Recursive Tetrahedral Meshes for Scientific Visualization

Benjamin Gregorski*
Center for Applied Scientific Computing Lawrence Livermore National Laboratory
Center for Image Processing and Integrated Computing University of California, Davis

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. All science and engineering disciplines are facing the same problem: how to archive, transmit, visualize, and explore the massive data sets resulting from modern computing. 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. Today's *impact problems* of science and engineering require new algorithms for the analysis of the massive data sets produced by computational simulations and new sensor technology. The exploration of truly massive data sets requires new techniques in compression, storage, transmission, retrieval, and visualization, as the existing techniques for small data sets do not scale well, or not at all. New systematic approaches are needed to address the interrelated problems of storage, visualization, and exploration of these massive data sets.

In this paper, we describe the use of recursive tetrahedral meshes based on longest edge bisection as a data structure for scientific visualization and its application to the approximation of datasets with material interfaces and interactive isosurface extraction from large volumetric datasets. The rest of this paper is structured as follows. Section 2 discusses previous work using longest edge bisection in the field of scientific visualization. Sections 3 and 4 detail the basic mesh refinement scheme and the adaptive refinement of the mesh. Section 5 discusses the application of this structure in the approximation of datasets that contain material interfaces. Section 6 describes its application to fast isosurface extraction from large datasets. Sections 7 and 8 give implementation details and Section 9 describes areas of future research.

2 Previous Work

The refinement of a tetrahedral mesh via longest edge bisection and its application in scientific visualization applications is described in detail in several papers. In Zhou et al. [1], 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 error tolerance. In addition they described a method for preserving the topology of isosurfaces extracted from coarser levels of the representation relative to the full resolution isosurface. For larger datasets, the fine-to-coarse algorithm 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 [2]. 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 Gerstner [3] and by Gerstner and Rumpf [4]. Both of these algorithm extract the isosurfaces from the mesh in a coarse-to-fine manner. In Roxborough and Nielson [5], 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 and approximate the original data within a user defined error tolerance.

*{gregorski1}@llnl.gov

3 Longest Edge Bisection

In this section we review longest edge bisection and establish terminology. In this scheme, a tetrahedron is described by a *level* and a *phase* with 3 phases at each level. The refinement begins at level 0, phase 0 with an initial configuration of a cube divided into 6 tetrahedra around the major diagonal of the cube. Figure 1 illustrates the three phases of the refinement process. Phase 0 introduces a new vertex at the center of the cube, phase 1 introduces a new vertex at the center of a face, and phase 2 introduces a vertex at the midpoint of an edge. After three refinements, the level is incremented by 1 and the original cube has been split into eight smaller cubes. This is equivalent to one refinement of an octree data structure, see [6] and [7]. After *n* refinements, the phase is $n \bmod 3$ and the level is $\lfloor n/3 \rfloor$. The number of refinements is called the *refinement level*. In the following sections, level refers to the value $\lfloor n/3 \rfloor$ not the *refinement level*. The *split edge* of a tetrahedron is the edge that is bisected when the mesh is refined. In phase 0 the split edge is the major diagonal of a cube, in phase 1 it is the diagonal of a cube's face and in phase 2 it is an edge of a cube. In each phase, a tetrahedron is subdivided into two *children* by introducing a vertex at the midpoint of the split edge and forming a triangular face with this vertex and the two vertices not on the split edge. This new face is shared by the two children. The new vertex at the midpoint of the split edge is called the *split vertex*.

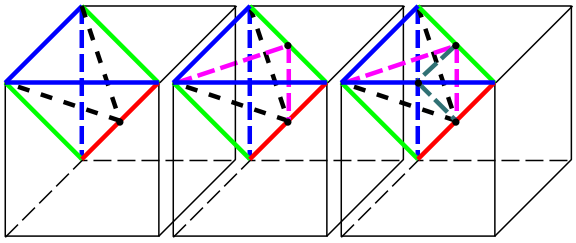

Figure 1: Three phases of refinement for a single tetrahedron of the initial configuration.

3.1 Diamonds

Tetrahedra are grouped into diamonds to simplify the refinement process. When a tetrahedron is split, all the tetrahedra that share its split edge must also be split in order to avoid introducing cracks

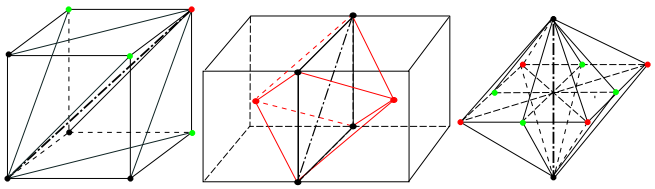

Figure 2: Shapes of phase 0, phase 1, and phase 2 diamonds. The refinement edge is the bold dotted and dashed line.

Type	Tetrahedra(phase,level)	Parents	Children
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 1: Number, phase, and level of tetrahedra, parents, and children for each type of diamond. l is the *level* of the diamond.

and T-intersections into the mesh. A group of tetrahedra that share a split edge is called a *diamond*. The *split edge* and *split vertex* of a diamond are defined to be the common split edge and split vertex of its tetrahedra. All diamonds in the mesh can be uniquely identified by their split edge or split vertex. In later sections, a diamond will be referenced by its split edge or split vertex. Phase 0, phase 1, and phase 2 diamonds are shown in Figure 2. A phase 0 diamond is a cube divided into six tetrahedra around its major diagonal, a phase 1 diamond consists of four tetrahedra around a face diagonal of a cube, and a phase 2 diamond consists of eight tetrahedra around an edge of a cube. By grouping tetrahedra into diamonds, we can easily locate all of the tetrahedra around a split edge. 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 1 lists the number of tetrahedra, their phase, and level for each diamond type.

Each diamond contains the following information: the phase and level of the diamond; the type of the diamond based on its split edge; if the diamond is a leaf, on a boundary of the root diamond, or the root diamond. The type of the diamond is used to efficiently encode the structure of the mesh (Section 7) including the location of parent and child diamonds (Section 3.2). The type of a diamond is determined by its split edge (SV_0, SV_1), where SV_0 and SV_1 are the vertices on the split edge. There are 26 different directions for the split edge of a diamond; 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. For example, the split edge ((64,64,0),(64,0,64)) gives the vector (0,-64,64). This corresponds to the direction vector (0,-1,1). The 26 directions can be defined by the different combinations of a vector (i, j, k) when the entries are restricted to -1, 0, and 1. The vector $(0, 0, 0)$ is not a valid split edge.

3.2 Parent and Child Diamonds

Given a diamond D, the parents of D are defined to be the diamonds that must be split to create D's tetrahedra. The parents of a diamond are diamonds from the previous refinement level. The parent information is summarized in Table 1. Figures 3-5 show the parents for each diamond type. In Figures 3-5, the split edge is $(sv0, sv1)$, the split vertex is SV, and the parents are shown as $p0, p1, \ldots$. A diamond is referenced by its split vertex.

For a diamond D, the diamonds that are created when D is split are called D's *children*. Figures 6-8 show the children for the diamond types. Similarly, the *grandchildren* of D are the children of D's children.

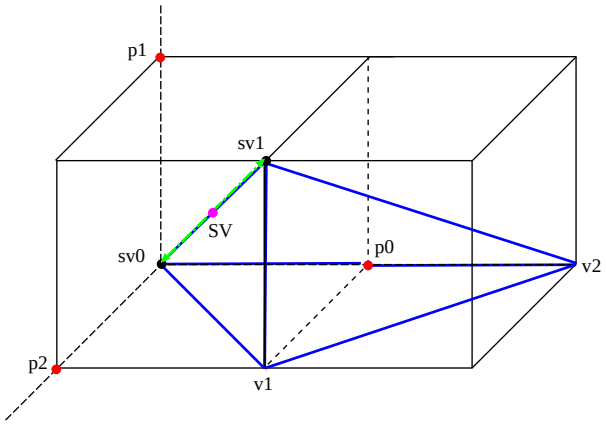

Figure 3: Phase 0 diamond and its phase 2 parents. The tetrahedron $(sv0, v2, sv1, v1)$ is one of the tetrahdra in the phase 2 diamond whose split vertex is $p0$.

