Benjamin F. Gregorski
June 2004
Computer Science

Multiresolution Tetrahedral Meshes for Scientific Visualization

Abstract

The ever increasing size of scientific datasets presents a significant challenge for visualization because computers are no longer capable of displaying entire datasets at interactive rates or at rates that do not significantly slow down the visualization pipeline. Thus, there is a need for algorithms that can quickly reduce the amount of data that must visualized so that it can be drawn quickly and accurately. This dissertation presents the use of tetrahedral meshes created by edge bisection for multiresolution scientific visualization. Volume datasets, such as scalar fields or scalar fields with material interface information, defined on rectilinear grids, are embedded into a hierarchy of tetrahedral meshes. This hierarchy, which has fast, local and crack-free refinement and coarsening algorithms, is used to create a locally-adaptive approximation of the original volume based on user defined refinement criteria or error metrics. This new approximate volume representation is then used for interactive visualization and exploration via incremental view-dependent refinement and coarsening.

Multiresolution Tetrahedral Meshes for Scientific Visualization

By

BENJAMIN F. GREGORSKI
B.S. (University of California Davis) 1998

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

_____

_____

_____

Committee in charge

2004

Multiresolution Tetrahedral Meshes for Scientific Visualization

Contents

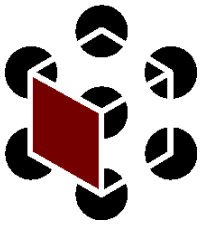
List of Figures

List of Tables

Acknowledgments

I would like to thank all of the professors, post-docs, students, system administrators and administrative staff in the Institute for Data Analysis and Visualization(IDAV) at UC Davis and especially my advisor Ken Joy for making this a wonderful place to spend the last five years. I would thank Mark Duchaineau, Valerio Pascucci and Peter Lindstrom from the Center for Applied Scientific Computing(CASC) at Lawrence Livermore National Laboratory(LLNL) as well as the people in the Institute for Scientific Computing Research(ISCR) and the Student Employee Graduate Research Fellowship(SEGRF) program at LLNL.

To friends and family and especially mommy, daddy, and brother John.



Multiresolution Tetrahedral Meshes for Scientific Visualization

Author: Benjamin F. Gregorski
Committee: Professor. Kenneth I. Joy, Chair
Professor. Bernd Hamann
Dr. Mark A. Duchaineau

VISUALIZATION AND GRAPHICS RESEARCH GROUP
INSTITUTE FOR DATA ANALYSIS AND VISUALIZATION
DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF CALIFORNIA, DAVIS

Chapter 1

Introduction

Science and engineering disciplines are facing the same problem: how to archive, transmit, visualize, and explore the massive datasets resulting from modern numerical computations and data collection devices. In the past, when only small amounts of data were processed, many researchers could accomplish some of these objectives at interactive rates on simple desktop machines. The *impact problems* of modern science and engineering require new algorithms for the analysis of the datasets produced by computational simulations and new sensor technology. The exploration of these datasets requires new techniques that scale with dataset size and computational resources because existing techniques for small datasets do not scale well, or not at all. New approaches are needed to address the interrelated problems of storage, visualization, and exploration of these massive datasets.

Scientific visualization is the process of generating meaningful images from collected scientific data such as numerical simulations, computerized tomography (CT) scans and sensor networks. The purpose of images and animations generated through the visualization process is to illuminate key features of the data in question and to improve the speed and efficiency with which people are able to understand and detect these features. A key to effective scientific visualization is

the interactivity at which the data can be visualized, the visualization parameters can be modified and the effects of those modifications can be shown to the user. Adaptive, multiresolution techniques that allow interactive visualization at varying levels-of-detail are a valuable technique for accomplishing these goals in the context of visualizing massive datasets. In this dissertation, I present new algorithms and data structures for interactive, multiresolution visualization of volume datasets. Adaptive tetrahedral meshes created by longest-edge bisection are used to build multiresolution representations of rectilinear volume datasets. The basic algorithms are integrated with modern state-of-the-art techniques for data compression, occlusion culling, hardware accelerated rendering and out-of-core visualization. At runtime this structure is adaptively refined and coarsened to interactively visualize datasets at varying levels-of-detail.

The algorithms developed here are based on the principles of selective, local refinement and coarsening. Figure 1.1 illustrates this principle for a two-dimensional triangle mesh. This type of refinement is used extensively in view-dependent terrain rendering and adaptive surface rendering applications where detail is added to specific local regions based on an error measurement. Starting with a square divided into four triangles, vertices are added at edge midpoints to refine the mesh. The middle picture shows the addition of three vertices with the numbers indicating the order of addition. In order to maintain a crack-free mesh, vertices 1 and 2 can be added in any order, but vertex 3 can be added only after vertices 1 and 2 have been added. The right picture shows the insertion of three more vertices to adaptively refine the mesh. As before, the vertices must be added in a specific order to maintain a crack-free mesh at all stages of refinement.

The inverse of the selective, local refinement process, selective, local coarsening is illustrated in Figure 1.2. Starting with the rightmost configuration from Figure 1.1, the triangle mesh is locally coarsened by removing vertices and triangles added during local refinement. The green triangles and shaded points indicate the vertex and triangles that are removed in each coarsening step.

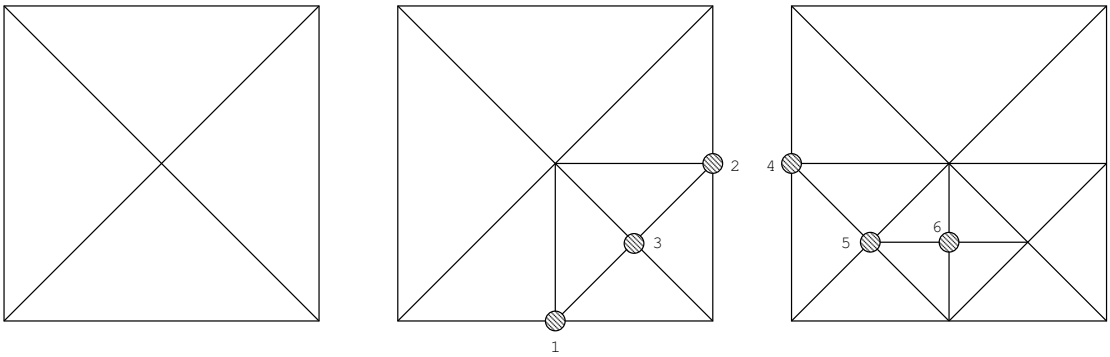


Figure 1.1: Left to right: local refinement of two-dimensional triangle mesh. Dashed circles indicate new vertices.

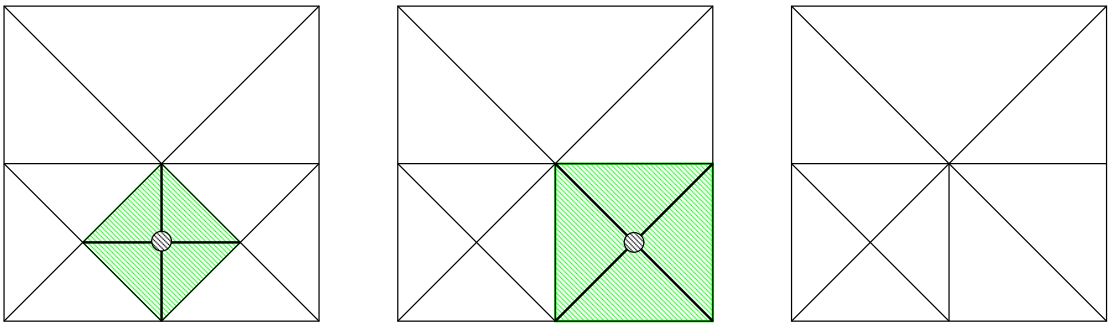


Figure 1.2: Left to right: local coarsening of two-dimensional triangle mesh. Green triangles and dashed vertices indicate those removed by mesh coarsening.

Both of these principles are illustrated in Figure 1.3 which shows how the mesh can be adapted around an arbitrary, moving position. The mesh is initially refined around the red circle. When the circle moves as indicated in the left image, the mesh is refined around the new position and coarsened away from it. The thick, dashed blue lines indicate when refinement occurs, and the thick green lines indicate where the mesh is coarsened.

These algorithms for selective, local refinement and coarsening can be extended to three-dimensional tetrahedral meshes which is described in Chapter 2. For visualization, this adaptive, three-dimensional structure is used to create a hierarchy of approximations to a volume dataset. The mesh structure is then refined and coarsened according to user specified criteria and the resulting tetrahedral mesh is visualized.

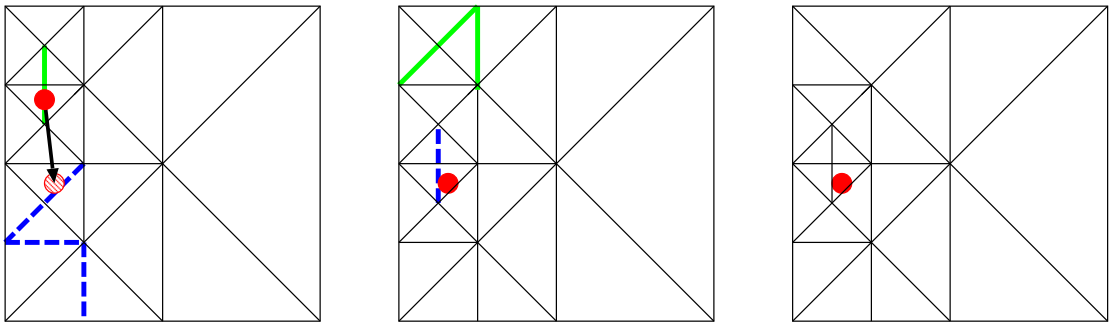


Figure 1.3: Left to right: example of selective, local refinement and coarsening. The mesh is adapted around the large, red circle; as it moves the mesh refines and coarsens accordingly.

In Figure 1.4, this technique is applied to isosurface visualization to allow the extraction of view-dependent isosurfaces (Chapter 4). The dataset is a $512 \times 512 \times 266$ CT scan of a monkey lung padded to brings its size to 512^3.

The isosurface for the bones has been extracted showing the vertebrae, ribs, and shoulder blades. Isosurfaces of decreasing resolution are shown from left to right. Since the refinement is view-dependent, the isosurface is extracted at a higher resolution closer to the viewpoint and at a lower resolution away from it. This is most noticeable in the lower resolution surfaces.

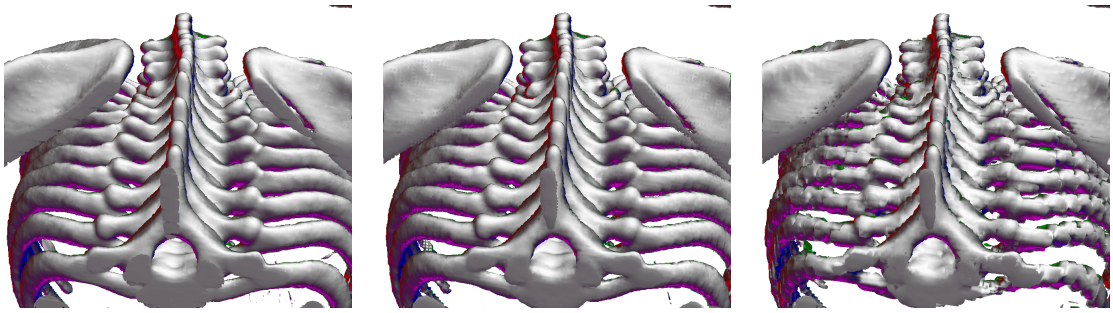


Figure 1.4: Left to right: View-dependent refinement is used to extract high, medium, and low resolution isosurfaces. The surfaces contain 2.21M, 812K, 242K triangles respectively.

In order to interactively visualize large isosurfaces with several million triangles, this adaptive mesh refinement algorithm must be integrated with several important techniques. These techniques allow multiresolution visualization algorithms based upon this mesh structure to effec-

tively utilize memory hierarchy coherence and graphics hardware improvements to speed up visualization.

A novel data layout scheme based on space filling curves (Section 4.5) is used to allow effective out-of-core visualization. The layout scheme reorganizes the data to more closely follow the order of the points needed by the mesh refinement. In Section 5.5 a spatial-temporal interlacing algorithm is used to adapt this data layout algorithm to time-varying data. High resolution isosurfaces from time-varying datasets, consisting of over a million triangles per timestep, can be visualized at several frames-per-second.

Dividing tetrahedra into chunks called subtrees (Section 4.9) allows efficient caching of extracted geometry for fast rendering using graphics hardware. The high-resolution isosurface of monkey bones shown in Figure 1.4, which contains 2.21M triangles, can be visualized at 5 frames-per-second. Data compression techniques (Section 4.11) are used to reduce the disk storage of large volumes and still allow for interactive visualization. Memory and processor speeds have been increasing faster that disk speeds; compression allows work to be transfered from the slower disk i/o system to the faster memory and processor systems to take advantage of the increasing performance gap. Data compression techniques using linear prediction and Huffman codes provide 2:1 to 5:1 compression of the original volume data. For the monkey dataset, the data (128 MB) is compressed at a ratio of 3.67:1, and the data plus quantized gradients (512MB) is compressed at a ratio of 2.44:1. Highly tuned decompression and reconstruction algorithms allow these volumes to be decompressed on-the-fly and visualized with the same interactivity as the uncompressed volumes.

The basic principle of selective, local refinement can be applied to a large range of visualization problems where visualization speed and quality need to be balanced. In Chapter 3 adaptive tetrahedral meshes are used to construct multiresolution representations of computational simulation datasets that contain material interface discontinuities. A new tetrahedral field representation is

introduced which explicitly represents the boundaries of discontinuities within a cell using a signed distance function. This allows for multiple, per-material interpolation functions to exist over a single cell making it possible to represent discontinuous functions without the need for large amounts of refinement around them. The contouring application shown in Figure 1.4 for static volumes is extended to time-varying datasets in Chapter 5. As with static volumes, adaptive tetrahedral meshes are used to construct an approximation which is changed over time according to changes in the isosurface. A new data layout scheme, that interleaves the time-varying data in the spatial and temporal dimensions, allows efficient streaming from disk so that large, compressed volumes ($256^3 \times 274$ timesteps) can be contoured at several frames-per-second.

Chapter 2

Longest-Edge Bisection

This chapter describes the refinement of tetrahedral meshes based on edge bisection used as the basis for several multiresolution visualization algorithms presented in this dissertation. These meshes have been used for many scientific visualization applications which are discussed in Section 4.2. These meshes and other types of meshes generated through local refinement have been used extensively in numerical integration, approximation and simulation applications for generating the adaptive grids needed for locally refined numerical solutions. Different mesh refinement schemes are described and analyzed in Liu and Joe [48],[49], Plaza et al. [59], Bey [2], Ruprect et al. [62] and Arnold et al. [1]. The refinement of tetrahedral meshes by edge bisection used in this dissertation is described by Maubach in [54] and [55] where edge bisection refinement is generalized for n-simplices. The following sections present the refinement algorithm for two and three dimensional simplices (i.e. triangles and tetrahedra) and describes a new formulation of the refinement strategy that eliminates non-conformant edges. In both cases, the initial configuration is a two-dimensional three-dimensional cube composed of simplicial elements.

2.0.1 Two-dimensional Refinement

To establish terminology and illustrate valuable concepts, we first describe the refinement in two dimensions for triangular meshes, see Figure 2.1. The mesh refinement starts with a square divided into two triangles along its major diagonal. The mesh is refined by inserting a new vertex called the *split vertex* at the midpoint of an element's (triangle or tetrahedraon) longest edge. This edge is called the *split edge* or *refinement edge*. In each phase, an element is subdivided into two *children*. To maintain a conformant mesh during refinement elements, are grouped into larger structures called *diamonds*. Diamonds are groups of elements that share the same split edge. All elements in a diamond are refined together. The bottom two images in Figure 2.1 show the two phases of diamonds that occur in two-dimensional refinement.

The meshes created by longest-edge bisection can be adaptively refined (adding elements) and coarsened (removing elements) using the *split* and *merge* operations. These operations are illustrated for two-dimensional Phase 0 and Phase 1 diamonds in Figures 2.2 and 2.3. Given a diamond D, the parents of D are the diamonds that must be split to create all of D's elements. $D's$ children are those diamonds that $D's$ child elements belong to (i.e., those elements created when D is split). Since the grids used have finite extent, boundary diamonds have fewer parents and children than interior diamonds.

In order for a diamond D to be split, all of its elements must be created. This is done by ensuring that all of D's parents are split before D is split. Splitting parent diamonds is used to maintain a conformant mesh (i.e., a mesh without cracks or t-intersections). This recursive splitting of parent diamonds is illustrated for two-dimensional refinement in Figure 2.4. Similarly, the mesh is coarsened by merging diamonds and removing tetrahedra from the mesh. In order for a diamond D to be merged, all of its elements must be split, but none of its child elements can be split. That

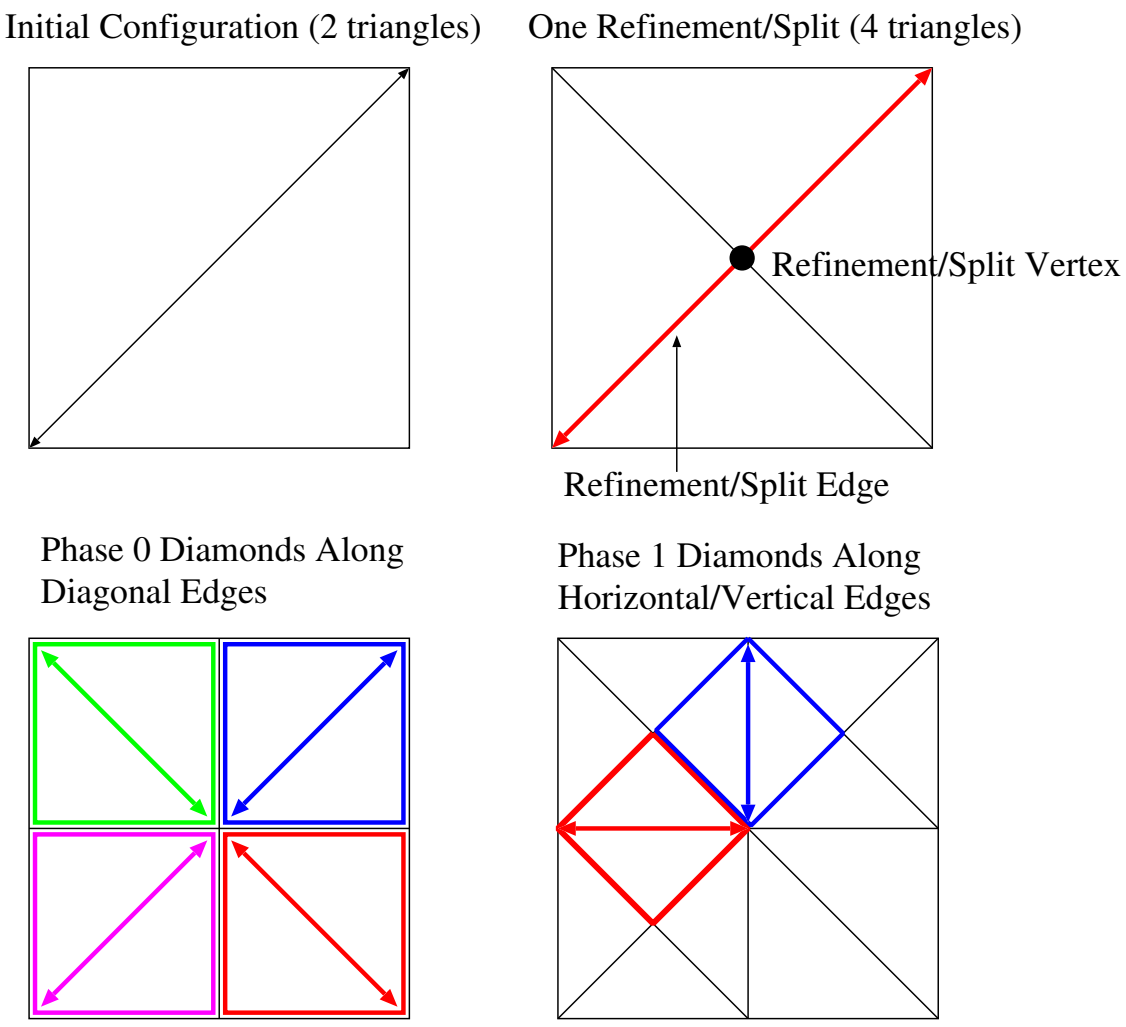


Figure 2.1: Mesh refinement in two dimensions showing refinement terminology. Left to right, top to bottom: Initial configuration, mesh after one refinement, two types of diamonds centered around face diagonals and edges.

is *D* has child elements but no grandchild elements. Elements here refers to triangles or tetrahedra and not diamonds because it is possible for a diamond's grandchild diamonds to exist even though none of its children have been split. Merging is illustrated in Figure 2.5.

2.0.2 Three-dimensional Refinement

In three dimensions, we refine a tetrahedral mesh. A tetrahedron is described by a *level* and a *phase*, with 3 phases at each level. The bisection begins with an initial configuration of a cube divided into

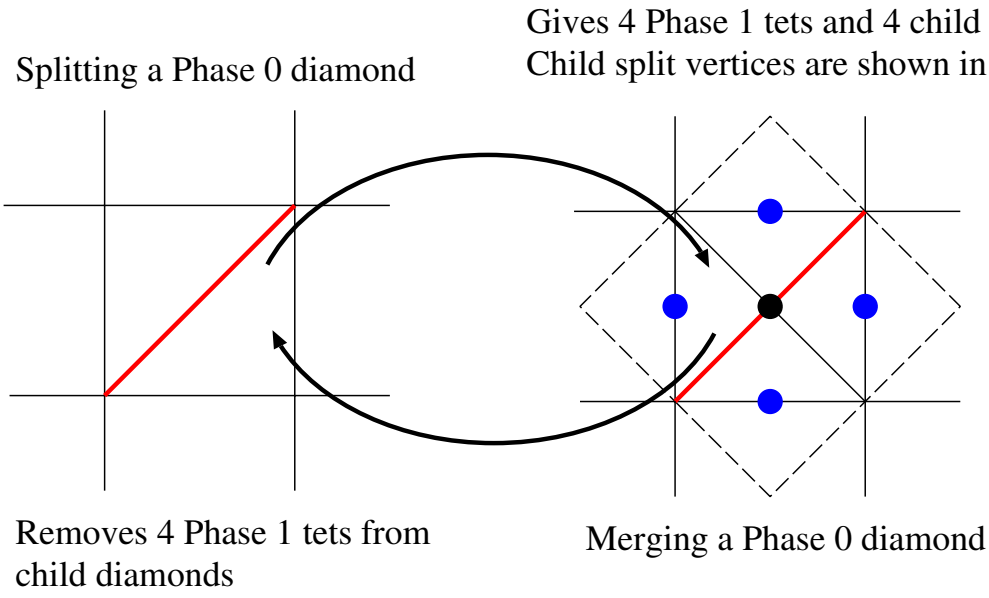


Figure 2.2: Splitting and merging operations for Phase 0 diamonds.

6 tetrahedra around a major diagonal. as shown in Figure 2.6.

Figures 2.10-2.12 illustrate the three phases of the refinement process. After three refinements, the level is incremented by 1. After n refinements, the phase is n mod 3 and the level is $\lfloor n/3 \rfloor$. As stated earlier for two dimensional refinement, the *split edge* of a tetrahedron is its longest edge. In each phase, a tetrahedron is subdivided into two child tetrahedra by inserting the split vertex at the midpoint of the split edge. Figure 2.7 shows a cube divided into six tetrahedra embedded in a larger cube to illustrate that the cube's points lie on the vertices, faces, and edges of a surrounding cube. This also illustrates that the three phases of 1:2 edge bisection refinement introduce the same points as one phase of 1:8 octree refinement.

Three-dimensional Diamonds

Tetrahedra are grouped into diamonds to simplify the refinement process and to ensure continuity of isosurfaces or other visualizations generated from the mesh. When a tetrahedron is split, all the

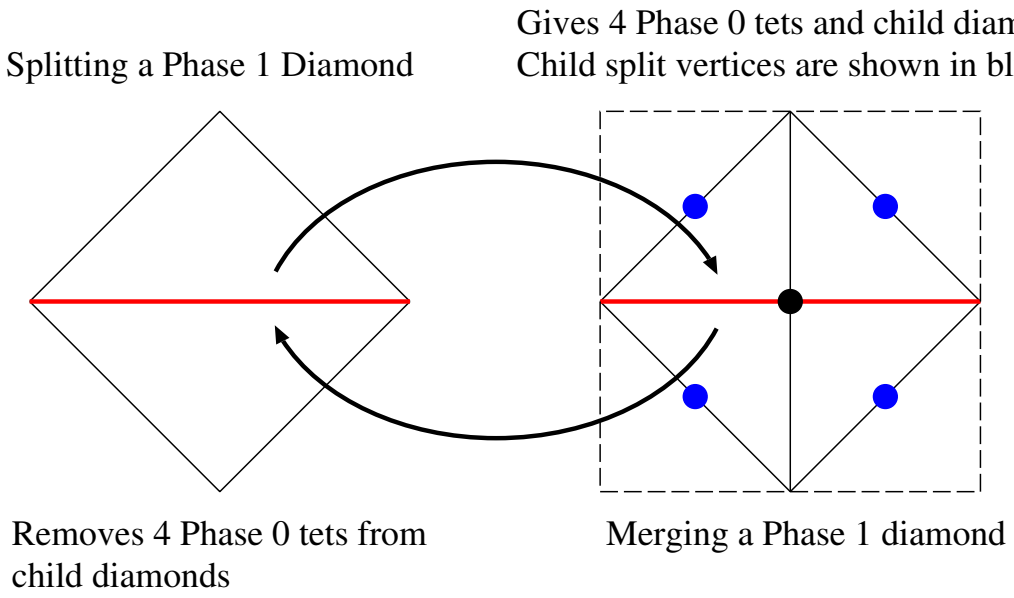


Figure 2.3: Splitting and merging operations for Phase 1 diamonds.

tetrahedra that share its split edge must also be split. A group of tetrahedra that share a split edge is called a *diamond*. The split edge and split vertex of a diamond are defined as the common split edge and split vertex of its tetrahedra. All diamonds in the mesh can be identified by their split edge or split vertex. Phase 2, phase 1, and phase 0 diamonds are shown in Figures 2.8 and 2.9. Given a three-dimensional dataset, each point in the dataset, except for the corner points of the original cube, is uniquely associated with the split vertex of one diamond. This is because each point in the dataset is introduced by the splitting of a diamond. By grouping tetrahedra into diamonds, we can easily locate all of the tetrahedra around a split edge. Any diamond in the hierarchy is uniquely associated with a split edge, a split vertex and the group of tetrahedra around its split edge. In later discussions, these are used interchangeably to identify diamonds. Splitting a diamond is equivalent to splitting all of the tetrahedra in the diamond. All tetrahedra within a diamond have the same level and phase. Table 2.1 lists the number of tetrahedra, their phase, and level for each diamond.

The *type* of a diamond is determined by its split edge (SV_0, SV_1), where SV_0 and SV_1 are

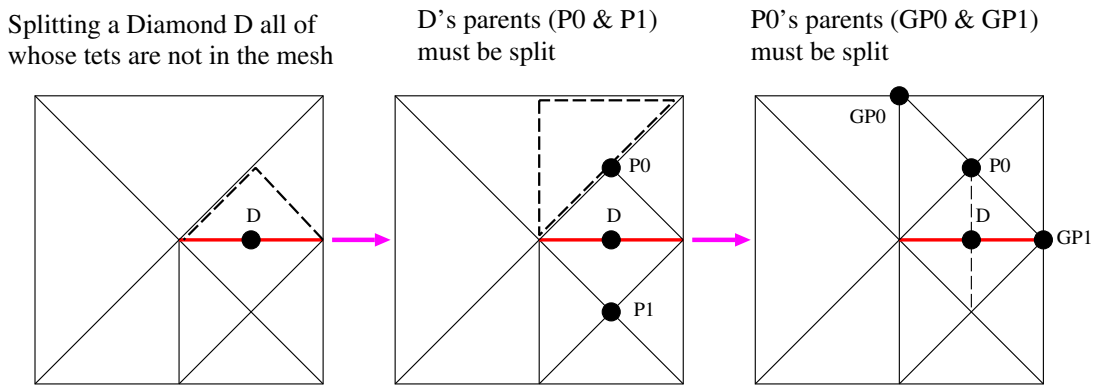


Figure 2.4: Left to right: Splitting a diamond first requires that all of the its tetrahedra are in the mesh. This can require several recursive splits on the diamond's parents.

the vertices on the split edge. Starting from the initial configuration of six tetrahedra in a cube, there are 26 different direction vectors (i.e., diamond types) for the split edge; there are 8 directions for the phase 0 diamonds, 12 for the phase 1 diamonds (4 each on the XY, XZ, and YZ planes), and 6 for the phase 2 diamonds. The type of a diamond is used to efficiently encode the structure of the mesh including the location of parent and child diamonds. A diamond's type is determine by the direction of its split edge which is encoded as a vector (i.e., $(1,1,1)$, $(1,0,-1)$ etc.). The locations of a diamond's parents and children are also described as vectors relative to the diamond's split vertex. These vectors are rescaled based upon the diamond's level to get their actual index. This is discussed in more detail in Section 4.7. Figure 2.8 shows the parents for each diamond. The diamonds that are created when *D* is split are called *D*'s *children*. It should be noted that when phase 1 diamonds lie on dataset boundaries they have one parent, and when phase 2 diamonds lie on a boundary they have only 2 or 3 parents. Similarly, when phasae Figure 2.9 shows the children for each diamond. In these figures, a diamond is indicated by its split vertex. The parent and child information is summarized in Table 2.1.

