Isosurface Rendering of Adaptive Resolution Data

BY

Robert S. Laramee
B.S., University of Massachusetts, Amherst, MA, 1997

THESIS

Submitted to the University of New Hampshire
in Partial Fulfillment of
the Requirements for the Degree of

Master of Science
in
Computer Science

December 2000

This thesis has been examined and approved.

Thesis Director, Dr. R. Daniel Bergeron
Professor of Computer Science

Dr. Radim Bartos
Assistant Professor of Computer Science

Dr. Ted M. Sparr
Professor and Chair of Computer Science

Date

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Isosurface Rendering of Adaptive Resolution Data

by Robert S Laramee
University of New Hampshire, December, 2000

We present an algorithm for isosurface volume rendering of adaptive resolution (AR) volume data in order to minimize the time taken for computation and the space needed for storage. Unnecessary computation is avoided by skipping over large sets of volume data deemed uninteresting. Memory space is saved by leaving the uninteresting voxels out of our octree data structure used to traverse the volume data. Our isosurface generation algorithm modifies the Marching Cubes Algorithm in order to handle inconsistencies that can arise between cells of different resolutions.

CHAPTER 1

OVERVIEW

Data analysis in the scientific community often deals with three dimensional data which we call volume data (see section 2.1 Applications on page 3. Volume data is usually sampled or generated at regular intervals as shown in Figure 1.1. Efficient handling of the volume data is critical especially since it tends to come in very large sets. One way of analyzing the volume is to investigate isosurface(s): surfaces of constant scalar value. We use an algorithm called marching cubes [15] to compute these surfaces.

The marching cubes algorithm, conceptually, is a filter, taking as input a volume data set and outputting surfaces of constant scalar value. From an implementation standpoint, it does so by introducing a cube topology onto the volume data and processing one cube at a time. Previous research has shown that 30% to 70% of the rendering time is spent processing empty cubes [26]. Hence a standard optimization technique uses an octree [27] to store the volume data. Each cube is stored as a leaf node and each internal node stores the minimum and maximum data values of all its children. The octree is traversed, examining each node to see if it contains a portion of the surface within the encompassed volume.

The standard octree representation can be converted to a multiresolution (MR) [27] representation by also storing common vertex values in the internal nodes. Such an MR representation can be used to produce isosurfaces at different resolutions by simply truncating or culling the octree at a particular level.

Instead of storing a full multiresolution octree representation of the volume data set (described in section 2.3), it is often desirable to create an adaptive resolution (AR) rep-

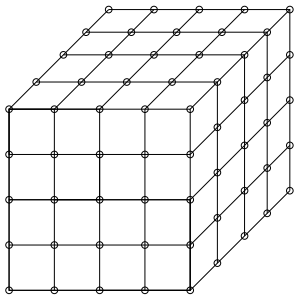

Figure 1.1: Volume data with a regular, rectilinear sampling.

resentation of the data. Conceptually, the AR representation stores higher resolution data in subsections of the volume deemed interesting, while the uninteresting higher resolution data is discarded leaving only a lower resolution representation in those areas. From an implementation standpoint, this amounts to chopping off branches of the octree at different levels, thus saving time spent traversing the tree, time spent processing cube data, and storage space. However, the volume now contains neighboring cubes of different sizes. If the basic marching cubes algorithm is applied to this data, we get several problems including: inconsistent interpolation between cube vertices, missing triangle vertices, and isosurface discontinuities. Consequently, neighboring cubes at different resolutions can no longer be treated independently. These problems are outlined in section(s) 4.2 Inconsistent Interpolation on page 13, 4.3 Missing Vertices on page 14, and 4.4 Discontinuities on 15, respectively. We present an adaptation of the marching cubes algorithm with added methodology in order to handle these new complexities. Sections 5 (page 19) and 6 (page 22) describe the algorithm and its implementation. Section 7 (page 31) presents some preliminary results and evaluation of the algorithm. Finally, section 8.3 (page 53) presents some conclusions along with ideas for future work.

CHAPTER 2

BACKGROUND

One operation we perform on volume data sets is isosurface computation — finding surfaces inside the volume data of constant scalar value. For example, a person's skin would map to an isosurface in medical image volume data. Another isosurface in the same volume data may identify a person's bone structure.

The volume data is a set of three dimensional data points usually with regularly spaced sampling points. Also, the sampling is usually from a continuous phenomena. From the regular sampling points we can introduce a conceptual division of the volume into rectilinear volume data elements or *voxels*. The sample point values may be viewed as either the center or corner of a voxel. We treat the sample point values as corners.

2.1 Applications

Applications of isosurface generation from volume data include [20, 21]:

- solid modeling
- computer-aided design/computer-aided manufacturing (CAD/CAM)
- robotics and computer vision
- medical imaging & medicine [16, 25]
- computational fluid dynamics (CFD) [4, 8]
- molecular dynamics [11, 13, 17]

2.2 Isosurface Rendering

The marching cubes algorithm was first presented by Lorensen and Cline [15] as a high resolution three dimensional surface construction algorithm. Given a three dimensional grid with scalar values defined at each intersection in the grid, a scalar (isovalue) value is chosen that corresponds to the isosurfaces generated. Figure 2.1 shows the 2 dimensional analogy to isosurfaces.

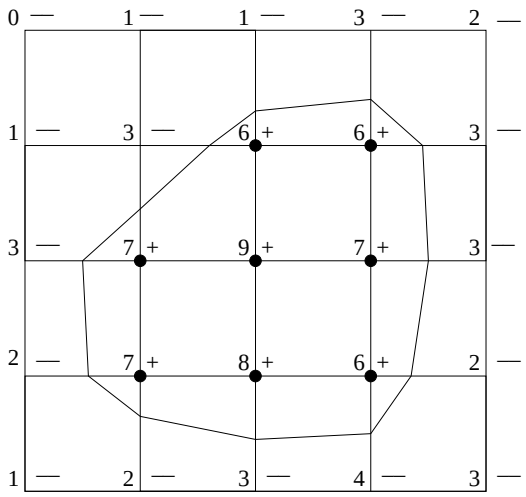

Figure 2.1: Contouring a two dimensional grid with a contour line of value five [21]

2.2.1 Computing a Surface with the Marching Cubes Algorithm

The marching cubes algorithm is very popular because of its simplicity of implementation. It is able to examine a large volume data set, one cube at a time, and output a surface. An important facet of the algorithm is that each cube is treated independently. As a result, there is a large reduction in complexity. Another advantage results when processing large volume data sets. Only small subsets of the data are required to be in memory at the same time for processing.

2.2.2 Basic Algorithm

The marching cubes (MC) algorithm identifies and constructs a boundary or surface of constant scalar value within a volume data set. In the MC algorithm:

- Each cell is treated independently.
- It is assumed that a surface can only pass through a cell in a finite number of ways.
- A case table is used to enumerate all possible topological states of a cell.
- A vertex is considered to be inside a surface if its scalar value is greater than the surface value.
- Once the proper case is computed, the location of the surface cell edge intersection is calculated using interpolation

The algorithm ends when all cells have been visited

2.2.3 A Cell's Topological States

The marching cubes algorithm is tractable because it assumes that the surface can only pass through each cube in simple ways. For example, if two adjacent vertices are

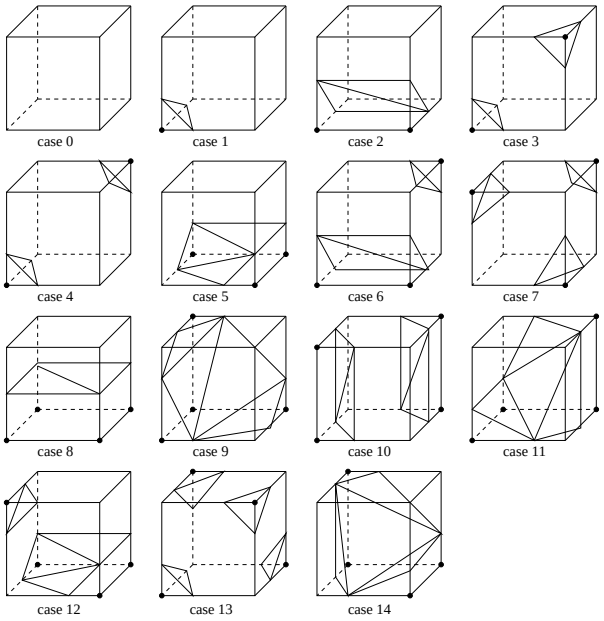

Figure 2.2: The 256 possible topological cube cases reduces to 15 by symmetry. Labeled vertices are greater than the isosuface value.

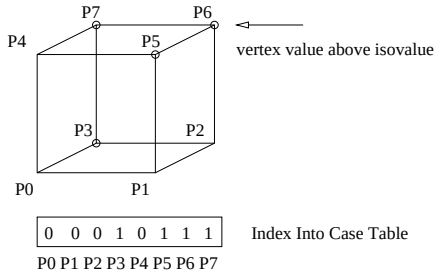

Figure 2.3: Generating an index into the case table: Vertices whose value is greater than the isosurface value are recorded in the index.

on the same side of the surface we assume that the surface does not pass through the adjoining edge at all. With these assumptions the surface can only pass through a cube in a finite number of ways. For a cube with eight vertices, the surface can pass through a cube in 256 (2^8) ways. But symmetry reduces the number of unique cases to the 15 shown in Figure 2.2. Thus, when the algorithm finds a cube containing the surface, it determines which topological case the cube falls into by associating a boolean value with each vertex. The boolean value is set to true if the vertex value is greater than the isosurface value, false otherwise. The 8 boolean values are converted to an 8-bit index into a case table of the 256 possible topological cases as shown in Figure 2.3.