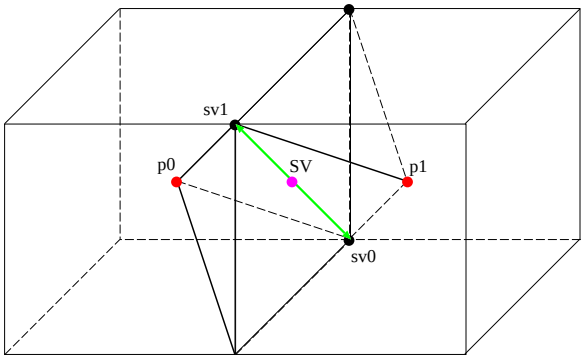

Figure 4: Phase 1 diamond and its phase 0 parents.

4 Split/Merge Refinement

The mesh supports the dual queue split/merge refinement strategy similar to that described by Duchaineau et al. [8]. In the applications described in Secions 5.2 and 6, this mesh structure is used to construct an approximation of a volumetric dataset. The tetrahedra which constitute the current approximation of the volume, and then are used to visualize the dataset.

The *current mesh* is a set of tetrahedra, possibly from different levels of the hierarchy, that approximates the volume dataset to within a certain error bound. This set of tetrahedra is free from cracks and T-intersections, and defines a C^0 continuous, piecewise linear approximation to the original data. Since a diamond represents a collection of tetrahedra which share a common refinement edge, it is possible for a diamond to have some but not all of its tetrahedra in the current mesh. The current mesh is generated using two queues. The *split queue* holds the diamonds containing the tetrahedra of the current mesh. The *merge queue* holds the diamonds that have been split whose children have not been split (i.e. diamonds that have children but no grandchildren). The diamonds in the queues are ordered by an application defined error value which measures how well the tetrahedra in a diamond ap-

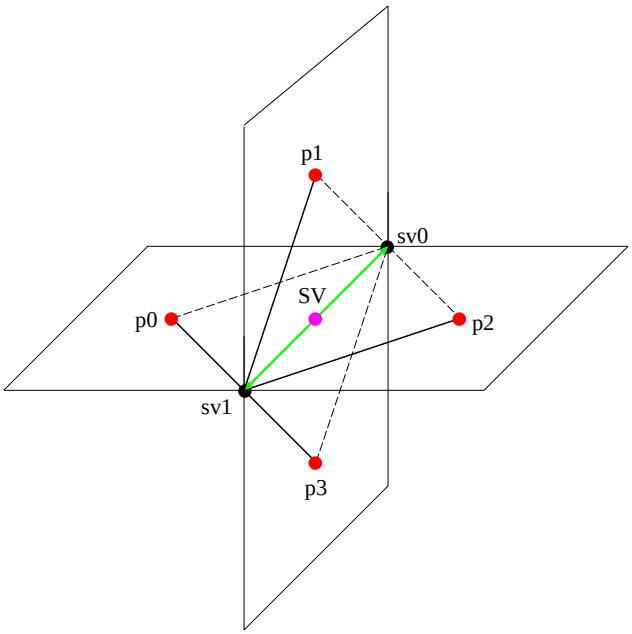

Figure 5: Phase 2 diamond and its phase 1 parents.

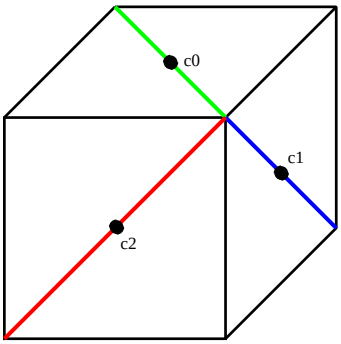

Figure 6: The children of a phase 0 diamond are the phase 1 diamonds located on the faces of a cube. Three of the children are shown here.

proximate a region of the volume for visualization purposes. Larger error values indicate a poorer approximation.

The split queue is initialized with the base configuration of six tetrahedra (the root diamond), and the merge queue is empty. Given an error tolerance E, the following steps are taken to construct an approximation:

1. Diamonds that do not contain regions of interest are marked as *empty*; they are assigned an error of zero. Errors values 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. *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 has elapsed.

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

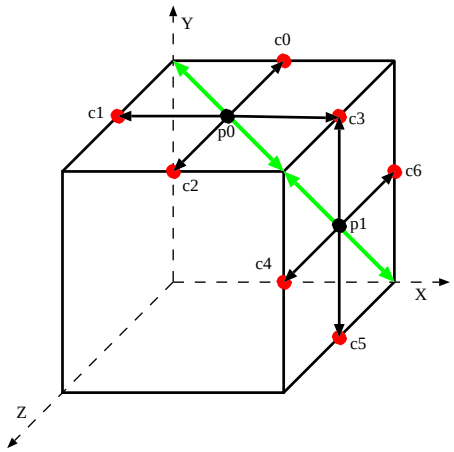

Figure 7: The children of a phase 1 diamond are the phase 2 diamonds located on the centers of the edges of the faces containing the split edge. Diamond p0 has children c0 - c3, and diamond p1 has children c3 - c6.

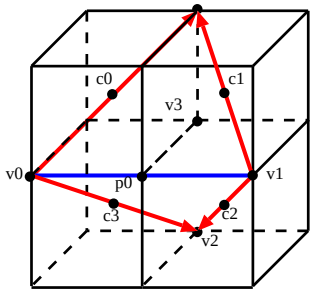

Figure 8: The children of a phase 2 diamond are the phase 0 diamonds from level $i + 1$ that touch the diamond's split edge. Four of the children for diamond p0 are c0 - c3. The other four are the centers of the octants that come out of the page.

A diamond D is split by splitting all of its tetrahedra and inserting the child tetrahedra into the split queue. A tetrahedron is placed in the split queue by creating an entry for its diamond and adding the tetrahedron to the diamond. When some of the tetrahedra in D do not exist (i.e. they are not in the current mesh), it is necessary to create them before D can be split. This is done 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.

A diamond D is merged by merging all of its tetrahedra and then 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 split queue. If so, the diamond is removed from the split queue. Lastly, D'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.

5 Representing Datasets Containing Material Interfaces

5.1 Introduction

Computational physics simulations 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 data sets contain special physical features such as material interfaces, physical boundaries, or thin slices of material that must be preserved when the field is simplified.

Material interfaces embedded in the meshes of computational data sets are often a source of error for simplification algorithms because they represent discontinuities in the scalar or vector field over mesh elements. Representing material interfaces explicitly allows separate field representations to be used for each material within 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. In order to ensure that these features are preserved, the simplified version of the data set is 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 data sets 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, since it is cumbersome to write algorithms for each specific type of dataset, the simplification algorithm must be able to deal with a wide range of possible input meshes. In order to effectively do this, the algorithm must be able to efficiently represent material interfaces and other explicit discontinuities.

The adaptive refinement of the tetrahedral mesh described in previous sections is used to construct a multiresolution representation of a computational dataset with explicit representation and approximation of material interfaces. Each tetrahedron approximates a region of the dataset. Associated with each tetrahedron is an approximation to the material interfaces that are wholly or partially contained within it and an approximation to the field variables (i.e. density, pressure, etc.) defined over the region of the volume approximated by the tetrahedron. In order to accomodate a large variety of input meshes, the input datasets are resampled at the vertices of the tetrahedral mesh in a manner that preserves user-specified as well as characteristic features in the data set and approximates the dependent field values to within a specified tolerance.

