilhelms and A. Van Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11(3):201–227, 1992.

[89] L. Wong, C. Dumont, and M. Abidi. Next best view system in a 3-d modeling task. In *Proc. of International Symposium on Computational Intelligence in Robotics and Automation (CIRA)*, pages 306–311, 1999.

[90] Eugene Zhang and Greg Turk. Visibility-guided simplification. In *Proc. of IEEE Visualization '02*, pages 215–222, 2002.

[91] Y. Zhao and G. Karypis. Criterion functions for document clustering: Experiments and analysis. Technical Report:TR 0140, Department of Computer Science, University of Minnesota, Minneapolis, MN, 2001.

[92] M. Zöckler, D. Stalling, and H.-C. Hege. Parallel line integral convolution. In *Proceedings of First Eurographics Workshop on Parallel Graphics and Visualisation*, pages 111–128, 1996.