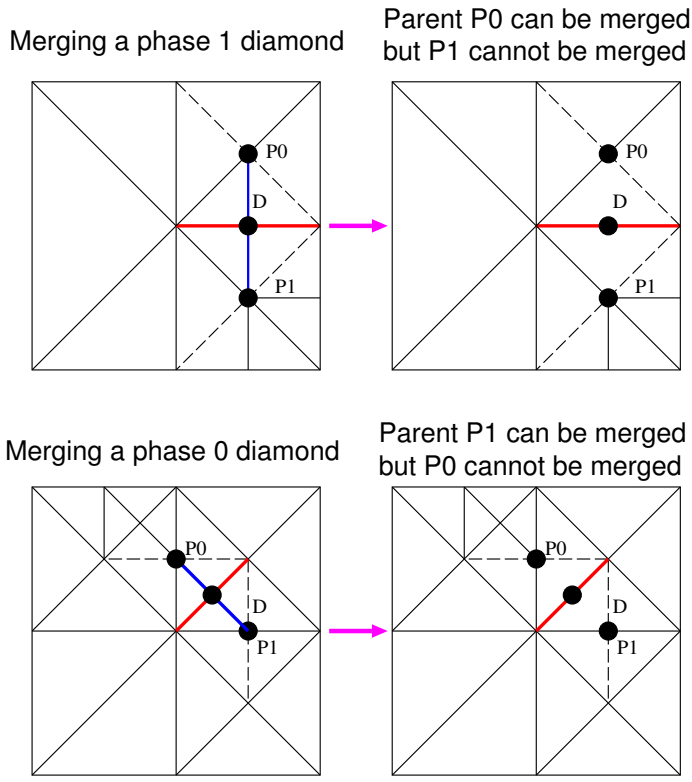


Figure 2.5: Top to bottom: Merging of Phase 1 and Phase 0 diamonds removes child tetrahedra from the mesh. After a diamond has been merged, parents are candidates for merger only if their children are not split (i.e. they have no tetrahedra grandchildren).

Refinement and Coarsening

The tetrahedral mesh hierarchy is refined and coarsened by split and merge operations respectively. The split and merge operations operate on diamonds, and the three refinement phases successively split three, two, and one simplices. Figures 2.10–2.12 show the split and merge operations for the three types of diamonds.

Phase	Tets(Phase,Level)	Parents(P,L)	Children(P,L)
0	6(0,L)	3(2,L-1)	6(1,L)
1	4(1,L)	2(0,L)	4(2,L)
2	8(2,L)	4(1,L)	8(0,L+1)

Table 2.1: Number, phase, and level of tetrahedra, parents, and children for the three different diamonds. L is the *level* of the diamond.

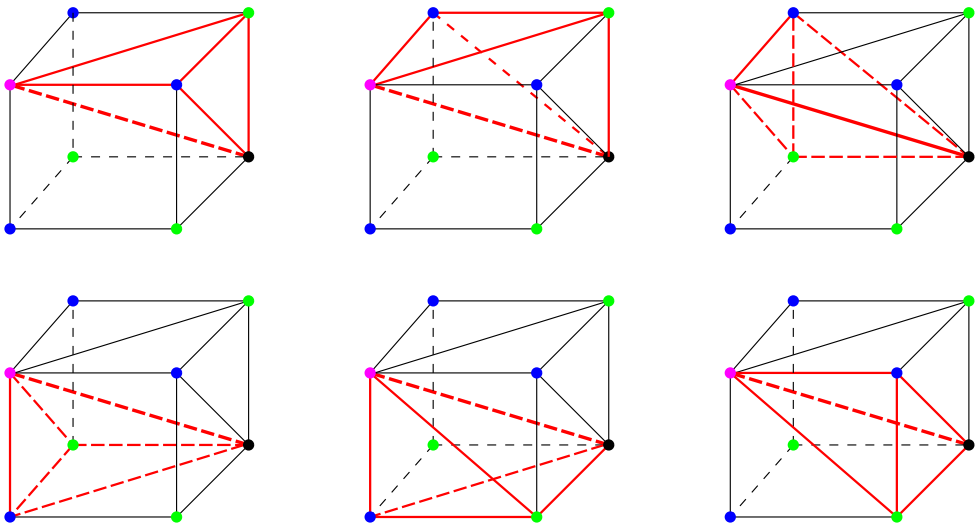


Figure 2.6: Tetrahedra in the initial configuration.

The refinement of the tetrahedral mesh using diamonds gives two important ways of thinking about the mesh structure. Using the structure indicated by the tetrahedra, the mesh hierarchy is described by a binary tree. In this tree, children are nested within the boundary of their parents and the mesh can be traversed using standard binary tree traversal algorithms such as in, pre, and post order enumeration. Following the structure of the diamonds represents the hierarchy as a directed acyclic graph. Parents have 6, 4 or 8 children, and children have 3, 2 or 4 parents. Children are not nested within the boundary of their parents. Given a diamond D, following children-parent paths enumerates which diamonds must be split before D can be split. The two relationships are illustrated in Figure 2.13.

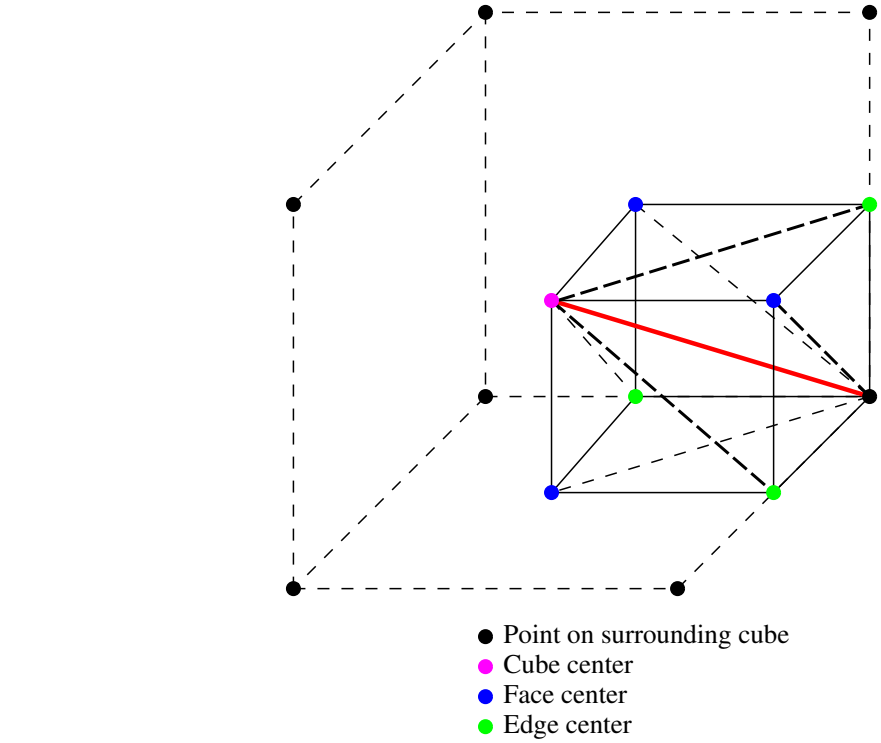


Figure 2.7: Embedding of an initial cube (Phase 2 diamond) in a larger cube.

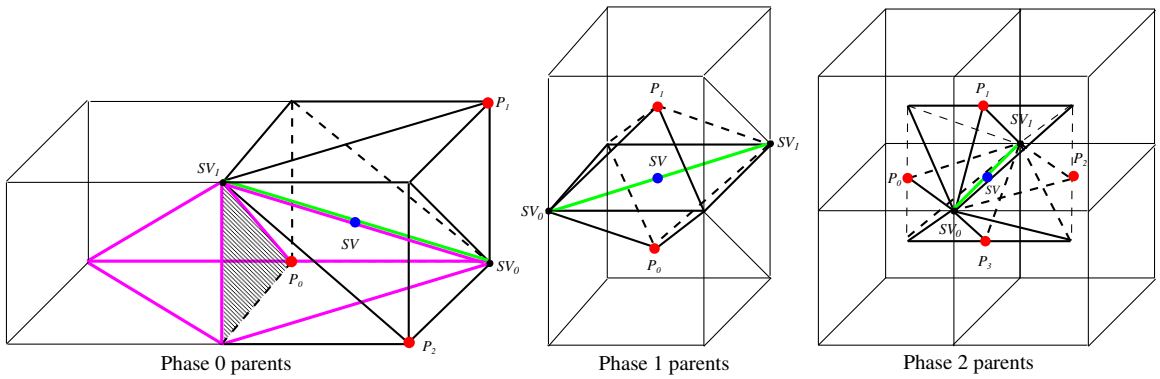


Figure 2.8: Parent diamonds: Phase 0 parents are phase 2 diamonds from the level $L-1$, phase 1 parents are located at cube centers, and phase 2 parents are located at face centers. The split edge is (SV_0, SV_1) (shown in green), the split vertex is SV (blue), and the parents are shown as P_0, P_1, P_2, and P_3 (red). The magenta tetrahedron is a tetrahedron in diamond P_0. The shaded triangle shows how it is split into two phase 0 tetrahedra.

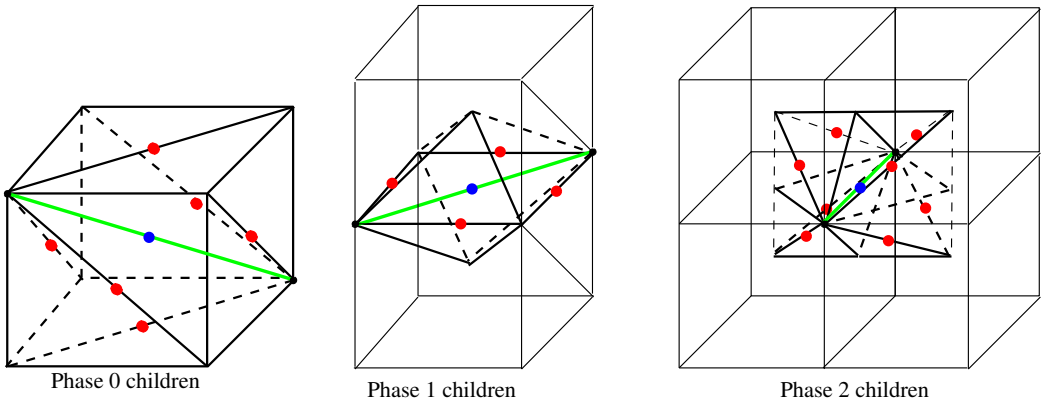


Figure 2.9: Child diamonds: Phase 0 children are located on the faces of a cube, phase 1 children are located on the centers of the edges of the face containing the split edge, and phase 2 children are the phase 0 diamonds from level $L+1$ that touch the diamond's split edge.

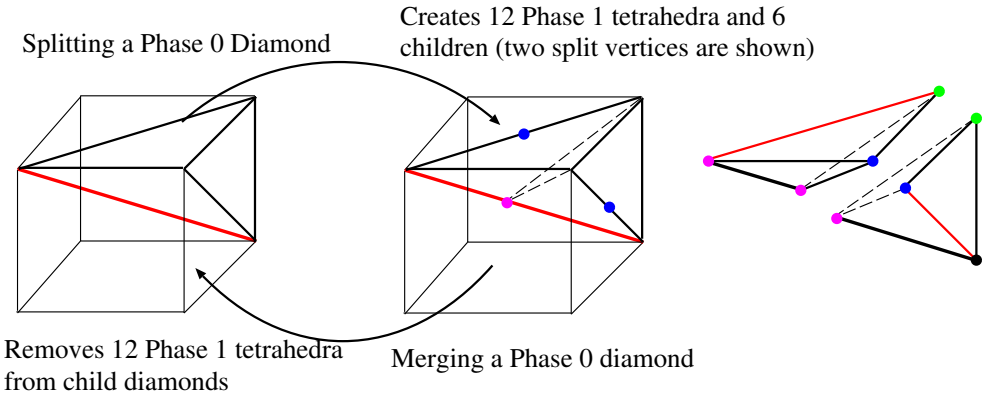


Figure 2.10: Splitting of a Phase 0 diamond inserts a new point at the midpoint of the cube diagonal.

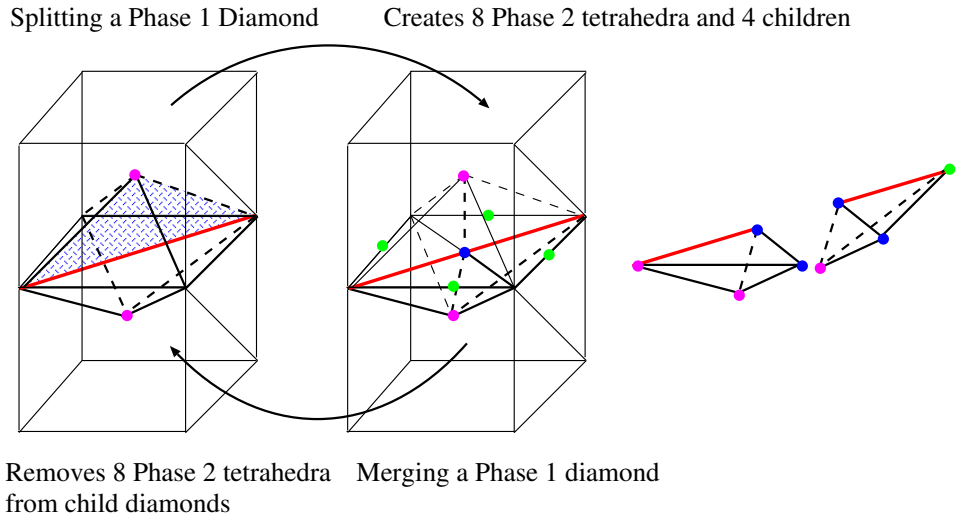


Figure 2.11: Splitting of a Phase 1 diamond inserts a new point at the midpoint of a face diagonal.

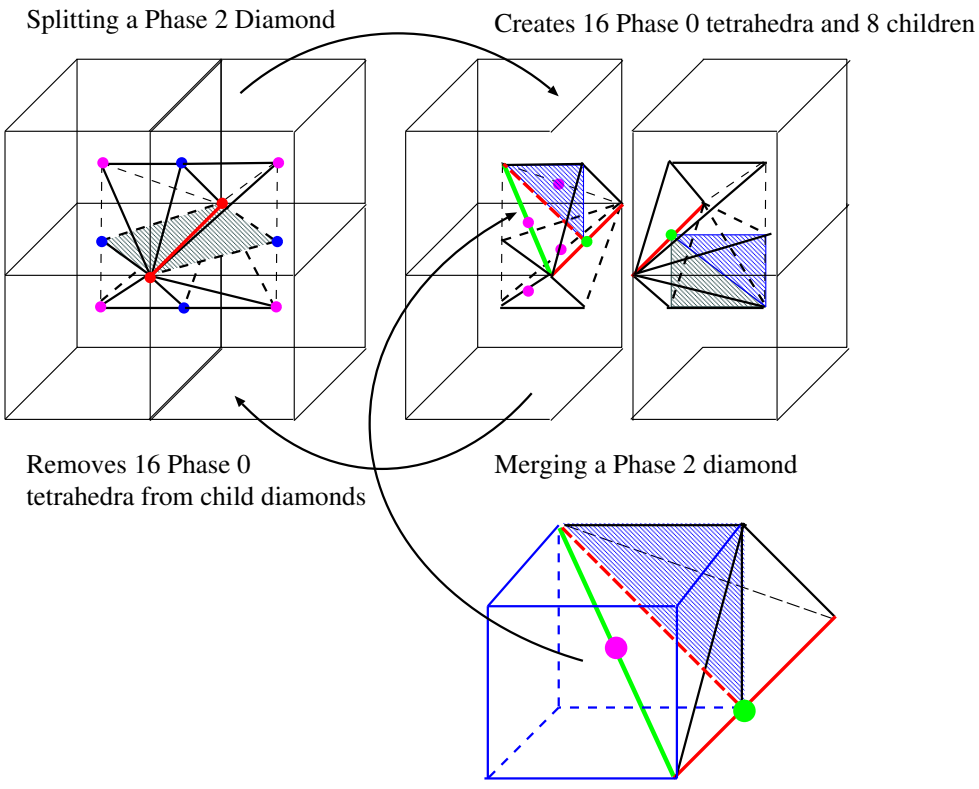


Figure 2.12: Splitting of a Phase 2 diamond inserts a new point at the midpoint of a cube edge.

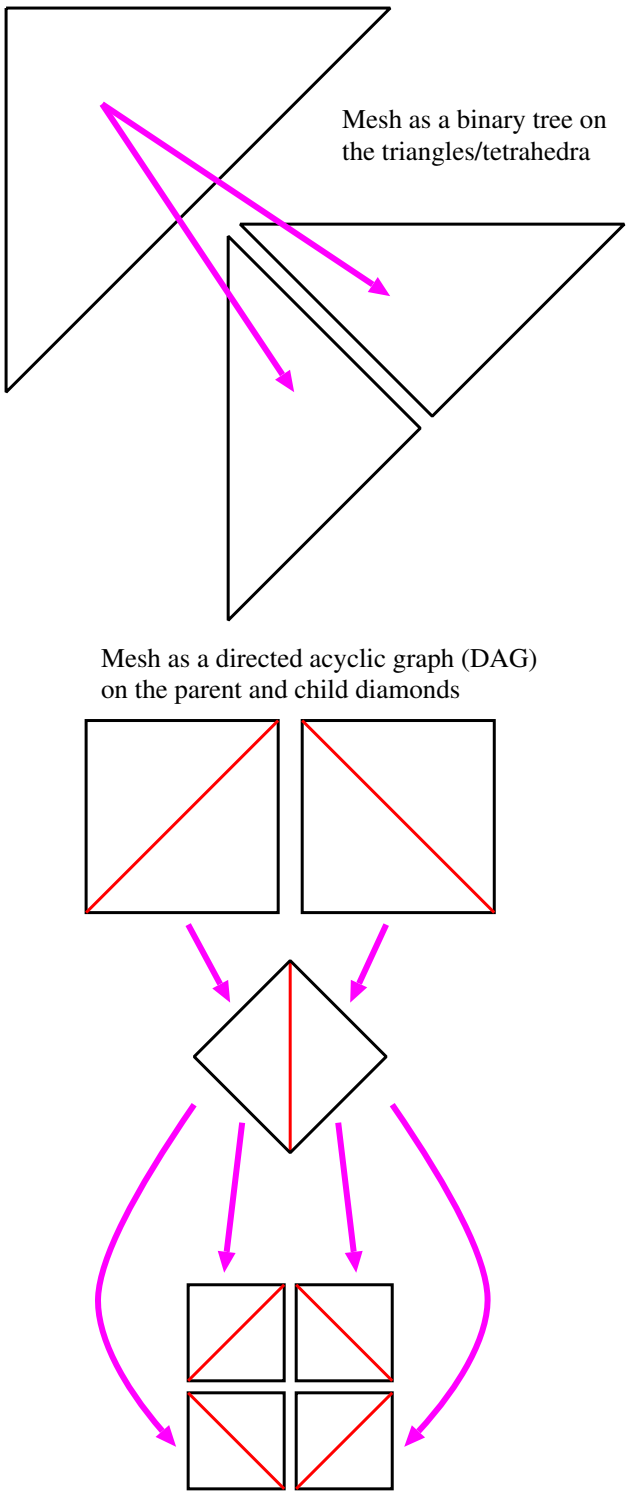


Figure 2.13: Top: Binary tree representation of tetrahedral mesh hierarchy. Bottom: Directed acyclic graph (DAG) representation of mesh hierarchy.

Chapter 3

Material Interfaces

3.1 Introduction

Computational physics simulations are generating larger and larger amounts of data. They operate on a wide variety of input meshes, for example rectilinear meshes, adaptively refined meshes for Eulerian hydrodynamics, unstructured meshes for Lagrangian hydrodynamics and arbitrary Lagrange-Eulerian meshes. Often, these datasets contain special physical features such as material interfaces, physical boundaries, or thin slices of material that must be preserved when the field is simplified. In order to ensure that these features are preserved, the simplified version of the dataset needs to be constructed using strict L^∞ error bounds that prevent small yet important features from being eliminated.

Data sets of this type require a simplification algorithm to approximate datasets with respect to several simplification criteria. The cells in the approximation must satisfy error bounds with respect to the dependent field variables over each mesh cell, and to the representation of the discontinuities within each cell. In addition, the simplification algorithm must be able to deal with a wide range of possible input meshes.

The algorithm presented here generates an approximation of a computational dataset that contains material interfaces. The new approximation can be used in place of the original high-resolution dataset generated by the simulation. The approximation is a resampling of the original dataset that preserves user-specified as well as characteristic features in the dataset and approximates the dependent field values to within a specified tolerance. Material interfaces embedded in the meshes of computational datasets are often a source of error for simplification algorithms because they represent discontinuities in the scalar or vector field over mesh elements. By representing material interfaces explicitly, this algorithm is able to provide separate field representations for each material over a single cell. Multiresolution representations utilizing separate field representations can accurately approximate datasets that contain discontinuities without placing a large percentage of cells around the discontinuous regions. This algorithm uses a multiresolution tetrahedral mesh supporting fast coarsening and refinement capabilities; error bounds for feature preservation; explicit representation of discontinuities within cells; and separate field representations for each material within a cell.

3.2 Related Work

The approximation of material interfaces is similar to the approximation or simplification of large polygonal meshes. The approximating mesh represents the original mesh to within a certain error tolerance using a substantially smaller number of triangles. Mesh approximation and simplification algorithms can be divided into decimation techniques and remeshing techniques.

Decimation based techniques attempt to simplify the existing geometry by removing vertices, edges or faces and evaluating an error function that determines the fidelity of the new mesh. Decimation based techniques start with the existing high resolution geometry and remove features until a specified error tolerance has been reached. Techniques for simplifying triangle meshes, scat-

tered data, and tetrahedral meshes have matured substantially. In [27] an iterative triangle mesh decimation method is introduced. Triangles in nearly linear regions are identified and collapsed to a point that is computed in a locally optimal fashion. Hoppe [31] describes a progressive mesh simplification method for triangle meshes. An arbitrary mesh is simplified through a series of edge collapse operations to yield a very simple base mesh. Heckbert and Garland present a comprehensive survey of these techniques in [28]. More recently, Heckbert and Garland [14] use a quadric error metric for surface simplification. They use vertex pair collapse operations to simplify triangle meshes and *quadric matrices* that define a quadric object at each vertex to control the error of the simplified surfaces. In [13], they use the same technique to simplify meshes that have associated colors and textures. Hoppe [33] uses a modified quadric metric for simplifying polygonal meshes while preserving surface attributes such as normal vectors, creases, and colors. Heckel and Uva [29] describe methods for constructing surface hierarchies based on adaptive clustering. Multiresolution methods for reconstruction and simplification have also been explored using subdivision techniques and wavelets.

Another approach to mesh approximation is to generate a completely new mesh through a remeshing or fitting process. The remeshing process starts with a coarse mesh and adds in geometry until the surface approximates the original model to within a specified error tolerance. Lee et al. [39] use a mesh parameterization method for surface remeshing. A simplified base mesh is constructed and a subdivision surface is fitted to the original model. Kobbelt et al. [37] use a shrink wrapping approach to remesh polygonal meshes with subdivison surfaces.

Our approximation of material interfaces falls into the category of remeshing techniques. As described in Section 3.4, the material interfaces are given as triangle meshes. Within each of our cells we construct a piecewise linear approximation of the material interfaces to within a specified error tolerance based on the distance between the original mesh and our approximation.

Zhou et al. [78] present the multiresolution tetrahedral framework that is the basis of our simplification algorithm. Cignoni at al. [5] describe a multiresolution tetrahedral mesh simplification technique built on scattered vertices obtained from the initial dataset. Their algorithm supports interactive level-of-detail selection for rendering purposes. Trotts et al. [69] simplify tetrahedral meshes through edge-collapse operations. They start with an initial high-resolution mesh that defines a linear spline function and simplify it until a specified tolerance is reached. Staadt and Gross [65] describe progressive tetrahedralizations as an extension of Hoppe's work. Their method simplifies a tetrahedral mesh through a sequence of edge collapse operations. They also describe error measurements and cost functions for preserving consistency with respect to volume and gradient calculations and techniques for ensuring that the simplification process does not introduce artifacts such as intersecting tetrahedra. Different error metrics for measuring the accuracy of simplified tetrahedral meshes have been proposed. Lein et al. [36] present a simplification algorithm for triangle meshes using the Hausdorff distance as an error measurement. They develop a symmetric adaption of the Hausdorff distance that is an intuitive measure for surface accuracy.

3.3 Multiresolution Framework

The simplification algorithm presented here is based on the refinement of a tetrahedral mesh as described in Chapter 2. In this application a generalized pointer-based implementation, that does not require the input data to be given on a regular rectilinear mesh consisting of (2^N+1)x(2^N+1)x(2^N+1) cells, is used. A variety of input meshes can be supported by interpolating field values to the vertices of the multiresolution tetrahedral mesh. In general, any interpolation procedure may be used. In some cases, the procedure may be deduced from the physics models underlying the simulation that produced the dataset. In other cases, a general-purpose interpolation algorithm will be appropriate. The approximation algorithm presented here uses the simplest representation for

a field within a tetrahedral cell as given by the unique linear function that interpolates field values specified at the cell's vertices. In the case of a conformant mesh, this natural field representation will be continuous across cell boundaries, resulting in a globally C^0-continuous representation. The multiresolution approximation is constructed in a top-down fashion. Data from the input dataset, including grid points and interface polygons, are assigned to child tetrahedra when their parent is split.

A multiresolution framework for approximating numerically simulated data needs to be a robust and extensible. The framework must be capable of supporting the wide range of possible input structures used in the simulations and the wide range of output data generated by these simulations. The following properties and operations are desirable for such a framework:

1. **Interactive transition between levels of detail**. The ability to quickly move between different levels of detail allows a user to select a desired approximation at which to perform a calculation (for a visualization application).

2. **Strict L^∞ error bounds**. Strict error bounds prevent small yet important features from being averaged or smoothed out by the approximation process.

3. **Local and adaptive mesh refinement and local error computations**. Local mesh refinement allows the representation to be refined only in the areas of interest. while keeping areas of little interest at lower resolutions. This is essential for maintaining interactivity and strict cell count on computers with limited resources. The error calculations for datasets consisting of millions or billions of cells should not involve a large amount of original data.

4. **Accommodating different mesh types**. Computational simulations are done on a large variety of mesh structures and it is cumbersome to write a multiresolution algorithm for each specific structure. In order for a framework to be useful it should be easily adaptable to a

broad class of input meshes.

5. **Explicit representation of field and/or material discontinuities**. Discontinuities are very important in scientific datasets and very often need to be preserved when the datasets are approximated. A multiresolution framework should support the explicit representation and approximation of these discontinuities.

The multiresolution framework described here satisfies these design criteria. Tetrahedral cells are the simplest of the polyhedral cells, and they allow us to use linear basis functions to approximate the material interfaces and the dependent field variables in a cell. A representation of the original data can be computed in a pre-processing step, and the algorithms developed in [9, 23] can be used to efficiently select a representation that satisfies an error bound or a desired cell count. This makes the framework ideal for interactive display.

The framework supports various input meshes by resampling them at the vertices of the tetrahedral mesh. The resampling error of the tetrahedral mesh is a user specified variable. This error defines the error between the field approximation using the tetrahedral mesh and the field defined by the input dataset and its interpolation method. The resampled mesh can be refined to approximate the underlying field to within a specified tolerance or until the mesh contains a specific number of tetrahedra. The resampled field is a linear approximation of the input field. The boundaries of the input mesh are represented in the same manner as the surfaces of discontinuity. The volume of space outside of the mesh boundary is considered as a separate material with constant field values. This region of *empty* space is easy to evaluate and approximate; no ghost values and no field approximations need to be computed. This allows a non-rectilinear input mesh to be embeded into the larger rectilinear mesh generated by the refinement of the tetrahedral grid. Discontinuities are supported at the cell level allowing local refinement of the representations of surfaces of discontinuity in geometrically complex areas. The convergence of the approximation depends upon

the complexity of the input field and the complexity of the input mesh. For meshes with complex geometry and interfaces but simple fields, the majority of the work is done approximating the input geometry and material interfaces. For meshes with simple geometry and interfaces but complex fields, the majority of work is done approximating the field values.

The framework has several advantages over other multiresolution spatial data structures such as an octree. The refinement method ensures that the tetrahedral mesh will always be free of cracks and *T-intersections*. This makes it easy to guarantee that representations of fields and surfaces of discontinuity are continuous across cell boundaries.

The resampling and error bounding algorithms require that an original dataset allow the extraction of the values of the field variables at any point and, for a given field, the maximum difference between the representation over one of the tetrahedral cells in the approximation and the representation in the original dataset over the same volume.

3.3.1 Building the Multiresolution Representation

Given a dataset and polygonal representations for the material interfaces, a multiresolution representation is constructed as follows:

1. The algorithm starts with the base configuration of six tetrahedra and associates with each one the interface polygons that intersect it.

2. The initial tetrahedral mesh is first subdivided so that the polygonal surface meshes describing the material interfaces are approximated within a certain tolerance. At each subdivision, the material interface polygons lying partially or entirely in a cell are associated with the cell's children; approximations for the polygons in each child cell are constructed, and interface approximation errors are computed for the two new child cells.

3. The mesh is further refined to approximate the field of interest, for example density or pressure, within a specified tolerance.

For the cells containing material interfaces, a field representation for each material passing through the cell is computed. This is done by extrapolating *ghost* values for each material at the vertices of the tetrahedron where the material does not exist. A ghost value is an educated guess of a field value at a point where the field does not exist. When material interfaces are present, field values for a given material do not exist at all of the tetrahedron's vertices. Since tri-linear approximation over a tetrahedron requires four field values at the vertices, extra field values are needed to perform the interpolation. Thus for a given field and material, the ghost values and the existing values are used to form the tri-linear approximation within the tetrahedron.

This is illustrated in Figure 3.1 for a field sampled over a triangular domain containing two materials. For the field sampled at V_0 and V_1, a ghost value at vertex V_2 is needed to compute a linear approximation of the field over the triangle. The approximation is used only for those sample points that belong to this material. Given a function sampled over a particular domain, the ghost value computation extrapolates a function value at a point outside of this domain. When the field approximation error for the cell is computed, the separate field representations, built using these ghost values, are used to calculate an error for each distinct material in the cell. Cell that contains multiple materials are refined as follows:

1. The signed distance values and ghost values for the new vertex are computed when the vertex is created during a split or subdivision operation. This is done by examining those cells that share the edge being split.

2. The interface representations, i.e., triangle meshes, are associated with the child cells, and their approximating representations and their associated errors are computed.

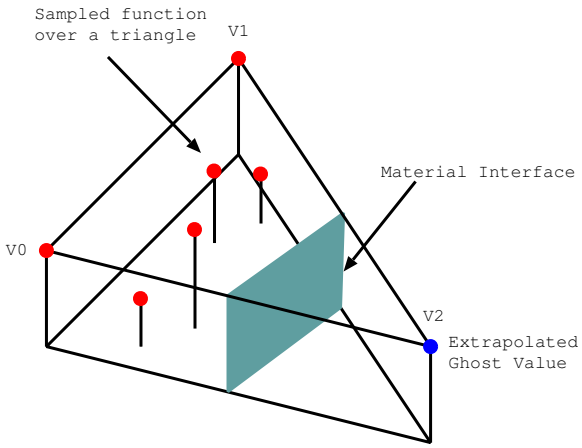


Figure 3.1: 2D Ghost Value Computation showing existing function values, material interface and extrapolated function value.

3. The field error for each of the materials is computed, and the maximum value of these errors is defined as the overall error associated with a cell containing multiple materials.