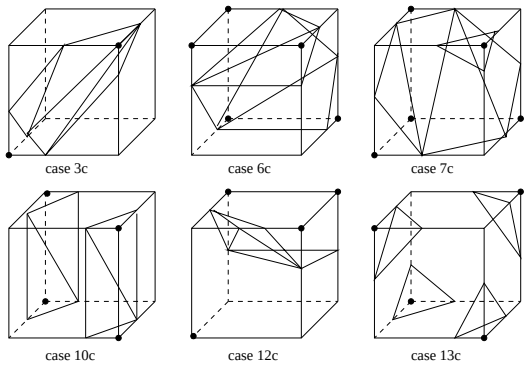

Figure 2.4: There are six unique marching cubes ambiguous cases.

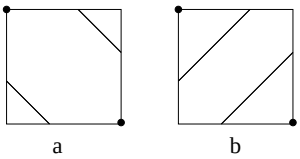

Figure 2.5: A 2D example of an ambiguous isocontour. Two vertex values above the isovalue can generate two different interpretations. The correct choice for the ambiguous cell depends on the nature of the surface in the adjoining cells, but we would like to preserve the independent cube processing model.

2.2.4 Surface Ambiguity

One problem with the marching cubes algorithm is surface ambiguity. Surface ambiguity arises when adjacent edge points are in different states, but diagonal vertices are in the same state. The 2D analogue this problem is illustrated in Figure 2.5. In 3D we need to extend the original fifteen marching cubes cases with the additional complementary cases shown in Figure 2.4. Correct selection between ambiguous choices insures that surface continuity is preserved, thus preventing accidental holes from appearing in the surface [21]. Other solutions include face patching, introducing tetrahedra, function dependent triangulation [12], and refinements described by Wilhelms and Van Gelder [27, 26, 18].

2.2.5 Marching Cubes Data Structures, the Octree

The octree is a hierarchical data structure well suited for the six-sided voxel structure of volume data. This is because octrees are based on the decomposition of space [27, 19, 20]. Conceptually, the root refers to the entire volume and from the root, the subvolume is divided into eight volumes recursively until we reach a level where each sample data point corresponds to a voxel vertex. See Figure B.1 in Appendix B for the octree representation of the volume data in Figure 1.1.

Octrees have the advantage that summary data can be used to prevent unneeded traver-

sal of uninteresting volumes. We use summary data such as the maximum and minimum values of data with each internal node's subvolume in the octree to avoid extra computation. However, the octree itself requires more memory than storing only the data.

2.2.6 Multiresolution Volume Data

One way of handling large volume data sets is to build a mulitresolution (MR) representation of the volume. We construct a data set that, in addition to the original highest resolution data, represents one or more levels of the volume that are coarser in resolution. The coarser resolutions are easier to handle in terms of processing time and memory storage. Often the average desktop computer does not have the resources necessary to process large volume data sets at the most detailed resolution. We may be able to render an isosurface in interactive time by running computations on a smaller data set of coarser resolution than on the original. However, we still are interested in preserving the original data in the case that it, or more likely a subset of the original data, becomes more interesting or needs more thorough investigation. We can extend this idea with the notion of an adaptive resolution (AR) representation. Conceptually, an AR representation is a volume data set with varying resolution. Some portions (subvolumes) of the volume are represented with high resolution while others are represented on a coarser level. We often find cases in which a particular subvolume is the focus of a scientist's observation or research.

2.3 Multiresolution Representation

The first stage of application development involves the generation of a multiresolution representation (MR) of the volume data set. If we begin with a 128^3 data set, we can build up to eight levels of resolution. Level 0 is our finest level of resolution containing a 128^3 representation of the data. The next coarser level of resolution, 64^3, may be computed in a variety of ways. The simplest approach is to simply discard every other sample point. Since each cube contains eight vertices, eight level 0 cubes are represented by one cube at level one. Level 7 is the coarsest level of resolution for a 128^3. Conceptually, level 7 is a cube containing the entire volume of the data set. In implementation, level 7 is the root node of the octree data structure used to store the volume data.

CHAPTER 3

RELATED WORK

Past research has focused on mulitresolution algorithms for isosurface rendering. This is because previous research has shown that 30% to 70% of the time spent in rendering is spent processing empty cubes [26]. In some instances, the isosurface has been reduced by 55% [22]. Unlike other octree algorithms [27], we address the use of octrees where certain regions of the volume data can be classified or pruned from the octree *a priori*. Reseach published by Engel, Wetermann, and Ertl [3] also utilizes a multiresolution data set, however their goal is to reduce the number of triangles generated by the standard MC algorithm, hence, compromising accuracy.

3.1 Static MR Representation

Past research deals with an MR representation which acts as the foundation for the rest of the rendering algorithm. Research published by Cox and Ellsworth [1, 2] observes that the amount of data generated by a visualization algorithm is relatively small compared to the total amount of data. This implies that sparse traversal methods can be created that reduce the amount of data needed to be accessed [14].

The entire octree is constructed and then traversed adaptively. In other words, the data is *not* adaptive but the traversal is. The isosurface value is examined inside blocks and the different resolutions are accessed adaptively. Isosurface traversal begins at the coarsest level of resolution. Whenever the isosurface value falls within the minimum-maximum boundary of a block of volume data, the next higher level of resolution in that block is traversed. If the isosurface value falls within the minimum-maximum boundary of one of those higher resolution blocks, then again, the next higher level of resolution in the corresponding block is traversed. This procedure is applied recursively as long as the isosurface value is found within the bounds of the volume data or until the lowest level resolution block is reached. Given an isosurface value, if it's not within the minimum-maximum boundary of a block, then entire branches of the tree data structure are skipped. Again, this is an MR representation with an AR traversal.

One advantage of this algorithm is that neighboring cubes used to generate the isosurface are always at the same resolution. When the isosurface passes from one cube to its neighbor, the neighbor is at the same level in the octree. This is a consequence of having a full MR representation to start with. However, the full MR representation also has disadvantages. In particular, the full MR representation takes up more storage space and requires more computation than an AR representation. It is often the case,

that a full MR representation is not needed due to volume data redundancy and areas in the volume that are simply not the focus for a scientist.

3.2 Adaptive Marching Cubes

R. Shu, C. Zhou, and M. Kankanhalli [22] were successful in speeding up the marching cubes algorithm with their version of an adaptive marching cubes (AMC). Their goal was to bring the MC algorithm to interactive time. They reduced the number of triangles by up to 55 percent by adapting the size of triangles to fit the shape of the isosurface. Similar to prior research, theirs uses a static uniform resolution representation with a dynamic MR traversal.

Their research differs from our approach in that their volume data is stored using a single level resolution representation with an MR traversal. And they resolve discontinuities in the isosurface differently then we do. Cracks may appear between two different neighboring cubes at different levels of subdivision (resolution). In what they called the "crack problem" [22], discontinuities in the surface are patched with polygons the same shape as those of the cracks. The cracks are abstracted into 22 basic configurations of different sizes, a solution that requires $O(n^2)$ of working memory space for an $n \times n \times n$ volume data set.

The basic notion behind the AMC strategy is to adjust the shape of an isosurface based on its curvature. This research starts off with the assumption that the volumetric data set size is $N_x \times N_y \times N_z$ in size where N_x, N_y, N_z are all powers of 2 (i.e. $N_x = 2^i$, $N_y = 2^j$, $N_z = 2^k$). The data set is then partitioned into cubes of equal size of 2^m where $m \leq min(i, j, k)$ (the details of how this is done are not presented in the paper). Then they choose an isosurface value and perform the basic MC algorithm. From this uniform resolution representation of the surface, they produce an MR representation of the surface based on the amount of curvature (or smoothness) within the cubes. This is done by partitioning the cubes at the original resolution into smaller and smaller cubes in a recursive fashion. The subdivision ends when either:

1. all the surfaces contained by the cubes are flat enough to fit into one of the MC cases or

2. the length of a subcube's side is unit length (finest resolution)

Figure 3.1 shows a 2D example of the AMC strategy. The curve PQR is to be approximated by a set of straight lines. By looking at the left half of the diagram, we can see that a uniform resolution approximation of the curve uses seven straight lines. However, when AMC is used, the same curve is approximated by three lines. The key is found by looking at the normals. Normals n_R and n_S are not too different from each other. Thus, we can approximate the surface between these two points with a straight line. However, normals n_Q and n_S are too different from each other, so the square encompassing the QS portion of the curve is subdivided and two lines are used to approximate the curve.