5.2 Material Interfaces

A material interface defines the boundary between two distinct materials. Figure 9 shows an example of two triangles crossed by a single interface (smooth curve). This interface specifies where the different materials exist within a cell. Field representations across a material interface are often discontinuous, and can introduce a large amount of error to cells that cross it. Instead of refining an approximation substantially in the neighborhood of an interface or attempting to place vertices and faces of cells directly on the interface, the discontinuity in the field is 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, our simplification algorithm does not need to refine regions in the spatial domain with a large number of tetrahedra.

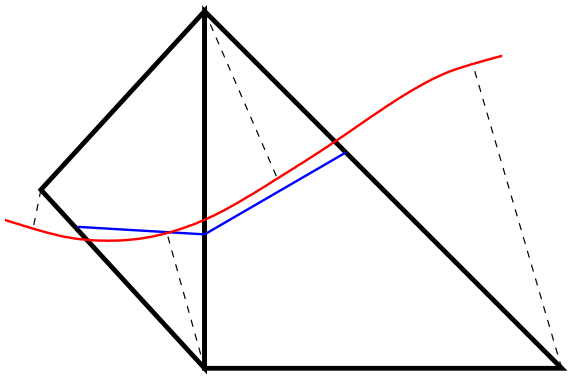

Figure 9: True and approximated interfaces.

5.2.1 Extraction and Approximation

For the datasets with which we are working, 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 [9]. (The volume fractions resulting from numerical simulations indicate what percentages of which materials are present in each cell.) This 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 tetrahedron, a material interface is approximated with the zero set of a signed distance function. Each vertex of a tetrahedron is given 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 finding the point $\mathbf{V}_i$ 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 tetrahedra in the mesh will contain a large number of interface triangles, and a fine tetrahedra will contain a small number of interface triangles. The signed distance values are computed as the mesh is subdivided. 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 (i.e. all of the tetrahedra in the diamond being split). If those tetrahedra do not contain any interfaces, no signed distance values needs to be computed for the new vertex.

In Figure 9, 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. 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. The situation in three dimensions is analogous.

Figure 10 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

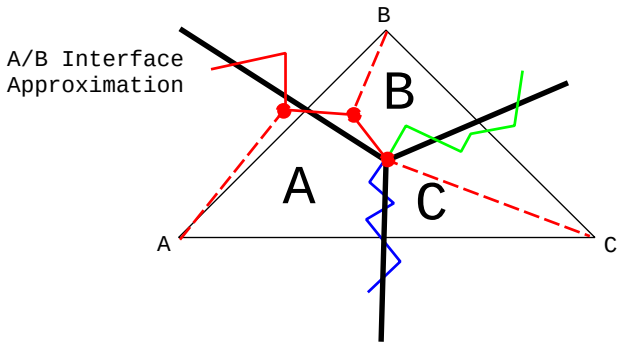

Figure 10: Triangle with three materials (A, B, and C) and three interfaces.

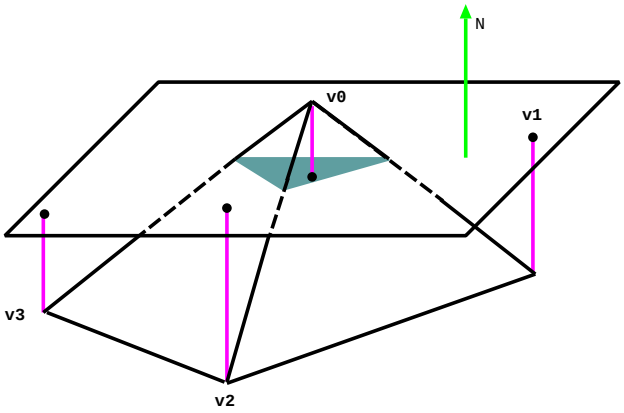

Figure 11: Tetrahedron showing signed distance values and the corresponding boundary approximation.

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 a tetrahedron. The distance function is a linear function $f(x, y, z) = Ax + By + Cz + D$. 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 11 shows a tetrahedron, a material interface approximation, and the signed distance values d_i for each vertex V_i. The approximation is shown 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. We note that a vertex has at most one signed distance value for each interface. This ensures that the interface representation is continuous across tetrahedron boundaries. If a tetrahedron does not contain a particular interface, the signed distance value for that interface is meaningless for that tetrahedron. Given a point **P** in an interface triangle and the interface 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 of a tetrahedron is the maximum of these distances, considering all the interfaces within the tetrahedron.

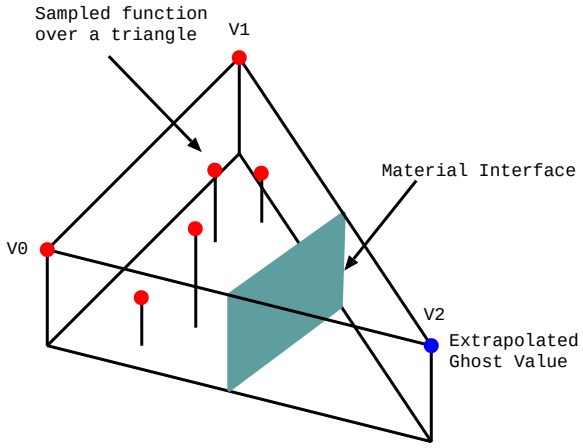

Figure 12: 2D Ghost Value Computation

5.2.2 Discontinuous Field Representations

Tetrahedra that contain material interfaces typically also have discontinuities in the fields defined over them. For example, the density field over a tetrahedron 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 over the tetrahedron 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 tetrahedron into two distinct cells at the material interface. In Figure 13, the triangle would be divided into a quadrilateral for material A and a triangle for material B. The disadvantage of this method is 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 tetrahedral mesh.

The discontinuous field is represented by constructing a field for each material in the tetrahedron. In order to do this, each of the vertices in the tetrahedron must have distinct field values for each material present in it. When material interfaces are present, field values for a given material do not exist at all of a tetrahedron's vertices. Since linear approximation over a tetrahedron requires four field values at the vertices, extra field values need to be extrapolated to perform the interpolation. These extrapolated values are called *ghost values*. A ghost value is an educated guess of a field value at a point where the field does not exist. This is illustrated in Figure 12 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. When the field approximation error for a cell (i.e. trianle or tetrahedron) is computed, the separate field representations, built using these ghost values, are used to calculate an error for each distinct material. The field approximation error for the cell is the maximum of these per-material errors.

5.2.3 Computation of 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 13. 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

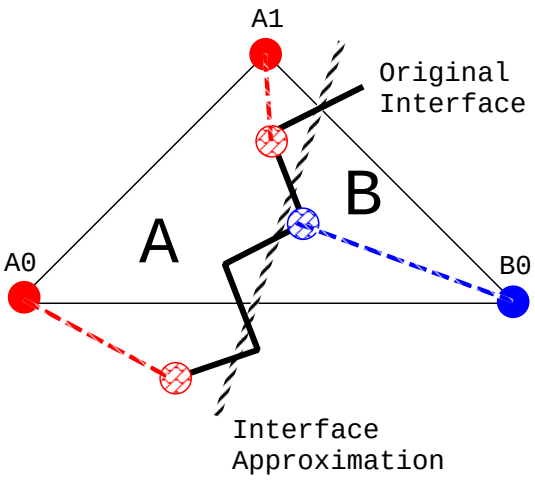

Figure 13: Ghost value computation for a triangle containing two materials. Ghost values for material B are computed at $A0$ and $A1$ and a ghost value for material A is computed at $B0$.