3.4 Material Interfaces

A material interface defines the boundary between two distinct materials. Figure 3.2 shows an example of two triangles crossed by an single interface (smooth curve). This interface specifies where the different materials exist within a cell. Field representations across a material interface are often discontinuous. Thus, an interface can introduce a large amount of error to cells that cross it. Instead of refining an approximation substantially in the neighborhood of an interface, the discontinuity in the field is better represented by explicitly representing the surface of discontinuity in each cell. Once the discontinuity is represented, two separate functions are used to describe the dependent field variables on either side of the discontinuity. By representing the surface of discontinuity exactly, this simplification algorithm does not need to refine regions in the spatial domain with a large number of tetrahedra.

3.4.1 Extraction and Approximation

In the class of input datasets considered here, material interfaces are represented as triangle meshes. In the case that these triangle meshes are not known, they are extracted from volume fraction data by a material interface reconstruction technique, see [3]. (The volume fractions resulting from numerical simulations indicate what percentages of which materials are present in each cell.) Such an interface reconstruction technique produces a set of crack-free triangle meshes and normal vector information that can be used to determine on which side and in which material a point lies.

Within a tetrahedra, an approximate material interface is represented as the zero set of a signed distance function. Each vertex of a tetrahedron is assigned a signed distance value for each of the material interfaces in the tetrahedron. The signed distance from a vertex $\mathbf{V}$ to an interface mesh $\mathbf{I}$ is determined by first finding a triangle mesh vertex $\mathbf{V}_i$ in the triangle mesh describing $\mathbf{I}$ that has minimal distance to $\mathbf{V}$. The sign of the distance is determined by considering the normal vector $\mathbf{N}_i$ at $\mathbf{V}_i$. If $\mathbf{N}_i$ points towards $\mathbf{V}$, then $\mathbf{V}$ is considered to be on the *positive side* of the interface; otherwise it is considered to be on the *negative side* of the interface. The complexity of this computation is proportional to the complexity of the material interfaces within a particular tetrahedra. In general a coarse cell in the mesh will contain a large number of interface polygons, and a fine cell in the mesh will contain a small number of interface polygons. The signed distance values are computed as the mesh is refined. When a new vertex is introduced via the mesh refinement, the computation of the signed distance for that vertex only needs to look at the interfaces that exist in the tetrahedra around the split edge. If those tetrahedra do not contain any interfaces, no signed distance value needs to be computed.

In Figure 3.2, the true material interface is given by the smooth curve and its approximation is given by the piecewise linear curve. The minimum distances from the vertices of the triangles to the interface are shown as dotted lines. The distances for vertices on one side of the interface (say,

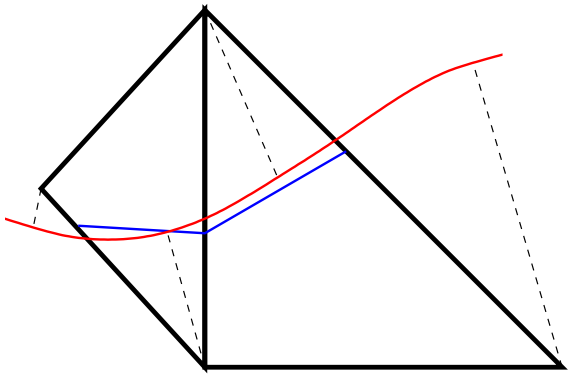


Figure 3.2: True and approximated material interfaces.

above the interface) are assigned positive values and those on the other side are assigned negative values. These signed distance values at the vertices determine linear functions in each of the triangles, and the approximated interface will be the zero set of these linear functions. Because the mesh is conformant, the linear functions in the two elements will agree on their common side, and the zero set is continuous across the boundary.

Figure 3.3 shows a two-dimensional example of a triangle with several material interfaces and their approximations. In this figure, the thin, jagged lines are the original boundaries and the thick, straight lines are the approximations derived from using the signed distance values. For the interface between materials A and B, the thin, dashed lines from vertices A, B, and C indicate the points on the interface used to compute the signed distance values. The signed distance function is assumed to vary linearly within the cell The coefficients for the linear function defining a boundary representation are found by solving a 4x4 system of equations, considering the requirement that the signed distance function over the tetrahedron must interpolate the signed distance values at the four vertices.

The three-dimensional example in Figure 3.4 shows a tetrahedron, a material interface approximation, and the signed distance values d_i for each vertex V_i. The approximation is shown

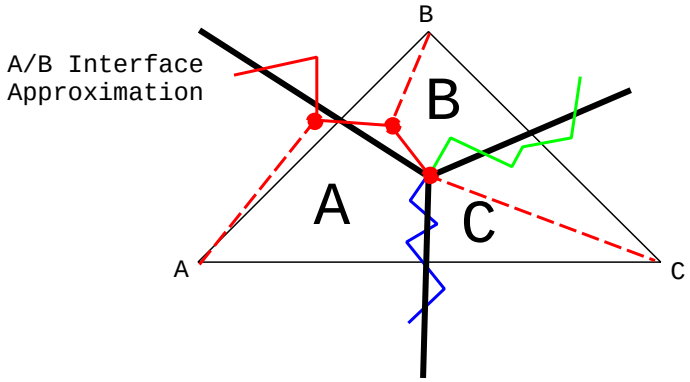


Figure 3.3: Triangle with three materials (A, B, and C) and three interfaces.

as a plane cutting through the tetrahedron. The normal vector N indicates the positive side of the material boundary approximation. Thus, the distance to V_0 is positive and the distances for V_1, V_2, and V_3 are negative.

It should be noted that a vertex has at most one signed distance value for each interface. This ensures that the interface representation is continuous across cell boundaries. If a cell does not contain a particular interface, the signed distance value for that interface is meaningless for that cell. Given a point **P** on an interface polygon and the associated approximation B_r, the error associated with **P** is the absolute value of the distance between **P** and B_r. The material interface approximation error associated with a cell is the maximum of these distances, considering all the interfaces within the cell.

3.5 Representing Discontinuous Fields

Cells that contain material interfaces typically also have discontinuities in the fields defined over them. For example, the density field over a cell that contains both steel and nickel is discontinuous exactly where the two materials meet. In these situations, it is better to represent the density field

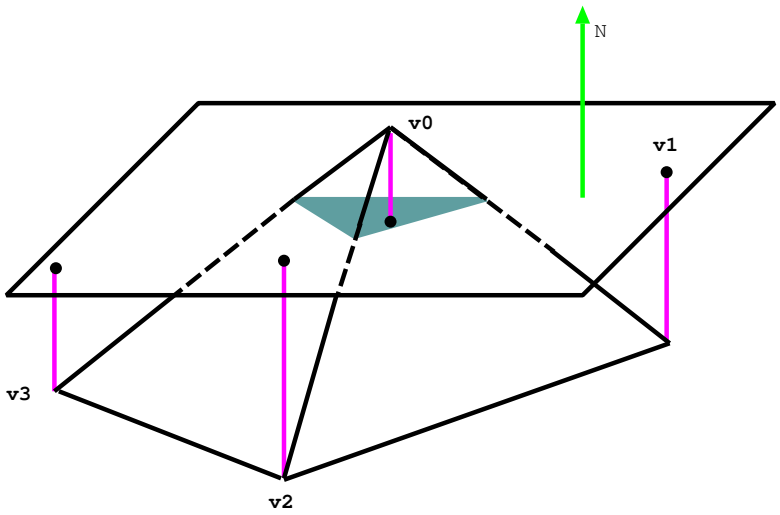


Figure 3.4: Tetrahedron showing signed distance values and the corresponding boundary approximation.

over the cell as two separate fields, one field for the region containing only the first material and one for the second material. One way to accomplish field separation is to divide the cell into two distinct cells at the material interface. In Figure 3.5, the triangle would be divided into a quadrilateral for material A and a triangle for material B. The disadvantages of this method are that it introduces new cell types into the mesh which makes it harder to have continuous field representations across cells. Furthermore, if new cell types are introduced, we loose the multiresolution structure and adaptive refinement capabilities of the recursive tetrahedral mesh structure. Our algorithm represents the discontinuity by constructing a field representation for each material in the cell. Each of the vertices in a cell must have distinct field values for each material in the cell. These extrapolated values are the ghost values.

For a vertex **V** that does not reside in material **M**, we compute a ghost value for the field associated with material **M** at vertex **V**. This ghost value is an extrapolation of the field value for **M** at **V**. The process is illustrated in Figure 3.5. The known field values are indicated by the solid circles. A_0 and A_1 represent the known field values for material A, and B_0 represents the known

field value for material B. Vertices A_0 and A_1 are in material A, and thus ghost values for material B must be calculated at their positions. Vertex B_0 lies in material B, and thus a ghost value for material A must be calculated at its position. As described in Section 3.3, the ghost value computation is performed when the vertex is inserted during the tetrahedral refinement process.

3.5.1 Computation of Ghost Values

The ghost values for a vertex **V** are computed as follows:

1. For each material interface present in the cells that share the vertex, find a vertex V_{min} in a triangle mesh representing an interface with minimal distance to **V**. (In Figure 3.5, these vertices are indicated by the dashed lines from A_0, A_1, and B_2 to the indicated points on the interface.)

2. Evaluate the dataset on the far side of the interface at V_{min} and use this as the ghost value at **V** for the material on the opposite side of the interface.

Only one ghost value exists for a given vertex, field and material. This ensures that the field representations are C^0 continuous across cell boundaries. For example, consider vertex V_0 of the triangle in Figure 3.3. The vertex V_0 lies in material A, and therefore we must compute ghost values for materials B and C at vertex A_0. The algorithm will examine the three material boundaries and determine the points from materials B and C that are closest to A_0. The fields for materials B and C are evaluated at these points, and these values are used as the ghost values for A_0. These points are exactly the points that were used to determine the distance map that defines the approximation to the interface. This computation assumes that the field remains constant on the other side of the interface. Alternatively, a linear or higher order function can be used to approximate the existing data points within the cell and to extrapolate the ghost value.

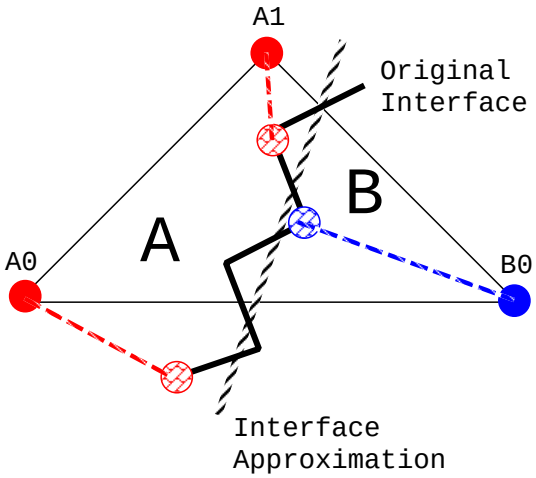


Figure 3.5: Ghost value computation for a triangle containing two materials.

3.6 Error Metrics

The error metrics employed are similar to the nested error bounds used in the ROAM system. Each cell has two associated error values: a field error and a material interface error. In order to calculate the field errors for a leaf cell in the tetrahedral mesh hierarchy, it is assumed that the original dataset can be divided into *native data elements*. Each of these is presumed to have a well defined spatial extent and a well defined representation for each field of interest over its spatial domain. The simplest example of a native data element is just a grid point that holds field values. Other possibilities are blocks of grid points treated as a unit, cells with a non-zero volume and a field representation defined over the entire cell, or blocks of such cells. Currently, only native data elements that are grid points of zero volume are considered. For a given field, it is assumed that it is possible to bound the difference between the representation over a tetrahedral leaf cell and the representation over each of the native data elements with which the given cell intersects. The error for the given field in the given cell is the maximum of the errors associated with each of the intersecting native data elements.

The field error e_T for a non-leaf cell is computed from the errors associated with its two children according to:

$$e_T = \max\{e_{T_0}, e_{T_1}\} + |z(v_c) - z_T(v_c)|, \tag{3.1}$$

where e_{T_0} and e_{T_1} are the errors of the children; v_c is the vertex that splits the parent into its children; $z(v_c)$ is the field value assigned to v_c; and $z_T(v_c)$ is the field value that the parent assigns to the spatial location of v_c. The approximated value at v_c, $z_T(v_c)$, is calculated as:

$$z_T(v_c) = \frac{1}{2}(z(v_0) + z(v_1)), \tag{3.2}$$

where v_0 and v_1 are the vertices of the parent's split edge. This error is still a genuine bound on the difference between the new approximate representation and the representation of the original dataset. However, it is looser than the bound computed directly from the data. The error computed from the children has the advantage that the error associated with a cell bounds not only the deviation from the original representation but also the deviation from the representation at any intermediate resolution level. Consequently, this error is *nested* or monotonic in the sense that the error of a child is guaranteed not to be greater than the error of the parent. Once the errors of the leaf cells are computed, the nested bound for all cells higher in the tree can be computed in time proportional to K, where K is the number of leaf cells in the tree. This can be accomplished by traversing the tree in a bottom-up fashion.

The material interface error associated with a leaf node is the maximum value of the errors associated with each of the material interfaces in the node. For each material interface, the error is the maximum value of the errors associated with the vertices constituting the triangle mesh defining the interface and being inside the cell. The error of a vertex is the absolute value of the distance between the vertex and the interface approximation. The material interface error **E** for a cell guarantees that no point in the original interface polygon mesh is further from its associated

approximation than a distance of **E**. This error metric is an upper bound on the deviation of the original interfaces from the approximated interfaces. A cell that does not contain a material interface is considered to have an interface error of zero.

Since both the function values $f(x,y,z)$ and the domain in which they exist are approximated, it is possible that a position $p = (x',y',z')$ which exists on one side of a material interface or discontinuity in the original dataset can move across the boundary in the approximation. This is very problematic because it leads to arbitrarily large L^∞ errors at these locations which makes it appear that the approximation is very bad. To avoid this problem, one must really consider a region of space $S = ([x_0,x_1],[y_0,y_1],[z_0,z_1])$ around p where the extent of the x,y and z dimensions is controlled by the approximation error of the original domain. This is because the approximation of the interface has warped the space around p and thus the position at which we evaluate the approximation to find the error at p must also be warped. The error associated with p is found by taking the minimum error between the original value at p, $f_o(p)$, and the function values at all points in S.

3.7 Results

The algorithm described here has been tested on a dataset resulting from a simulation of a hypersonic impact between a projectile and a metal block. The simulation was based on a logically rectilinear mesh of dimensions 32x32x52. For each cell, the average density and pressure values are available, as well as the per-material densities and volume fractions. The physical dimensions in x, y, and z directions are [0,12] [0,12] and [-16,4.8].

There are three materials in the simulation: the projectile, the block, and empty space. The interface between the projectile and the block consists of 38 polygons, the interface between the projectile and empty space consists of 118 polygons and the interface between empty space and the block consists of 17574 polygons. Figure 3.6 shows the original interface meshes determined

from the volume fraction information. The largest mesh is the interface between the metal block and empty space; the next largest mesh in the top, left, front corner is the interface between the projectile and empty space; the smallest mesh is the interface between the projectile and the block.

Figure 3.7 shows a cross-section view of the mesh created by a cutting plane through the tetrahedral mesh. The darker lines are the original interface polygons, and the lighter lines are the approximation to the interface. The interface approximation error is 0.15. (An error of 0.15 means that all of the vertices in the original material interface meshes are no more that a physical distance of 0.15 from their associated interface approximation. This is equivalent to an error of (0.5 - 1.5)% when considered against the physical dimensions.) A total of 3174 tetrahedra were required to approximate the interface within an error of 0.15. The overall mesh contained a total of 5390 tetrahedra. A total of 11990 tetrahedra were required to approximate the interface to an error of 0.15 and the density field within an error of 3. The maximum field approximation error in the cells containing material interfaces is 2.84, and the average field error for these cells is 0.007. These error measurements indicate that separate field representations for the materials on either side of a discontinuity can accurately reconstruct the field.

Figures 3.7 and 3.8 compare the density fields generated using linear interpolation of the density values and explicit field representations on either side of the discontinuity. These images are generated by intersecting the cutting plane with the tetrahedra and evaluating the density field at the intersection points. A polygon is drawn through the intersection points to visualize the density field. In the cells where material interfaces are present, the cutting plane is also intersected with the interface representation to generate data points on the cutting plane that are also on the interface. These data points are used to draw a polygon for each material that the cutting plane intersects.

Figure 3.8 shows that using explicit field representations in the presence of discontinuities can improve the quality of the field approximation. This can be seen in the flat horizontal and ver-

tical sections of the block where the cells approximate a region that contains the block and empty space. In these cells, the use of explicit representations of the discontinuities leads to an exact representation of the density field. The corresponding field representations using linear interpolation, shown in Figure 3.7, capture the discontinuities poorly. Furthermore, Figure 3.8 captures more of the dynamics in the area where the projectile is entering the block (upper-left corner). The linear interpolation of the density values in the region where the projectile is impacting the block smooths out the density field, and does not capture the distinct interface between the block and the projectile.

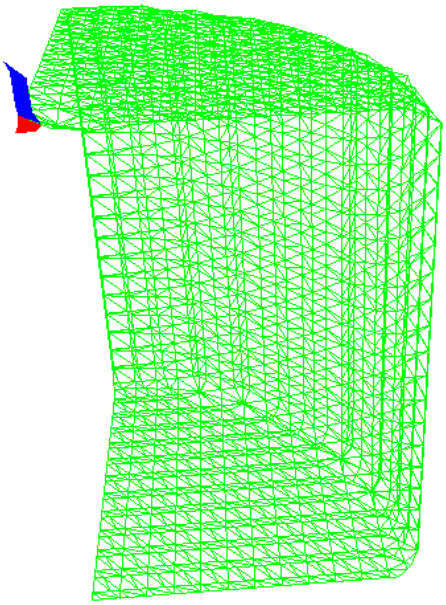


Figure 3.6: Original triangular meshes representing material interfaces.

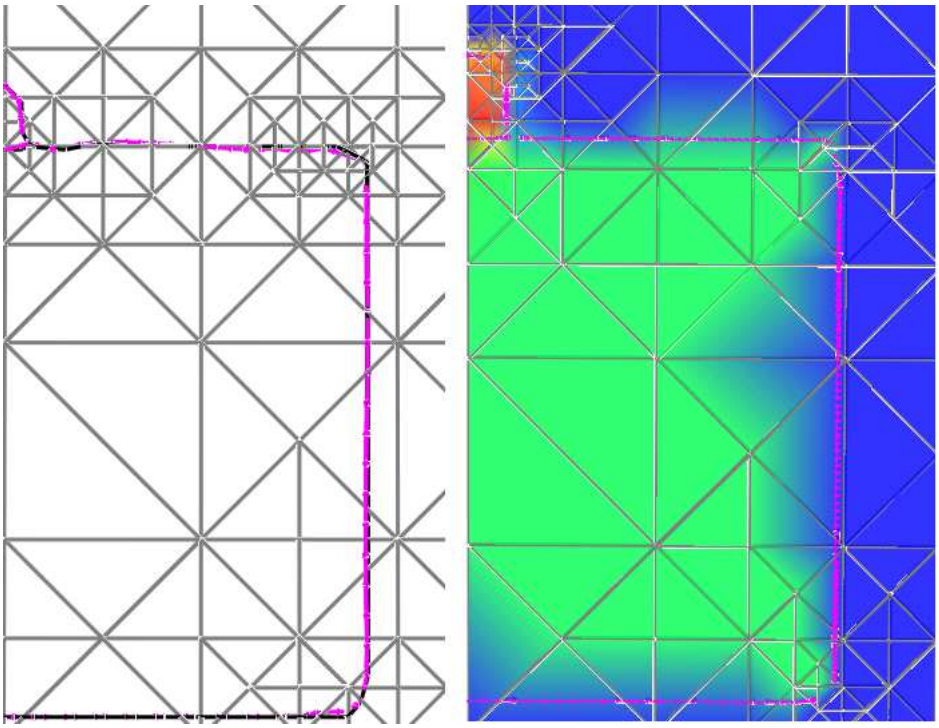


Figure 3.7: Cross section of the tetrahedral mesh. The left picture shows the original interfaces and their approximations. The picture on the right shows density field using linear interpolation.

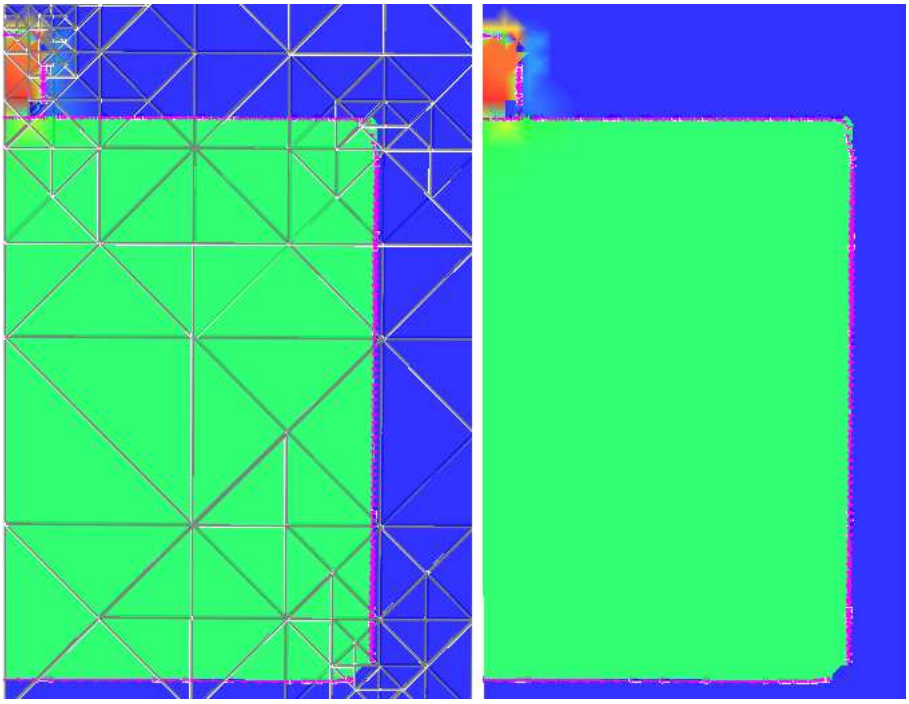


Figure 3.8: Density field using explicit interface representations and separate field representations. The left picture shows the field along with the approximating tetrahedral mesh. (interface error = 0.15).

Chapter 4

Contouring Time-independent Scalar Fields Defined over Three-dimensional Grids

4.1 Introduction

The advent of high-performance computing has completely transformed the nature of most scientific and engineering disciplines making the study of complex problems from experimental and theoretical disciplines computationally feasible. Traditionally, with smaller and simpler datasets, researchers have developed *in-core* isosurface visualization methods that work well on small or medium-scale datasets. They can quickly generate isosurfaces, and treat each isosurface independently. However, isosurfaces extracted from today's datasets require a different approach to address their ever increasing size and complexity.

This chapter presents algorithms for interactively extracting and rendering isosurfaces of large datasets in a view-dependent manner. Our algorithm generates isosurfaces "on-the-fly" using

a view-dependent error measure, a recursive tetrahedral mesh refinement scheme, and a unique data layout scheme suitable for out-of-core visualization of large datasets.

Surface based level-of-detail techniques such as [15] and [74] extract a coarse isosurface and iteratively build a multiresolution surface model. For large volume datasets that contain topologically complex isosurfaces with millions and millions of triangles, these techniques need to be combined with out-of-core simplification techniques such as those developed by Lindstrom [44] and Lindstrom and Silva [46] in order to operate. In some cases, the storage requirements needed to extract, simplify, and visualize these surfaces can actually exceed those of the volume data from which they are derived [8]. Interactively visualizing these types of isosurfaces requires algorithms such as those developed by Duchaineau et al. [7, 8], that combine multiresolution representations, compression, and view-dependent optimizations. Surface based techniques are not suitable for visualizing volumes that contain a large number of isosurfaces that are important to the user because they must extract all of the interesting surfaces which would take far too much storage to be practical. On the other hand, volume based techniques, which extract and render the isosurfaces directly, do not require the precomputation of selected isosurfaces and can easily switch between isovalues.

We utilize the refinement of a tetrahedral mesh via longest-edge bisection (Chapter 2) to build a multiresolution hierarchy of a volume dataset. We combine coarse-to-fine and fine-to-coarse refinement schemes for this mesh to create an adaptively refinable tetrahedral mesh. This adaptive mesh supports a dual priority queue split/merge algorithm similar to the ROAM system [9] for view-dependent terrain visualization. It has fast coarsening and refinement operations which allow for localized, incremental mesh updates, strict frame-to-frame triangle counts, progressive improvements of mesh quality, and guaranteed frame rates. The refinement scheme is coupled with a data storage scheme which aligns the data on disk and in main memory with the access pattern dictated by the mesh refinement. Sets of tetrahedra that share a common refinement edge

are grouped into an aggregate structure called a diamond as described in Section 2.0.2. Diamonds, as opposed to tetrahedra, function as the unit of operation in the mesh hierarchy and simplify the process of refining and coarsening the mesh.

At runtime, the split-merge refinement algorithm is used to create a lower resolution dataset that approximates the original dataset to within a given error tolerance. The error tolerance is a measure of how much an isosurface, extracted from the lower resolution dataset, deviates from the finest level isosurface. The error tolerance is measured in pixels on the view screen. The lower resolution dataset is a set of tetrahedra, possibly from different levels of the hierarchy, that approximates the volume dataset to within this isosurface error tolerance. This set of tetrahedra is free from cracks and T-intersections, and it defines a piecewise linear approximation of the original data. The isosurface is extracted from the tetrahedra in this lower resolution representation using linear interpolation.

4.2 Previous Work

The refinement of a tetrahedral mesh via longest edge bisection is described in detail in several papers. In Zhou et al. [78], a fine-to-coarse merging of groups of tetrahedra is used to construct a multi-level representation of a dataset. Their representation approximates the original dataset to within a specified tolerance and preserves the topology of the finest level mesh. For larger datasets, this fine-to-coarse strategy is not practical because storing the finest level mesh would require too much memory.

An improved algorithm for preserving the topology of an extracted isosurface is presented by Gerstner and Pajarola [18]. This algorithm is combined with a coarse-to-fine splitting of tetrahedra to extract topology preserving isosurfaces or to perform controlled topology simplification. Rendering of multiple transparent isosurfaces and parallel extraction of isosurfaces are presented by

Gerstner [16] and by Gerstner and Rumpf [17]. Both of these algorithms extract the isosurfaces from the mesh in a coarse-to-fine manner. In Roxborough and Nielson [61], the coarse-to-fine refinement algorithm is used to model 3-dimensional ultrasound data. The adaptivity of the mesh refinement is used to create a model of the volume that conforms to the complexity of the underlying data.

View-dependent extraction of isosurfaces utilizes multiresolution representations to extract surfaces that satisfy certain visual requirements. These requirements are usually based on the distance of the surface from the viewpoint, the position of the surface relative to the view-frustum, and the occlusion of the surface. Duchaineau et al. [9] control refinement using the screen space projection error, view-frustum culling, and line of site corrections. Occlusion culling supplements view-frustum culling by finding areas within the visible region that cannot be seen. In Livnat and Hansen [51], a hierarchical visibility test is used to determine regions of the volume that are occluded. The volume is decomposed using an octree, and the visibility test is performed using hierarchical tiles based on coverage masks (see Greene [22]). A shear warp transformation is used to perform the screen space projection. Their visibility algorithm requires that the octree be traversed from front to back. A ray tracing approach is also used in [50] to find an initial set of voxels from which to propagate the isosurface. The algorithm starts extracting the isosurface from these seed sets, and detects when the surface folds back on itself and becomes occluded. In our algorithm, we use the screen space projection error of the isosurface, view-frustum culling, and occlusion culling to control the view-dependent refinement.

The algorithms described in this chapter focus on level-of-detail based, interactive exploration of large, complex isosurfaces. Surface based methods such as [8, 74] construct level-of-detail surface models that are suitable for interactive view-dependent rendering. Volume based techniques such as [51, 75] speed up the search for cells that contain the isosurface and cells that do not need to be rendered, but they extract the isosurface from the finest level cells. Our algorithm differs from

these approaches by utilizing a level-of-detail volumetric model which extracts the isosurface from coarser representations of the volume that meet certain requirements. Isosurface extraction techniques based upon level-of-detail allow the isosurface to be progressively refined over time, see for example [11, 57]. Visualizing large, complex isosurfaces often requires the ability to fly through the dataset and closely inspect areas of interest. Level-of-detail methods that support strict triangle counts per frame for efficient rendering, progressive improvement of mesh quality to provide guaranteed frame rates, and coherent access to data to minimize memory faults are well suited to this task.

4.3 Preprocessing

In a preprocessing phase, the following information, used to drive the runtime mesh refinement, is computed for each diamond (Chapter 2):

1. The isosurface approximation error of the region enclosed by the diamond. (Section 4.6)

2. The min and max data values within the diamond including the diamond's boundary. The precomputed min/max ranges are used to quickly cull regions of the dataset that do not contain the isosurface.

3. The normalized gradient vector at the center point or split vertex of the diamond (including the 8 unused points of the coarsest cube). The precomputed gradient vectors are used to shade the isosurface using texture mapping. The gradient is used as a texture coordinate to perform diffuse lighting and to highlight the regions orthogonal to the viewing direction such as internal or silhouette edges. At each frame the texture coordinates (i.e. gradients) are multiplied by the inverse of the rotation component of the modeling matrix, translated and scaled so that all components lie within the range $[0,1]$. The texture used for rendering is

shown in Figure 4.1.

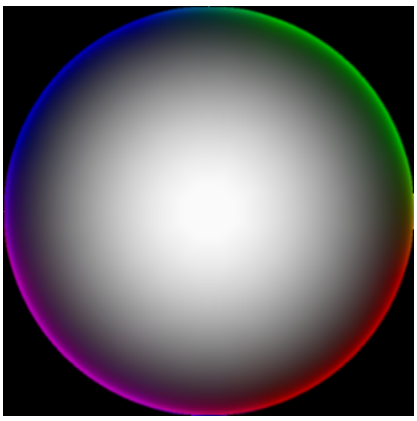


Figure 4.1: Diffusely illuminated sphere texture map used for shading. The colored edges are mapped to regions where the gradient vector is nearly perpendicular to the viewing direction.

Gradients can be computed at runtime and stored in a hash table; however, our data layout algorithm (Section 4.5) makes computing gradients via central differences expensive and so the gradients are precomputed. Assuming byte scalar field data, floating point errors and floating point gradient components, 18 additional bytes are required per input value. A 1024^3 (1 Gb) dataset would be inflated to an enormous 19 Gb dataset.