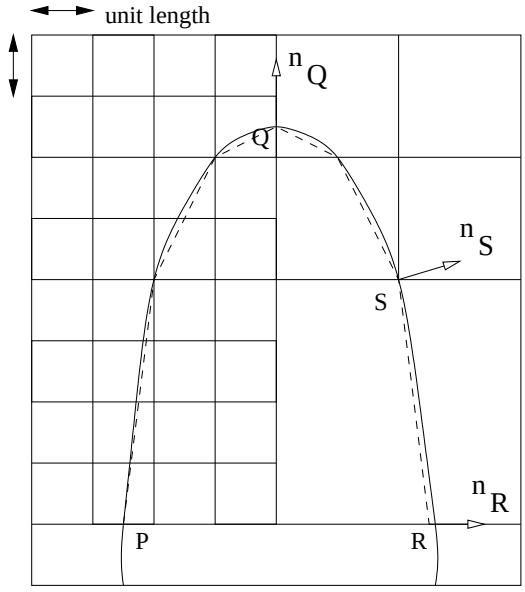

Figure 3.1: An example of AMC strategy in two dimensions

CHAPTER 4

ISOSURFACE RENDERING OF ADAPTIVE RESOLUTION DATA

Our approach uses an AR representation and an AR traversal algorithm. We render a static AR representation. Any value from the dataset may be chosen to generate an isosurface. Whether that choice happens to coincide with a slowly changing region of volume data or highly changing region is irrelevant in our case.

4.1 Adaptive Resolution Representation

An example AR data set is shown in Figure 4.1. Conceptually, this involves combining cubes whose vertex values are within a specified delta range. The adaptive resolution representation is decided by the change in values in a local area. We store a finer resolution representation in areas of rapidly changing sample data points, whereas areas with little change are stored with a coarser resolution representation. In other words, we can define a scalar value that represents an amount of change, δ, in the volume data set. When examining a volume cube, if the cube's vertices do not encompass a change greater than or equal to chosen threshold value δ, its volume is represented by a larger cube (of coarser resolution). From an implementation standpoint, this requires a depth-first tree traversal (DFS) of our octree data structure. Each node of the octree includes a cube. If the difference between the maximum and minimum vertex scalar values is less than the threshold value, all of the octree node's children are pruned from the tree (Figure 4.2). With this adaptive representation of the volume data, we minimize data which we may find to be uninteresting. In this fashion, we save both memory space resulting from a reduction in the amount of data stored and computation time during the rendering phase because many cubes have been eliminated from the data set. Figures 4.1 and 4.2 show an AR data set along with its corresponding octree representation. In this example, there are three levels of resolution. Each level of the octree stores a different level of resolution —the lower down the tree, the higher the resolution.

Since we are rendering an adaptive representation of the volume data, complications arise from the adjacency of blocks at different levels of resolution. The elegance of the conventional marching cubes algorithm is that each cube is treated independently.

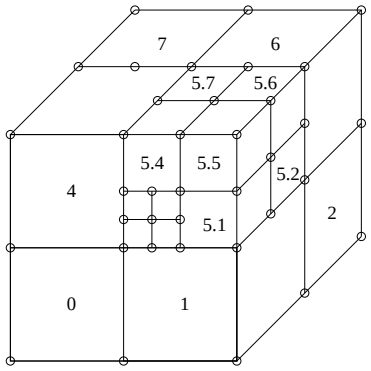

Figure 4.1: An example of an AR data set with numbered octants. Octant 3 is not shown (left, down, back)

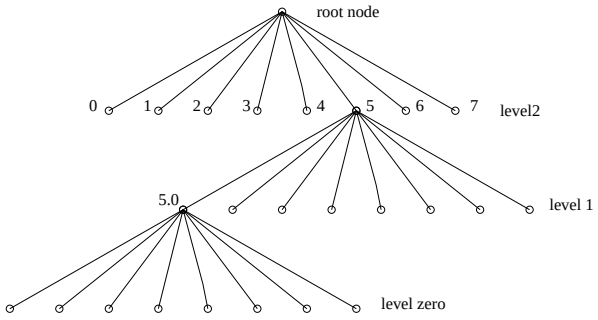

Figure 4.2: An octree data structure representing the sample AR data shown in Figure 4.1

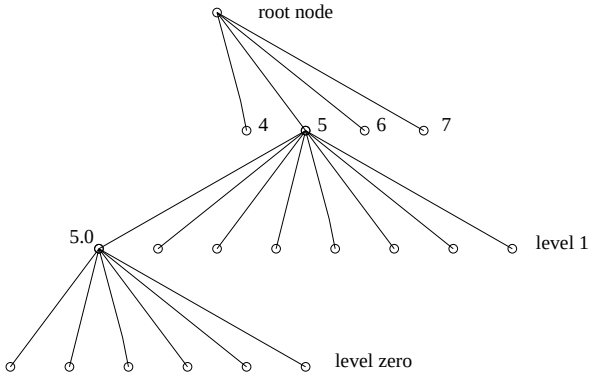

Figure 4.3: An incomplete AR representation of AR data shown in Figure 4.4

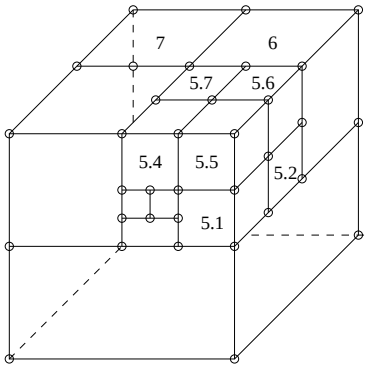

Figure 4.4: An example of an incomplete AR data set

This elegance is lost when we render adjacent blocks at different levels of resolution. We must introduce additional methodology into the marching cubes algorithm in order to overcome these complications. The following sections describe this in further detail.

Figure 4.2 shows a *complete* AR representation —- an octree whose internal nodes all have 8 children. However, not all AR representation are necessarily regular. Figure 4.4 shows an *incomplete* AR representation —- an octree whose internal nodes may have less than 8 children. Our algorithm assumes a complete AR data set, however, we can produce a complete AR from an incomplete AR. One way our AR rendering algorithm can handle this is by subdividing the coarser level cube(s) until we have a uniform resolution within the subvolume. This requires interpolation between the known vertices in order to recompute the missing nodes in the incomplete AR representation.

4.2 Inconsistent Interpolation

The standard algorithm for computing the intersection of a surface with a cube edge is to compute the linear interpolation between the endpoints of the edge. In a conventional volume data set, all cells are the same resolution, hence all shared edges have the same

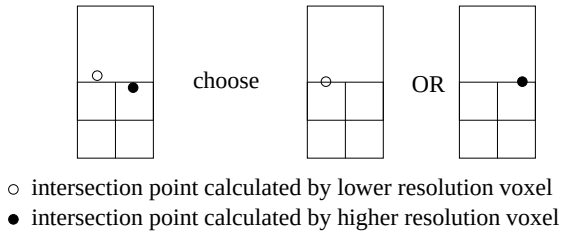

Figure 4.5: A special case of two cubes calculating different intersection points illustrating inconsistent interpolation

vertex values and thus neighboring cells interpolate to the same position. However, in our data set, two neighboring voxels at different resolutions may compute different linear interpolations as shown in Figure 4.5. This ambiguity is easy to resolve. We always use the interpolated point of the higher resolution voxel. We process the cubes in a regular fashion, even though they are different sizes, in order to avoid excessive traversal of our data structure.

In order to identify an instance of inconsistent interpolation, each octree node inspects the neighboring nodes that share the edge on which an edge intersection occurs. We use a case table to identify the appropriate neighbors to inspect and another case table to identify shared edges. See Appendix E on page 67 for the full details of the case table(s).

As soon as an instance of inconsistent interpolation is identified, the finer resolution node adds the edge intersection it computed onto a *fine edge intersection list* maintained by the coarser node as shown in Figure 4.9. The coarser node is then added to a 2^{nd} *pass list* of nodes whose triangle generation is postponed until we have performed the MC algorithm for *all* of the nodes. Processing of the 2^{nd} pass list is described in section 4.5 (page 16). We introduce another data structure to our octree nodes in order to address the problems created by adjacent cubes of differing resolution. Each octree node includes a cube as a member. Also, each node includes a list of triangles computed by the MC algorithm. The array of triangles points to an array of vertices, we call the *intersection list*, that may be updated at a later time, during traversal of neighboring coarser resolution voxels on the 2^{nd} pass list that share this edge.

4.3 Missing Vertices

Sometimes a coarser resolution cube identifies no intersection point along an edge, but intersections are found by two neighboring higher resolution cubes. Figure 4.6 illustrates this case. When the first finer resolution cube calculates an intersection point, it may search the coarser resolution cube's *edge vertex list*. However, there is no coarse vertex along the same edge. In this case the vertices of coarse resolution cube are on the same side of the isosurface. The second finer resolution cube may go through the same process, resulting in a missing vertex. To address this, both the first and

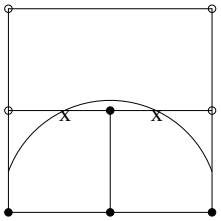

Figure 4.6: An intersection points is calculated by both higher resolution cubes, but not by their lower resolution neighbor illustrating a missing vertex

second finer resolution cubes create a new entry in the coarser resolution neighbor's *fine edge vertex list* shown in Figure 4.9. The coarser resolution cube is added to a *second pass list* of octree nodes. All of the octree nodes on the second pass list are processed after the entire octree has been processed. If a coarser resolution node on the second pass list discovers that it has two vertices on the same edge of its edge vertex list, it is subdivided.

4.4 Isosurface Discontinuities

Because we are dealing with adaptive resolution data, we have, in some instances, a larger, unsubdivided volume with a triangle, next to a smaller, neighboring cube with another triangle sharing a vertex with the larger triangle. This can introduce discontinuities in the isosurface.