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. The ghost value computation is performed when the vertex is inserted during the mesh refinement process. The ghost values for a vertex **V** are computed as follows:

1. For each material interface present in the tetrahedra of the diamond being split, find a vertex V_{min} in a triangle mesh representing an interface with minimal distance to **V**. (In Figure 13, 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 data set on the far side of the interface at V_{min} and use this as the ghost value at **V**.

Only one ghost value exists for a given field and material at the vertex **V**. This ensures that the field representations are C^0 continuous across cell boundaries. In Figure 10, 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 at A_0. This computation assumes that the field remains constant on the other side of the interface. Alternatively, a linear or higher order interpolation technique, taken from the simulation used to generate the dataset, can be used to extrapolate the ghost values.

5.3 Multiresolution Representation of Datasets

Given a dataset and triangle mesh representations for the material interfaces, our algorithm constructs a multiresolution representation as follows:

1. Our algorithm starts with a base mesh of six tetrahedra and associates with each one the interface triangles that intersect it.

2. The initial tetrahedral mesh is first subdivided so that the triangle meshes describing the material interfaces are approximated within a certain tolerance. At each subdivision, the material interface triangles lying partially or entirely in a tetrahedron are associated with the tetrahedron's children; approximations for the triangles in each child tetrahedron are constructed, and interface approximation errors are computed for the two new child tetrahedra, and ghost values are computed for the split vertex.

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

5.4 Error Metrics

Each tetrahedron has two associated error values: a field error and a material interface error. In order to calculate the field errors for a leaf tetrahedron in our mesh hierarchy, we assume 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. For a given field, we assume that it is possible to bound the difference between the representation over one of our leaf tetrahedra and the representation over each of the native data elements with which the given tetrahedron intersects. The error for the given field in the given tetrahedron is the maximum of the errors associated with each of the intersecting native data elements. Since tetrahedra are grouped into diamonds, the error values ,i.e. interface error and field error, associated with a diamond are the maximum of the error values of its tetrahedra.

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

$$e_T = \max\{e_{T_0}, e_{T_1}\} + |z(v_c) - z_T(v_c)|, \tag{1}$$

where e_{T_0} and e_{T_1} are the errors of the children; v_c is the split vertex; $z(v_c)$ 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:

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

where v_0 and v_1 are the vertices of the parent's split edge.

This error is looser than the bound computed directly from the data. It has the advantage that the error associated with a tetrahedron bounds the deviation from the original representation and the deviation from any intermediate resolution. This error is *nested* or monotonic because the error of a child is guaranteed not to be greater than the error of its parent.

The material interface error associated with a leaf tetrahedron is the maximum value of the errors associated with each of the material interfaces in it. 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 it . 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 tetrahedron 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 our approximated interfaces. A tetrahedron that does not contain any material interfaces has an interface error of zero.

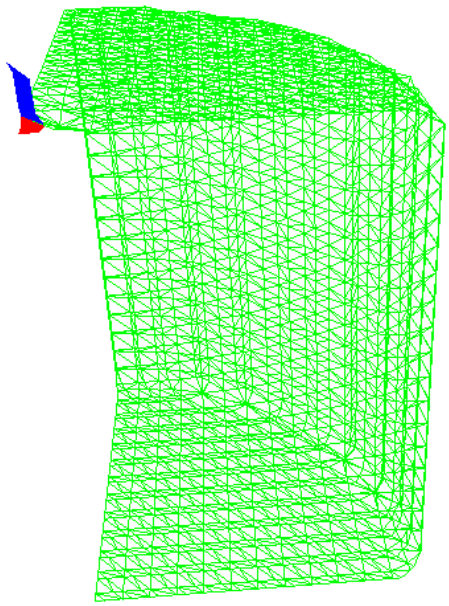

Figure 14: Original triangular meshes representing material interfaces.

5.5 Results

Our sample dataset is 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 a total of 53248 nodes. 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 14 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 15 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 than 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 tetrahedra containing material interfaces is 2.84, and the average field error for these tetrahedra 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 15 and 16 compare the density fields generated using linear interpolation of the density values and explicit field representations on either side of the material interface. 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 tetrahedra 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 render the separate field prepresentations for each material that the cutting plane intersects.

Figure 16 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 vertical sections of the block where the tetrahedra approximate a region that contains the block and empty space. In these tetrahedra, the use of explicit representations for the discontinuities leads to an exact representation of the density field. The corresponding field representations using linear interpolation, shown in Figure 15, captures the discontinuities poorly. Furthermore, Figure 16 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.

6 View-dependent Extraction of Iso-Surfaces

Isosurface extraction is a fundamental method for visualizing volume datasets. Traditionally, with smaller and simpler data sets, researchers developed *in-core* isosurface extraction techniques that work well on small or medium-scale data sets. These techniques can quickly generate isosurfaces, and treat each isosurface independently. Today's scientific and engineering problems require a different approach to address the massive data problems in organization, storage, transmission, visualization, exploration, and analysis.

Surface based level-of-detail techniques such as [10] and [11] extract the isosurfaces and build multiresolution models from these surfaces. 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 techniques such as those developed by Lindstrom [12], [13] 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. Processing and interactively visualizing these types of isosurfaces requires algorithms such as those developed by Duchaineau [14], [15], that combine multiresolution representations, compression, and view-dependent optimizations. Furthermore, these surface based techniques are not suitable for visualizing volumes that contain a large number of interesting isosurfaces because they must extract all of the interesting surfaces which would take far too much disk 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.

The refinement of a tetrahedral mesh via longest edge bisection is utilized to build a multiresolution hierarchy of the volume dataset. This multiresolution representation is used to generate isosurfaces "on-the-fly" using a view-dependent error measure and an adaptive refinement algorithm to generate and explore the isosurfaces. In this algorithm, we combine coarse-to-fine and fine-to-coarse refinement schemes for the tetrahedral mesh to create an adaptively refinable mesh. This adaptive mesh supports a dual queue split/merge al-

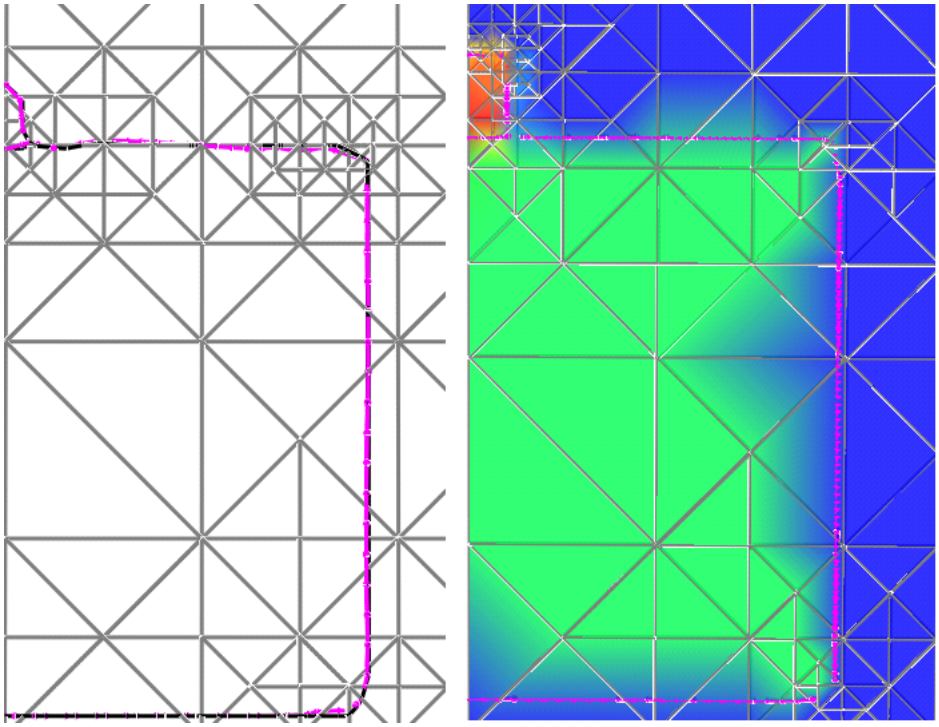

Figure 15: Cross section of the tetrahedral mesh. The left picture shows the original interfaces and their approximations. The picture on the right shows the density field using linear interpolation.

gorithm similar to the ROAM system [8] for terrain visualization. It has fast coarsening and refinement operations which allow for 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 patterns dictated by the mesh refinement.