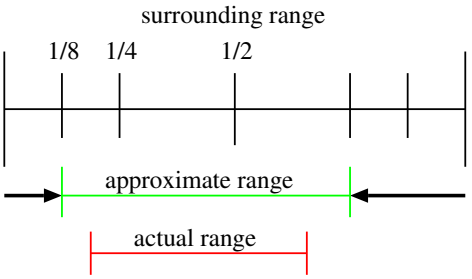


Figure 4.2: The min/max values of a diamond are encoded relative to the min and max values of an enclosing diamond using 4 bits (2 each for min/max) to encode 0/8, 1/8, 1/4, or 1/2 of the enclosing interval. (The offset 3/8 is not encoded.) In this example min = 1/8, max = 1/4.

In order to reduce the size of the precomputed data, the isosurface errors are compressed on a logarithmic scale on a per-level basis and represented in six bits. The min and max values for

a diamond D are compressed in relation to a diamond S that completely contains D. A diamond S contains a diamond D if the polyhedron for D is completely enclosed by the polyhedron for S. The tetrahedra created by recursively refining D's tetrahedra are all contained within S's polyhedron. Either of the two diamonds whose split vertices are the vertices of $D's$ split edge can be used for S. This is illustrated in Figure 4.3.

Many algorithms exist for quantizing gradients. The gradients can be uniformly distributed over a unit cube or sphere or a more expensive codebook based quantization algorithm can be used to create an optimal set of gradients. In these techniques, the gradient value is used as an index into a lookup table. In the initial implementation, gradient vectors were quantized on a unit cube using fourteen bits. Gradients quantized in this manner often do not have a lot of coherence. That is, two gradients that are very similar, as measured by their dot product, can have very different indices and vice versa. For data compression, as discussed in Sections 4.11 and 5.7, we need a gradient representation that has some coherence over space for static volumes and over time for time-varying volumes and a meaningful delta between two gradients. To meet these requirements, the gradients are quantized component-wise. For compression of static isosurfaces, each component of the normalized gradient is quantized with 8 bits. A different quantization scheme used for the compression of time-varying data is described in Section 5.7.

Using these compression techniques, the precomputed information for each diamond is stored in three bytes. (6 bits for error, 4 for min/max, 14 for gradient). A 1 Gb dataset is inflated to 4 Gb instead of 19 Gb. Error values and gradients are found at runtime using lookup tables. Since the errors are encoded in 6 bits, the table for the error values contains $2^6 \times n$ floating point values where n is the number of levels in the mesh. For a 512^3 dataset, n equals 9. The gradient table contains 3×2^{14} floating point values. The use of subtrees, described later in Section 4.9, allows the precomputed min and max values to be stored uncompressed.

4.3.1 Gradient Processing

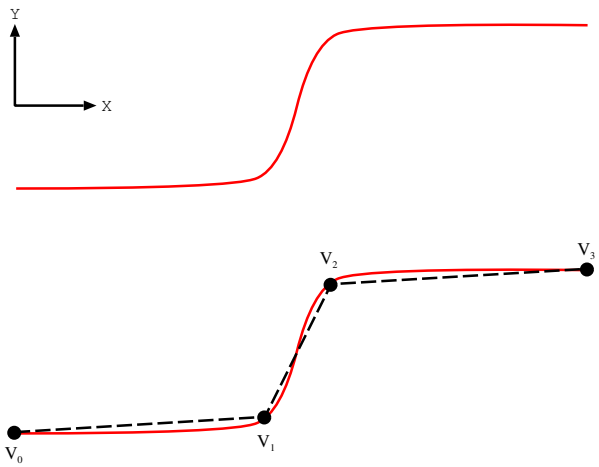


Figure 4.3: Top: A smooth 1-D function. Bottom: The function is approximated with three line segments. At the endpoints V_0 and V_3, the gradient computed from the original function is 0.

Gradients for shading are estimated from the volume dataset using central differences. These gradients can be very noisy and in constant-valued regions they are degenerate (i.e. their magnitude is zero). This is illustrated in Figure 4.3. The function is approximated with three linear segments. Since the function is flat at the ends, the gradients at the endpoints V_0 and V_3 are degenerate. For isosurfaces extracted from different levels of detail, noisy and degenerate gradients are especially bad for shading. In order to improve the quality of the gradients for shading purposes, the standard central differences algorithm for gradient computation is modified to use a larger set of neighboring values (Section 4.3.2), and the gradients are smoothed in the spatial and temporal domains(for time-varying data) to diffuse non-degenerate gradients into the degenerate regions (Section 4.3.3).

4.3.2 Calculating Gradients

For a univariate function $y = f(x)$ the gradient is defined as:

$$\nabla f = \frac{\partial f}{\partial x}. \tag{4.1}$$

It can approximated using central differences as:

$$\nabla f = \frac{f(x+h) - f(x-h)}{2h}. \tag{4.2}$$

The gradient of a trivariate function $y = f(x,y,z)$ is defined as:

$$\nabla f(x,y,z) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right). \tag{4.3}$$

It is approximated using central differences as:

$$\begin{aligned} \nabla f(x,y,z) = & \\ & \left(\frac{f(x+h,y,z) - f(x-h,y,z)}{2h}, \right. \\ & \frac{f(x,y+h,z) - f(x,y-h,z)}{2h}, \\ & \left. \frac{f(x,y,z+h) - f(x,y,z-h)}{2h} \right). \end{aligned} \tag{4.4}$$

This gradient calculation uses six data values surrounding the point (x,y,z). In the case of regular volume data, a spatial location (x,y,z) has 26 surrounding data points and it is desirable to use all of these data points to improve the gradient calculation's quality.

The standard central difference method for computing ∇f shown in Equation 4.4 use six points or three directions. The 26 surrounding points have a total of 13 directions. (i.e. directions $(1,1,1)$ and $(-1,-1,-1)$ are considered the same.) In order to use the additional surrounding points to compute the gradient, we use directional derivatives.

Given a function $y = f(x,y,z)$ and a direction vector $v = (A,B,C)$, the derivative of $f(x,y,z)$ in direction (A,B,C) is defined as:

$$\frac{\partial f}{\partial v} = \nabla f \cdot (A,B,C) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right) \cdot (A,B,C) \tag{4.5}$$

For example the calculation of $\frac{\partial f}{\partial x}$ can be expressed in terms of directional derivatives as:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial v} = \nabla f \cdot v \tag{4.6}$$

where v is the vector $(1,0,0)$.

As another example, consider the direction vector $v = (1,1,0)$ in the xy plane passing through the spatial location (x,y,z). The directional derivative $\frac{\partial f}{\partial v}$ at (x,y,z) can be computed as:

$$\begin{aligned} \frac{\partial f}{\partial v} &= \nabla f \cdot \left(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, 0 \right) \\ &= \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right) \cdot \left(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, 0 \right) \end{aligned} \tag{4.7}$$

This derivative can be approximated as:

$$\frac{\partial f}{\partial v} = \frac{f(x+1,y+1,z) - f(x-1,y-1,z)}{2\sqrt{2}}. \tag{4.8}$$

In matrix form the directional derivative's equation is:

$$\frac{\partial f}{\partial v} = \begin{bmatrix} 1/\sqrt{2} & 1/\sqrt{2} & 0 \end{bmatrix} \begin{bmatrix} \partial f/\partial x \\ \partial f/\partial y \\ \partial f/\partial z \end{bmatrix} \tag{4.9}$$

Numbering the 13 directions $S_0, S_1 \ldots S_{12}$, the equation for the gradient is written in matrix

form as:

$$\begin{bmatrix} 1/\sqrt{3} & 1/\sqrt{3} & 1/\sqrt{3} \\ -1/\sqrt{3} & 1/\sqrt{3} & 1/\sqrt{3} \\ -1/\sqrt{3} & 1/\sqrt{3} & -1/\sqrt{3} \\ 1/\sqrt{3} & 1/\sqrt{3} & -1/\sqrt{3} \\ 1/\sqrt{2} & 1/\sqrt{2} & 0 \\ 1/\sqrt{2} & -1/\sqrt{2} & 0 \\ 1/\sqrt{2} & 0 & 1/\sqrt{2} \\ -1/\sqrt{2} & 0 & 1/\sqrt{2} \\ 0 & 1/\sqrt{2} & 1/\sqrt{2} \\ 0 & -1/\sqrt{2} & 1/\sqrt{2} \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \partial f/\partial x \\ \partial f/\partial y \\ \partial f/\partial z \end{bmatrix} = \begin{bmatrix} \partial f/\partial S_0 \\ \partial f/\partial S_1 \\ \partial f/\partial S_2 \\ \partial f/\partial S_3 \\ \partial f/\partial S_4 \\ \partial f/\partial S_5 \\ \partial f/\partial S_6 \\ \partial f/\partial S_7 \\ \partial f/\partial S_8 \\ \partial f/\partial S_9 \\ \partial f/\partial S_{10} \\ \partial f/\partial S_{11} \\ \partial f/\partial S_{12} \end{bmatrix} \tag{4.10}$$

Where $\partial f/\partial S_0 \ldots \partial f/\partial S_{12}$ are the directional derivatives in the 13 directions calculated using Equation 4.2. These directions, shown on left side of Equation 4.10, have been normalized for the computation of the directional derivative.

Rewriting this as $Ax = b$ and multiplying by A^T to make the matrices square yields:

$$\left[A^T\right][A][x] = \left[A^T\right][b]. \tag{4.11}$$

This equation can now be solved using standard techniques to get the gradient vector $\nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}\right)$.

4.3.3 Gradient Smoothing

Figure 4.4 shows images of the raw and smoothed gradients from the Richtmyer-Meshkov dataset. The normalized gradient (g_x, g_y, g_z) is converted to an rgb color as follows:

$$[r, g, b] = ([g_x, g_y, g_z] + 1.0) \times 0.5$$

Gray areas indicate regions where the gradient is $(0, 0, 0)$.

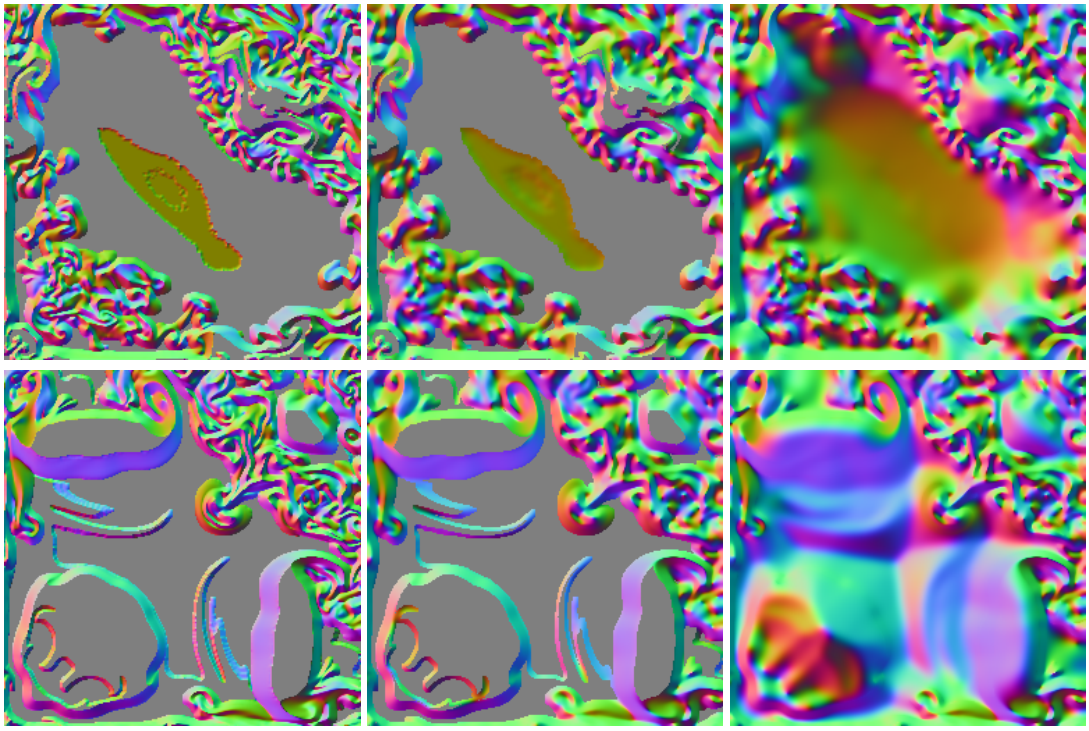


Figure 4.4: Top and bottom: images of raw and smoothed gradients from time steps 160 and 273, slice $z = 100$ of a $256^3 \times 274$ dataset. Left to Right: Original gradients, gradients after smoothing phase one, and after smoothing phase two.

The gradient smoothing process consists of two separate phases. The first phase is a symmetric diffusion of the gradients where the gradient at each data point is replaced with a weighted average of itself and the surrounding gradients. In this phase, if the gradient at a data point is degenerate, it is not updated. The second phase is a constrained symmetric diffusion, the new gradient, again a weighted average of itself and the surrounding gradients, is not allowed to move to far from

the value obtained after the first diffusion phase. The constrained diffusion is only performed where the original gradients are not degenerate. In the degenerate regions, the symmetric diffusion from the first phase is performed. Results of this process are illustrated in Figure 4.4. which shows the results after both smoothing phases.

The constrained diffusion process works as follows: Given the unnormalized gradient g_o from phase 1 and the weighted average gradient g_a, we compute the difference gradient g_d as:

$$g_d = g_a - g_o$$

The new gradient g_n is given as

$$g_n = \begin{cases} |g_d| > \frac{K}{|g_o|} & : \quad g_o + \frac{g_d}{|g_d|} \times \frac{K}{|g_o|} \\ otherwise & : \quad g_o \end{cases} \quad ,K > 0$$

Where $K > 0$. Regions where the original gradient has zero magnitude continue to undergo the diffusion process of the first phase. In both phases of the smoothing process, if the weighted average gradient has zero or near zero magnitude it is not used. This occurs in regions where the data varies very slowly and the gradients are not meaningful.

4.4 Split-Merge Refinement

The tetrahedral mesh supports the dual queue split-merge refinement strategy similar to that described by Duchaineau et al. [9]. This strategy provides more frame-to-frame coherence than a coarse-to-fine only algorithm. It allows us to control the triangle count per frame and to effectively cache previously computed geometry to minimize expensive interpolation calculations. In most interactive applications, the viewing position does not change significantly between consecutive frames. In frame $i+1$, many diamonds from frame i will have a view-dependent error that is still within the error tolerance. These diamonds can be reused in frame $i+1$. A small fraction of the

diamonds must be split or merged to satisfy the error tolerance. By starting the refinement process for frame $i+1$ with the mesh from frame i instead of the base mesh, fewer splits and merges are performed.

The *current mesh* is a set of tetrahedra that approximates the volume dataset to within a certain view-dependent error bound. The mesh is generated using two priority queues. The split queue holds the diamonds containing the tetrahedra of the current mesh. The merge queue holds the diamonds that have been split and whose children have not been split (i.e. diamonds with children but no grandchildren).

At frame 0, the split queue is initialized with the base configuration of six tetrahedra (the root diamond), and the merge queue is empty. At each frame, given a view-dependent error tolerance E, the following steps are taken:

1. Diamonds not within the view frustum and diamonds that are occluded (Section 4.10) are marked as *invisible* and diamonds that do not contain the isosurface are marked as *empty*; they are assigned a view-dependent error of zero. View-dependent errors are recomputed for all other diamonds in the split and merge queues.

2. Diamonds in the split queue whose error is greater than E are split. Diamonds in the merge queue whose error is less than E are merged. *Invisible* and *empty* diamonds in the split queue are never split. In the merge queue, they are the first diamonds to be merged.

3. The refinement process is stopped when all diamonds in the split queue have an error below E and all diamonds in the merge queue have an error above E, or when the time allowed for processing the current frame has elapsed.

4. The isosurface is extracted from the tetrahedra that belong to the visible, non-empty diamonds in the split queue.

A diamond D is split by splitting all of its tetrahedra, and inserting the child tetrahedra into the split queue. A tetrahedron is placed into the split queue by creating an entry for its diamond and adding the tetrahedron to the diamond. There is only one entry for a diamond in the split queue. When some of the tetrahedra in a diamond do not exist (i.e. they are not in the current mesh), it is necessary to create them before the diamond can be split. This situation is shown in 2D in Figure 4.5. The tetrahedra are created by splitting the parents of D that have not been split. When all the parents and tetrahedra of D have been split, D is removed from the split queue and added to the merge queue.

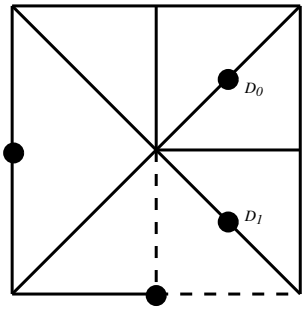


Figure 4.5: Diamond D_0 has two triangles in the mesh. Diamond D_1 has two triangles, one of which in the mesh. The triangle not in the mesh is shown with the dashed lines. This is a 2D analogy of the 3D tetrahedral mesh.

Merging a diamond is done by merging all of its tetrahedra, and adding them to the split queue. A tetrahedron is merged by removing its two children from the split queue. A tetrahedron is removed from the split queue by locating its diamond's entry in the split queue and removing it from the diamond. When a tetrahedron is removed from the mesh, its diamond is checked to see if all the tetrahedra of the diamond have been removed from the queue. If so, the diamond is removed from the split queue. Lastly, the diamond's parents are checked to see if they can be added to the merge queue. A diamond can be added to the merge queue only if all of its children are in the split queue.

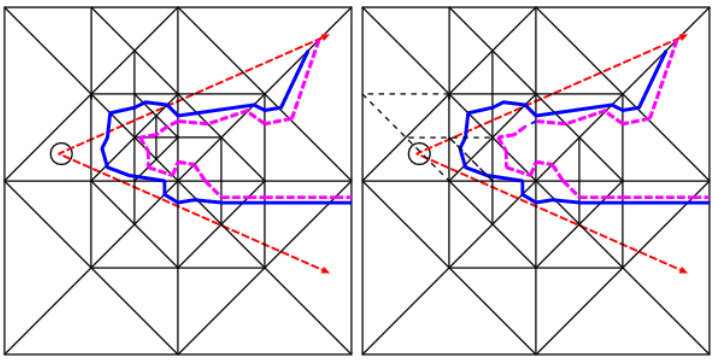


Figure 4.6: The new contour (solid line) is close to the old contour (dashed line). The new contour is extracted from the current mesh and refinement continues.

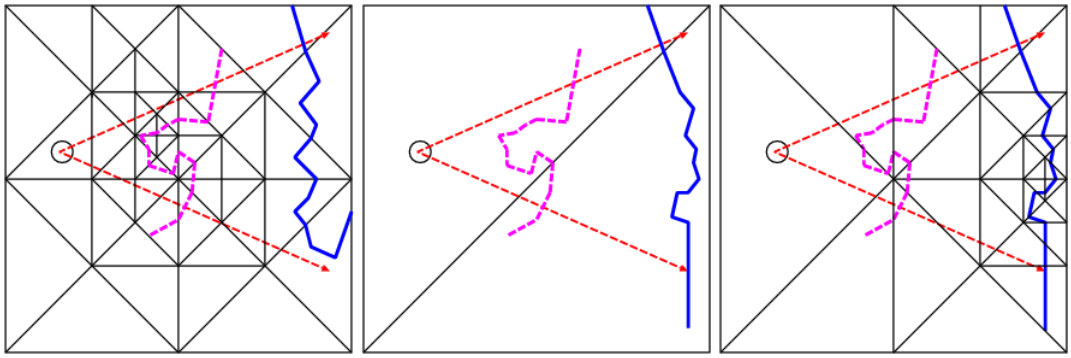


Figure 4.7: The two contours are far away. The old mesh is discarded, and refinement starts from the base mesh.

4.4.1 Changing the Isovalue

When the isovalue is changed, the new isosurface can be extracted by starting at the root diamond or starting from the current mesh as shown in Figures 4.7 and 4.6. In the first case, the split queue, merge queue and isosurface are invalidated and initialized with the root diamond. Refinement then begins from this initial configuration. In the second case, the split queue and merge queue remain the same, but the old isosurface is discarded. The diamonds in the split and merge queues are checked to determine if they contain the new isovalue. Diamonds that do not contain the isovalue are marked as empty and given an approximation error of zero. Error values are recomputed for diamonds that contain the new isosurface but did not contain the old isosurface (i.e. diamonds added to the mesh

because of forced splits). Isosurface errors are computed for those diamonds that contain the new isovalue. The split-merge refinement continues from this new configuration. Regions that contained the old surface but do not contain the new surface will be coarsened since they are not needed, and regions that contain the new surface will be refined and coarsened as necessary to meet the error threshold.

The performance and effectiveness of both of these methods depends on the distance between the old and new isosurfaces in the mesh hierarchy. Given two isosurfaces I_0 and I_1 and their corresponding meshes M_0 and M_1, the distance between the isosurfaces is measured as the number of splits and merges necessary to convert M_0 to M_1. Starting from the current configuration makes sense if they are close together, and starting from the base configuration makes sense if they are far apart. Similarly if a new isosurface is close to the existing surface, extracting the new surface and refining the mesh will be very coherent since much of the necessary data is already in memory and the data structures do not need to be rebuilt. On the other hand, if the two surfaces are far apart, extracting the new surface will be very incoherent and will probably generate a large number of page faults as the system loads data from regions stored on disk.

4.5 Memory Layout

When visualizing very large datasets, memory performance is a key bottleneck that must be overcome to achieve interactivity. In order to improve cache performance and effectively utilize the available memory bandwidth, we arrange our data on disk and in memory in a hierarchical z-order layout which follows the data ordering indicated by the mesh refinement.

Figure 4.8 illustrates the connection between the mesh refinement and the z-order space filling curve. Starting with the root configuration on the left, the dots indicate which data points are introduced at each refinement step. The numbers indicate the order in which the points are stored in

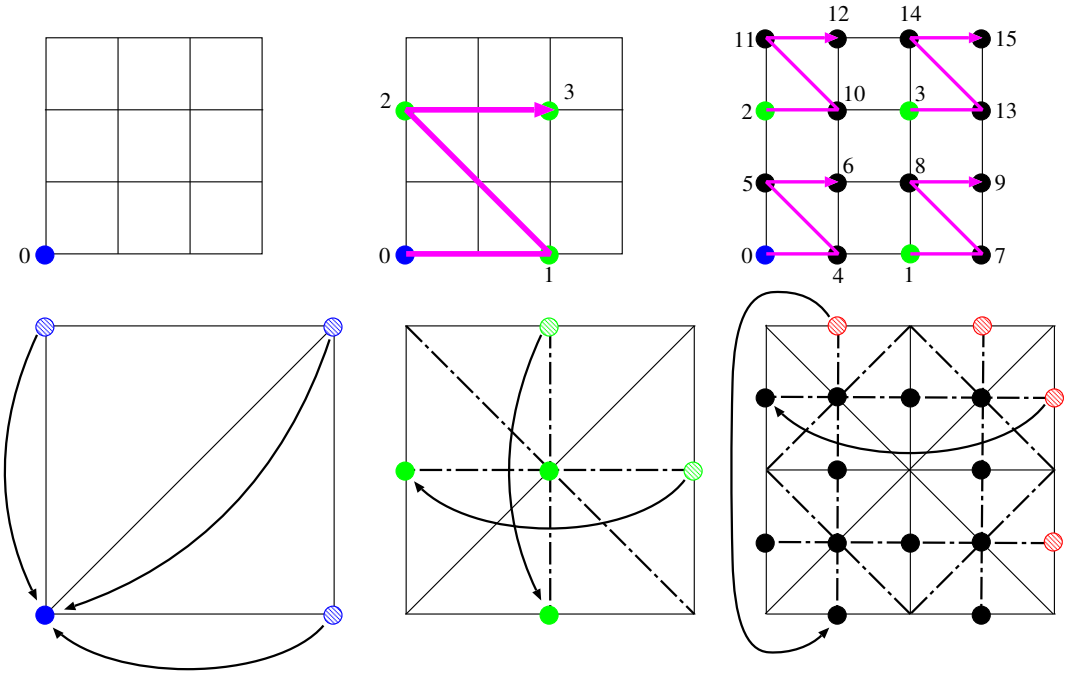


Figure 4.8: 2-D example of z-order and mesh refinement. Top: the order of the data points on the z-order curve. Bottom: The data points introduced by the mesh refinement at each hierachy level. The dashed circles and curved arrows illustrate how (2^k+1) indices in the mesh hierarchy wrap to 0 on the z-order curve.

a one-dimensional array. The dataset is stored first by level-of-detail (i.e. quadtree or octree level) and then by geometric proximity within each level. The order of the points introduced by each level of mesh refinement is essentially the same as the order given by the z-order curve. The only difference is that the mesh refinement operates on $(2^k+1)^3$ grids, and the z-order curve works on $(2^k)^3$ grids. The z-order tiles the grid in space causing index 2^k to wrap to 0. Each (i,j,k) index of point P in the dataset corresponds to a $z-order$ index on the one dimensionsl z-order curve. It is easy and fast to convert between (i,j,k) and $z-order$ indices, see Appendix A.1 and [58].

This data layout scheme and its performance benefits are detailed in [45] and [58]. Storing the data in this manner improves the coherence of the data access which is essential when working with large datasets. The original dataset and the information computed in the preprocessing phase (Section 4.3) of our algorithm are stored on disk in this manner. An explicit paging scheme using a

fixed sized chunk of main memory and Least Recently Used replacement is used to load data from disk as needed. Alternatively, the data can be mapped from disk to main memory at runtime using the Unix **mmap** command. The mmap command establishes a mapping between a process's address space and a virtual memory object represented as a disk file. It provides us with a basic out-of-core paging algorithm. These paging schemes allow us to keep in memory the data that is currently being used by the split/merge process and the isosurface extraction process. On GNU/Linux systems, mmap can currently be used to access an address space of up to 1.4GB. For larger files, explicit paging must be used. The combined features of the mesh hierarchy and the z-order curve give excellent disk and memory coherence making them well suited for adaptive, out-of-core visualization of large datasets.

4.6 Error Metrics

Each diamond in the mesh has an associated approximation error, isosurface error, and view-dependent error. The approximation error e_a for a tetrahedron T is the maximum difference between the linear approximation over T that interpolates the values at $T's$ vertices and the actual data values for the points inside T and on its boundary (i.e. faces, edges, and vertices). The approximation error for a diamond D is the maximum of the approximation errors of its tetrahedra. Leaf tetrahedra and leaf diamonds have an approximation error of zero.

The isosurface error of a tetrahedron T is the maximum deviation of an isosurface generated using the scalar values at the vertices of T from the true isosurface passing through T. This calculation is illustrated in Figure 4.9 for univariate functions. The original function is $f(x)$ and it is approximated by $L(x)$. The upper and lower bounds on the approximation, given by the approximation error e_a, are $a_1(x)$ and $a_2(x)$. For a given function value y, the isocontour using $L(x)$ occurs at point a where $y = L(a)$, while the true isocontour using $f(x)$ occurs at the point b where $y = f(b)$.

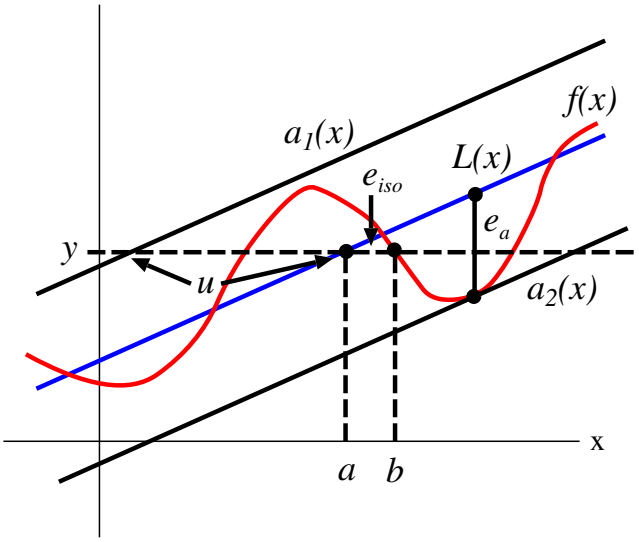


Figure 4.9: Contour error estimation in univariate case.

The error in the isocontour is given by:

$$e_{iso} = |a - b| \tag{4.12}$$

An upper bound u for the isosurface error can be computed by:

$$u = e_a/k \geq e_{iso}, \tag{4.13}$$

where k is slope of the linear approximation L. As f approaches a vertical line the slope of L increases, and f is approximated with increasing accuracy by L. As the slope of f decreases, the isocontour approximation a and the true isocontour b can be far apart even if e_a is small. In higher dimensions, the slope of the approximation translates to the magnitude of the gradient. In three-dimensions, this is the gradient of the field through a tetrahedron as given by the linear function that interpolates the values at the tetrahedron's vertices. The isosurface error is clamped at the physical size of the tetrahedron because the isosurface drawn through a tetrahedron can never be outside the tetrahedron's boundaries. The isosurface error for a tetrahedron T is given by:

$$e_{iso}(T) = \min(e_a/\|\nabla f(T)\|, diam(T)), \tag{4.14}$$

The isosurface error $e_{iso}(D)$ for a diamond is:

$$e_{iso}(D) = \max(e_{iso}(T), \forall T \in D). \tag{4.15}$$

The view-dependent error of a diamond is the projection of its isosurface error onto the view screen. This projection is done by creating a sphere at the diamond's split vertex of radius $e_{iso}(D)$ and projecting this sphere onto the view screen. The size of the projected sphere (i.e. width or height in pixels) is the view-dependent error. Details on view-dependent error metrics can be found in Hoppe [32], Lindstrom and Pascucci [45], and Luebke and Erikson [52]. All of these error metrics are easily incorporated into our refinement strategy.

4.7 Mesh Encoding

The mesh structure can be encoded in a very compact manner assuming that the data points lie on a $(2^n+1) \times (2^n+1) \times (2^n+1)$ grid. In this case, the offsets, relative to the split vertex of the diamond, to compute a tetrahedron's vertices, parent diamonds and child diamonds are all powers of two. Data that do not lie on such a grid can either be resampled to lie on a grid of the proper size or embedded in a *virtual grid* of the proper size.

Since each data point corresponds to a diamond, we represent a diamond using an (i, j, k) index. This index corresponds to the index used to access the precomputed diamond information and data values if they were stored in a C-style 3-dimensional array. The vertices defining the split edge of a diamond are encoded in a single byte as an offset vector from the split vertex. For example, the split edge with $SV_0 = (64, 64, 0)$ and $SV_1 = (64, 0, 64)$ has the vector $(0, -64, 64)$ and split vertex $(64, 32, 32)$. Dividing this vector by 64 yields $(0, -1, 1)$. These values are stored as 2 bit quantities in a single byte. SV_0 and SV_1 are computed by rescaling the vector and adding/subtracting it from the split vertex. In this example, $(0, -1, 1)$ is rescaled to $(0, -32, 32)$. The rescaling factor is easily

determined from the level of the diamond. For a mesh with l levels, the scaling factor for a diamond at level j is given by 2^{l-j-1}. The split edge encodings are stored in a lookup table and accessed at runtime based upon the type of the diamond. Since a diamond is identified by its split vertex, the vertices on the split edge can be computed by knowing the diamond's type and level. The parents, tetrahedra, and children of a diamond are encoded and stored in the same manner as the split edge. For any diamond, the (i, j, k) index for a parent, child or vertex of a tetrahedron can be computed from the diamond's split vertex and the proper encoding. There is one set of encodings for each of the 26 types of diamonds.