Looking at Figure 4.7a, we may generate a lower resolution triangle entirely in the larger voxel. Then we generate the triangles in the neighboring higher resolution voxel (entirely in the smaller cube). Another possible configuration is shown in Figure 4.7b. In both cases the these two triangles introduce a surface discontinuity. The key to identifying this problem is recognizing that one (or more) triangle vertices are on the face, not an edge, of the larger voxel. The difference between the two cases is that the first one has 2 cube vertices above the isovalue on the edge of its coarser neighbor, while the second case has, in addition an additional cube vertex (above the isovalue) on the face of its coarser neighbor. In both cases, there are *triangle facial vertices* i.e., triangle vertices on the face of a coarser neighbor. See Appendix E for the figure(s) and table(s) used to identify facial vertex intersections.

We add the triangle facial vertices onto the coarser octree node's *facial vertex list* as shown in Figure 4.9. We then add the coarser node on to the second pass list of octree nodes. When a coarser node in the second pass list discovers that it has triangle facial vertices it is subdivided. We subdivide by interpolating between the known cube vertex values at the coarser resolution as shown in Figure 4.8. In the case of the cube vertices that a shared with finer resolution neighbors, we use the the finer resolution cube vertices rather than interpolating. See Figure F.1 for the specifics used to identify shared cube vertices used for subdivision.

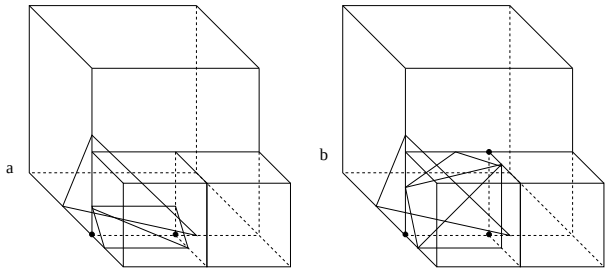

Figure 4.7: Two triangles generated at different resolutions that introduce isosurface discontinuity (a) the higher resolution cube has 2 cube vertices above the isovalue on the edge of its coarser neighbor (b) the higher resolution cube has 3 cube vertices above the isovalue on the edge and face of its coarser neighbor

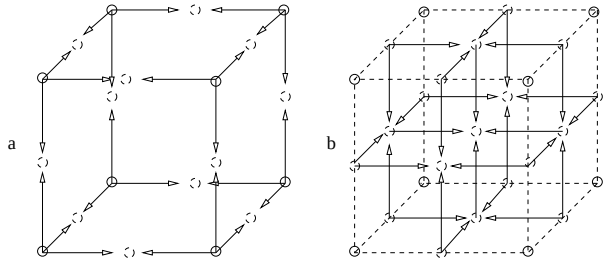

Figure 4.8: The process of subdividing a cube using interpolation. The arrows indicate direction of interpolation. The process can be broken down into two stages: a) shows the first round of interpolation calculations and resulting vertices b) shows the second set of interpolation calculations and resulting vertices

4.5 The 2^{nd} Pass List

The 2^{nd} pass list, as shown in Figure 4.9, is not processed until the entire AR octree has been traversed. The algorithm for processing the 2^{nd} pass list is as follows:

```
FOR EACH octree node in 2nd pass list
    IF facial vertex list is nonempty
      THEN subdivide
    ELSE IF there are 2 fine edge vertices on same edge
      THEN subdivide
    ELSE IF there is 1 coarse & 1 fine vertex on same edge
      THEN replace coarser vertex with fine vertex
END FOR EACH octree node
```

The 2^{nd} pass list is dynamic. More octree nodes may be added to the end of the list if another discontinuity of the type we have classified is discovered. This may be the result of subdividing an octree node on the list.

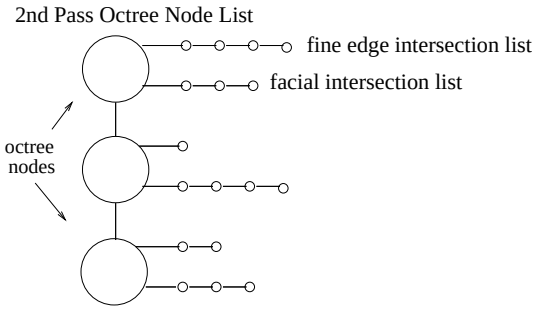

Figure 4.9: The 2^{nd} pass list of octree nodes. Each octree node has a list of finer edge intersections and facial intersections computed by their finer resolution neighbor(s).

4.6 VisAD

Our test environment for implementation is based on VisAD ("Visualization for Algorithm Development") [9]. We chose VisAD for its rendering capabilities and feature set.

4.6.1 VisAD Features

The important features of VisAD include:

1. It is implemented in 100% Java, taking advantage of Java's platform independence and network capabilities to support data sharing and real-time collaboration among geographically distributed users.

2. Support for distributed computing is integrated into the VisAD library utilizing Java RMI (remote method invocation) distributed objects.

3. It is based on a mathematical data model that can be adapted to virtually any numerical data and provides transparent access to data independent of storage format and location.

4. It provides a display model that supports 3-D rendering, multiple data views, direct manipulation, collaboration, and virtual reality by utilizing Java 3D^{TM} and Java 2D^{TM}. One of the goals in our proposal was to incorporate Java 3D^{TM} into our existing application.

5. It supports two distinct communities: developers who create domain-specific systems based on VisAD, and users of those domain-specific systems. VisAD supports a wide variety of user interfaces, ranging from simple data browser applets to complex applications that allow groups of scientists to collaboratively develop data analysis algorithms. [9]

4.6.2 Porting Issues

VisAD's data model incorporates the use of *Field* objects and *Set* objects. Fields approximate a function by interpolating values in a finite subset of the functions own

domain [7, 9]. A Field object contains a Set object that stores the sampled data. A Set object's responsibilities also include a coordinate system and data units. The Set class has many subclasses one of which is the *Gridded3DSet*. The Gridded3DSet has a rectangular topology but not necessarily a rectangular geometry. It is the Gridded3DSet that contains isosurface generating methods. We extended the Gridded3DSet class with our own *Gridded3DMRSet* class. Our Gridded3DMRSet class over-rides the `makeIsosurface()` method and adds support for MR processing and visualization. It was also necessary to modify a VisAD display component, the *DisplayImplJ3D* and the user interface component, *ContourWidget*.

CHAPTER 5

ALGORITHM OVERVIEW

5.1 Cube Processing

The input to our isosurface rendering application is cube data. The cube data is either an MR or AR representation of the volume in which we search for an isosurface(s). The demands of volume data are very high. Therefore, we have taken steps *a posteriori* to reduce processing time, storage space, and paging while we populate the octree with the given data.

In order to reduce paging we need to control access to each cube. Rather than force all input into a certain ordering, we implemented a tool to re-order the cells in an octree. The goal is to minimize the paging that occurs when the cubes are processed.

We build the octree by adding one cube at a time from the input file. If the entire octree does not fit in memory, paging will occur to access different nodes. Each time the operating system accesses secondary storage we incur a relatively large cost in processing time. If the cube data is read in an unordered fashion and ends up in locations that are scattered throughout the octree, excessive paging may result and hence processing time increases. This is based on the presumption that nodes that are topologically close to each other in the octree should be close to each other in memory. Therefore, we partition and sort the cubes as a preprocessing measure before we populate the octree.

5.2 Cube Partitioning

The cube partitioning utility reads any resolution level of cube data, either MR or AR, and places the cubes into one of their eight respective octants or subvolumes. Figure 5.1 shows the ordering. Any cube at any level in the volume is a member of one of these octants. Within each octant we sort the subcubes using either the *coordinate* or *octant-based* sorting techniques described in sections 5.3 and 5.4.

In addition to more efficient access to storage, this ordering is useful for testing our AR processing on AR data. With this utility it is easy to combine different data at different levels of resolution in a very deterministic fashion. For example, let's say we have a uniform data set consisting of 64^3 cubes partitioned into its eight octants. We can easily replace the octant 0 data at the 64^3 level with the corresponding coarser octant 0 data at the 32^3 level. This turns the uniform 64^3 data into an AR data set in

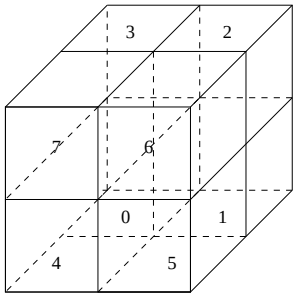

Figure 5.1: A cube volume is partitioned into eight octants

a very predictable fashion. In this way we know exactly where the boundary is between different levels of resolution in the data set. It is precisely this boundary that is of most interest to us when resolving discontinuities in the AR isosurface.

5.3 Coordinate-Based Sort

We implemented coordinate-based sorting in order to reduce paging while populating the octree. The cubes are sorted in x, y, z order based on the coordinate of their 0^{th} vertex. Now, cubes that are topologically close to each other in the octree are also close to each other in memory. This resulted in a significant reduction in the amount of paging.

5.4 Octant-Based Sort

Octant-based sorting is another utility we wrote in order to further reduce paging while populating the octree. The cubes are sorted by octant starting with octant 0 based on the subvolume they belong to with respect to their parent. This is an improvement over coordinate-based sorting because two cubes with the same parent in octants 0 and 4 respectively, while close to each other in the octree, may be rather far from each other in memory because they are in different layers (z axis) in Cartesian coordinate space. The result of octant-based sorting is that the cubes are added to the octree in precisely the same order as if performing a BFS traversal of the octree. This also resulted in a further reduction in the amount of paging.