Our algorithm is divided into two phases: a preprocessing phase that collects general information for each diamond, and a run-time phase that uses this information to generate the isosurfaces. In the preprocessing phase, we compute the following information for the diamonds (Section 3.1):

1. The isosurface approximation error of the region enclosed by the diamond. (Section 6.2)

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 gradient vector at the split vertex of the diamond. (Section 3.1)

This precomputed information is used to drive the run time mesh refinement.

At run time, the split/merge refinement algorithm (Section 4) 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 drawn through the lower resolution dataset deviates from the finest level isosurface. The error tolerance is measured in pixels on the view screen. The isosurface is extracted from the tetrahedra in this lower resolution representation using linear interpolation. The precomputed gradient vectors are used to shade the isosurface.

6.1 Mesh Refinement and Isosurface Extraction

As described in Section 4, the mesh supports the dual queue split/merge refinement strategy [8]. This strategy provides more frame-to-frame coherence than a coarse-to-fine 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, a large number of splits and merges do not have to be performed.

For the split merge refinement process described in Section 4, the error metric E is a view-dependent measure of the screen space projection deviation between the real isosurface and the isosurface extracted from the tets in the current mesh. A diamond that does not contain the isosurface of interest is considered *empty*. In addition, diamonds that are outside of the view-frustum are considered *invisible*. Similar to empty diamonds, invisible diamonds are never split and are the first to be merged. The final isosurface is extracted from the visible, non-empty diamonds in the split queue.

The isosurface approximation error, min and max values, and gradient vector at the split vertex are precomputed for each diamond in the hierarchy. The isosurface errors are compressed on a logarithmic scale and represented in six bits. There is one set of error values for each level of the mesh (Section 3). The gradient vectors are quantized on a unit cube using fourteen bits. In addition, we use an iterative relaxation process to smooth the gradient

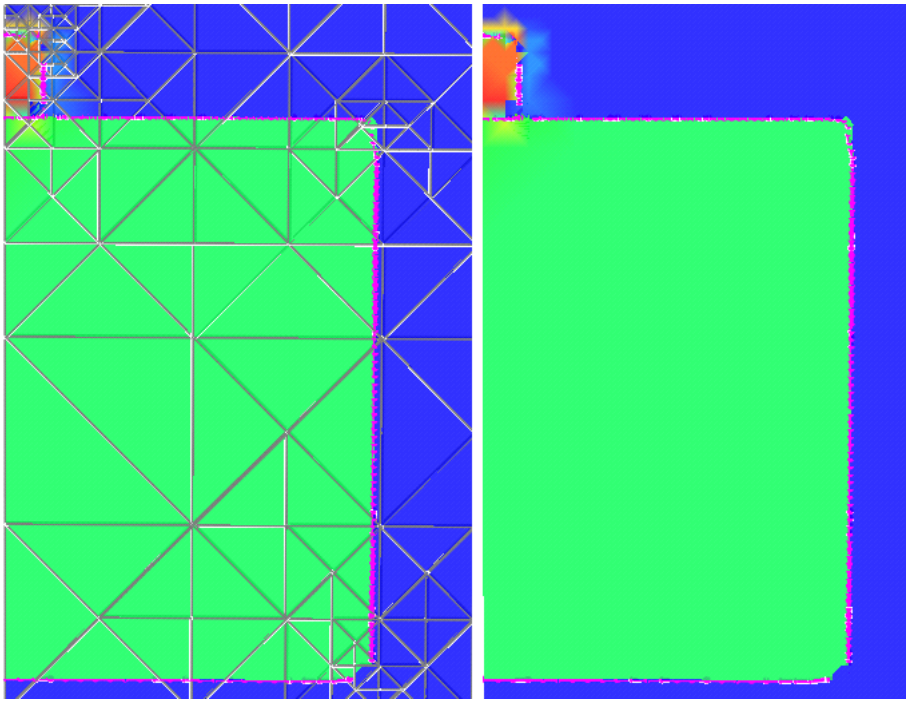

Figure 16: Cross section of a density field approximated using explicit interface representations and separate field representations. The left picture shows the field along with the approximating tetrahedral mesh. (interface error = 0.15).

vectors which are computed directly from the byte datasets we use. The min and max values for a diamond D are compressed in relation to a diamond S that completely contains D. This is illustrated in Figure 17. A diamond S that completely surrounds D can be found by examining the two diamonds whose split vertices are the vertices of D's split edge.

We assume that the data points of the input dataset lie on a $(2^n+1)\times(2^n+1)\times(2^n+1)$ grid. In this setting, each data point corresponds to the split vertex of some diamond. We also assume that the dataset obeys periodic boundary conditions (i.e. the value at index (2^n) equals the value at index 0). For the dataset used in our tests this is a valid assumption. For each diamond, the precomputed information is stored in three bytes. Including the original data itself which is byte data, this gives us thirty two bits of information for each input data point.

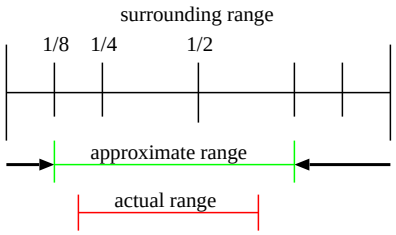

Figure 17: The min/max values of a diamond are encoded relative to the min and max values of an enclosing diamond using 4 bits to encode 0/8, 1/8, 1/4, or 1/2 of the enclosing interval.

When the isovalue changes, the new isosurface can be extracted by starting at the root diamond or starting from the current mesh. In the first case, the split and merge queues, hash tables, and caches are

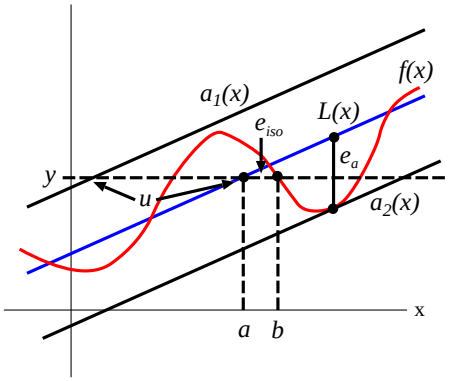

Figure 18: Isosurface error calculation in 1D.

emptied and initialized with the root diamond. The split/merge refinement is then started from this initial configuration. In the second case, the diamonds in the split and merge queues must be checked to determine if they contain the current isovalue. Diamonds that do not contain the isovalue are marked as empty and given an approximation error of zero. Once these diamonds have been marked, the split/merge refinement continues from this new configuration. The effectiveness of both of these methods depends on the locality of the old and new isosurfaces in the mesh hierarchy. Starting from the current configuration makes sense if the are close together, and starting from the top makes sense if they are far apart.

When a tetrahedron is added to the split queue, the isosurface is extracted 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. This caching method duplicates normals and vertices along edges. Its advantage is that it has better memory coherence than hash table based caches which cache the vertices and normals on a per-edge basis (See Gerstner and Rumpf [4]). In each frame the mesh is drawn simply by traversing the triangle cache. By using the split/merge refinement scheme and the geometry cache, we only have to lookup and compute the changes between consecutive frames.