Encoding the mesh in this manner allows us to quickly compute the (i, j, k) index of a diamond's parents and children. Instead of storing pointers to the parents and children of a diamond, we store all of the diamonds in a hash table and use the (i, j, k) index of the split vertex to locate a diamond. In the case of a phase 2 diamond, this saves twelve pointers (4 parents, 8 children) or 48 bytes per diamond. Since each (i, j, k) index corresponds to a data point, we can quickly compute indices for the vertices of a tetrahedron and use them to get the data values needed to extract the isosurface.

4.8 Data Structures

The split and merge queues are implemented as hash tables using a fixed number of buckets and chaining to handle collisions. Each bucket corresponds to a range of the projected screen space error as measured in pixels. Each entry in the bucket corresponds to a diamond whose screen space error falls within the bucket's range. The buckets are not sorted internally by error value. Hash tables can be used instead of priority queues because it is not necessary to split the diamond in the split queue with the highest error, or to merge the diamond in the merge queue with the lowest error. Instead it is sufficient to split a diamond whose error is greater than the current tolerance and to

merge a diamond whose error is less than the current tolerance. Hash tables with $O(1)$ operations provide better performance than a priority queue with $O(\log n)$ operations. A separate hash table, the queue hash table, is used to map diamond indices to their entries in the queue. There is one hash table for the split queue and one hash table for the merge queue. This second hash table is necessary because the split and merge queues are ordered by view-dependent error. In order to quickly locate a specific diamond in either queue, we need to be able to access the queue based upon the (i, j, k) index of the diamond. Accessing the diamonds in the queues based on view-dependent error would require computing the view-dependent error, locating the bucket that the diamond is in, and then traversing the bucket to get the appropriate entry.

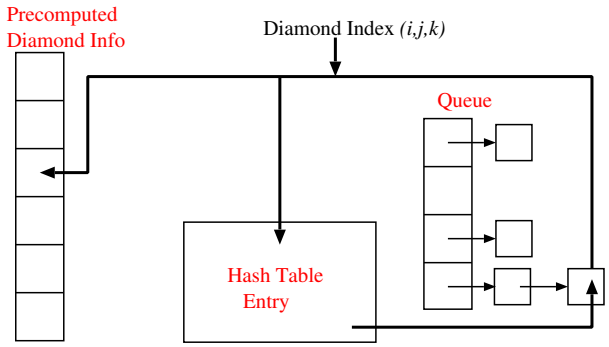


Figure 4.10: Relationship between precomputed data, queue entries, and queue hash table.

The data structures are illustrated in Figure 4.10. The hash table maps a diamond index to an entry in the queue. The diamond index associated with the queue entry maps back to the precomputed diamond information and the same hash table entry. When a tetrahedron is added or removed from the mesh, its diamond's index is used to locate the corresponding entry in the split queue via the split queue's hash table. Each diamond in the split queue contains flags indicating which of its tetrahedra are actually in the current mesh. The reason for these flags is shown in Figure 4.5. Each bucket entry in the split and merge queues stores the diamond's level, (i, j, k)

index, isosurface and view-dependent errors, and *invisible* and *empty* bits.

When a tetrahedron is added to the split queue, the isosurface passing through it is computed and stored in the *geometry cache*. The geometry is cached in an array so that it is in a contiguous region of memory. New geometry is appended to the end of the array. Geometry is removed from the cache by replacing the removed geometry with geometry at the end of the array. A hash table is used to map a diamond to the geometry cache entries associated with its tetrahedra. This caching method duplicates normals and vertices along edges. Its advantage is that it has better memory coherence than hash table based caches which store the vertices and normals on a per-edge basis (see Gerstner and Rumpf [17]). On newer graphics hardware, storing the geometry in a contiguous region of memory allows one to stream the data from memory to the graphics card for improved rendering performance. The mesh is drawn by traversing the geometry cache.

4.9 Subtrees

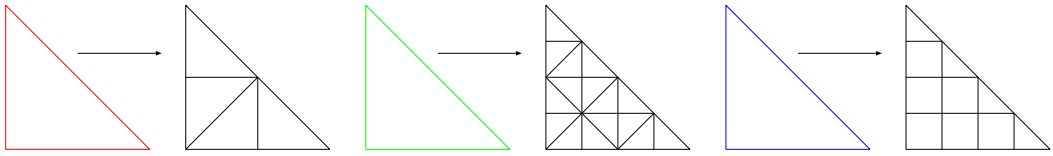


Figure 4.11: Division of a triangle into different subtrees. Left to right: Subtree of depth 1, subtree of depth 2, and subtree using triangular and quadrilateral elements that maintains the same boundary refinement as the depth 2 subtree.

In order to reduce the storage cost of the precomputed data and the granularity of the refinement process, we implicitly replace a single coarse tetrahedron T with a set of smaller tetrahedra S_T called a subtree. Figure 4.11 shows two-dimensional examples of a triangle divided into subtrees of depths one and two. The use of subtrees is similar to performing several refinements at once. The difference is that instead of performing the refinements by using the split and merge queues, we

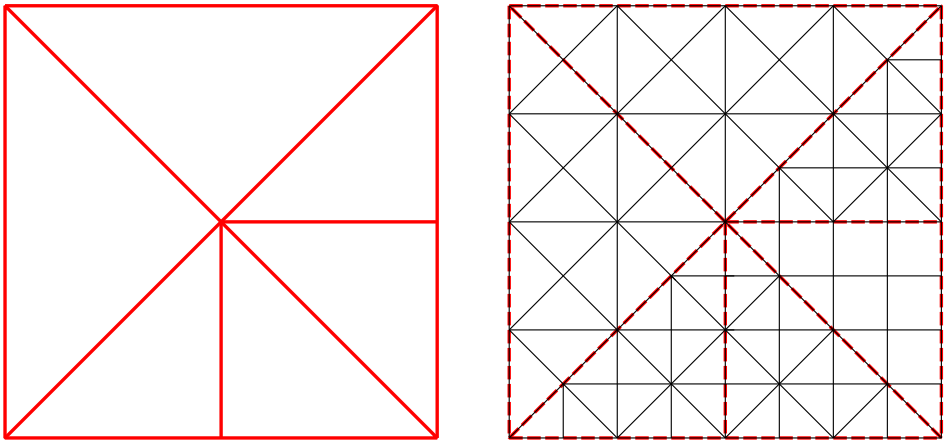


Figure 4.12: Two-dimensional example of subtrees in an adaptively refined mesh. A coarse adaptively refined mesh's elements are replaced with subtrees. Coarse triangles are implicitly replaced with a set of finer elements, and contouring is performed on the finer mesh. Triangular and quadrilateral elements are used in the subtrees, but because the boundaries are consistent there will be no cracks in the contour.

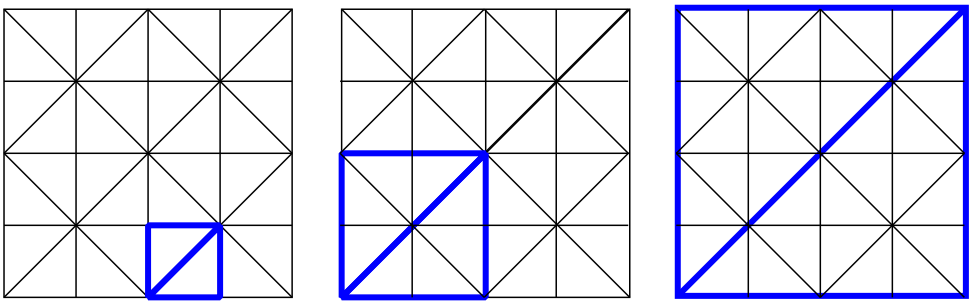


Figure 4.13: Left to right: Leaf nodes for subtrees of depth zero, one, two for a two-dimensional 5x5 grid. The bold lines indicate the leaf diamonds which are at levels two, one, and zero respectively.

precompute the set of tetrahedra added to the mesh by these extra refinements and implicitly replace the coarse tetrahedron with this set of finer tetrahedra. Each tetrahedron in the mesh now represents an aggregate set of tetrahedra called a *chunk*. This is similar to aggregate triangle structures or chunks used to improve performance in terrain rendering applications such as those by Levenberg [41], Cignoni et al. [4] and Pomeranz [60]. T's min and max values remain the same, and T's error value is the maximum of its subtree's tetrahedras' errors. The error value is given as:

$$e_T \quad = \quad \max(e_t) \;\; \forall\, t \in S_T .$$

In general, as shown on the right in Figure 4.11, a subtree does not have to be composed of triangles or tetrahedra. It can included any element type, hexahedra, prisms etc., and any algorithm can be used to extract the isosurface from the subtree elements as long as there is a consistent tesselation of the boundary edges and faces. Figure 4.12 shows an adaptively refined mesh before and after the use of subtrees. Two different types of subtrees are used but because the boundaries are consistent there will be no cracks in the contour. One triangle on the lower right has a subtree composed of six quadrilaterals and four triangles, and the other triangles have subtrees composed of 16 triangles. The isosurface is extracted from the elements in S_T instead of T itself.

The size of a subtree is 2^{3d} where d is called the depth of the subtree. A depth of two divides one tet into 64 tets. In general, the subtrees do not need to be a power of two as long as adjacent edges and faces have the same tessellation. These more general cases are not considered here. The creation of adaptive, consistent subtrees with varying edge tesselations and is described for progressive terrain applications by Pomeranz [60]. The division of tetrahedra into subtrees occurs at all levels of the mesh. The finest or leaf level of the mesh is the one with the smallest tetrahedra. For a $(2^k+1)^3$ mesh (i.e. a 2^{3k} volume) the leaf diamonds are at level k.

The division of the mesh into subtrees has several important performance benefits. Subtrees reduce the size of the mesh for which data is precomputed by a factor of 2^{3d}. Figure 4.13 shows how the leaf nodes move to coarser and coarser levels of the mesh as the depth of the subtrees increases. Tetrahedra at finer (smaller) levels than the leaf nodes cannot be divided into a subtree because all of the data points at the vertices introduced by the subtree do not exist. Thus, the error, min, and max values do not need to be computed and stored for the diamonds that exist at finer levels than the leaf diamonds. For a 1024^3 volume only 256^3 errors, min, and max values need to be computed. Subtrees reduce the storage cost of the precomputed errors, min, and max values (Section 4.3) by a factor of 2^{3d}. Furthermore, subtrees reduce the granularity of the mesh

refinement, the number of splits and merges that need to be performed, and the size of the runtime data structures. Since once tetrahedron actually represents 64 smaller tetrahedra, its error is smaller and thus larger changes in the error bound and viewing parameters must occur before it it removed from the mesh. This reduced granularity also means that a tetrahedron tends to remain in the mesh over several time steps.

The use of fixed-size subtrees makes it possible to easily cache the extracted triangles and gradients in vertex arrays for improved rendering performance. A subtree of depth two has 64 tetrahedra with at most 128 triangles and 92 vertices. With these fixed sizes, we allocate fixed-size chunks of triangles and vertices, and use lookup tables to quickly place the extracted vertices and triangles into vertex arrays. These chunks of triangles are cached in AGP memory and can be rendered extremely quickly using graphics hardware [41].

The disadvantage of using subtrees is that it increases the number of tetrahedra in the mesh for a given error bound. Since a coarse tetrahedron is replaced with a set of finer tetrahedra, its single error value is really the range of error values as defined by its subtree elements. Thus, for a given error bound E, it is possible to add tetrahedra into the mesh whose error is less than E. This results in the contouring algorithm being run on more tetrahedra than necessary and also being run on more tetrahedra that do not contain the isosurface.

At finer resolutions (i.e. lower errors), there is a tradeoff between the smaller runtime data structures and reduced number of splits and merges given by larger subtrees and the reduced number of tetrahedra contoured by smaller subtrees. Table 4.1 shows the effect of subtree depth on the number of tetrahedra contoured and the number of tetrahedra actually respresented in the mesh. The *Tetrahedra* and *Diamonds* rows indicate the number of tetrahedra and diamonds in the split queue. The *Contoured* row indicates the number of tetrahedra contoured which is the number of tetrahedra in the split queue multiplied by the subtree size. The *With Surface* row is the number of

Error	1.0	0.7	0.5	0.3	0.2	0.1
Tetrahedra(2)	51612	51612	220176	220176	844886	963786
Diamonds	23879	23879	100409	100409	409287	431033
Contoured	3303168	3303168	14091264	14091264	54072704	61682304
With Surface	236452	236452	1037746	1037746	3995653	4389896
Tetrahedra(1)	51612	51612	220176	220176	978916	3771636
Diamonds	23879	23879	100409	100409	433567	1771639
Contoured	412896	412896	1761408	1761408	7831328	30173088
With Surface	47220	47220	236452	236452	1037746	3938786
Tetrahedra(0)	51612	51612	220176	220176	978916	4388324
Diamonds	23879	23879	100409	100409	433567	1853489
Contoured	51612	51612	220176	220176	978916	4388324
With Surface	8094	8094	47220	47220	236452	1037746

Table 4.1: Table showing the number of tetrahedra, diamonds, contoured elements and elements containing the isosurface for the monkey lung dataset in Figure 1.4. Top to bottom: data for subtree depths of 2, 1, and 0 with 64, 8, and 1 tetrahedra respectively. The error value refers to the isosurface approximation error not a view-dependent error (Section 4.6).

contoured tetrahedra that actually contained the surface. With larger subtrees, fewer tetrahedra and diamonds need to be represented in the runtime data structures making them smaller and quicker to search. This is most important when an isosurface is viewed at a high resolution. For example at error value 0.1, the mesh with depth 2 subtrees stores 431033 diamonds whereas the mesh with depth 0 subtrees must store 1853489 diamonds. The downside is that a lot more tetrahedra that do not contain the surface are contoured resulting in wasted computation and excess data being loaded from disk. Similar tradeoffs occur when an isosurface with a specific triangle count is desired or when a target frame rate must be maintained. In practice, subtrees of depth 1 and 2 (i.e. 8 or 64 tetrahedra) appear to work well.

4.10 Occlusion Culling

4.10.1 Previous Work

Another problem with contouring massive volumes is that the extracted isosurfaces contain millions of triangles and almost always have a depth complexity greater than one. This means that a pixel on the screen can have a large number of fragments rendered to it even though many of those fragments are invisible. In general, a large number of occluded triangles, triangles within the view frustum that do not affect the final image, are extracted and rendered. When viewing an isosurface at a high resolution, extracting and rendering a large number of occluded triangles consumes a large amount of processing, memory, and rendering resources. Occlusion culling can prevent the invisible triangles from being extracted and rendered, thus saving memory and rendering time. This allows valuable system resources to be allocated to the visible regions of the surface, and it is essential for maintaining interactivity during isosurface extraction from massive datasets.

A large amount of work has been done on occlusion for polygonal models. Software based occlusion methods precompute a visibility database to help determine occluded areas. In [10], the dataset is divided against a grid and a solidity value is determine for each cell. A cell's visibility is based on the solidity values of the cells that intersect the line segment between the view point and the cell's center. In [77], a hierarchical occlusion map is used to select occluders from a database. Occluders are rendered as white polygons on a black background. A estimated depth buffer is constructed to allow occluded regions to be detected. Modern graphics hardware has the ability to perform occlusion queries and to report the number of pixels affected by some geometry. This new feature has been used extensively for occlusion culling in large polygonal environments, see [20, 76, 30, 26].

Occlusion culling methods have also been extended to volume datasets for isosurface

extraction and volume rendering applications. Livnat and Hansen [51] traverse an octree decomposition of the dataset front-to-back and use a hierarchical visibility test based on coverage masks to determine occluded regions. As occluded regions are determined during the traversal, blocks of the volume are culled and no isosurface is extracted from them. In [75], the dataset is decomposed into a set of blocks, and ray casting is used to select a group of blocks that are occluders which are then rendered to create a mask of covered screen pixels. The remaining unoccluded blocks, as determined by this mask, are rendered. Recent work such as [42] and [12] has extended occlusion culling techniques to volume rendering applications.

Our occlusion culling algorithm is similar to the algorithm of Livnat and Hansen [51]. We replace their octree with our tetrahedral mesh hierarchy based on diamonds (Chapter 2). Since the diamond hierarchy does not allow for front-to-back rendering, we utilize frame-to-frame coherence between successive view-points to create an approximate depth buffer against which occlusion tests are performed. The software occlusion tests are replaced with hardware occlusion queries, and the mesh structure is dynamically refined and coarsened as regions become visible and occluded.

4.10.2 Hardware Accelerated Occlusion Culling

Occlusion queries on modern graphics hardware allow one to render geometry and determine the number of samples that pass the depth and stencil tests. This indicates the number of screen pixels affected by the rendered geometry. By disabling writes to the depth and color buffers, one can determine the number of screen pixels that would have been modified by the given geometry had it actually been rendered. Since the results must be read back from the graphics hardware, there is a noticeable delay between the time when the query is issued and the time when the results are available. The key to efficient hardware occlusion queries is to fill this time gap with useful computations that can be used later on. In order to minimize this delay, we issue several queries and

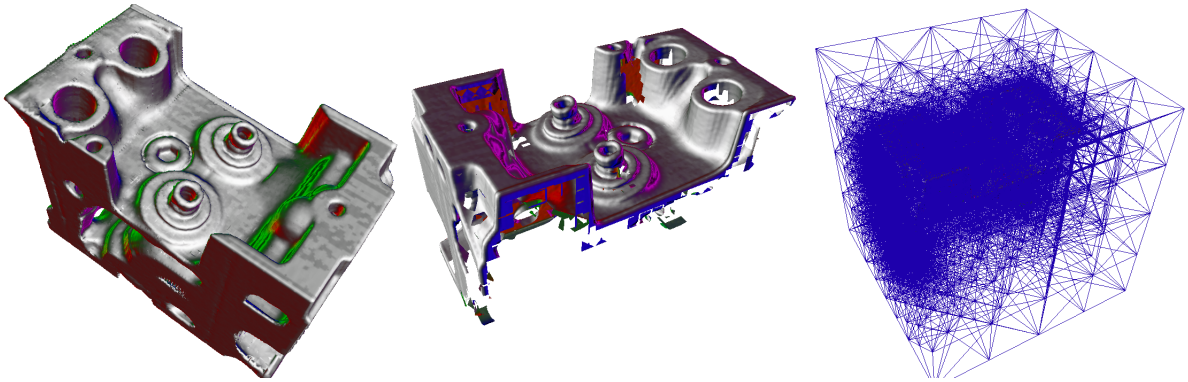


Figure 4.14: Left: high resolution rendering of the engine dataset, Isovalue = 100, Error = 0.6. The mesh contains 1.23 million triangles. Middle: Backside view showing the parts of the dataset removed by occlusion culling; the mesh contains 680K triangles. Right: The tetrahedral mesh showing that the occluded areas (lower right) are at a lower resolution.

perform some speculative error calculations before reading the results of the first query.

Occlusion culling is performed on the diamonds which drive the mesh refinement and coarsening process. The advantage of using diamonds instead of tetrahedra for occlusion culling is that diamonds, being the unit of refinement in our hierarchy, allow us to quickly eliminate whole regions of refinement with a single occlusion test. Unlike a tetrahedron, a diamond's children are not contained within its convex hull. This means that the occlusion queries cannot be performed top-down using the parent-child relationship between diamonds. Various nesting relationships do exist between a diamond and coarser diamonds. For example, a nesting relationship exists between a diamond and its great-grandparent. Given a diamond D, the great-grandparent is a coarser diamond of the same phase as D that when subdivided three times will create D. The great-grandparent's split vertex is the second vertex (SV_1) along $D's$ split edge. Performing occlusion culling on the diamonds allows the mesh to be coarsened enough so that the number of occlusion tests performed is minimized.

Our occlusion culling algorithm works as follows:

1. At the start of a frame f_i, we render the isosurface from frame f_{i-1} and then initiate an

occlusion query for all diamonds in the split and merge queues (Chapter 2) within the current view frustum that were occluded in frame f_{i-1}. This is done by rendering the diamond's bounding box. Visible diamonds are allowed to remain visible for several frames before occlusion queries are performed on them. Queries are performed on occluded diamonds every frame.

2. After all of the queries have been issued, we read back the results of the occlusion queries in the order that they were executed and recompute the error for diamonds in the split and merge queues. Occluded diamonds are given an error of zero. Thus, occluded diamonds in the split queue are never split, and occluded diamonds in the merge queue can be merged as necessary to simplify the mesh.

3. To exploit the frame-to-frame coherence present in view-dependent refinement, the occlusion queries in frame f_i are performed against the depth buffer created when the isosurface from frame f_{i-1} is rendered from f_i's view point. This provides a good initial guess for the occluders and gives the correct depth buffer for those occluders.

4.10.3 Changing Isovalues

When the isovalue is changed (Section 4.4.1), the currently extracted isosurface can no longer be used as a valid set of occluders against which occlusion queries can be performed. This is because the new isosurface can be dramatically different from the previous one. When the isovalue is changed by the user, all triangles for the diamonds in the split queue and outstanding occlusion queries for the diamonds in the split and merge queues are invalidated. All diamonds in the split and merge queues are marked as visible and the new isosurface is extracted from the diamonds in the split queue. The depth buffer given by the new isosurface can now be used for occlusion queries, and the refinement continues as described above in Section 4.10. As new error values are computed

and new occluded regions are discovered, the mesh structure refines around the visible region of the new isosurface and coarsens everywhere else.

4.11 Contouring Compressed Volumes

4.11.1 Overview

The visualization of massive volume datasets is made difficult by the fact that the entire dataset cannot fit into a computer's core memory. In addition, disk storage speed has been growing at a much slower rate when compared with the speed of CPUs, GPUs and direct memory transfers. Thus, visualization algorithms need to effectively organize data on disk so that it can be paged in when needed. Data compression techniques are essential to overcoming this performance gap; they reduce the number of times data must be loaded from slower disks, and they increase the amount of data that can be read at a time. In order to be effective for interactive applications, decompression must be fast and localized to the region of interest. As CPUs and GPUs improve in performance, the cost of decompression becomes much smaller than the cost of reading massive, uncompressed data from disk. Thus, regions of the compressed volume can be prefetched and cached in memory until they are needed for visualization.

Hierarchical visualization, which allows regions of the dataset to be accessed separately from others and at lower resolutions if necessary, is a very effective technique for visualizing large volumes. This is especially true when subregions of the dataset are being visualized and when lower resolution versions of the dataset may be acceptable for visualization, e.g. when interactive frame rates can be maintained. Guthe and Strasser [25] utilize wavelet based compression to adaptively render volume datasets using texture hardware. An octree decomposition breaks the volume into a hierarchy of 2^{3k} blocks and linear spline wavelets are used to transform the data. Arithmetic

and Huffman coding are used to compress the wavelet coefficients. Schneider and Westermann [72] utilize a hierarchical vector quantization scheme for compressing static and time-varying volumes. Their algorithm could also be used for hierarchical visualization and it has the advantage that rendering can occur directly from the compressed representation removing the overhead of decompression. Their algorithm also uses an octree-based (cubical) decomposition scheme to divide the dataset.

Our data compression algorithm uses a tetrahedral mesh hierarchy defined by longest-edge bisection. The cubical blocks of an octree decomposition, used to define the data prediction algorithms, are replaced with the diamond shapes developed in Gregorski et al. [23] and used by Linsen et al. [47] for hierarchical representation of large dataset. We use the top-down traversal of the mesh hierarchy to predict data values and gradient components for lossless compression of large volume datasets. This prediction scheme follows the data access pattern dictated by the mesh refinement. Combined with the z-order data layout scheme of Pascucci et al. [58], it ensures that the reconstruction masks are of minimal size and that data values necessary for reconstruction can be accessed in a memory and disk coherent manner. Furthermore, it ensures that the decompression can be confined to local regions of the volume which is essential for visualization of large datasets. The delta values used for runtime reconstruction are compressed in chunks and stored in z-order. The levels of our hierarchy are created from subsampled versions of the dataset. Thus, as finer (higher resolution) levels of the volume are reconstructed, values at coarser levels do not need to be updated (as they must be when wavelet lifting schemes are used).

Many types of data cannot be compressed easily because the number of different data values present is high and the corresponding redundancy is low. For better compression, data is often transformed from its regular domain to another domain in order to increase its redundancy. Data transforms include the FFT, DCT [19], and the well-known families of wavelet transforms

[21, 66].

Figure 4.15 shows a block diagram of the general data compression process. In the *transform* stage, the data is transformed to another representation to decrease its entropy for better compression. In the *modeling* stage, the data is scanned to collect information on its contents. This information is used in the *code generation* stage to generate codewords which are used in the *encoding* stage to compress the data. Not all coders may follow this process exactly. In particular the modeling stage often can be integrated with the transformation stage, and in adaptive coders the coding process repeatedly loops through the modeling, code generation, and encoding stages.

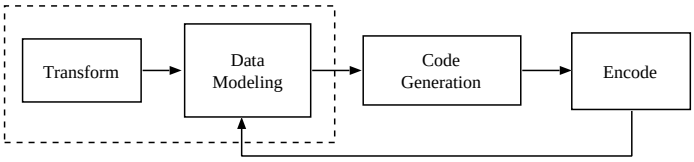


Figure 4.15: Flow diagram of the basic encoding process.

4.11.2 Compression and Decompression

Since we are visualizing data that has been processed and encoded, we focus on simplifying and streamlining the decoding process. Because the datasets have already been quantized, we are interested only in lossless compression. We choose to avoid more sophisticated methods and use the minimum number of operations needed, keeping the operations used to the most basic; thus a simple direct table lookup scheme is used. In a table lookup scheme, assuming there are N possible input symbols, a prefix–free codeword is assigned to each symbol (often using the Huffman algorithm [34]) and an encoding lookup table (LUT) is created with N entries, each entry giving a symbol–codeword association. The encoding process takes an input symbol, uses the symbol as an index into the LUT, and the code given in the LUT entry is written to output. Decoding is similar;

assuming the longest codeword is L bits long, a decoding LUT is created with 2^L entries, each entry giving a codeword–symbol association. Then the encoded bits are read in blocks of size L, and those L bits are used as an index into the LUT, which gives the associated symbol and the number of those L bits actually used. One problem with using a direct table lookup scheme is that the decoding LUT must be a size that is a power of 2 of the longest code length. In the worst case the longest code is equal to the number of possible input symbols minus 1. For a difference transform performed on 8–bit values, there are 512 possible transform coefficient magnitudes, and a possible worst–case code length of 511 bits, although in practice this code length is not likely to be reached. Even if the longest code length is of a more "reasonable" length, it may be that the decode table is larger than desired. This is especially important if memory is tight or performance is critical. Another problem is that the longest code length may be longer than a standard word size (32 bits on most common architectures). Handling these longer code lengths requires extra operations that slow down the encoding and decoding processes. In order to have small tables for fast decoding, length–limited codes must be used. This can be accomplished using codes generated by the Package–Merge algorithm [38] and its variants [43, 70], or by reducing the number of possible input symbols. The algorithm presented here follows the latter approach, and reduces the number of possible input symbols.

When examining a set of transform coefficient magnitudes in binary representation, most of the redundancy in a coefficient lies in the position of its leading 1. Given a 9–bit transform coefficient, such as 000001010, the leading 000001 compresses rather well, and the following 010 does not. Lead–1 encoding is utilized to take advantage of this. Instead of encoding all 512 possible transform coefficient magnitudes, only the position of the magnitude's leading 1 is encoded. This reduces the total number of input symbols that must be coded from 512 to 10 (nine locations for the leading 1 and the zero magnitude). The codewords for the leading 1 positions are generated using the Huffman algorithm. Since there are only 10 symbols to encode, a codeword can be at most nine

bits long and the decoding table can have at most 512 (2^9) entries.

To use Lead–1 encoding, the difference transform coefficients are treated as a signed–magnitude representation. During encoding, the transformed coefficient's magnitude is used as the index into the encoding LUT. The table entry gives the position of the coefficient's leading 1, the length of the associated codeword, and the codeword itself. The codeword is output, and if the coefficient magnitude is nonzero the sign bit is output also. Finally, if the magnitude is greater than 1 all bits following the leading 1 in the magnitude are output unchanged. The encoded data consists of the codeword for the leading 1 position, and, if the magnitude is non-zero, a sign bit and all bits following the leading 1. The decoding process is similar, and is shown in Figure 4.16. An entry in the decoding table stores the length of the codeword that indexes to that entry and the leading 1 position indicated by the code. From this information the decoder can determine the number of bits after the leading 1 and how far to shift the encoded bit stream during decompression. This information is packed into a single byte. Table 4.2 shows the Lead–1 counts for the transformed and untransformed data of a $256^3 \times 274$ time-varying dataset. After the difference transform, a much larger number of the leading 1 positions are zero, indicating that the data will compress nicely and that the decoding loop will be efficient. Transformation and compression of time-varying data are discussed in Section 5.7.

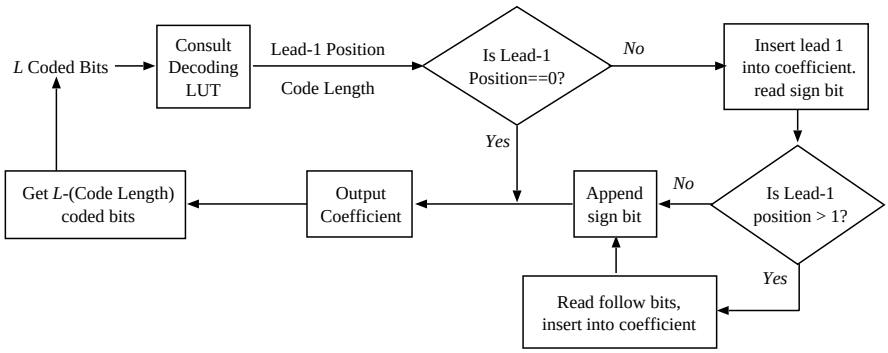


Figure 4.16: Coding loop of basic Lead–1 decoding.

Lead–1	Untransformed	Transformed
0	4743194	6093309314
1	9297339	1075445689
2	61897415	1453200739
3	4119536406	1010756643
4	408098714	880931513
5	282209288	704350384
6	412069938	479577928
7	2753591178	251551499
8	3977820400	78543126
9	0	1597037

Table 4.2: Leading 1 position counts for a $256^3 \times 274$ time-varying dataset.