5.5 ASCII to Binary Conversion

We also have a preprocessing utility program that converts cube data from ASCII file format to binary file format. Storing the cubes in binary format reduces reading time significantly. If we read the cubes in ASCII format, a considerable amount of conversion computation is performed. In particular, the cube's state is parsed and set including the cube's level of resolution, minimum and maximum scalar values, and the x, y, and z coordinates and scalar values of each of its eight vertices. Also error checking is performed on every number that is read. The error checking includes performing a test

to verify that the number is in the proper range. When we read the cubes in binary format, none of this computation is performed. The cube's state is already set and no error checking is performed. This amounts to considerable savings in processing time. Cube objects stored in binary format are read much faster than those read in ASCII format. Reading 64^3 cubes of data in binary format takes our application an average of 441 seconds. Reading 64^3 cubes of data in ASCII format takes an average of 2,290 seconds i.e. at least 5 times longer.

CHAPTER 6

ALGORITHM IMPLEMENTATION

We use an octree data structure to store the volume data (see section 1). The implementation of the octree raised performance cost issues with respect to processing time and storage space. Specifically, performance costs are at a premium when rendering volume data at resolution levels 64^3 and higher. These issues resulted in modifications to our initial naive implementation of the octree and its associated data structures.

6.1 Basic Algorithm

The basic algorithm for isosurface rendering is:

- read the volume data
- store the volume data in the octree
- traverse the octree, applying the Marching Cubes algorithm; recording cubes that need to be revisited because of potential discontinuities
- process the recorded cubes
- output the resulting polygons

6.1.1 Octree Population

The finest resolution cubes are stored as leaf nodes in the octree, while the coarser resolution cubes are stored as internal nodes. As cubes (described in section 5.5 on page 20) are read from the file, we start at the root node of the octree, classify the cube into one of the root node's octants, $0-7$, pass the cube to the node representing the computed octant, and recurse down the tree until the cube's resolution is one level finer than its parent octree node's resolution level. This octree node classifies the cube into one of its own octants, and stores the cube data as a data member for one of its child nodes, $0-7$.

The algorithm we use for adding cube data to the octree does not assume coarser resolution data will be read in before finer resolution data. Therefore, if a finer resolution cube is read in before one or more of its parents, we create the internal parent nodes on an as-needed basis, without their cube data. Then, if one of those internal node's

OCTANT	L	R	D	U	B	F
0	T	F	T	F	T	F
1	F	T	T	F	T	F
2	F	T	F	T	T	F
3	T	F	F	T	T	F
4	T	F	T	F	F	T
5	F	T	T	F	F	T
6	F	T	F	T	F	T
7	T	F	F	T	F	T

Table 6.1: The lookup table used in order for two octree nodes to find their common ancestor. (L = LEFT, R = RIGHT, D = DOWN, U = UP, B = BACK, F = FRONT)

cube data is read in, the internal node's cube data member is merely updated with the new cube read from secondary storage.

6.1.2 Neighbor Finding Technique

One way in which we modified the marching cubes algorithm is with the addition of neighbor finding techniques. This is because, unlike in the standard Marching Cubes, AR cube data cannot be treated independently. Each octree node inspects its neighbors for inconsistencies and discontinuities in the isosurface (see sections 4.2, 4.3, and 4.4). We used H. Samet's neighbor finding techniques [19].

Samet presents a neighbor finding algorithm through the use of two basic cube functions: the `adjacent()` and `reflect()` methods. The `adjacent()` method is responsible for finding the first common ancestor of the two neighboring nodes and the `reflect()` method is responsible for picking out the actual neighboring node amongst all of the common ancestor's children. In this case, we are referring to *face* neighbors —- neighboring cubes that share one (of eight) face. This is to be distinguished from *edge* neighbors —- neighboring cubes that share one (of twelve) edges.

The `adjacent()` method finds a common ancestor by taking advantage of each cube's (and hence octree node) octant position. Each cube must be in one of its parent cube's octants $0-7$. The `adjacent()` method returns *true* if a cube in octant O is adjacent to the face F of the cube's containing parent cube. For example, the left face of a cube in octant 0 is adjacent to its parent's left face. Table 6.1 shows the results of all the cases.

The `reflect()` method is what actually returns the neighboring cube. It also utilizes each cube's octant position. The `reflect()` method returns the face neighbor of the given cube in direction D. For example, the cube to the left of a cube in octant 0 is in octant 1. Table 6.2 below shows the results of all the cases.

OCTANT	L	R	D	U	B	F
0	1	1	3	3	4	4
1	0	0	2	2	5	5
2	3	3	1	1	6	6
3	2	2	0	0	7	7
4	5	5	7	7	0	0
5	4	4	6	6	1	1
6	7	7	5	5	2	2
7	6	6	4	4	3	3

Table 6.2: The lookup table used in order for one octree node to find its neighbor. (L = LEFT, R = RIGHT, D = DOWN, U = UP, B = BACK, F = FRONT)

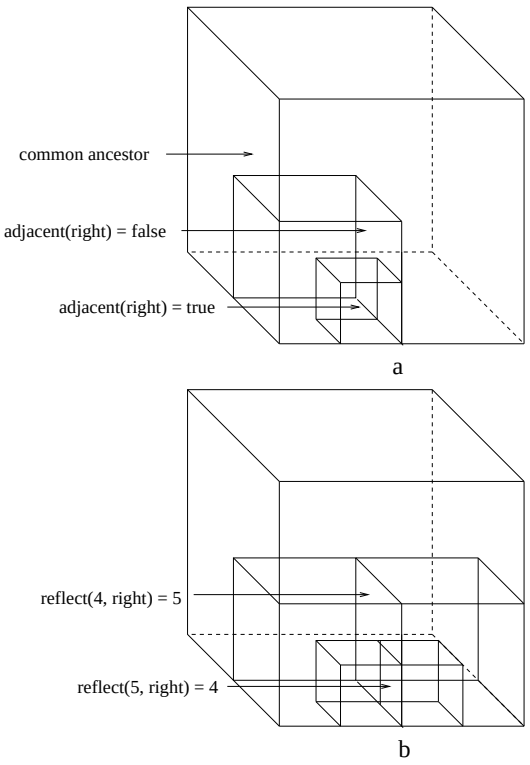

Figure 6.1: a) the result of two calls to the adjacent method and b) the result of two calls to the reflect method yields a right neighbor

The `adjacent()` and `reflect()` methods work together because the `adjacent()` method outputs a path of nodes from the calling octree node to a common ancestor and the `reflect()` method recurses back down that path reflecting each node identified by `adjacent()`. Figure 6.1 illustrates one example of the neighbor finding technique. In Figure 6.1a) the cube at the finest resolution asks for its neighbor to the right. Two calls

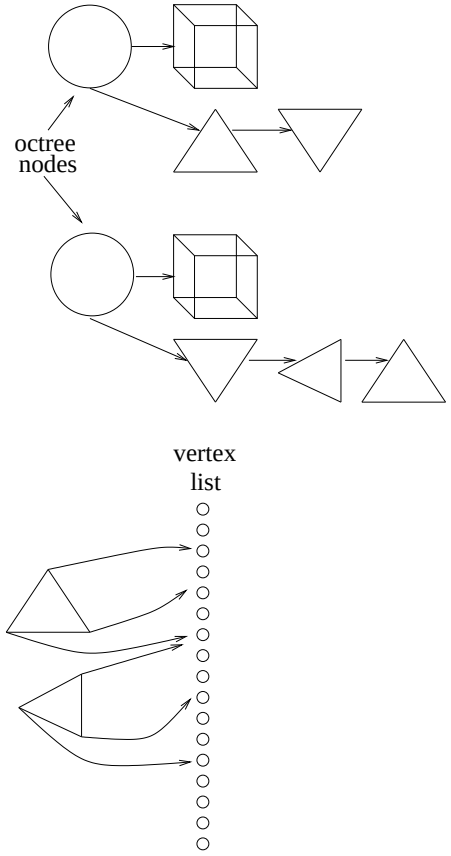

Figure 6.2: These are the data structures used to correct inconsistencies in an isosurface. Each octree node has a pointer to a cube and its associated polygons. Each polygon vertex is an index into a vertex list.

to `adjacent()` return their common ancestor (Figure 6.1a), and two calls to `reflect()` return the right neighbor (Figure 6.1b).

6.1.3 Storing Polygons With Octree Nodes

One of our goals is the implementation of a data structure to store the vertices of polygons that require updating by their higher resolution neighbors. Each node in the octree contains a cube, and a list of polygons. The list of polygons contains the triangles generated by the Marching Cubes algorithm. Each triangle has, in turn, three indices into a vertex list. The vertex list contains all *unique* triangle vertices. The individual triangle lists and the vertex list are precisely the data structures we need in order for higher resolution nodes to update their coarser resolution neighbors. If a node inspects it neighbors and finds an isosurface discontinuity or an inconsistency, it may access and update its neighbor's triangle vertices through these data structures (see Figure 6.2).

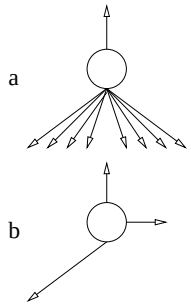

Figure 6.3: a) an octree node with 9 pointers and b) an octree node with 3 pointers The latter uses 24 fewer bytes of memory.