6.2 Error Metrics

Each diamond in the mesh has an associated approximation error, isosurface error, and view-dependent error. The approximation error ae_T for a tetrahedron T is the maximum difference between a linear approximation of the scalar values at the vertices of T and the actual data values for the points inside T and on its boundary. The approximation error ae_D for a diamond D is the maximum of the approximation errors of its tetrahedra. Leaf tetrahedra and leaf diamonds have an approximation error of 0.

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 18 for the one-dimensional case. The original isosurface is $f(x)$ and it is approximated by $l(x)$. The upper and lower bounds on the approximation, given by the approximation error ae, are $a1(x)$ and $a2(x)$. 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 $y = f(b)$. The error in the isocontour is given by:

$$ie = |a - b| \tag{3}$$

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

$$u = ae/m, \tag{4}$$

where ae is the approximation error and m is slope of the linear approximation l. As the slope of l increases, f must converge to a vertical line and will be approximated with increasing accuracy by l, implying that the approximation errors get smaller and smaller. As the slope of l decreases, the isocontour approximation a and the true isocontour b can be far apart even if ae 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 its linear approximation. 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:

$$ie_T = ae_T/\|\nabla T\|, \tag{5}$$

The isosurface error ie_D for a diamond is the maximum of its tetrahedras' isosurface errors. The view-dependent error of a diamond is the projection of its isosurface error onto the view screen. This is done by creating a sphere at the diamond's split vertex of radius ie_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 [16] , Lindstrom [17], and Luebke [18]. All of these error metrics are easily incorporated into our refinement strategy.

6.3 Memory Layout

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

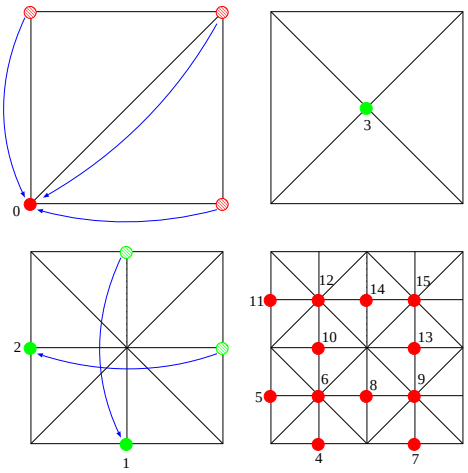

Figure 19: New data points required at each refinement level for 2D edge bisection. The lower right mesh is the lower left mesh after two refinements.

Figure 19 shows how the mesh refinement algorithm accesses the data. Starting with the root configuration in the upper left, the dots indicate which data points are introduced at each refinement step. As the mesh is refined, the dataset is accessed along grids whose points are separated by decreasing powers of two. This data layout scheme and its performance benefits are detailed in [19] and [17]. 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 of our algorithm are stored on disk in this manner. They are mapped to main memory at run time. This allows us to keep in memory the data that is currently being used by the split/merge process and the isosurface extraction process.

6.4 Results

We have tested our methods on an SGI Onyx with 44-250 MHZ R10000 processors and IR2 graphics boards. At run time the algorithm uses one processor and one graphics pipe. The preprocessing was done in parallel on an SGI Onyx.

Our test dataset is the Gorden Bell Prize winning simulation of a Richtmyer-Meshkov instability in a shock tube experiment [20]. A full resolution isosurface of the mixing interface produces a mesh with 460 million triangles. The dataset consists of 274 time steps with each time step divided into a grid of 8x8x15 bricks where each brick consists of 256x256x128 byte data values for a total time step resolution of 2048x2048x1920 byte data values. In our examples, we are looking at isosurfaces of entropy values calculated as two fluids mix over time. Our examples are from 512^3 datasets. Figure 20 shows a view of the turbulent mixing in a region of high entropy at time step 200. Fixing this viewpoint, zooming out, and drawing the isosurface outside of the view-frustum reveals how the triangulation and the mesh change as we move away from the point of interest. This is shown in Figure 21. Figure 22 shows a closeup view of the surface of entropy value 186 at time step 273. The ability to zoom in on regions of the dataset and refine the isosurface accordingly allows one to closely inspect the features of the mixing process. Figure 23 is created from Figure 22, by turning off view-

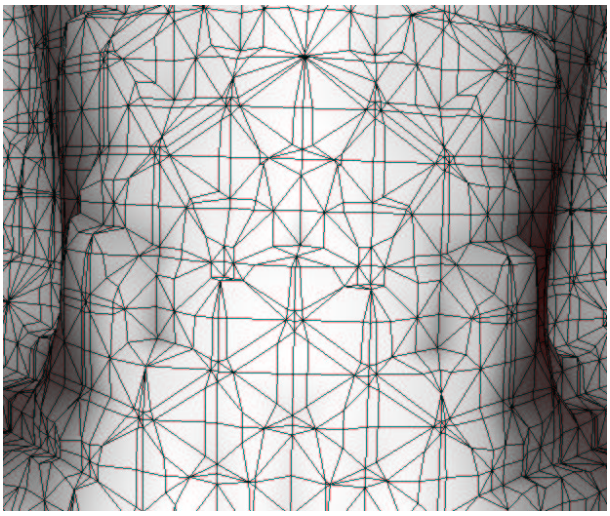

Figure 20: Close up inspection of turbulent mixing in Richtmeyer-Meshkov instability(time step 200/273). Iso value (Entropy) = 180.

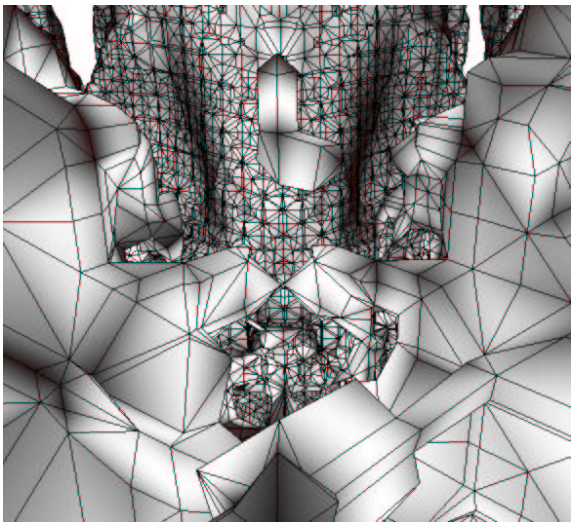

Figure 21: Zoomimg out from the viewpoint in Figure 20 reveals the triangulation at coarser levels of the hierarchy.

dependent refinement, zomming out and drawing the isosurface including those parts culled by the view frustum. This shows how the mesh and triangulation adaptively refine around the viewpoint.

Table 2 shows the performance measurements for the visibility culling and priority recomputation, the split merge refinement, and the rendering sections of our algorithm. The time for culling and priority recomputation depends on the number of computations and the memory performance of the hash table. We can perform about 700K - 1.1e6 computations a second. Rendering at 7 FPS and allowing at most half of the frame time for culling and priority recomputation, we are allowed 50K - 78K computations per frame. Currently, we recompute the priorities for all visible, non-empty diamonds in the queues. This is an expensive operation and can be improved using hierarchical, defered priority recomputation. The split/merge performance is determined by the number of recursive splits and the coherency of the data access. Merges only have to look at children and parents and perform $O(1)$ lookups to find them. Splits look at children and parents and may have to look at $O(n)$ diamonds where n is the number of levels in the tree. To test the performance of these operations, we fixed the time for doing splits and merges to 0.01s. The algorithm performs around 1000 - 4000 updates per second or 10 - 40 updates per timeslice. The time for drawing depends on the number of elements drawn. In immediate mode, the SGI graphics system can draw 50K triangles at a rate of 1.05e6 triangles per second which is about 20 frames per second. This gives us about 66K triangles per frame at 7 FPS. The slowest portion of the algorithm is the culling and priority computation. Limiting the triangle count in the extracted isosurface to around 50K triangles gives us roughly 65K - 85K diamonds in the queues. This allows us to render 5 - 7 frames per second or about 250K - 350K triangles per second. In practice when exploring inside a dataset and looking closely at features, the maximum number of triangles is not used because large regions are culled. Our results are similar to previous results on isosurface extraction using edge bisection, and we show that they can be achieved on large datasets.