4.11.3 Improving the Coding Rate

The table generation step for the basic Lead–1 coder collects the number of occurrences of each leading 1 position in the transformed data and uses those counts to create the coding tables. This method of data collection and table generation answers the following question: "given a coefficient magnitude, what is the probability that the leading 1 position will be *x*?" To improve the coding rate, the coder uses one position of context. When collecting data about the leading 1 positions, the table generator separates the Lead–1 counts based on the leading 1 position of the previous coefficient. This answers the question "given that the previous leading 1 position was *x*, what is the probability that the current one is *y*?" Using context in this manner allows us to take advantage of relationships in the data that were not visible with the simple coding method. Table 4.3 shows a selection of the conditional counts for a $256^3 \times 274$ time-varying dataset. For example, if the previous leading 1 position was a two the current position is most likely to be a zero. Instead of having a single encoding and decoding table, there is now one for each context (i.e. previous position).

	Prev leading 1		
Next leading 1	0	1	2
0	4257612902	440481133	689695448
1	649598731	71511545	112961554
2	525024182	321374553	216536125
3	232651401	109045949	176819114
4	170946802	64646151	122123553
5	125473481	39082448	76907193

Table 4.3: Conditional leading 1 position counts for a $256^3 \times 274$ time-varying dataset.

4.11.4 Design Considerations

Lead-1 encoding and Huffman codes were chosen for several reasons. Encoding and especially decoding needs to be fast. These algorithms uses direct table lookups which satisfies this criteria. To minimize the size of the encoding and decoding lookup tables, one must decrease the number of symbols that need to be encoded. The generated Huffman codes are limited in length by the number of input symbols. If there are n possible symbols to be encoded, the longest code length will be at most $n-1$ bits. Lead–1 encoding reduces the number of symbols that need to be encoded thus reducing the size of the codes and the table size. As described in Section 4.11.2, the decoding table entries are single bytes which make the decoding table very small and capable of fitting into cache memory. Since the goal is to decompress data as it is needed for contouring, a balance must be struck between compression ratio and decoding performance. Compared with arithmetic style encoders that have better compression performance, this encoding scheme using Lead–1 encoding and Huffman codes achieves decent compression with superior speed and throughput.

4.11.5 Compressing Volumes

The data compression algorithm is divided into three phases. In the first phase, the original volume is traversed along the tetrahedral mesh hierarchy, and a prediction scheme is used to build a histogram

of the differences between the actual data and the data given by the prediction. In the second phase, these delta values are passed to the encoder which builds a set of codes used to compress the delta values. In the final phase, the stream of delta values is divided into pages and compressed using the codes generated in phase two.

For multiresolution extraction of isosurfaces, it is important to use hierarchical prediction that follows the refinement of the multiresolution mesh. This ensures that the runtime decompression algorithm can efficiently locate the data needed for reconstruction. It also ensures and that decompression is localized across the space and scale of the volume hierarchy. Decompressing a region at a high resolution only occurs in a small localized space around the region and in similarly localized spaces at coarser levels of the hierarchy.

The data prediction algorithm, based on difference from linear prediction along a diamond's refinement edge, is illustrated for two-dimensional refinement in Figure 4.17. The predicted data is

$$D_p = \frac{(D_{SV_0} + D_{SV_1})}{2}. \tag{4.16}$$

The delta value encoded later is

$$\delta = D_V - D_p. \tag{4.17}$$

In three dimensions, the predicted value is computed in the same manner since the diamond whose split vertex is V always has two points on its split edge regardless of dimension. This simple predictor handles boundary conditions easily because the diamonds on the faces of the volume are always phase one and zero diamonds, and the diamonds on the edges (except for the corners which are not predicted) are phase zero diamonds (Chapter 2). Thus, there are no special cases.

Since datasets are stored in hierarchical z-order, the transformed coefficients are also stored in z-order. To allow for local decompression, the transformed coefficients are divided into

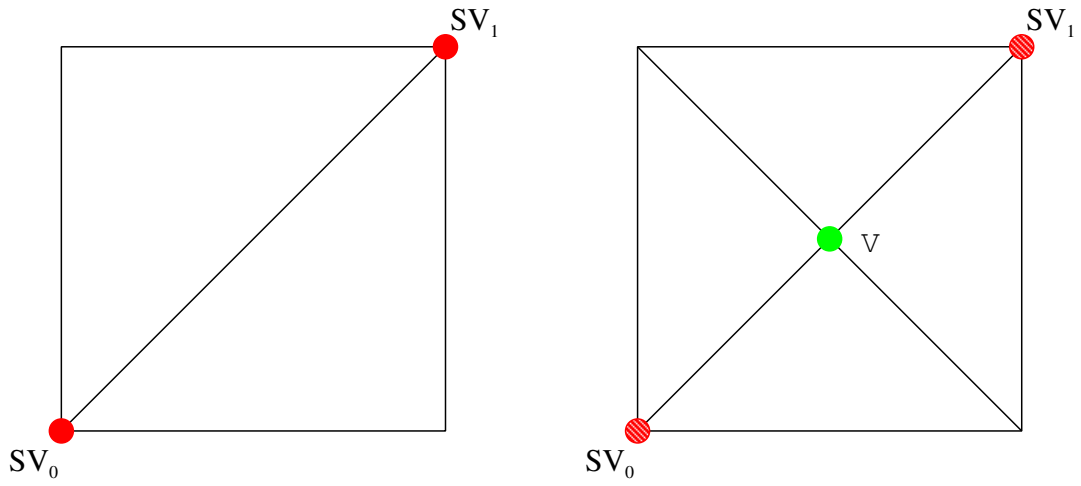


Figure 4.17: In difference from linear prediction, the value at a point V, which is also the split vertex of a diamond, is predicted as the midpoint of its two split edge values. The red, dashed circles, SV_0 and SV_1, indicate the split edge vertices.

Dataset	Uncompressed	Compressed	Data+Gradients	Data
Bucky 1024^3	4096	1334	3.071	5.51
PPM 512^3	512	284.9	1.797	2.43
XMasTree 512^3	640	378.1	1.707	4.06

Table 4.4: Compressed and Uncompressed sizes of test datasets in MB. The Uncomp and Comp columns show the uncompressed and compressed sizes of the combined data and gradients file. The Data+G column shows the compression ratio for the data and gradients combined and the Data column shows the compression ratio for the data only.

pages of 2^{12} data points and encode each page separately. A lookup table is used at runtime to find the disk offset for loading a particular page.

Given an index $P_{(i,j,k)}$ for a point P, the data value and gradient at P are reconstructed as follows:

1. Convert the (i,j,k) index $P_{(i,j,k)}$ to its z-order index P_z. Using P_z, locate the disk page DP that contains P, and decompress DP to get the delta values associated with DP's data points.

2. Since the deltas are stored in z-order, the z-order index P_z for all points in DP is known. For each point in DP, compute its (i,j,k) index $P_{(i,j,k)}$ from P_z.

3. Given that this (i, j, k) index, $P_{(i,j,k)}$, is the split vertex of a diamond, compute the (i, j, k) indices of the vertices necessary to reconstruct the value at P.

4. Fetch the surrounding values needed for reconstruction and compute the original data value. This may require recursive decompression as necessary to obtain all of the surrounding values.

The data values d_i needed to reconstruct the data at a point $P_{(i,j,k)}$ all have z-order indices z_i such that $z_i < P_z$. This means that they come from coarser levels of the mesh and from earlier positions (relative to the file offset) in the data file than P. Furthermore, z-order stores the data points first by level-of-detail and then by spatial proximity within each level-of-detail, and it has been shown to have good memory and disk coherence properties. Accessing the coarse and fine data points, from memory and disk, needed for reconstruction, exploits these coherence properties of the storage order. Compression results for test datasets are given in Table 4.4.

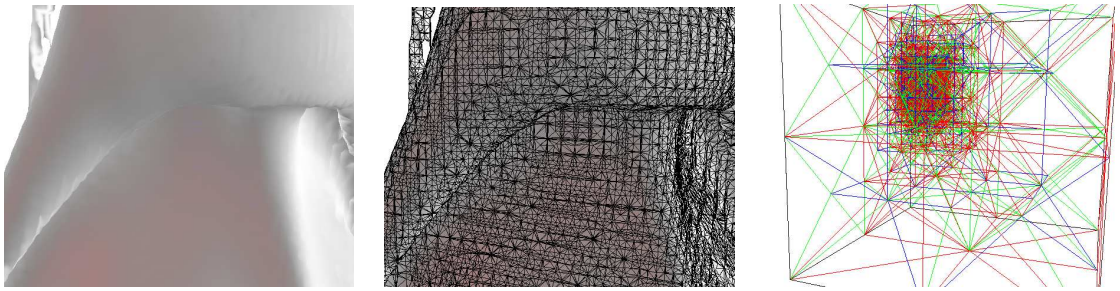


Figure 4.18: Closeup view of an isosurface feature in the mixing interface of two gases showing the texture mapped surface, underlying triangle mesh, and the adaptively refined tetrahedral mesh around the region of interest. Time step = 273, Isovalue = 206, Isosurface error = 1.5, 50K Triangles.

4.12 Results

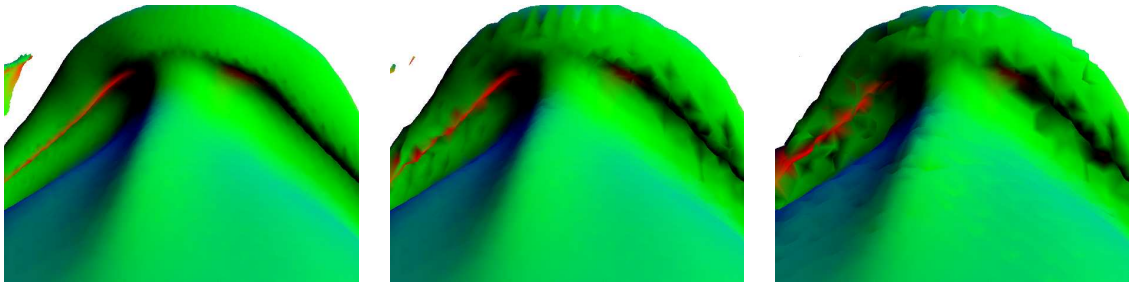


Figure 4.19: Isosurface with varying screen space error at Time Step = 273, Isovalue = 213. From left to right: Error = 0.56, 95K Triangles; Error = 1.7, 30K Triangles; Error = 2.7, 13K Triangles.

Our test machine is a 2 Ghz Pentium with 1 GB of main memory and a GeForce4 Ti 4600 graphics card. The amount of main memory available for uncompressed data was limited to the lesser of 512 MB or $(1/2)$ of the uncompressed dataset size. The buckyball dataset is a synthetic dataset made from Gaussian functions. The 1024^3 dataset was made by a $2 \times 2 \times 2$ tiling of a 512^3 dataset. The PPM (Piecewise Parabolic Method) dataset is a Richtmyer-Meshkov simulation dataset [56]. For the Christmas Tree CT dataset [35], the high-resolution version of the near isotropic and rotated dataset is used. Its initial dimensions are $512 \times 512 \times 499$, and it has been padded to make it 512^3. As stated by Lee et al. in [40], the extension and padding of the dataset comes with a small increase in entropy. All isosurfaces were extracted from the compressed volumes. Reported error values are

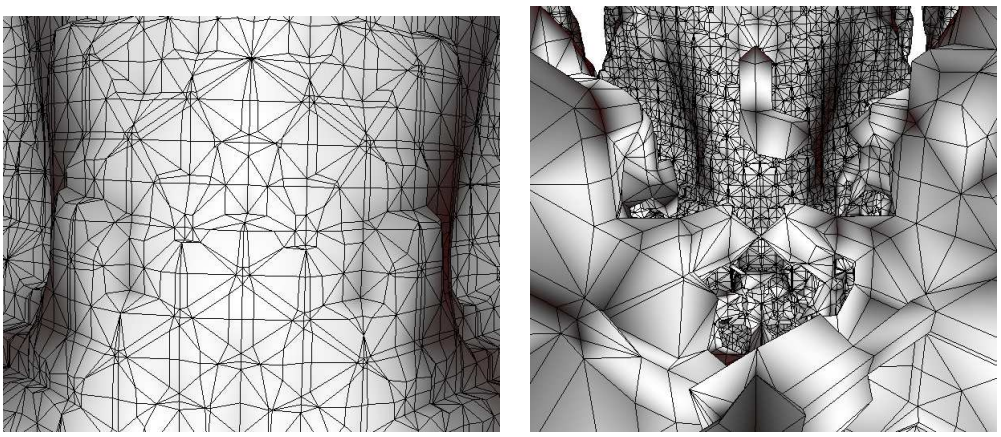


Figure 4.20: Closeup view of a mixing feature. Time Step = 273, Isovalue = 186, Isosurface error = 0.5. On the right, a zoomed out view shows the portion of the isosurface culled by the view frustum.

calculated using the metric described in Section 4.6 and indicate an approximation to the screen space deviation of the extracted isosurface from original isosurface as measured in screen pixels. Texture mapping is used to simulate diffuse lighting, highlight edges and reduce the visual artifacts present in the lower resolution surfaces.

Figure 4.18 shows a closeup view of a high-resolution isosurface from the PPM dataset. Using AGP memory and vertex arrays this surface of 50K triangles can be rendered at 15M TPS or around 30 FPS. Figure 4.19 shows an isosurface similar to the one shown in Figure 4.18 at different screen space errors. The isosurface representing the mixing interface contains a large number of topological components and small features. In the two lower resolution surfaces, the small feature in the top left is not preserved. Figure 4.20 shows the effects of view-frustum culling on the tetrahedral mesh and the resulting isosurface. In the image on the right, the surface is extracted at a lower resolution in the regions outside of the view-frustum. Normally, no isosurface is extracted in these regions, but is is extracted in this example to show how the mesh is coarser in the invisible regions.

Each dataset includes the actual data and three bytes for a quantized gradient, one byte for each component. The four values are predicted independently, and one encoding table is used for

compression. Tests using separate encoding tables for each component or separate tables for the data and gradients did not yield significant improvements in compression. Before compression, the raw floating point gradients are run through the diffusion based smoothing process described in Section 4.3.1 to ensure that there are no degenerate gradients that can cause problems for shading. Figure 4.21 shows histograms of the raw and transformed data for two test datasets. The transformed data is the set of delta values generated by the prediction step (Section 4.11.5). As the histograms show, the data transformation greatly increases the number of values with small magnitudes and reduces the number of values with large magnitudes. This indicates that the entropy on the data has been reduced by the transformation and that the transformed data should compress better than the raw data.

4.12.1 Compression

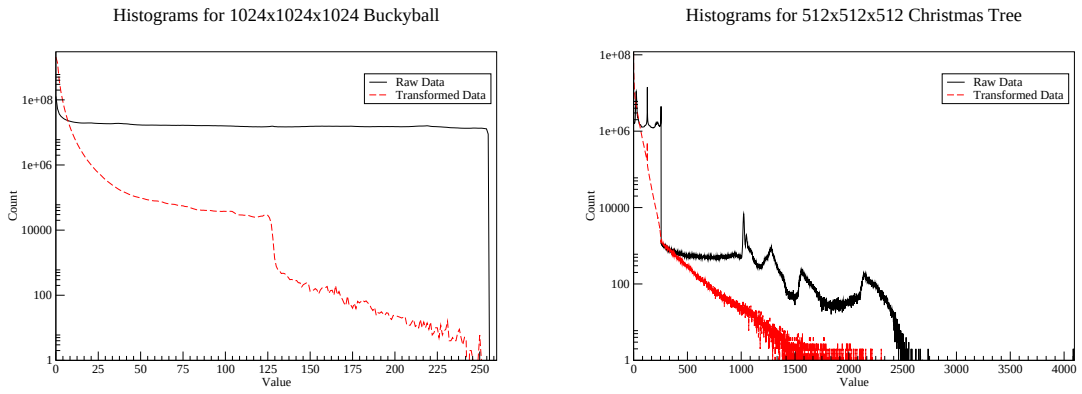


Figure 4.21: Histograms showing data values and gradient component magnitudes of raw and transformed datasets. (Gradient component magnitudes are unsigned values.) Left: Synthetic buckyball dataset. Right: Christmas Tree dataset. The buckball dataset is 8-bit data, and the Christmas Tree dataset is 12-bit data represented in 16-bit unsigned shorts. The count axis (y-axis) is on a logarithmic scale.

Table 4.4 (Section 4.11.5) summarizes the performance of our compression algorithm. For each dataset, the compression of the data and gradients as well as the compression of just the data is

shown. In general, the data compresses much better than the gradients since the gradients, even after our smoothing process, are still very noisy. For several test datasets, we are able to achieve lossless compression ratios between 2:1 and 5:1 for the data and between 1.17:1 and 2.7:1 for the gradients using the simple split-edge predictor and Lead-1 encoding with Huffman codes. To test the speed of our runtime decompression and reconstruction algorithms, measurements were recorded over several interaction sessions in which the viewing position, isovalue, and error level were all changed at some point. To measure the decompresion speed, the time to decompress a single page was recorded. This takes k input bytes, where k is the size of the compressed page, and produces 4×2^{12} deltas or 32768 bytes. This does not included the time to load data from disk. Over several test runs the decompression speed ranged from $10 - 100$ MB/sec with the average decompression speed between $50 - 70$ MB/sec. The decoder's speed is $O(n)$ where n is the number of output values. It is affected mainly by memory coherence of the input and output data arrays. The reconstruction rate was measured as the time to compute the actual data and gradients using the deltas. This includes the time to invert the z-order indices, compute the vertices needed for reconstruction and fetch their associated data and gradients. The reconstruction speed varied from $1 - 12$ MB/sec with the average rate between $8 - 10$ MB/sec. The reconstruction speed is affected by the index conversion and the cost of fetching the needed data needed. This requires two memory accesses per data point or a total of 2^{13} accesses to reconstruct one page. Furthermore, it can cause recursive decompression when needed values must be reconstructed themselves.

4.12.2 Occlusion Culling

Results for occlusion culling are summarized in Table 4.5. It shows the number of triangles, tetrahedra, and diamonds in the mesh for surfaces extracted with and without occlusion culling. It also

shows the number of occluded diamonds in the split queue and the total number of occlusion queries performed per frame. This includes queries on diamonds in the merge and split queues. Occluded diamonds exist in the split queue because of splits necessary to maintain the mesh continuity and to meet the given error threshold. However, contouring is never performed on these occluded diamonds. Our occlusion method is conservative because we allow visible regions to remain visible for several frames. Conservative occlusion culling greatly reduces the number of occlusion tests performed per frame and randomizes at which frame the queries are performed, allowing us to maintain interactive frame rates.

Figures 4.14 and 4.22 show the benefits of occlusion culling on the engine dataset and the buckball dataset. The isosurfaces are extracted at a high-resolution and thus contain a large number of occluded triangles. In most areas, the engine has a depth complexity of 2-3, and the buckball has depth complexity of 4-6. For these datasets, occlusion culling greatly reduces the number of triangles, tetrahedra, and diamonds. A reduction in triangle count yields a reduction in rendering time and storage space for the triangles. The reduction in mesh size reduces the number of error calculations that need to be performed and the number of data pages that need to be decompressed; both of which improve performance.

Figure 4.23 shows two high-resolution isosurfaces extracted from different view points and isovalues from a 512^3 chunk of the PPM dataset. These isosurfaces are much more complex than the engine and buckball isosurfaces, they have high depth complexity, and they contain a large number of very small features. As shown in Table 4.5, for the PPM dataset occlusion culling reduces the number of extracted triangles from 30-50% and the size of the mesh by 20-30%. For the Christmas Tree dataset shown in Figure 4.24, occlusion culling is not as useful because the isosurfaces contain a large number of very thin features and fewer large features which are good occluders. As the results indicate, occlusion culling is able to remove some triangles, but it is not

nearly as effective as it is for the other datasets.

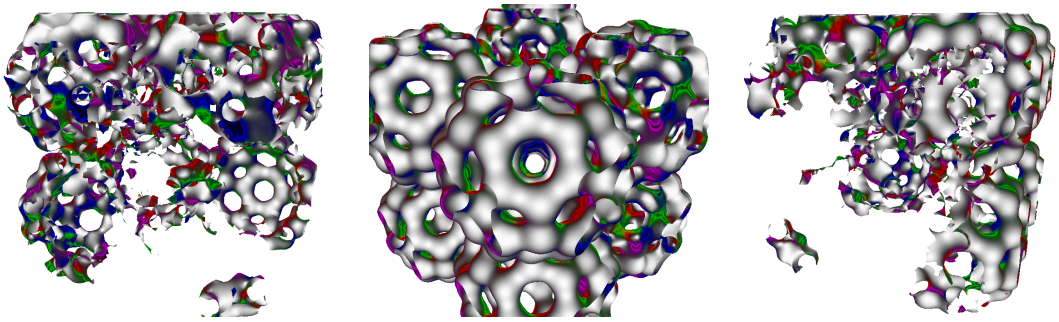


Figure 4.22: High resolution rendering of the 1024^3 synthetic buckball dataset. Left and Right: Backside views showing that the occluded areas have been removed. Middle: The surface rendered with occlusion culling, Isovalue = 104, Error = 0.71. Occlusion culling results are given in Table 4.5.

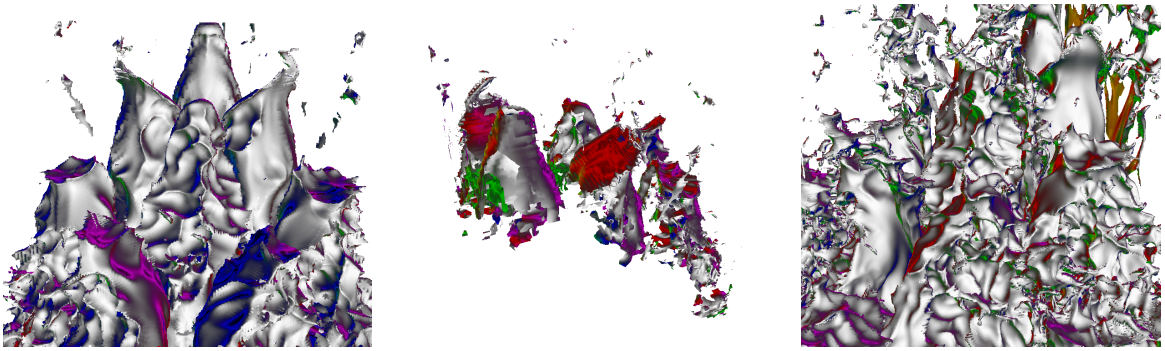


Figure 4.23: Left: High resolution isosurface from the PPM dataset. Isovalue = 228, Error = 0.78. Middle: A backside view of the left image showing the occluded regions. Right: Another isosurface from the PPM dataset. Isovalue = 200, Error = 0.638. Results with and without occlusion culling are given in Table 4.5.

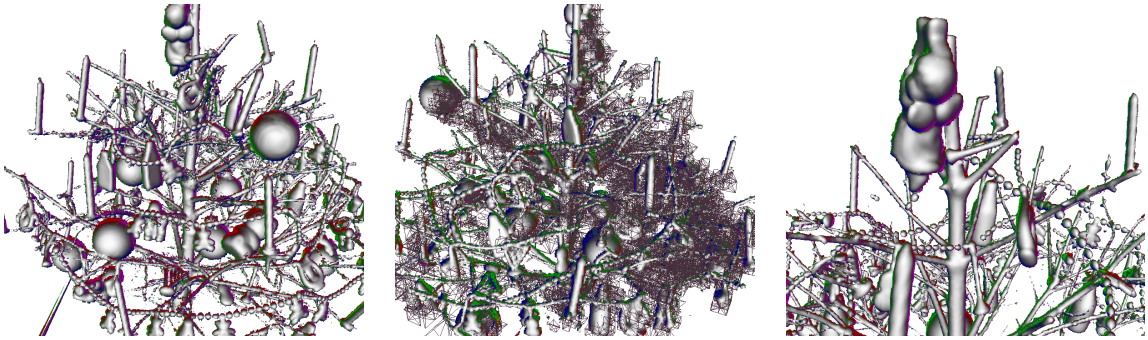


Figure 4.24: Isosurfaces from the Christmas Tree dataset. The large number of thin features in the surface, caused by the needles, branches, and tinsel, make occlusion culling hard for these isosurfaces. Left: Isovalue = 175, Error = 0.50. Middle: A backside view of the left image showing the occluded diamonds in the split queue. This indicates where occluded triangles have been removed. Right: Isovalue = 300, Error = 0.496. Results with and without occlusion culling are given in Table 4.5.

Dataset	Triangles	Tetrahedra	Diamonds	Occluded Diamonds	Occlusion Tests Per Frame
PPM (Left)	2.5M	306K	128K	–	–
	1.02M	206K	96K	16K	10-12K
PPM (Right)	2.9M	400K	175K	–	–
	2.0M	312K	143K	22K	16-17K
XMasTree (Left)	1.6M	322K	153K	–	–
	1.45M	309	146K	5K	8-10K
XMasTree (Right)	1.06M	246K	124K	–	–
	893K	234K	119K	7K	7-9K
Buckyball	2.24M	254K	102K	–	–
	1.23M	183K	82K	17K	10-14K

Table 4.5: Occlusion culling performance for various test datasets. Top row shows values without occlusion culling and the bottom row shows values with occlusion culling. For the PPM and Christmas Tree dataset, left and right refer to the two surfaces in Figures 4.23 and 4.24.

Chapter 5

Contouring Time-varying Data

5.1 Introduction

High-performance computing on large shared memory machines and PC clusters makes it possible to study complex problems through large scale numerical simulations. These simulations, of such things as weather patterns, fluid dynamics, and material deformations, are being carried out on larger scales and are producing larger datasets that need to be visualized. An example is the simulation of a Richtmyer-Meshkov instability in a shock tube experiment [56] conducted at Lawrence Livermore National Laboratory. The output of the simulation is a series of Brick-of-byte (BOB) volumes that measure entropy. Each output file is divided into an $8 \times 8 \times 15$ grid of $256 \times 256 \times 128$ bricks. The resolution of a single time step is $2048 \times 2048 \times 1920$ or about 7.7 GB. The simulation was run for 27,000 time steps, and the output consists of 274 BOB files for a total data size of about 2.1 TB. A full resolution isosurface of the mixing interface for a single time step produces a mesh with several hundred million triangles. The visualization of a single time step presents a significant challenge because of the size and topological complexity of the surfaces. Since these problems are time-varying in nature, a true understanding can only come from a time-varying visualization.

Visualization of these datasets requires techniques that exploit view-dependent, hierarchical and out-of-core algorithms to minimize the amount of runtime computation and maximize the use of available resources.

The algorithm presented here for visualizing time-varying data utilizes the diamond-based, crack-free refinement strategy presented in [23]. The mesh refinement scheme has fast coarsening and refinement operations necessary for adaptive, view-dependent refinement. To improve the efficiency of the mesh refinement and the isosurface visualization, a tetrahedron in the hierarchy is represented as a set of tetrahedra called a subtree. Subtrees, as described in Section 4.9, reduce the granularity of the refinement, allow for efficient temporal caching, and fast rendering from vertex arrays. Our data storage scheme, Section 5.5, aligns the data with the access pattern indicated by the mesh refinement. It exploits spatial and temporal locality of reference, and allows one to progressively extract isosurfaces for single time steps and to quickly extract new surfaces as the time step changes. This storage scheme is coupled with a time-varying compression algorithm that allows the data to be progressively decoded over time and only in the regions needed for the isosurface visualization.

The input is a series of volume datasets that are called *timesteps*. We assume the intervals between timesteps are constant. At runtime, given an isovalue and a timestep, we adapt the previous timestep's mesh in the visible, high error regions that contain the isosurface. This refinement strategy allows one to move through the timesteps at a coarse resolution to get a general idea of how a surface changes over time or to move through the timesteps at a high resolution and closely inspect the surface at each time step.

5.2 Previous Work

Techniques for visualizing time-varying datasets focus on exploiting spatial and temporal locality. Spatial techniques reduce the time to locate the regions needed for visualization and the amount of extra storage needed for these search structures. The Temporal Hierarchical Index Tree [63] exploits temporal coherence by adaptively joining cells whose extreme values do not change much over time. A tree structure is used to quickly locate the regions of the dataset that contain the isosurface. The approximation of actual extreme values with extreme values over time can cause regions that do not contain the isosurface to be loaded. Once these regions are found, the actual values for the indicated time step are used to extract the isosurface. In [67, 68] Sutton et al. extend the Branch-on-Need Octree(BONO) [73] to a Temporal Branch-on-Need Octree for time-varying isosurface extraction. They build a BONO for each time step of the dataset and separately store the extreme values and data values for each time step. Instead of exploiting temporal coherence, they minimize the I/O bottleneck, associated with reading in a new time step for isosurfacing, by dividing the dataset into bricks for better spatial coherence. The Time-Space Partitioning Tree [64] extends the Branch-on-Need Octree [73] to take advantage of spatial and temporal coherence. The nodes in the octree are binary time trees which measure the spatial and temporal coherence of the region over time. The structure exploits spatial coherence by approximating regions with coarser volumes, and it exploits temporal coherence by reusing rendered images between time steps. The data structure is adapted for out-of-core visualization by dividing the dataset into bricks and utilizing a demand paging system [6].

Compression based techniques utilize coherence over time to reduce the size of the datasets for faster disk loads and rendering. Westermann [71] separately encodes each time step using a wavelet transform, and uses the Lipschitz exponents to detect regions with low and high temporal variation. This method allows the volume to be decompressed locally in regions of interest

for multiresolution and progressive rendering. In [24], Guthe and Strasser use wavelet transforms, run-length encoding, and arithmetic coding to compress time-varying volume datasets. Individual volumes are encoded using a wavelet transform and compressed using run-length and arithmetic coding. A sequence of compressed volumes is further compressed using MPEG-style encoding and compression. Unlike [71] their technique decompresses the whole volume at each time step. Lum et al. [53] use a DCT and Lloyd-Max quantization to compress time-varying volumes. A group of values over time is first transformed and then compressed using a lossy quantization process. Rendering is performed directly from the compressed volumes allowing interactive rendering and manipulation of colormaps.

Our algorithm for time-varying isosurface extraction extracts an arbitrary isosurface at various levels of detail based upon user specified criteria. Additionally we are only interested in moving forwards and backwards in the temporal domain in discreet steps to *play* the volume from start to finish. Volumetric methods for time-varying isosurface extraction [67, 68, 63] can extract arbitrary isosurfaces, but because they do not incorporate level-of-detail approximations, they must extract the surface from the finest level cells in the volume even when a coarser approximation can be used (i.e view-dependent or other error-based rendering).