6.2 Memory Considerations

Conserving memory was of primary importance when rendering 64^3 or higher resolution volumes. We found ourselves running out of memory and looked for ways to conserve memory throughout the implementation of our data structures. We took measures in order to 1) reduce memory 2) reduce paging and 3) reduce computation. A reduction in memory was often in direct contrast with a reduction in computation.

6.2.1 An Octree Node With Three Pointers

Our original implementation of the octree and its nodes used 9 pointers per node: 1) pointer to the parent node and 2) 8 pointers to child nodes. We replaced this with an octree that contains 3 pointers: 1) a pointer to its parent, 2) a pointer to its 0^{th} child, and 3) a pointer to its next sibling. This change is illustrated in Figure 6.3.

In Java, each pointer uses 4 bytes of memory. So this change represents a savings of 24 bytes per octree node. At 64^3 we use 262,144 nodes for a total savings of approximately 6.3 megabytes of memory. At 128^3 we use 2,097,152 nodes for a total savings of approximately 50.3 megabytes of memory. This comes at the expense of computation since we have to traverse 7 sibling nodes in order to reach child number 7 of any octree node. Overall, however, the reduction of paging costs far outweighs the additional computation costs.

6.2.2 The Vertex Hierarchy

We also conserve memory by distinguishing between different types of vertices. We take advantage of the fact that a cube vertex does not need as much storage space as a triangle vertex. The resulting implementation is the vertex hierarchy shown in Figure 6.4. The vertex hierarchy consists of an abstract Vertex Class, a Cube Vertex class, a Triangle Vertex Class, and the IsoXvertex, IsoYvertex, and IsoZvertex classes. These are described in detail in the sections that follow.

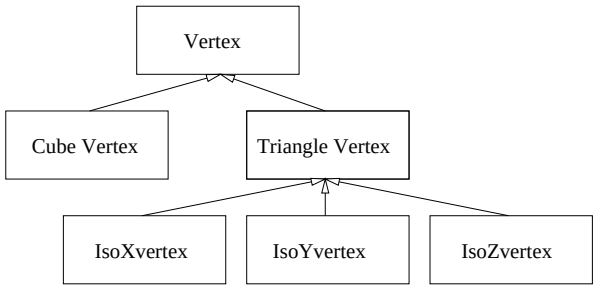

Figure 6.4: The vertex hierarchy: we save memory by distinguishing between vertices.

6.2.3 The CubeVertex Object

The most distinguishing property of a cube vertex is that each of the x, y, and z coordinate values needs only k bits for a $2^k \times 2^k \times 2^k$ volume. This is because our original sampled data is a regular gridded, rectilinear set of points in 3D space. A natural choice for storing such values is an integer. However, an integer in Java requires 4 bytes of memory. Since we only need 2^k unique values, we can use a Java `byte` instead of an `int` for $k \leq 8$. This represents a difference of 3 bytes per coordinate value. At 64^3 we use 6,201,456 cube coordinate values (3 for each cube vertex) for a total difference of approximately 18.8 megabytes of memory. At $k = 8$ we use 50,331,648 cube coordinate values for a total difference of approximately 150.9 megabytes of memory.

6.2.4 The IsoXvertex, IsoYvertex, and IsoZvertex Objects

The IsoXvertex, IsoYvertex, and IsoZvertex are vertices that make up each triangle. They are named as such because they are what ultimately define the isosurface itself, i.e., the output of the Marching Cubes algorithm. The key to conserving memory with these vertices is recognizing that each interpolated Triangle Vertex in the MC algorithm lies on an edge of its containing cube. Therefore, two of the x, y and z coordinate values are the same as the two Cube Vertex objects from which the new Triangle Vertex is interpolated. Only one of the x, y, or z coordinate values is interpolated and hence has a new value. For example, an interpolated Triangle Vertex on edge 0 (between Cube Vertex 0 and Cube Vertex 1) is interpolated along the x axis and hence has an interpolated x coordinate value. We use the same y and z coordinate values as Cube Vertex 0 and Cube Vertex 1. This new Triangle Vertex is what we call an *IsoXvertex* —- a vertex whose x coordinate value is interpolated. It has the unique property that its x coordinate value is stored as a Java `short` and its y and z coordinate values are each stored as a Java `byte`. A natural choice for saving 3 triangle coordinate values might be using 3 Java `float` values requiring 12 bytes total. However, taking advantage of edge values requires storing only 4 bytes —- a difference of 8 bytes. At 128^3 resolution, a typical isosurface may contain approximately 80,000 unique Triangle Vertex objects (see Table 7.5). This represents a difference of about 640 kilobytes.

6.2.5 Internal Nodes versus Leaf Nodes

There is an opportunity to save memory by distinguishing between internal octree nodes and leaf nodes. Internal octree nodes contain:

1. a pointer to their parent node
2. a pointer to their sibling node
3. a pointer to their 0^{th} child
4. a cube
5. a list of polygons
6. their child number
7. minimum and maximum vertex values

Leaf nodes however, do not need to store all of this information. Leaf nodes don't need to store minimum and maximum (1 `float` is 4 bytes) vertex values nor a pointer (4 bytes) to their 0^{th} child. The minimum and maximum vertex values may simply be computed. This represents a savings of 12 bytes for each leaf node. At 64^3 there may be up to 262,144 leaf nodes for a potential savings of approximately 3.1 megabytes. At 128^3 there may be up to 2,087,152 leaf nodes for a potential savings of about 25.2 megabytes.

6.3 Processing Considerations

We also tried different approaches to save processing time. We recognize that building cubes, building octree nodes, and building the octree all take a certain amount of processing time. We tried to save processing by using Java serialization.

Cube serialization [10] represents a considerable savings in processing time. This is described in detail in section 5.5 ASCII to Binary Conversion (page 20). Cubes stored in binary format are processed at least twice as fast as those read in ASCII format.

On the other hand, Octree Node serialization actually represents a performance hit in terms of processing time. If we serialize the octree nodes, reading them takes longer than if we just serialize the cubes. The only state information we save is the cube and minimum and maximum vertex values. The pointers to an octree node's parent, sibling, and 0^{th} child as well as the list of polygons are only computed in the octree. We can conclude from this, that simply allocating a new Octree Node object with a Cube object takes less time than the file I/O incurred by reading the Octree Node object from secondary storage.

The same holds true for the entire octree. Although by serializing the entire octree, we do save a lot of state information (see section 6.2.5 —- Internal Nodes And Leaf Nodes on page 28 for a full list), it still takes longer to read the octree from secondary

storage than to build the octree from just Cube objects. From this we can infer that, on average, it takes less time to place a cube in an octree than the file I/O time incurred from reading the Octree Node object (with all of its state information) from secondary storage.

6.4 Paging During Octree Construction

One of the problems we ran into while implementing the octree was excessive paging. The more nodes that are added to the octree, the more paging there is. Some of the ideas we use to minimize excessive paging are bottom-up octree population, rendering the octree by octant, and increasing virtual memory and heap size.

6.4.1 Top-Down Versus Bottom-Up Octree Population

It used to be the case that after approximately 200,000 nodes were added to the octree, the process would gradually slow down to a virtual halt due to thrashing. We speculated that this was due to the top-down recursion that took place every time an octree node was added to the octree. This process is described in detail in section 6.1.1 (Octree Population). Specifically, with top-down population, an octree node starts at the root node and traverses, in most cases, all the way down the tree until it finds its proper position.

Accessing the root node and several coarser resolution octree levels every time a node is added is not necessary. We replaced the top-down recursive method with a bottom-up approach. We added an Octree Node pointer array to the octree. Each entry in the array stores a pointer to a unique octree node and the array can be resized to store an arbitrary number of resolution levels. Whenever a node is added to an Octree containing an Octree Node array, the index of its parent node in the array is computed. If the parent node has already been added, the new Octree Node is immediately placed in the Octree and its own position in the Octree Node array is computed. If a pointer to the parent node is not found in the octree node array, a parent node (minus the cube data) is created and placed in the octree node array. This procedure recurses up the Octree until an existing parent node is found. Then the new Octree Node is set as one of the newly created parent's children and again its own position in the Octree Node array is computed.

Using this method we do not have to recurse down the entire Octree every time a node is added. Furthermore, we save processing time by taking advantage of the fact that the nodes being read in are already in sorted order as described in section 5.3 and 5.4 (Coordinate and Octant-Based Sort on page 5.4). The Octree Node array saves processing time and time due to paging at the expense of more memory. A disadvantage to this approach comes because the minimum and maximum octree node data values may have to recurse up the octree in order that the internal nodes have the proper values. The performance improvement of a bottom-up approach is still greater than the disadvantage due to bottom-up recursion of minimum and maximum scalar values.

6.4.2 Increasing Memory Size, Heap Size

Another factor we experimented with was the Java Virtual Machine (JVM) heap size. The JVM has the option of setting an initial heap size and maximum heap size at run time. The JVM runs out of memory if we do not set a maximum heap size of 256 megabytes. It turns out that the larger we set the initial heap size, the slower the application runs. If the initial heap size is set too large (e.g. 128 megabytes), the JVM starts paging before the program even starts.

After all of our memory reduction techniques, we were able to render an isosurface at 64^3 within a reasonable amount of time and with a reasonable amount of paging. For 128^3 we increased the size of our memory from 128 megabytes to 256 megabytes. The likelihood of us being able to store, simultaneously, an entire octree at 128^3 resolution with 256 megabytes is small.