Time(s)	Operation	# Elements	Elem/Sec
0.07	Cull/Priority	50K - 78K	700K - 1.1e6
0.06	Drawing	66K	1.05e6
0.01	Split/Merge	10-40	1000 - 4000

Table 2: Timings results for algorithm sections.

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 to compute the tetrahedron vertices, parents and children of a diamond are all powers of two relative to the split vertex of the diamond. 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.

A diamond is represented by an (i,j,k) index. 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 case,$(0,-1,1)$ 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}. Since this factor is always a power of 2, computing the indices can be done with shifts and adds. The split edge encodings are stored in a lookup table and accessed at run time based upon the type (Section 3.1) 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, tetrahedron vertices, and children of a diamond are also encoded relative to the split vertex of the diamond in the same manner that the split edge is encoded. These encodings are stored in a lookup table based on the type of the diamond. For any diamond, the (i,j,k) index for a parent, child or vertex 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.

In the case of adaptively resampled grids, such as those used in the material interface approximation, the vertices of the initial cube can be assigned indices on a $(2^{31}+1)\times(2^{31}+1)\times(2^{31}+1)$ grid.

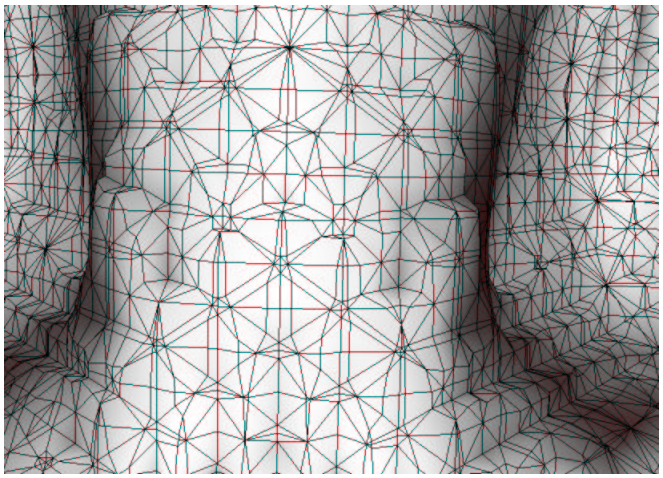

Figure 22: Closeup view of a mixing feature at time step 273. Entropy value = 186, Iso error = 0.5. This shows at a fine resolution the smoothness of the flow and the dimpled shape created by the mixing.

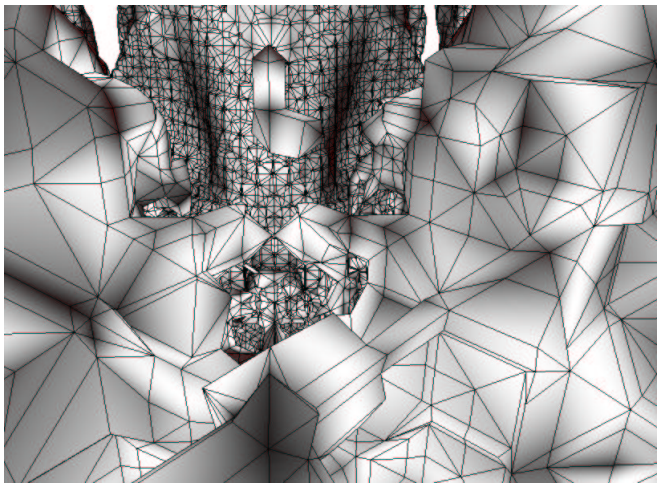

Figure 23: Zoomed out view of Figure 22 shows the refinement of the mesh structure in the vicinity of the viewpoint and away from the viewpoint.

While this theoretically limits the maximum size of the adaptive grid, in practice the grid will ever be subdivided the ninety three times necessary to run out of bits in the indices.

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 possible error values. Each entry in the bucket corresponds to a diamond whose error falls within the bucket's range. In the case of isosurface extraction, the error corresponds to the projected screen space error as measured in pixels on the screen. For building multiresolution representations of datasets with material interfaces, the error can be the interface approximation error or the field approximation error. The buckets are not sorted 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 (i, j, k) 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 error values. 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 error would require locating the bucket that the diamond is in, and then traversing the bucket to get the appropriate entry.

The data structures are illustrated in Figure 24. 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

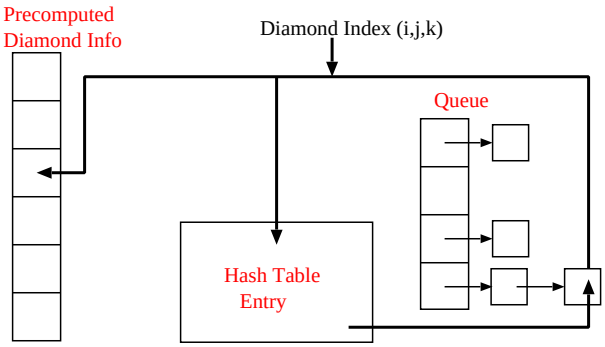

Figure 24: Relationship between precomputed data, queue entries, and queue hash table.

flags indicating which of its tetrahedra are actually in the current mesh. These flags are called the diamond's *tetrahedron flags*. The reason for these flags is illustrated for the 2D case in Figure 25. The mesh has four diamonds $d0 - d3$. Diamond $d1$ has two triangles that are both in the mesh. Diamond $d2$ has two triangles only one of which is in the current mesh. The triangle not in the mesh is shown with the dashed lines. The diamond's tetrahedron flags are used to record this information. Each bucket entry in the split and merge queues stores the diamond's level, (i, j, k) index, error values, *invisible* and *empty* bits where applicable and application specific diamond information.

9 Future Work

9.1 IsoSurface Extraction for Time-Varying Data

The work on isosurface extraction was done only for static data. The visualization of time varying data presents an even bigger challenge especially for large datasets such as the Richtymyer-Meshkov dataset. Extending our algorithms to time varying data requires

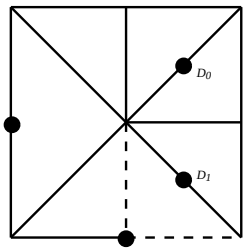

Figure 25: Use of *tetrahedron flags* to indicate which tetrahedra are in the current mesh. Diamond $d1$ has 2 triangles in the mesh. Diamond $d2$ has one triangle in the mesh.

time varying encoding and compression of the data as well as fast decoding and decompression to update the mesh as quickly as possible and to minimize the amount of information that must be loaded from disk. Since the approximation and view-dependent error values are changing between frames, more information needs to be updated at each frame to maintain interactivity. For time-varying data, the tetrahedral mesh in three dimensions becomes a mesh of four dimensional simplices which approximate a region of the volume over time. Future work would focus on efficient encoding/decoding, compression/decompression methods, storage and retrieval schemes. In addition parallel rendering and retained mode rendering algorithms can be used to increase the number of triangles drawn per frame.

9.2 IsoSurface Extraction on Datasets of Arbitrary Size

Currently the input datasets are assumed to be on power of 2 grids. Arbitrary grids can be handled either by dividing them into chunks and operating on the chunks independently or by adaptively resampling the dataset on the vertices of the tetrahedral mesh. Future work would focus on accurately resampling the data to ensure that small features are not lost, and on efficient storage and run-time retrieval schemes for the adaptively sampled data.