Our algorithm combines a hierarchical, volumetric data structure, an adaptive refinement strategy, and a cache coherent data ordering to perform time-varying isosurface extraction. This allows us to extract arbitrary time-varying isosurfaces and to adjust the resolution of the surface based upon user specified criteria. The extraction of new isosurfaces is accelerated by vertex programs on modern graphics hardware to perform the interpolation of positions and gradients.

5.3 Preprocessing

As described in Section 4.3, in a preprocessing phase, the following information is computed for each timestep:

1. The isosurface approximation error, minimum data value, and maximum data value of the region enclosed by each diamond (Chapter 2) in the mesh.

2. The normalized gradient vector at each data point.

For our time-varying datasets, the data values are bytes, and the gradients are normalized and quantized in 16 bits. For data compression purposes, we need a gradient representation that has temporal coherence and a meaningful delta between consecutive values over time. To meet these requirements, the gradients are quantized component-wise with eight bits for the x-component, seven for the y-component, and one bit for the z-component's sign. This 16 bit gradient is split into two 8 bit values; the first being the x-component and the second being the y-component with the sign of the z-component as the least significant bit. This format gives high accuracy to two components and less accuracy to the third. The max x, y, and z component errors are 0.004, 0.008, and 0.13. Higher z-component errors occurs when it is near 0. The maximum error in our quantized gradients, as measured by the deviation of the dot product between the original and quantized vectors, is 0.012 (i.e. the dot product is 0.988).

5.3.1 Gradient Processing for Time-varying Data

The gradient smoothing process described in Section 4.3.1 is also done for time-varying data to smooth the gradient vectors over the spatial and temporal dimensions. Ideally, the gradient smoothing would be performed in the spatial and temporal domains on a four-dimensional (i, j, k, t) grid. However this computation can be extremely time consuming given the size of our test datasets,

and so we approximate it with a three-dimensional smoothing in the spatial domain and a one-dimensional smoothing in the temporal domain.

1. For a dataset with k time steps, perform the gradient smoothing process for each time step separately until no degenerate gradients exist.

2. For each spatial location in the volume, smooth the gradients associated with this spatial location in the temporal domain.

3. Repeat the above steps until a sufficiently smooth gradient field is obtained.

4. The smoothing process for a time step t can be accelerated by using the smoothed gradients from time steps $t+1$ or $t-1$ as the initial solution to time step t. Using the solution from time step $t+1$ or $t-1$ as the initial solution to time step t allows time step $k-1$ to be smoothed using fewer iterations.

5.4 Runtime Algorithm

The runtime algorithm for time-varying data is similar to that for static volumes. Given an error bound E, isovalue I, and time step T, the refinement process creates a set of tetrahedra S_t that approximates the regions of the dataset containing I at time T to within E. Given a mesh at a time step T and an error tolerance E, the following steps are taken as long as the time step does not change:

1. Diamonds outside the view frustum and diamonds that do not contain I are assigned an error of zero. Errors are recomputed for all other diamonds in the split and merge queues.

2. Diamonds in the split queue whose error is greater than E are split. Diamonds in the merge queue whose error is less than E are merged.

3. The split/merge refinement process is used to create S_t. The refinement process is stopped when no more diamonds can be split or merged, or when the time allocated for the current frame has elapsed.

4. The isosurface is extracted from the tetrahedra that belong to the visible, non-empty diamonds in the split queue.

5.4.1 Changing the Time Step

When the time step changes the isosurface for the new time step is extracted by starting from the previous mesh as follows:

1. Get the error, min, and max values for the new time step for all visible diamonds in the split and merge queues.

2. Refine and coarsen the mesh as described in Section 4.4.

3. Extract a new isosurface from the tetrahedra in the current mesh.

Once the error, min, and max values have been updated, the refinement procedure for a static volume is used until the time step is changed again. Diamonds whose error, min, and max values were not updated in Step 1 are updated the next time they could be used for visualization. Each diamond stores the time step that its error, min, and max values correspond to. When the diamond is needed, the stored time step is checked against the current time step and the data is updated as necessary. For invisible diamonds in the split and merge queues, the values are updated when they become visible. For diamonds at coarser levels of the hierarchy, we perform a lazy evaluation and update the values only when they are added to the merge queue.

Adaptive refinement over time allows us to exploit temporal coherence in the volume. Tetrahedra with low temporal variation will remain in the mesh over several time steps. When

moving from time step T to $T+1$, many diamonds from time step T will have an error that is still within the error tolerance, and a small fraction of the diamonds must be split or merged to satisfy the error tolerance. While the isosurface for $T+1$ still needs to be extracted from all tetrahedra in the current mesh, initiating the refinement process for time step $T+1$ with the mesh from time step T requires fewer splits and merges than beginning from the root diamond and allows speculative paging of data from future time steps (Section 5.5). Similarly, as the viewing parameters change from frame F to $F+1$, tetrahedra where the isosurface is simple will remain in the mesh for several frames, and tetrahedra where the isosurface is more complex with be refined and coarsened as they move toward and away from the view point.

The performance of the refinement process as the time step and viewing parameters change depends on the spatial and temporal coherence of the volume, the error threshold, and the number of error calculations. When the viewing parameters change slowly and smoothly, the system takes advantage of frame-to-frame coherence as many tetrahedra remain in the mesh between frames and much of the data needed to extract the new isosurface is already loaded from disk. This is true even when the isosurface is being extracted at a high resolution as the amount of mesh reuse between frames is high although it often decreases as the resolution of the extracted surface increases. Additionally, the isosurface resolution affects system performance because the cost of extracting the new isosurface is directly proportional to its resolution. Conversely large changes in the viewing parameters are often very incoherent and will subsequently cause performance to decrease. The cost of the error calculations performed each frame is reduced by the use of Subtrees detailed in Section 4.9.

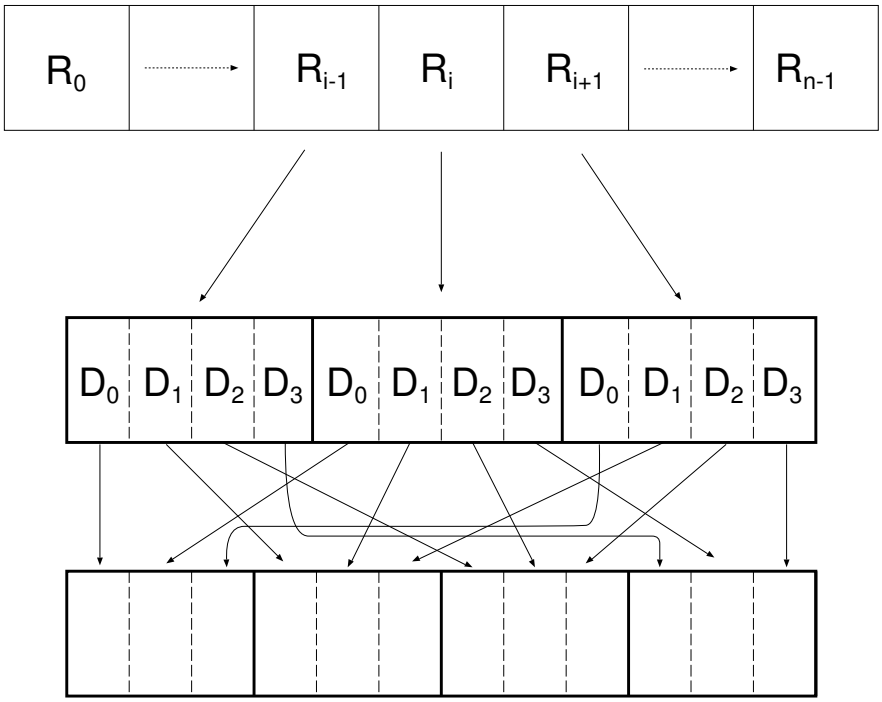


Figure 5.1: Top: Meta-records on disk in z-order. Middle: The meta-records are grouped into pages. Bottom: The values in each page are interleaved in time.

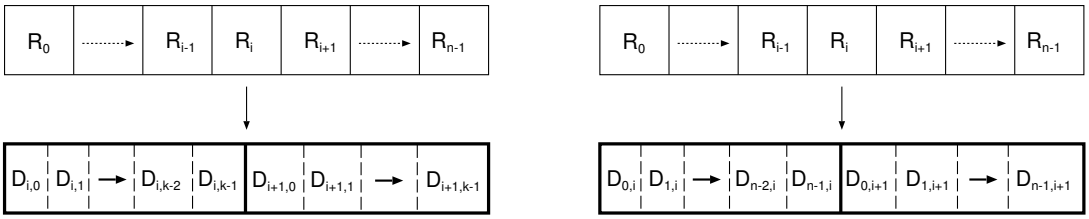


Figure 5.2: Left: Page size $k = 0, n = 1$, all timesteps for single data point are store together. Right: Page size $k = i, n = 2^i$, where 2^i is the size of the dataset, all data points for a single timestep are stored together. $D_{i,j}$ indicates data at index i time step j.

5.5 Memory Layout for Time-varying Data

To improve memory performance, the data is arranged in hierarchical z-order as described in Section 4.5. For time-varying data, we have a sequence of datasets that needs to be stored in a spatially and temporally coherent manner. As shown in in Figure 5.1, each data point is a meta-record which stores the data values for all time steps at the given spatial location. On disk the records are grouped into pages of $n = 2^k$ elements. Since our data layout scheme orders the data points first by level of detail and then by geometric proximity, each of these pages corresponds to a spatially coherent region within the data hierarchy. Within these pages, the data is interleaved in time so that time steps can be effectively prefetched for the spatial region covered by the page. The size of the pages is a tunable parameter that can be optimized for the capabilities of different storage systems and specific datasets.

Figure 5.2 illustrates the two extremes of this storage scheme. For $k = 0$ and $n = 1$, each page contains a single element and thus all timesteps for a single data point are stored together. In this case, the storage scheme has extrememly fine granularity allowing data values to be prefetched over time only for a single data point. While this minimizes the amount of unused data that is loaded from disk, prefetching time steps for a group of data points requires a large number of small disk reads which is very inefficient. If we load several pages at once in a single disk read, all of the time steps for a single data point must be loaded from disk and kept in memory at the same time. This creates a large amount of wasted space as time step 200 must be loaded even if timestep 10 is currently being visualized.

When $k = 3i$ and $n = 2^{3i}$, 2^{3i} is the size of the dataset, each page contains all data points for a single time step and the storage scheme has coarse granularity allowing all data values for a single time step to be loaded at once. This storage scheme minimizes the number of disk reads, but results in a large amount of unused data being loaded since certain regions will not contain the isosurface

and others may be visualized at coarser resolutions. This layout also makes it more expensive to do temporal prefetching for a given spatial region since a disk read is required for each prefetched time step and data for these time steps can be far apart on disk even if the dataset is stored in contiguous disk blocks. Thus, the page size must be selected to balance the disk and memory performance tradeoffs between these two extremes and to allow for efficient prefetching in the temporal domain.

Page faults occur when the requested page or time step is not in memory. The number of time steps loaded when a page fault occurs is slightly randomized to prevent a large number of faults from occurring at the same time. A fixed amount of memory is allocated for the pages, and they are loaded from disk as needed and replaced with a least-recently-used replacement scheme. Since the amount of space needed for k compressed time steps is not known at runtime, for each page that resides in main memory, we create fixed-size buffers for storing compressed and uncompressed data. When compressed data is loaded from disk, the compressed data buffer is filled and the range of loaded time steps is saved. Similarly, when the compressed data is decompressed, we uncompress a fixed number of time steps into the uncompressed data buffer.

As with static volumes, since data values and gradients are used only during the isosurface extraction and rendering phases, and the error, min, and max values are used only during the mesh refinement phase, the original data values and gradients are stored in one file, and the error, min, and max values are stored in a separate data file

5.5.1 Compressed Disk Layout

To reduce the storage space and the number of the disk reads, the data and gradients are stored in a compressed form on disk and decoded at runtime. The difference transformation described in Section 5.7 is applied to the data values and the components of the gradient (see Section 5.3) with respect to the temporal domain. The data for every kth time step is stored uncompressed to avoid

decompressing time steps $[0, t-1]$ when time step t is requested. The decompression for time step t starts at time step $(t - (t \bmod k))$ unless time step $t-1$ has already be decompressed. For each data point, in place of the original uncompressed data, we have uncompressed data and difference transformed values. This data is stored in z-order, grouped into pages, and interlaced as shown in Figure 5.1. The interlaced difference values are compressed using the algorithm described in Sections 4.11 and 5.7.

A two-dimensional lookup table is used to access the compressed data. The first dimension is the disk offset for the start of a specific page. For pages of size 2^k, the page number for a data point is computed by shifting out the low order k bits from the point's z-order address. The second dimension of the lookup table is the disk offset for the start of a specific time step within the disk page. At runtime, this lookup table is used to determine the location at which to start reading data and the amount of data to read in order to prefetch a certain number of time steps for a given page.

Since the amount of space needed for k compressed time steps is not known at runtime, for each page that resides in main memory, we create fixed-size buffers for storing compressed and uncompressed data. When compressed data is loaded from disk, the compressed data buffer is filled, and the range of loaded time steps is saved. Similarly, when the compressed data is decompressed, we uncompress a fixed number of time steps into the uncompressed data buffer.

5.6 Hardware Accelerated Contouring

When we move from one time step to the next, the isosurface needs to be reextracted from the tetrahedra in the current mesh. This process is very computationally intensive since it must touch every tetrahedron. For linear interpolation, the equations for positions and gradients along an edge

are:

$$\begin{aligned} p &= (1.0-\alpha)\times p_0 + \alpha \times p_1 \\ g &= (1.0-\alpha)\times g_0 + \alpha \times g_1 \\ \alpha &= \frac{(isovalue - d_0)}{d_1 - d_0} \end{aligned}$$

where $p_0, p_1, g_0, g_1, d_0, d_1$ are the positions, gradients and data values at the end points. Performing the interpolations on the CPU requires a lot of data movement and a large number of floating point operations. Modern graphics cards are optimized to perform these vector operations in parallel, and thus it makes sense to perform these interpolations on the graphics card.

When interpolations are performed on the CPU, six floating point values, three for position and three for gradient, are stored for one vertex. To perform these interpolations on the graphics card, we need to send it two positions, two gradients, two data values, and the isovalue for a total of 15 values. However, sending 15 floating point values per vertex increase the amount of data that needs to be transfered by 250% which significantly degrades performance.

Since we are working with fixed point data on a regular grid, the vertex positions and data values are integers. The gradient components as described in Section 5.3 are also integers. The positions, gradients, and data are packed into seven values as follows:

$$\begin{aligned} p_{packed} &= p_0 + p_1 \times 2^k \\ g_{packed} &= g_0 + g_1 \times 2^k \\ d_{packed} &= d_0 + d_1 \times 2^k \end{aligned}$$

Since all the data are integers and we are always scaling by a power of two, the packing is done using only shifts, masks, and logical operations. These packed values are converted to floating point and stored in vertex arrays as described in Section 4.9. The interpolation is performed on the graphics card using a vertex program. The positions, gradients, and data values are unpacked using

mod operations. The gradient components are then transformed to the range $[-1,1]$ and normalized. The α for linear interpolation is computed from the data values by passing the current isovalue as a parameter to the program. By packing the data and decoding it on the graphics card, we send 7 floating point values per vertex which increases the amount of transfered data by 17%.

Hardware contouring can also be applied to isosurface extraction from static volumes. In the static case, it is less effective because a large number of triangles are reused between successive frames, and the the vertex program decreases the rendering performance of the hardware. However, hardware contouring can be effectively utilized to speed up the extraction of a new isosurface when the isovalue is changed. This is similar to the time-varying situation when the time step changes because a new isosurface must be extracted for every tetrahedron in the split queue.

5.7 Compression

For data that varies gradually and smoothly a difference transform between adjacent values is very effective for increasing the data's redundancy or decreasing its entropy. The difference transform T_L that we use assumes that the data varies linearly over time.

$$T_C(i) = D(i) - D(i-1), 0 < i < n. \tag{5.1}$$

$$T_L(i) = \begin{cases} T_C(i) & : \quad i = 1 \\ T_C(i) - (T_C(i-1)) & : \quad 1 < i \end{cases} \tag{5.2}$$

The first data value at $i = 0$ is passed along unchanged.

Data transformation and compression are performed in the temporal domain. The set of data D, transformed using Equation 5.2, is the sequence of data and gradient values over time for a single spatial location. Figure 5.9 shows the coefficient histogram for our test dataset, a $256^3 \times 274$ time-varying simulation [56], before and after the difference transformation. Even though the magnitude range of the transformed coefficients has increased, a much larger number of the

magnitudes are closer to zero. This fact indicates that the transformed coefficients will compress better than the original data values. Data compression is done with Lead–1 encoding and Huffman codes as described in Section 4.11.2. The algorithms that we use to compress the transformed data are described in Section 4.11. Tables 4.2 and 4.3 show statistics for leading 1 counts and conditional leading one counts for our test dataset.

5.8 Results

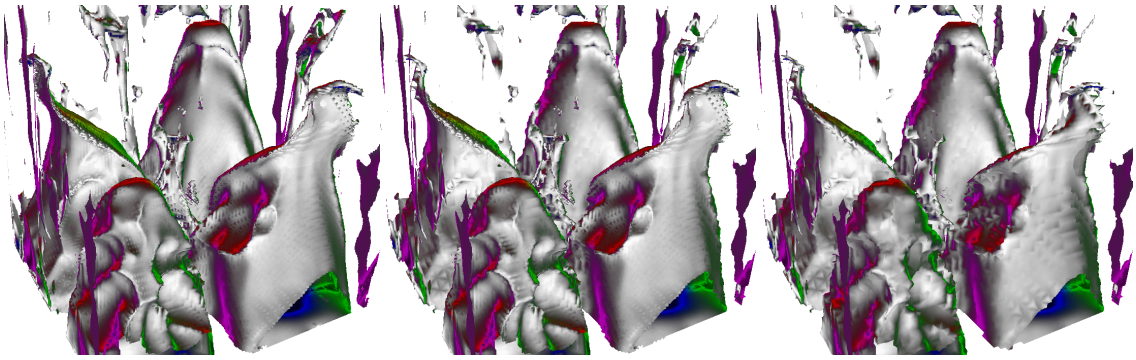


Figure 5.3: Error based rendering with fixed viewpoint at time step 260, isovalue 223.5. Left to right: Error = 0.78, 1.11M Triangles. Error = 1.2, 420K Triangles. Error = 1.8, 235K Triangles.

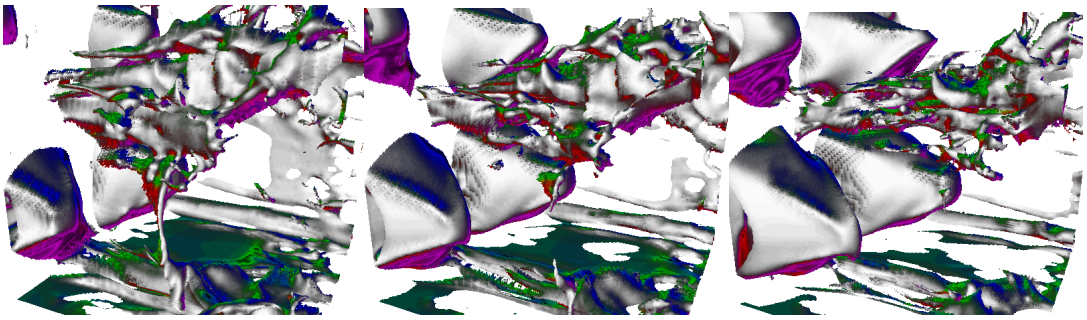


Figure 5.4: Left to right: Isosurfaces extracted from time steps 160, 180, and 200 at isovalue 227.9, error 0.5.

Our test machine is a 2 GHz Pentium IV with 1 GB of main memory and a GeForce4 Ti 4600 graphics card. We have tested our algorithm on a $256^3 \times 274$ chunk of the Richtmyer-Meshkov

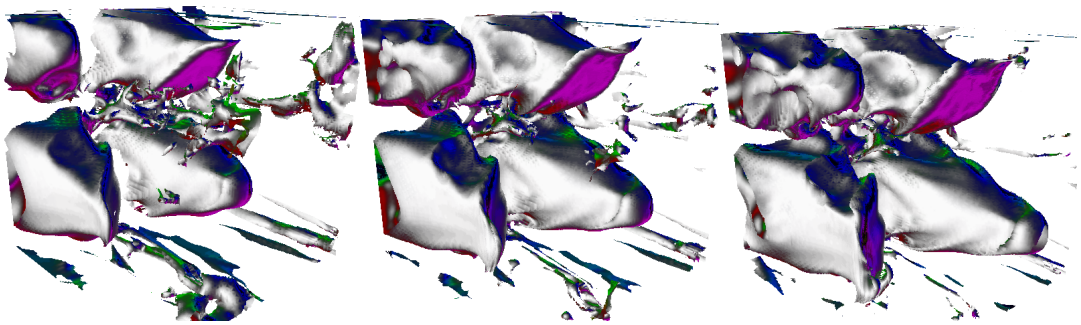


Figure 5.5: Left to right: Isosurfaces extracted from time steps 223, 248, and 273 at isovalue 227.9, error 0.85.

instability dataset from Lawrence Livermore National Lab. The simulation is of a shock passing through two gases of different densities and equal pressures initially separated by a membrane. In the simulation, the membrane is modeled as a contact discontinuity. As the shock passes through the gases, the mixing becomes more turbulent and the surfaces describing the mixing interface become very complex. As described in Section 5.1, the full resolution simulation contains 27,000 time steps and the output consists of only 274 time steps. Compared to the actual simulation, our test dataset has a much coarser temporal resolution and thus reduced coherence between time steps. The coarse temporal resolution decreases the amount of temporal coherence that can be exploited for compression and during mesh refinement.

Our test dataset is cropped from the full resolution dataset. The data values measure the entropy of the mixture. They are created by scaling the full range values into the range $[0, 255]$ such that 128 represents the value halfway between the minimum and maximum. Isosurfaces of the dataset contain a large number of small topological components and features as well as larger components with intricate mixing features.

A subtree depth of 2 is used; the uncompressed data and gradients take up 12.844 GB or 48 MB per time step, and the error, min, and max values take up 205 MB. Disk pages for the data and gradients have 2^{12} data points with 274 time steps and 3 bytes per time step for a total of

3.21 MB per page. The compressed data and gradients take up 6.99 GB or 54% of the original size. As described in Section 5.5.1, we store every 8th time step uncompressed to allow for progressive decompression. This results in 34 uncompressed time steps which takes up 1.6 GB. The runtime lookup tables for the compressed volume require $1.12MB$. The size of the error, min, and max values is $\frac{1}{64}$ the size of the data and gradients because subtrees of depth 2 perform 1 to 64 refinement. The error, min, and max values are stored uncompressed and loaded into memory at runtime.

Figure 5.3 shows contours from time step 260 at different levels of resolution. At finer resolutions one can see that the surface contains a large number of small topological components. As we move to coarser resolutions, the overall shape of the surface is maintained, but the topology is not preserved and many of the small topological components disappear. Figures 5.4 and 5.5 show a series of contours from the same physical region in the simulation extracted at different time steps. These pictures illustrate the formation of the numerous large and small scale features that are present in the mixing interface. Figure 5.6 shows adaptive refinement of the isosurface over a series of time steps. Images from time steps 150, 211, and 270 are shown at different levels of resolution for a fixed viewpoint as the volume is played forward in time from timestep 0 to 273. The surface at time step 150 is refined with an error of 1.5 to capture some of the intricate folds and topological components of the surface. As time progresses, the large surface patch in the front breaks apart into smaller topologically separate surfaces. To capture these smaller components, the error is changed to 0.5, and finally as the surface becomes less complex, as shown in the image for time step 270, the error is changed to 2.7.

5.8.1 Memory Layout Performance

To compare the performance of our algorithm using compressed and uncompressed volumes, we play the volume from $t = 0$ to $t = 273$ from a fixed viewpoint with a small error value of 0.7.

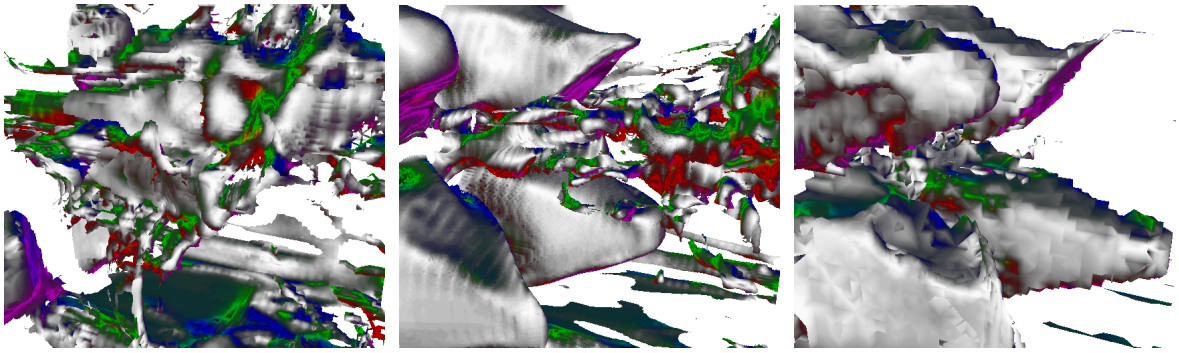


Figure 5.6: Adaptive error-based refinement of an isosurface over time for a fixed viewpoint. Isovalue = 225. From left to right: Time step 150, Error = 1.5, 220K Triangles. Time step 211, Error = 0.5, 850K Triangles. Time step 270, Error = 2.7, 51K Triangles.

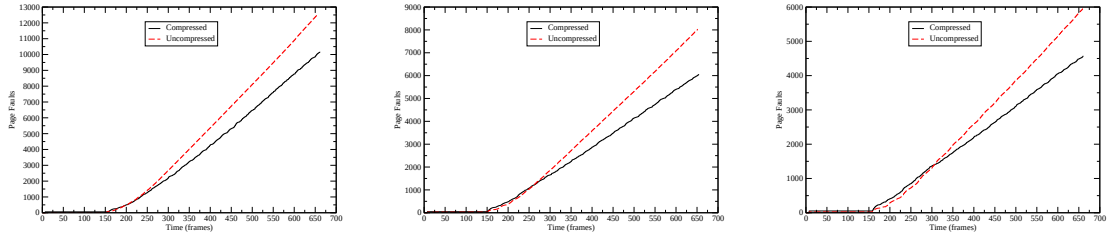


Figure 5.7: Page faults over time for compressed and uncompressed volumes. Left to right: disk read sizes of 10, 15, and 20 uncompressed time steps.

A small error ensures that a high resolution surface is extracted and that the finest levels of the hierarchy must be accessed for the isosurface extraction. To evaluate the amount of disk I/O, the tests are run so that the amount of data loaded from disk is roughly the same when compressed and uncompressed volumes are used. The amount of data loaded is a multiple of the size of a single time step in an uncompressed page. A single time step in an uncompressed page has 2^{12} or 4096 elements with 3 bytes per element for a total of 12288 bytes. In our tests, we use load sizes of 10, 15, and 20 uncompressed times steps. When the compressed volume is used, 7 time steps are decompressed at a time and kept in memory in addition to the compressed time steps loaded from disk. Table 5.1 summarizes the size of the main memory pages and the amount of data read for the compressed and uncompressed volumes at the different load sizes. Even when viewing

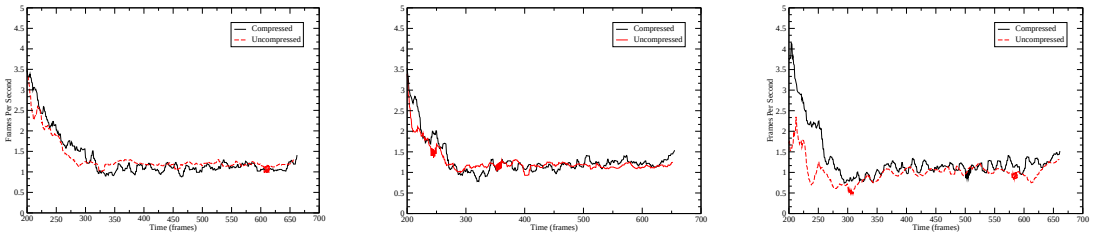


Figure 5.8: Frame rate over time for compressed and uncompressed volumes. Left to right: disk read sizes of 10, 15, and 20 uncompressed time steps.

Load Size	10	15	20
Data Loaded(C)	1.05(GB)(15%)	0.9(12.87%)	0.87(12.4%)
Data Loaded(U)	1.43(GB)(11.1%)	1.38(10.72%)	1.36(10.6%)
Page Size(C)	208904(bytes)	270344	331784
Page Size(U)	122888(bytes)	184328	245768

Table 5.1: Total data loaded and main memory page sizes for each run. The size of the uncompressed volume is 12.844 GB, and the size of the compressed volume is 6.99 GB.

a surface at a very high resolution, for many time steps the extracted surface has over a million triangles, the actual amount of data loaded is relatively small compared to the total size of the datasets. As expected, the percentage of data loaded when using the compressed volume is higher because it uses less storage, and the amount of data loaded in a single read is the same between compressed and uncompressed playback. Figure 5.7 compares the total number of page faults over time for compressed and uncompressed volumes. Time is the number of frames recorded as the volume is played from start to finish. There are between 660 and 670 frames for each playback sequence. As is expected, the use of the compressed volume reduces the amount of data read from disk and the frequency of the disk reads. The linear increase in the amount of data read and the number of page faults during the playback indicates that the data storage scheme, which balances spatial and temporal locality, does a good job of streaming the needed data from disk. A total of 510 different disk pages were loaded during playback. The frame rates over time for the compressed and uncompressed volumes are shown Figure 5.8. The data for the initial 200 frames is

not shown because the surfaces in those frames contain fewer triangles and have higher frame rates which are not indicative of the overall performance. The compressed and uncompressed volumes both playback at 1-2 frames per second. The extra cost of decompressing and reconstructing the compressed volume is roughly equivalent to the extra disk reads when using the uncompressed volume. Using the compressed volume, we achieve the same performance as the uncompressed volume using about half the storage.

5.8.2 Hardware Accelerated Contouring

Table 5.2 summarizes the improvements in triangle extraction rate for hardware interpolation versus software interpolation for different error bounds and triangle counts. The numbers are calculated by playing the uncompressed volume from time step 0 to time step 273, advancing the time step at regular intervals, and recording the time to generate the new isosurface from the current mesh.