6.4.3 Java Garbage Collection

As described in section 8.2.1 on page 52 (Rendering By Octant), our application grinds to a virtual halt from thrashing even if we read the volume data in from eight separate files, each of which contains one octant's worth of the same data. This leads us to believe that even though an entire octant's worth of data is not being used by our algorithm, it is not being released from memory as we would expect. This in turn leads us to believe that it is the Java garbage collector that is ultimately responsible for this. We speculate that the garbage collector, when looking for more memory to free, references every octree node preventing it from being swapped out of memory, or causing excessive unnecessary paging. We would like to test out our hypothesis by turning off the garbage collector or running it only at fixed times that we may schedule. However, the current version of the classic JVM for Linux does not support turning off the garbage collector.

CHAPTER 7

EVALUATION

7.1 Platform Specifications

Our evaluation experiments were run on a Dell PC with a 450 MHz pentium microprocessor and 256 Mbytes of RAM running Red Hat Linux 6.1. We use Sun Microsystem's Java 1.2/2.0 and VisAD which utilizes Java 3D. The isosurface generation algorithm is an Adaptive Marching Cubes derived from *The Visualization Toolkit* by Schroeder, Martin, and Lorensen [21], The display algorithm uses VisAD [9].

7.2 The Data Sets

One of our test data sets is a $113 \times 256 \times 256$ slice CAT scan of a cadaver head taken on a General Electric CAT Scanner and provided courtesy of North Carolina Memorial Hospital and Siemens Medical Systems, Inc., Iselin, NJ. It is volume data. We generated two coarser resolutions of the data, 128^3 and 64^3 for testing purposes.

Figure 7.1 shows a 64^3 resolution rendering of the medical image data with an isovalue of 0.185 (out of a normalized range of 0.000 to 1.000). We used an AR representation to render the image. The AR threshold δ is the difference between the minimum and maximum scalar vertex values of a node's cube and all of its children. A δ value of 10% results in all octree nodes whose scalar vertex values vary by less than 10% being trimmed from the original MR representation. By setting the AR threshold, δ, to 10% we were able to trim the octree data structure to about 47,000 cubes. The resulting isosurface consists of approximately 44,000 triangles encompassed by approximately 22,000 cubes of volume data.

Figure 7.2 shows a 128^3 resolution rendering of the medical image data with an isovalue of 0.185. By setting δ to 10% we were able to trim the octree data structure to about 292,000 cubes. The resulting isosurface consists of 199,000 triangles encompassed by 99,000 cubes of volume data.

Our secondary test data set is that of a lobster from the visualization laboratory at the State University of New York (SUNY), Stonybrook and Advanced Visual Systems (AVS) Inc in Waltham, MA (http://www.avs.com). The original data set was $128 \times 128 \times 64$. We expanded this to a shape of a uniform cube at resolutions 128^3 and 64^3 by padding the original data set with 64 layers in the z dimension. See section 7.3.3 on page 36 for pictures of the lobster data set.

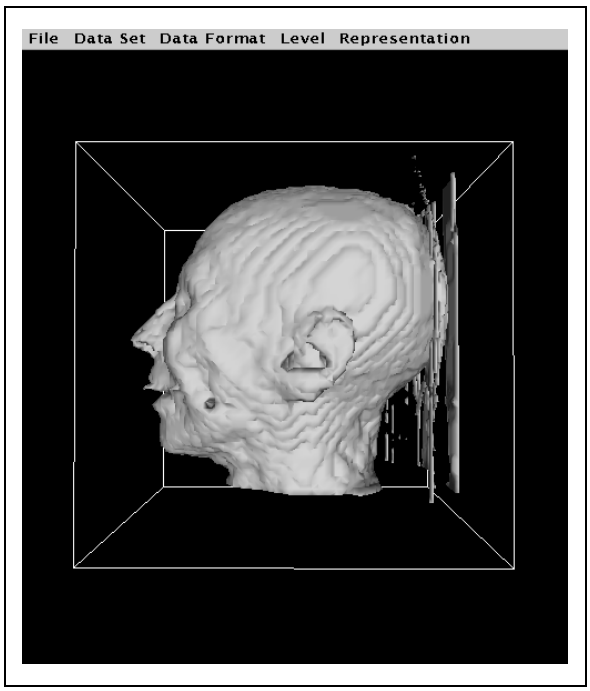

Figure 7.1: This is an AR isosurface with resolution 64^3, isovalue = 0.185, and $\delta = 10\%$.

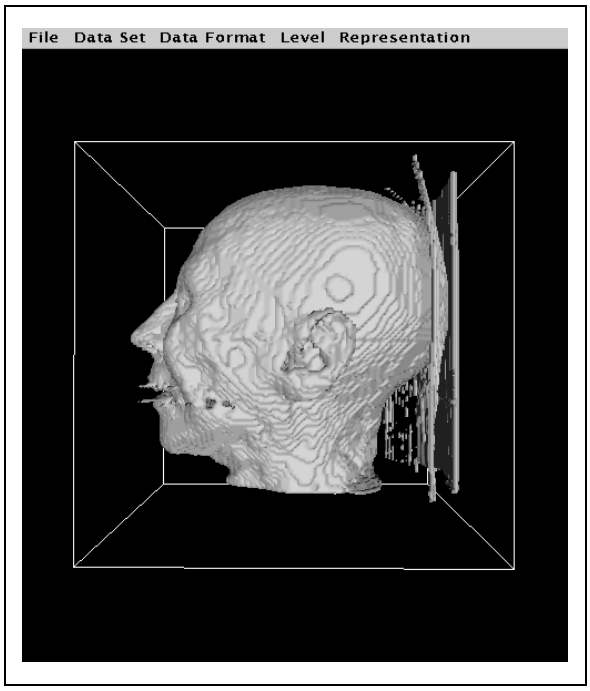

Figure 7.2: This is an isosurface with resolution 128^3 , isovalue = 0.185, and $\delta = 10\%$.

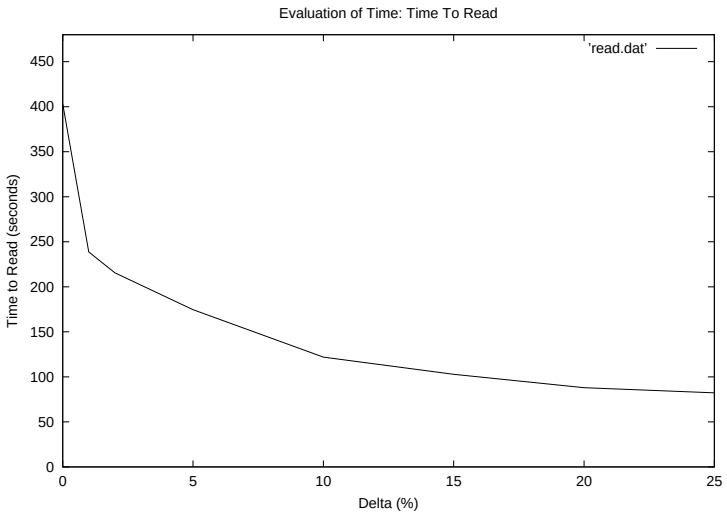

Figure 7.3: This plot shows δ (%) vs time to read (seconds)

7.3 Evaluation Strategy

We have a few different strategies for evaluating our performance. We make comparisons based on time, storage space, and image complexity.

7.3.1 Evaluation of Time

Our primary focus is the evaluation of the AR algorithm, i.e., the isosurface rendering of an AR representation. As our most fundamental evaluation, we compare this algorithm with a full MR representation algorithm, i.e. a representation with uniform resolution. A full MR representation stored as an octree still has the advantage that entire branches are culled by storing maximum-minimum data values with the internal tree nodes.

Table 7.1 (Evaluation of Time) and Figures 7.3 and Figure 7.4 compare the run time of our algorithm to that of a full MR data representation algorithm. There is a significant improvement in time when using an AR representation of the volume data. We can see that even at $\delta = 1\%$ we cut our reading time by over one half and our processing time is reduced by approximately 15%-20%. Time savings increase as δ increases. At $\delta = 10\%$, our time to read the data is reduced by a factor of about 5, and our processing time is reduced by about 20%-25%. This is our most interesting δ value because we obtain our biggest savings in time and preserve image quality at the same time. See section 7.3.3 (Evaluation of Accuracy). Using a full, uniform, MR representation takes approximately 5 times longer than the time to use an AR representation for a complete, single read and render cycle. We can also see from the table that 84% of the total time is spent reading the data of an AR representation while 95% of the total time is spent just reading the data of a full, uniform, MR representation.

octree	δ	isovalue	time to read (sec)	time to process (sec)
res 64^3 MR		0.185	403.3	30.7
		0.378		30.3
		0.408		36.0
AR	1%	0.185	238.7	18.9
		0.378		21.5
		0.408		19.6
	2%	0.185	215.5	18.6
		0.378		20.1
		0.408		19.8
	5%	0.185	174.6	16.6
		0.378		16.1
		0.408		15.8
	10%	0.185	121.9	16.3
		0.378		17.8
		0.408		16.7
	15%	0.185	102.9	14.8
		0.378		17.7
		0.408		17.1
	20%	0.185	88.0	15.7
		0.378		19.3
		0.408		17.8
	25%	0.185	82.3	16.2
		0.378		21.5
		0.408		19.3
res 128^3 AR	10%	0.185	803.7	730.8
		0.378		940.2
		0.408		755.8
	15%	0.185	623.6	259.2
		0.378		337.2
		0.408		256.1

Table 7.1: This table compares the run time of our algorithm verses the run time of an adaptive traversal of a full, uniform MR octree on the *cadaver head* data set. The 4^{th} column is the average time(s) taken to read the data. The 5^{th} column is the average time(s) to process the data.

octree	δ	time to read (sec)	time to process (sec)
res 64^3 MR		361.0	6.0
AR	1%	44.0	5.9
	2%	39.3	6.5
	5%	33.1	6.2
	10%	30.1	6.1
	15%	30.1	6.0
	20%	28.3	6.6
	25%	27.3	6.3

Table 7.2: This table compares the run time of our algorithm verses the run time of an adaptive traversal of a full, uniform MR octree on the *lobster* data set with an isovalue of 0.051 (out of a normalized range of 0.000 - 1.000). The 3^{rd} column is the average time(s) taken to read the data. The 4^{th} column is the average time(s) to process the data.