9.3 Higher Order Field Representations

Currently our tetrahedra are linear elements. Higher order elements can approximate complex regions using fewer tetrahedra than linear field representations at the expense of increased computation time for preprocessing and visualization. Higher order elements include tetrahedra that define quadratic or cubic interpolation methods and tetrahedra that explicitly represent discontinuities or other interesting features such as those used to construct multiresolution representations of datasets containing material interfaces. Future work would focus on constructing these higher order representations from the input datasets and on efficiently storing and rendering them.

9.4 Parallelization

The algorithm for isosurface extraction can easily be parallelized. The overall split/merge refinement scheme can be threaded so that mesh refinement happens asynchronously from drawing. Synchronization only needs to occur when the geometry is updated from one frame to the next. Large datasets can be divided into chunks, and each chunk can be handled by a different processor. The only information passed between processors are splits/merges that occur on common boundaries. Future work would focus on dividing up datasets and distributing them between processors, parallel rendering, and efficient message passing between processors.

9.5 Isosurface and Volume Rendering Techniques

Since the isosurface geometry is constantly changing from frame to frame, traditional lighting methods do not provide the best images because of popping artifacts. Since modern hardware is optimized for texture based rendering, future work would focus on using texturing for non-photorealistic rendering to enhance features of the dataset, and software and hardware techniques for multiresolution volume rendering.

9.6 Geometric Modeling

Adaptive distance fields have recently been introduced as a new modeling primitive for computer graphics. Current research has focused on using adaptively subdivided octrees to represent the ADF. This presents problems when trying to extract polygonal meshes. Recursive tetrahedral meshes do not have this problem, and polygonal meshes can be extracted using simple isosurface extraction techniques.

REFERENCES

[1] Y. Zhou, B. Chen, and A. Kaufman, "Multiresolution tetrahedral framework for visualizing regular volume data," in *IEEE Visualization '97*, IEEE Computer Society Technical Committee on Computer Graphics, Oct. 1997.

[2] T. Gerstner and R. Pajarola, "Topology preserving and controlled topology simplifying multiresolution isosurface extraction," in *Proceedings Visualization 2000* (T. Ertl, B. Hamann, and A. Varshney, eds.), pp. 259–266, IEEE Computer Society Technical Committee on Computer Graphics, 2000.

[3] T. Gerstner, "Fast multiresolution extraction of multiple transparent isosurfaces," in *Proceedings of EG+IEEE VisSym* (R. P. David S. Ebert, Jean M. Favre, ed.), Annual Conference Series, EG+IEEE, Springer, 2001.

[4] T. Gerstner and M. Rumpf, "Multiresolution parallel isosurface extraction based on tetrahedral bisection," in *Volume Graphics* (M. Chen, A. Kaufman, and R. Yagel, eds.), Computer Graphics Proceedings, Annual Conference Series, pp. 267–278, ACM, Springer, 2000.

[5] T. Roxborough and G. M. Nielson, "Tetrahedron based, least squares, progressive volume models with application to freehand ultrasound data," in *Proceedings Visualization 2000* (T. Ertl, B. Hamann, and A. Varshney, eds.), pp. 93–100, IEEE Computer Society Technical Committee on Computer Graphics, 2000.

[6] H. Samet, *Introduction to Spatial Data Structures*. Reading, Massachusetts: Computer Graphics, Image Processing, and GIS-Addison-Wesley, 1990.

[7] H. Samet, *The Design and Analysis of Spatial Data Structures*. Reading, Massachusetts: Addison-Wesley, 1990.

[8] M. A. Duchaineau, M. Wolinsky, D. E. Sigeti, M. C. Miller, C. Aldrich, and M. B. Mineev-Weinstein, "ROAMing terrain: Real-time optimally adapting meshes," in *IEEE Visualization*

'97, IEEE Computer Society Technical Committee on Computer Graphics, Oct. 1997.

[9] K. S. Bonnell, D. R. Schikore, K. I. Joy, M. Duchaineau, and B. Hamann, "Constructing material interfaces from data sets with volume-fraction information," in *Proceedings Visualization 2000*, pp. 367–372, IEEE Computer Society Technical Committee on Computer Graphics, 2000.

[10] Z. J. Wood, M. Desbrun, P. Schröder, and D. Breen, "Semi-regular mesh extraction from volumes," in *Proceedings Visualization 2000* (T. Ertl, B. Hamann, and A. Varshney, eds.), pp. 275–282, IEEE Computer Society Technical Committee on Computer Graphics, 2000.

[11] M. Gavriliu, J. Carrance, D. E. Breen, and A. H. Barr, "Fast extraction of adaptive multiresolution meshes with guaranteed properties from volumetric data," in *Proceedings Visualization 2001* (T. Ertl, K. I. Joy, and A. Varshney, eds.), pp. 295–302, IEEE Computer Society Technical Committee on Computer Graphics, 2001.

[12] P. Lindstrom, "Out-of-core simplification of large polygonal models," in *Siggraph 20000, Computer Graphics Proceedings* (K. Akeley, ed.), pp. 259–262, ACM Siggraph, ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[13] P. Lindstrom and C. T. Silva, "A memory insensitive technique for large model simplification," in *Proceedings Visualization 2001* (T. Ertl, K. I. Joy, and A. Varshney, eds.), pp. 121–126, IEEE Computer Society Technical Committee on Computer Graphics, 2001.

[14] M. A. Duchaineau, S. Porumbescu, M. Bertram, B. Hamann, and K. I. Joy, "Dataflow and re-mapping for wavelet compression and view-dependent optimization of billion-triangle isosurfaces," in *Hierarchical Approximation and Geometrical Methods for Scientific Visualization* (G. Farin, H. Hagen, and B. Hamann, eds.), Springer-Verlag, Berlin, Germany, (to appear)., 1999.

[15] M. A. Duchaineau, M. Bertram, S. Porumbescu, B. Hamann, and K. I. Joy, "Interactive display of surfaces using subdivision surfaces and wavelets," in *Proceedings of 16th Spring Conference on Computer Graphics, Comenius University, Bratislava, Slovak Republic* (T. Kunii, ed.), 2001.

[16] H. Hoppe, "View-dependent refinement of progressive meshes," in *SIGGRAPH 97 Conference Proceedings* (T. Whitted, ed.), Annual Conference Series, pp. 189–198, ACM SIGGRAPH, Addison Wesley, Aug. 1997. ISBN 0-89791-896-7.

[17] P. Lindstrom and V. Pascucci, "Visualization of large terrains made easy," in *Proceedings of IEEE Visualization 2001* (T. Ertl, K. Joy, and A. Varshney, eds.), IEEE Computer Society Technical Committee on Computer Graphics, 2001.

[18] D. Luebke and C. Erikson, "View-dependent simplification of arbitrary polygonal environments," in *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, (Los Angeles, California), pp. 199–208, ACM Siggraph, ACM SIGGRAPH / Addison Wesley, August 1997. ISBN 0-89791-896-7.

[19] V. Pascucci, "Multi-resolution indexing for out-of-core adaptive traversal of regular grids," in *Proceedings of the NSF/DoE Lake Tahoe Workshop on Hierarchical Approximation and Geometric Methods for Scientific Visualization* (H. Hagen, G. Farin, and B. Hamann, eds.), (Tahoe City, California), Oct. 2000. Available as LLNL technical report UCRL-JC-140581.

[20] A. A. Mirin, R. H. Cohen, B. C. Curtis, W. P. Dannevik, A. M. Dimits, M. A. Duchaineau, D. E. Eliason, D. R. Schikore, S. E. Anderson, D. H. Porter, , and P. R. Woodward, "Very high resolution simulation of compressible turbulence on the ibm-sp system," in *Proceedings of SC99*, Also available as Lawrence Livermore National Laboratory technical report UCRL-MI-134237, 1999.