The isosurface is rendered from a fixed viewpoint in a 570×530 screen. To ensure consistent memory and disk performance, the page table and all caches are cleared at the start of each test. For error bounds of 1.8 and 1.3, hardware interpolation is two to three times faster than software interpolation. For an error bound of 0.8, hardware interpolation is about 70% faster. At higher errors, there are fewer page faults and thus the performance benefits of hardware over software interpolation dominate the performance of the extraction process. At lower errors, the performance benefit decreases as more data must be accessed from memory and disk. Despite the increased cost of data access at low error bounds, the hardware interpolation still improves the extraction rate by about 455K triangles per second. Recalculating the interpolations every frame reduces the rendering performance of the hardware by about 35%. For static scenes, where the view point and time step do not change and the mesh is not refined, we can render 16–18 million textured triangles a second. When hardware interpolations are used, the rendering performance is 10–12 million triangles a sec-

Error	Triangles	Hardware Interpolation	Reextract Tris/Sec
1.8	75-100K	no	755K
		yes	2.02M
1.3	200-300K	no	812K
		yes	1.9M
0.8	650-750K	no	625K
		yes	1.08M

Table 5.2: Comparison of triangle extraction rates for hardware and software interpolation at different error bounds.

ond. For a typical surface with 750K triangles, this corresponds to a increase in the time to render a frame from 0.047 to 0.07 seconds. However, as graphics card performance improves, the effect of the vertex program will become less and less significant.

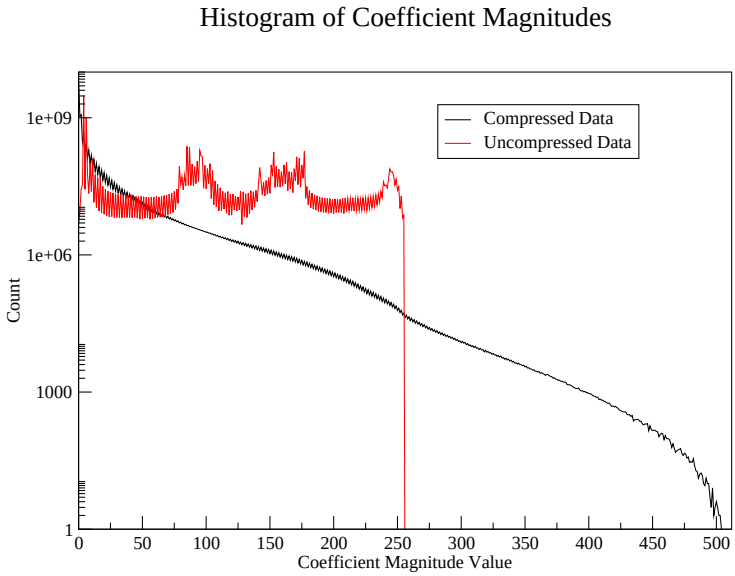


Figure 5.9: Histogram of coefficient magnitudes in a 256^3 volume with 274 timesteps. The vertical axis is on a logarithmic scale.

Chapter 6

Conclusions

This dissertation has presented a new crack-free refinement algorithm for tetrahedral meshes created by edge bisection. Previous algorithms based on splitting tetrahedra required additional refinement steps to locate neighboring tetrahedra in order to maintain a conformant mesh. This new formulation based on aggregate structures of tetrahedra called diamonds simplifies the mesh refinement and the search for neighboring tetrahedra and easily generates conformant meshes. This new mesh structure can be used for multiresolution scientific visualization across a wide range of applications including volume simplification, isosurface extraction, and volume rendering.

Chapter 3 describes the use of these meshes for creating resampled, multiresolution representations of volume datasets that contain boundaries defined by material interfaces. In this application, we developed a new linear field approximation for tetrahedral cells that contained explicit discontinuities. Separate linear functions are created for each field that exists within a tetrahedron. This allowed the accurate approximation of fields that were discontinuous across boundaries internal to the cells of the mesh.

In Chapter 4 these adaptive tetrahedral meshes are used to perform interactive, adaptive isosurface extraction from large, compressed volume datasets. In this application we show the

intimate relationship between the edge bisection refinement algorithm and hierarchical z-order space filling curves. These space filling curves are used to order the volume data in the same order that the mesh refinement accesses it, allowing for efficient out-of-core visualization which is critical for large data visualization.

Lastly, in Chapter 5 the algorithms presented in Chapter 4 have been extended to compressed time-varying datasets. The space filling curve data layout used for static volumes is extended to a time-interlaced data layout that organizes the data in discrete chunks; the data is ordered first by spatial locality according to the z-order space filling curve and then by temporal locality within each chunk. This new ordering scheme allows data for sucessive timesteps to be streamed from disk to memory over time as an isosurface is continuously extracted over all timesteps in a volume.

Bibliography

[1] Douglas N. Arnold, Arup Mukherjee, and Luc Pouly. Locally adapted tetrahedral meshes using bisection. *SIAM Journal on Scientific Computing*, 22(2):431–448, 2001.

[2] J. Bey. Tetrahedral grid refinement. *Computing*, 55(4):355–378, 1995.

[3] Kathleen S. Bonnell, Daniel R. Schikore, Kenneth I. Joy, Mark Duchaineau, and Bernd Hamann. Constructing material interfaces from data sets with :1 volume-fraction information. In *Proceedings Visualization 2000*, pages 367–372, 2000.

[4] Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno. Planet-sized batched dynamic adaptive meshes (p-bdam). In *Proceedings of IEEE Visualization 2003*, pages 147–154, 2003.

[5] Paolo Cignoni, Enrico Puppo, and Roberto Scopigno. Multiresolution Representation and Visualization of Volume Data. *IEEE Transactions on Visualization and Computer Graphics*, 3:352–369, October 1997.

[6] Michael Cox and David Ellsworth. Application-controlled demand paging for out-of-core visualization. In *IEEE Visualization*, pages 235–244, 1997.

[7] Mark A. Duchaineau, Martin Bertram, Serban Porumbescu, Bernd Hamann, and Kenneth I. Joy. Interactive Display Of Surfaces Using Subdivision Surfaces And Wavelets. In T.L. Kunii, editor, *Proceedings of 16th Spring Conference on Computer Graphics, Comenius University, Bratislava, Slovak Republic*, pages 22–34, 2001.

[8] Mark A. Duchaineau, Serban Porumbescu, Martin Bertram, Bernd Hamann, and Kenneth I. Joy. Dataflow And Re-Mapping For Wavelet Compression And View-Dependent Optimization Of Billion-Triangle Isosurfaces. In G. Farin, H. Hagen, and B. Hamann, editors, *Hierarchical Approximation and Geometrical Methods for Scientific Visualization*. Springer-Verlag, Berlin, Germany, (to appear)., 2002.

[9] Mark A. Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. ROAMing Terrain: Real-time Optimally Adapting Meshes. In *IEEE Visualization '97*, pages 81–88. IEEE Computer Society Press, October 1997.

[10] J. El-Sana, N. Sokolovsky, and C. Silva. Integrating occlusion culling with view-dependent rendering. *Proc. of IEEE Visualization*, 2001.

[11] Klaus Engel, Rudiger Westermann, and Thomas Ertl. Isosurface Extraction Techniques For Web-Based Volume Visualization. In *Proceedings of IEEE Visualization 1999*, pages 139–146. IEEE Computer Society Press, 1999.

[12] Jinzhu Gao, Jian Huang, Han-Wei Shen, and James Arthur Kohl. Visibility culling using plenoptic opacity functions for large volume visualization. In *Proceedings of IEEE Visualization 2003*, pages 341–348, 2003.

[13] M. Garland and P. S. Heckbert. Simplifying surfaces with color and texture using quadric error metrics. In *IEEE Visualization '98 (VIS '98)*, pages 263–270, Washington - Brussels - Tokyo, October 1998. IEEE.

[14] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 209–216. ACM SIGGRAPH, Addison Wesley, August 1997.

[15] Marcel Gavriliu, Joel Carrance, David E. Breen, and Alan H. Barr. Fast Extraction Of Adaptive Multiresolution Meshes With Guaranteed Properties From Volumetric Data. In T. Ertl, K. I. Joy, and A. Varshney, editors, *Proceedings Visualization 2001*, pages 295–302. IEEE Computer Society Press, 2001.

[16] T. Gerstner. Fast Multiresolution Extraction Of Multiple Transparent Isosurfaces. In Ronald Peikert David S. Ebert, Jean M. Favre, editor, *In Data Visualization 2001 Proceedings of VisSim 2001*, Annual Conference Series. Springer-Verlag, 2001.

[17] T. Gerstner and M. Rumpf. Multiresolution Parallel Isosurface Extraction Based On Tetrahedral Bisection. In M. Chen, A. Kaufman, and R. Yagel, editors, *Volume Graphics*, pages 267–278. Springer-Verlag, 2000.

[18] Thomas Gerstner and Renato Pajarola. Topology Preserving And Controlled Topology Simplifying Multiresolution Isosurface Extraction. In T. Ertl, B. Hamann, and A. Varshney, editors, *Proceedings of IEEE Visualization 2000*, pages 259–266. IEEE Computer Society Press, 2000.

[19] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Addison–Wesley, 1992. See Chapter 3.

[20] Naga K. Govindaraju, Brandon Lloyd, Sung-Eui Yoon, Avneesh Sud, and Dinesh Manocha. Interactive shadow generation in complex environments. In *Proceedings of SIGGRAPH 2003*, pages 501–510, 2003.

[21] Amara Graps. An introduction to wavelets. *IEEE Computational Science and Engineering*, 2(2), 1995.

[22] Ned Greene. Hierarchical Polygon Tiling With Coverage Masks. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 65–74. Addison Wesley, 1996.

[23] Benjamin Gregorski, Mark A. Duchaineau, Peter Lindstrom, Valerio Pascucci, and Kenneth I. Joy. Interactive view-dependent rendering of large isosurfaces. In *Proceedings of the IEEE Visualization 2002*, 2002.

[24] Stefan Guthe and Wolfgang Staser. Real-time decompression and visualization of animated volume data. In *Proceedings of IEEE Visualization 2001*, pages 349–358, 2001.

[25] Stefan Guthe, Michael Wand, Julius Gonser, and Wolfgang Staser. Interactive rendering of large volume data sets. In *Proceedings of IEEE Visualization 2002*, pages 53–60, 2002.

[26] Haeyoung Ha. Out-of-core interactive display of large meshes using an oriented bounding box-based hardware depth query. Master's thesis, University of California, Davis, September 2003. Available as Department of Computer Science Technical Report CSE-2003-25.

[27] B. Hamann. A data reduction scheme for triangulated surfaces. In *Computer Aided Geometric Design*, volume 11, pages 197–214, 1994.

[28] Paul S. Heckbert and Michael Garland. Survey of polygonal surface simplification algorithms. Technical report.

[29] B. Heckel, A. E. Uva, and B. Hamann. Clustering-based generation of hierarchical surface models. In *Proceedings IEEE Visualization '98 – Late Breaking Hot Topics*, 1998.

[30] K. Hillesland, B. Salomon, A. Lastra, and D. Manocha. Fast and simple occlusion culling using hardware-based depth queries. Technical Report UNC-CH-TR02-039, Computer Science Department, University of North Carolina at Chapel Hill, Chapel Hill, North Carolina, 2002.

[31] Hugues Hoppe. Progressive meshes. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 99–108. ACM SIGGRAPH, Addison Wesley, August 1996.

[32] Hugues Hoppe. View-Dependent Refinement Of Progressive Meshes. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 189–198. Addison Wesley, 1997. Los Angeles, California, 03-08 August 1997.

[33] Hugues H. Hoppe. New quadric metric for simplifying meshes with appearance attributes. In David Ebert, Markus Gross, and Bernd Hamann, editors, *IEEE Visualization '99*, pages 59–66, San Francisco, 1999. IEEE.

[34] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[35] Armin Kanitsar, Thomas Theußl, Lukas Mroz, Milos Srámek, Anna Vilanova Bartrolí, Balázs Csébfalvi, Jirí Hladùvka, Dominik Fleischmann, Michael Knapp, Rainer Wegenkittl, Petr Felkel, Stefan Guthe, Werner Purgathofer, and Meister Eduard Gröller. Christmas tree case study: Computed tomography as a tool for mastering complex real world objects with applications in computer graphics. In Robert Moorhead, Markus Gross, and Kenneth I. Joy, editors, *Proceedings of the 13th IEEE Visualization 2002 Conference (VIS-02)*, pages 489–492, Piscataway, NJ, October 27– November 1 2002. IEEE Computer Society.

[36] Reinhard Klein, Gunther Liebich, and Wolfgang Straßer. Mesh reduction with error control. In Roni Yagel and Gregory M. Nielson, editors, *Proceedings of the Conference on Visualization*, pages 311–318, Los Alamitos, October 27–November 1 1996. IEEE.

[37] Leif P. Kobbelt, Jens Vorsatz, Ulf Labsik, and Hans-Peter Seidel. A shrink wrapping approach to remeshing polygonal surface. In P. Brunet and R. Scopigno, editors, *Computer Graphics Forum (Eurographics '99)*, volume 18(3), pages 119–130. The Eurographics Association and Blackwell Publishers, 1999.

[38] Lawrence Larmore and Daniel Hirschberg. A fast algorithm for optimal length-limited huffman codes. *Journal of the Association for Computing Machinery*, 37(3):464–473, Jul 1990.

[39] Aaron W. F. Lee, Wim Sweldens, Peter Schröder, Lawrence Cowsar, and David Dobkin. MAPS: Multiresolution adaptive parameterization of surfaces. In Michael Cohen, editor, *SIGGRAPH 98 Conference Proceedings*, Annual Conference Series, pages 95–104. ACM SIGGRAPH, Addison Wesley, July 1998.

[40] Haeyoung Lee, Mathieu Desbrun, and Peter Schroder. Progressive encoding of complex isosurfaces. In *Proceedings of SIGGRAPH 2003*, pages 471–476, 2003.

[41] Joshua Levenberg. Fast view-dependent level-of-detail rendering using cached geometry. In *Proceedings of IEEE Visualization 2002*, pages 259–265, 2002.

[42] Wei Li, Klaus Mueller, and Ari Kaufman. Empty space skippping and occlusion clippinf for texture-based volume rendering. In *Proceedings of IEEE Visualization 2003*, pages 317–326, 2003.

[43] Mike Liddell and Alistair Moffat. Incremental calculation of minimum–redundancy length–restricted codes. In *Proceedings of the Data Compression Conference (DCC 2002)*, 2002.

[44] Peter Lindstrom. Out-Of-Core Simplification Of Large Polygonal Models. In Kurt Akeley, editor, *SIGGRAPH 2000 Conference Proceedings*, pages 259–262. ACM Press / Addison Wesley, 2000.

[45] Peter Lindstrom and Valerio Pascucci. Visualization Of Large Terrains Made Easy. In T. Ertl, K. Joy, and A. Varshney, editors, *Proceedings of IEEE Visualization 2001*, pages 363–370. IEEE Computer Society Press, 2001.

[46] Peter Lindstrom and Claudio T. Silva. A Memory Insensitive Technique For Large Model Simplification. In T. Ertl, K. I. Joy, and A. Varshney, editors, *Proceedings of IEEE Visualization 2001*, pages 121–126. IEEE Computer Society Press, 2001.

[47] Lars Linsen, Jevan T. Gray, Valerio Pascucci, Mark A. Duchaineau, Bernd Hamann, and Kenneth I. Joy. Hierarchical large-scale volume representation with ’3rd-root-of-2’ subdivision and trivariate b-spline wavelets. In *Geometric Modeling for Scientific Visualization*. Springer-Verlag, Heidelberg, Germany, 2003.

[48] Anwei Liu and Barry Joe. On the shape of tetrahedra from bisection. *Math. Comput.*, 63(207):141–154, 1994.

[49] Anwei Liu and Barry Joe. Quality local refinement of tetrahedral meshes based on bisection. *SIAM J. Sci. Comput.*, 16(6):1269–1291, 1995.

[50] Zhiyan Liu, Adam Finkelstein, and Kai Li. Progressive View-Dependent Isosurface Propagation. In D. Ebert, J. M. Favre, and R. Peikert, editors, *Proceedings of the Joint Eurographics - IEEE TCVG Symposium on Visualizatation (VisSym-01)*, pages 223–232. Springer-Verlag, 2001.

[51] Y. Livnat and C. Hansen. View Dependent Isosurface Extraction. In David Ebert, Hans Hagen, and Holly Rushmeier, editors, *Proceedings of IEEE Visualization 1998*, pages 175–180. IEEE Computer Society Press, 1998.

[52] David Luebke and Carl Erikson. View-Dependent Simplification Of Arbitrary Polygonal Environments. In *SIGGRAPH 97 Conference Proceedings*, pages 199–208, Los Angeles, California, 1997. Addison Wesley.

[53] Eric Lum, Kwan-Liu Ma, and John Clyne. Texture hardware assisted rendering of time-varying volume data. In *Proceedings of IEEE Visualization 2001*, pages 263–270, 2001.

[54] Joseph Maubach. Local bisection refinement for n-simplicial grids generated by reflection. *SIAM J. Sci. Comput.*, 16(1):210–227, 1995.

[55] Joseph Maubach. The efficient location of neighbors for locally refined n-simplicial grids. In *Proc. 5th Int. Meshing Roundtable, Sandia National Laboratories*, pages 137–153, 1996.

[56] Arthur A. Mirin, Ron H. Cohen, Bruce C. Curtis, William P. Dannevik, Andris M. Dimits, Mark A. Duchaineau, D. E. Eliason, Daniel R. Schikore, S. E. Anderson, D. H. Porter, , and Paul R. Woodward. Very High Resolution Simulation Of Compressible Turbulence On The IBM-SP System. In *Proceedings of SuperComputing 1999*. (Also available as Lawrence Livermore National Laboratory technical report UCRL-MI-134237), 1999.

[57] V. Pascucci and C. L. Bajaj. Time Critical Isosurface Refinement And Smoothing. In *Proceedings of the 2000 IEEE symposium on Volume Visualization*, pages 33–42. ACM Press, 2000.

[58] Valerio Pascucci. Multi-Resolution Indexing For Out-Of-Core Adaptive Traversal Of Regular Grids. In G. Farin, H. Hagen, and B. Hamann, editors, *Proceedings of the NSF/DoE Lake Tahoe Workshop on Hierarchical Approximation and Geometric Methods for Scientific Visualization*. Springer-Verlag, Berlin, Germany, (to appear)., 2002. (Available as LLNL technical report UCRL-JC-140581).

[59] A. Plaza and G. Carey. About local refinement of tetrahedral grids based on bisection. In *Proc. 5th International Meshing Roundtable, Sandia National Laboratories*, pages 123–136, 1996.

[60] Alex Pomeranz. Roam using surface triangle clusters (rustic). Master's thesis, University of California, Davis, June 1998.

[61] Tom Roxborough and Gregory M. Nielson. Tetrahedron Based, Least Squares, Progressive Volume Models With Application To Freehand Ultrasound Data. In T. Ertl, B. Hamann, and A. Varshney, editors, *Proceedings Visualization 2000*, pages 93–100. IEEE Computer Society Press, 2000.

[62] Detlef Ruprecht and Heinrich Müller. A scheme for edge-based adaptive tetrahedron subdivision. In Hans-Christian Hege and Konrad Polthier, editors, *Mathematical Visualization*, pages 61–70. Springer Verlag, Heidelberg, 1998.

[63] Han-Wei Shen. Isosurface extraction in time-varying fields using a temporal hierarchical index tree. In David Ebert, Hans Hagen, and Holly Rushmeier, editors, *IEEE Visualization '98*, pages 159–166. IEEE, 1998.

[64] Han-Wei Shen, Ling-Jan Chiang, and Kwan-Liu Ma. A fast volume rendering algorithm for time-varying fields using a time-space partitioning (TSP) tree. In David Ebert, Markus Gross, and Bernd Hamann, editors, *IEEE Visualization '99*, pages 371–378, San Francisco, 1999.

[65] Oliver G. Staadt and Markus H. Gross. Progressive tetrahedralizations. In David Ebert, Hans Hagen, and Holly Rushmeier, editors, *Proceedings of Visualization 98*, pages 397–402. IEEE Computer Society Press, October 1998.

[66] Stollnitz, DeRose, and Salesin. *Wavelets for Computer Graphics*. Morgan Kaufffmann, 1996.

[67] Philip Sutton and Charles Hansen. Isosurface extraction in time-varying fields using a temporal branch-on-need tree (T-BON). In David Ebert, Markus Gross, and Bernd Hamann, editors, *EEE Visualization '99*, pages 147–154, San Francisco, 1999.

[68] Philip Sutton and Charles Hansen. Accelerated isosurface extraction in time-varying fields. *IEEE Transactions on Visualization and Computer Graphics*, 6(2):98–107, April/June 2000.

[69] Isaac J. Trotts, Bernd Hamann, Kenneth I. Joy, and David F. Wiley. Simplification of tetrahedral meshes. In Holly Rushmeier David Ebert and Hans Hagen, editors, *Proceedings IEEE Visualization '98*, pages 287–296. IEEE Computer Society Press, October 18–23 1998.

[70] Andrew Turpin and Alistair Moffat. Efficient implementation of the package-merge paradigm for generating lengthlimited codes. In *Proceedings of Conference on Computing: The Australian Theory Symposium*, pages 187–195, Jan 1996.

[71] Rudiger Westermann. Compression domain rendering of time-resolved volume data. In *Proceedings of IEEE Visualization 1995*, pages 168–175, 1995.

[72] Rudiger Westermann. Compression domain volume rendering. In *Proceedings of IEEE Visualization 2003*, pages 293–300, 2003.

[73] Jane Wilhelms and Allen Van Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11(3):201–227, July 1992.

[74] Zoë J. Wood, Mathieu Desbrun, Peter Schröder, and David Breen. Semi-Regular Mesh Extraction From Volumes. In T. Ertl, B. Hamann, and A. Varshney, editors, *Proceedings of IEEE Visualization 2000*, pages 275–282. IEEE Computer Society Press, 2000.

[75] Vijaya Ramachandran Xiaoyu Zhang, Chandrajit Bajaj. Paralel And Out-Of-Core View-Dependent Isocontour Visualization. In David Ebert, Pere Brunet, and Isabel Navaz, editors, *Proceedings of the Joint Eurographics - IEEE TCVG Symposium on Visualizatation (VisSym-02)*, Vienna, Austria, May 27–29 2002. Springer-Verlag.

[76] Sung-Eui Yoon, Brian Salomon, and Dinesh Manocha. Interactive view-dependent rendering with conservative occlusion culling in complex environments. In *Proceedings of IEEE Visualization 2003*, 2003.

[77] H. Zhang, D. Manocha, T. Hudson, and K. Hoff III. Visibility culling using hierarchical occlusion maps. *Proceedings of SIGGRAPH*, pages 77–88, August 1997.

[78] Yong Zhou, Baoquan Chen, and Arie Kaufman. Multiresolution Tetrahedral Framework For Visualizing Regular Volume Data. In *Proceedings of IEEE Visualization 1997*, pages 135–142. IEEE Computer Society Press, 1997.

Appendix A

Index Calculations

A.1 Z-order Space Filling Curve

The z-order space filling curves for two and three dimensions are shown in Figure A.1. In general, a k-dimensional z-order curve is created by connecting two (k-1)-dimensional curves. For example, a zero-dimension curve is a single point, a one-dimension curve is a line segment, and a two-dimensional curve has the familiar z-shape shown in Figure A.1.

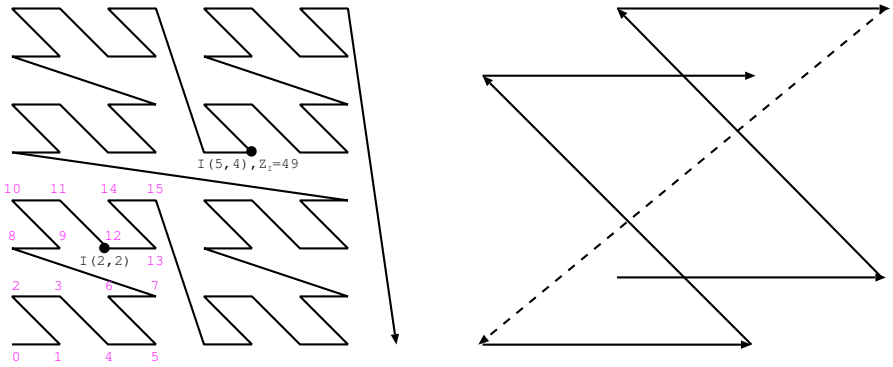


Figure A.1: Left: Two-dimensional z-order space filling curve. Right: The three-dimensional curve connects two two-dimensional curves.

A.2 Index Computations

A.2.1 Computing Z-Indices

One advantage of the z-order space filling curve is that it is easy and fast to convert between k-dimensional regular grid indices and the one-dimensional index along the space filling curve. Let $I(i_0, i_1 \ldots i_{k-1})$ be a point P's index in a k-dimensional regular grid. Let Z_I be the one-dimensional index of I on the z-order space filling curve that traverses the k-dimensional grid. Given the binary representation of I (i.e. $i_0, i_1 \ldots i_{k-1}$ are represented in binary), Z_I is computed by interlacing the bits of $i_0, i_1 \ldots i_{k-1}$. This is illustrated in Figure A.2. For example, in the two-dimensional curve in Figure A.2, the point $I(5,4)$ with binary representation $(101, 100)$ has $Z_I = \mathbf{1}1\mathbf{0}0\mathbf{0}1 = 49$.

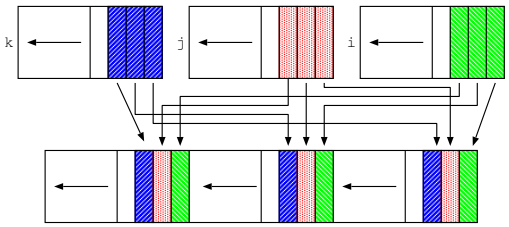


Figure A.2: The one-dimensional index Z_I along the z-order space filling curve is computed by interlacing the bits of the binary representations of $i_0, i_1 \ldots i_{k-1}$. In three dimensions, Z_I is created from the original index $I(i, j, k)$ by interlacing i, j, and k's bits.

A.2.2 Hierarchical Indices

For multiresolution applications, the z-order space filling curve can be used for fast, cache coherent access to a hierarchy of subsampled volumes. The coarsest level of the hierarchy is a single point. The hierarchical z-order space filling curve is illustrated in Figure A.3.

Level l_t, the top-down level, of k-dimensional hierarchy is a regular grid with 2^{kl_t} points. For example, for $k = 3, l_t = 8$, the regular grid is 256^3 and has 2^{24} points. The number of points in

coarser levels than l_t is

$$C_l = 2^{k(l_t-1)}, \tag{A.1}$$

and the number of points added at level l_t is

$$2^{kl} - 2^{k(l_t-1)}. \tag{A.2}$$

A 2^{kw} grid has $w+1$ levels with the finest level(smallest cells) being level 0, and the coarsest level(biggest cells) being level w. Level l_b, the bottom-up level, is computed as $w - l_t$. Given a point $I(i_0, i_1 \ldots i_{k-1})$ with z index Z_I, l_b for I is computed as $z \mod k$ where z is the number of trailing zeros in the binary representation of Z_I. For example, when $k = 2, w = 3$ on a 2^3x2^3 grid, $I(5,4)$ with binary representation $(101,100)$ has $Z_I = 110001$, and is in l_b = 0 or $l_t = 3$. The point $I(2,2)$ with binary representation $(10,10)$ has $Z_I = 1100$, and is in $l_b = 1$ or $l_t = 2$.

Now that we have Z_I, the index along the z-order curve, we want to compute ZH_I the index of the point along the hierarchical z-order curve. Table A.1 shows the regular and hierarchical z-order indices for the data points on a 4×4 grid as well as the local index L_{Z_I} of each z-order index Z_I. Figures A.1 and A.3 illustrate the traversal orders of the regular and hierarchical z-order curves.

At level 1 the indices (4,8,12) map to the local indices (0,1,2), and at level 2 the indices (1,2,3,5,6,7,9,10,11,13,14,15) map to the sequence (0,1,2,3,4,5,6,7,8,9,10,11). One is subtracted from (1,2,3) to get (0,1,2), two is subtracted from (5,6,7) to get (3,4,5), and so forth. At coarser levels, Z_I is normalized by a factor of 2^{l_b}. The general computation of L_{Z_I} on a k-dimensional grid is given as:

$$N = \frac{Z_I}{2^{kl_b}} L_{Z_I} = N - ((N/2^k) + 1). \tag{A.3}$$

For example: $Z_I = 13, N = 13/4 = 3\ L_Z = 13 - (3+1) = 9$. Given a regular z index Z_I and hierarhical z index ZH_I is computed as:

$$ZH_I = C_l + L_{Z_I}. \tag{A.4}$$

Level l_t	0	1			2											
Level l_b	2	1			0											
Local Index L_Z	0	0	1	2	0	1	2	3	4	5	6	7	8	9	10	11
Regular Z_I	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hierarchical ZH_I	0	4	8	12	1	2	3	5	6	7	9	10	11	13	14	15

Table A.1: Regular and hierarhical z-order for a 4×4 grid. The top row is the points's index in the top-down level l_t..

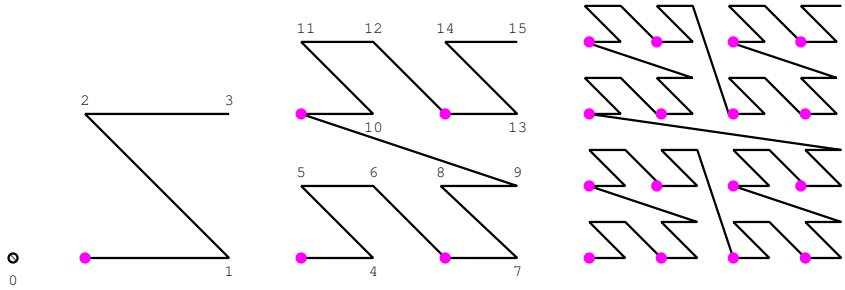


Figure A.3: Hierarchical z-order space filling curve in two-dimensions showing the one-dimensional order of the points. Left to right: coarse to fine levels of the hierarchy. The solid circles indicate the points introduced at coarser levels of the hierarchy.

A.2.3 Inverting Hierarchical Indices

For a hierarchical index ZH_I, the level l_b is computed as $z \mod k$ where z is the number of leading zeros in the binary representation of ZH_I. This is the reverse of level computation for a regular z-order index Z_I where the low order zeros in the binary representation are examined.

Given the hierarchical z-order index ZH_I for a point I, we want to compute the regular z-order index Z_I. This is done as follows:

1. Calculate l_t and l_b. This requires knowing the grid size, i.e. $64^3, 1024^3$ etc., that indices are being calculated for.

2. Calculate the offset for the start of the level as $GO_I = 2^{k*(l_t-1)}$, and the index of the points in its level as $LO = ZH_I - GO$.

3. To bring the local index LO into the sequence of elements associated with its level compute $d = LO/(2^k - 1) - 1$.

4. The regular z index Z_I is calculated as $Z_I = (LO - d)2^{k*l_b}$

5. The grid index $I(i_0, i_1 \ldots i_{k-1})$ is computed by uninterlacing Z_I.