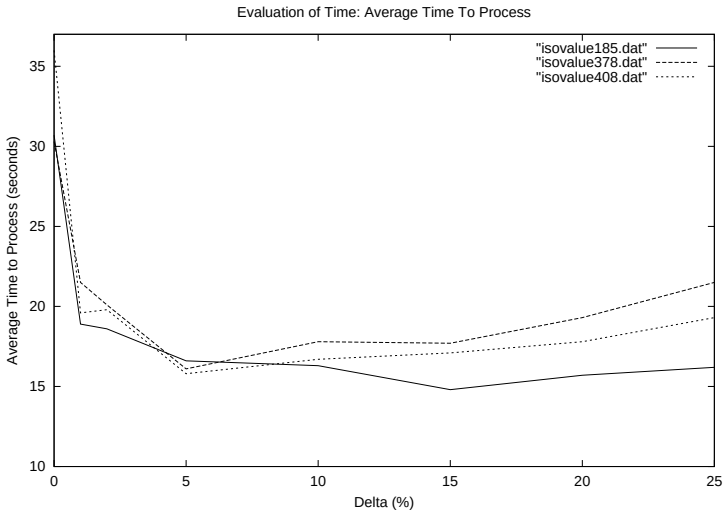

Figure 7.4: This plot shows δ (%) vs time to render 3 different isosurfaces (seconds) in the cadaver head data set

octree	δ	storage space (MB)	cubes read
res 64^3 MR		34.08	262,144
AR	1%	21.56	165,800
	2%	19.87	152,316
	5%	16.14	124,128
	10%	11.24	86,480
	15%	9.59	73,728
	20%	8.45	64,952
	25%	7.68	58.721
res 128^3 MR		272.63	2,097,152
AR	10%	51.37	481,640
	15%	41.62	393,888

Table 7.3: This table compares the storage requirements for an AR octree verses a full, uniform MR octree for the cadaver head data set.

7.3.2 Evaluation of Space

Storage requirements are presented in Table 7.3. The storage requirements are consistent with the processing times presented in Table 7.1. At $\delta = 1\%$ we already cut our storage requirements by over one half as well as the number of cubes. And at $\delta = 10\%$ we obtain our greatest savings in storage space while at the same time preserving image quality. We reduce our storage requirements by a factor of just over 5 at $\delta = 10\%$. Again, this is indicative of the time spent reading each representation shown in Table 7.1.

7.3.3 Evaluation of Accuracy

Our evaluation of accuracy and quality is based on image quality and data accuracy. Image quality is a subjective evaluation. We compare pictures of the same volume data rendered as both a full MR representation versus a series of AR representations. Tables 7.5 and 7.7 quantify the differences between a full MR representation versus this series of AR representations.

We expect to see no differences between the images of our full MR rendering and AR renderings at $\delta = 1\%$, $\delta = 2\%$, and $\delta = 5\%$ with an isovalue of 0.185. This is because at $\delta = 1\%$, $\delta = 2\%$, and $\delta = 5\%$ there are no discontinuities caused by AR data. This is shown in the 2^{nd} pass column of Table 7.7. At $\delta = 10\%$ (Figure 7.8) the differences in images are not discernible even though we do incur some discontinuities caused by AR data. However, at $\delta = 15\%$ (Figure 7.9) there are discernible differences. There are differences near the top of the ear, and differences in image quality where the back of the head meets the resting surface. And finally at $\delta = 20\%$ and $\delta = 25\%$ (Figures

octree	δ	storage space (MB)	cubes read
res 64^3 MR		34.08	262,144
AR	1%	4.22	32,456
	2%	3.68	28,336
	5%	3.12	24,144
	10%	2.84	21,880
	15%	2.77	21,304
	20%	2.69	20,709
	25%	2.62	20,152

Table 7.4: This table compares the storage requirements for an AR octree verses a full, uniform MR octree for the lobster data set.

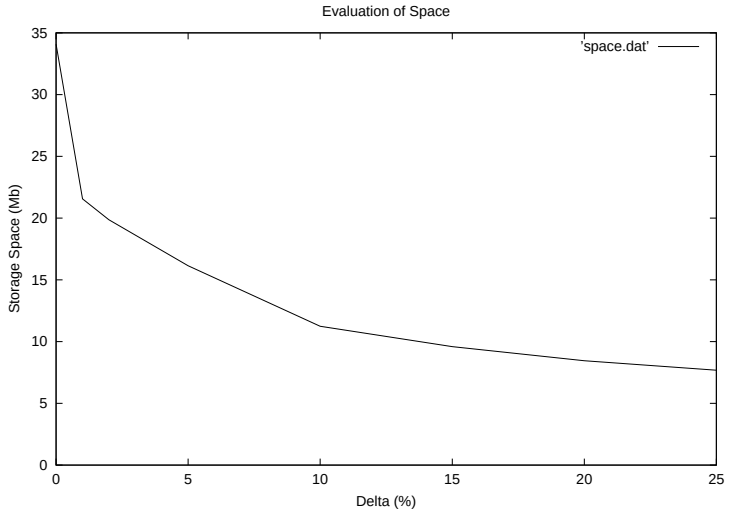

Figure 7.5: This plot shows δ (%) vs storage space (in Mb).

7.10 and 7.11) we lose our image quality beyond an acceptable level.

Also, we expect to see no differences between the images of our full MR rendering and AR renderings at $\delta = 1\%$ and $\delta = 2\%$ with an isovalue of 0.378. This is because at $\delta = 1\%$ and $\delta = 2\%$, there are no discontinuities caused by AR data. This is recorded in Table 7.7. At $\delta = 5\%$ and $\delta = 10\%$ (Figures 7.13 and 7.14) the differences in images are hardly discernible even though we do incur some discontinuities caused by AR data. At $\delta = 10\%$ we can see some differences in the image in the neck area and the bottom, rear of the skull. Finding differences at $\delta = 5\%$ is much more challenging however. At $\delta = 20\%$ (Figure 7.16) and greater, the differences in image quality become obvious. See Appendix G (Images) for more images. The isosurface value 0.408 is very similar in appearance to that of 0.378 however it contains significantly fewer discontinuities caused by AR data. See Table 7.7 for those results.

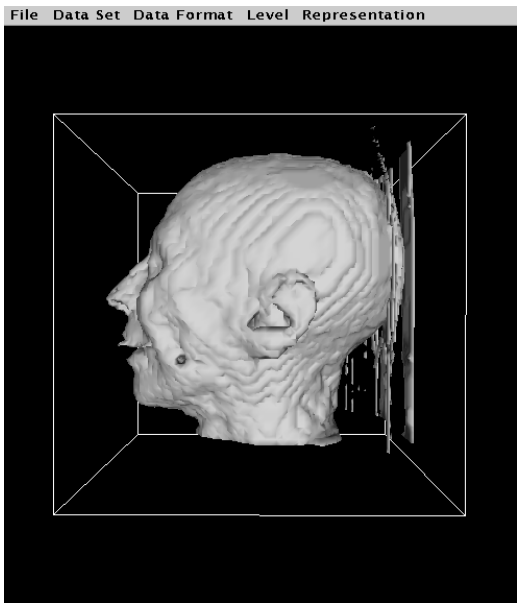

Figure 7.6: This is an MR isosurface with resolution 64^3 and isovalue 0.185.

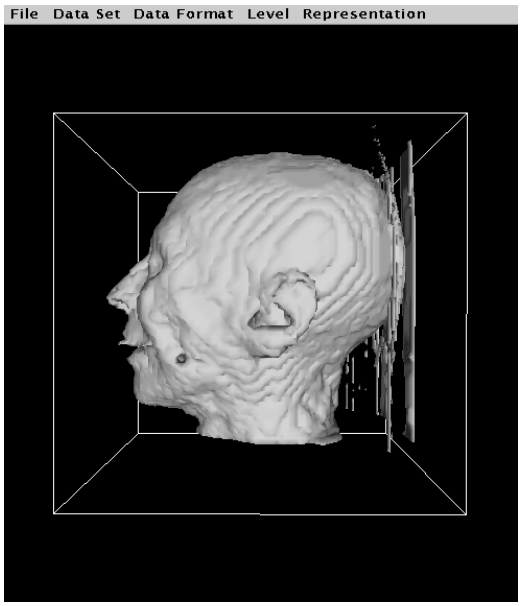

Figure 7.7: This is an AR isosurface with resolution 64^3, isovalue 0.185, and $\delta = 5\%$.

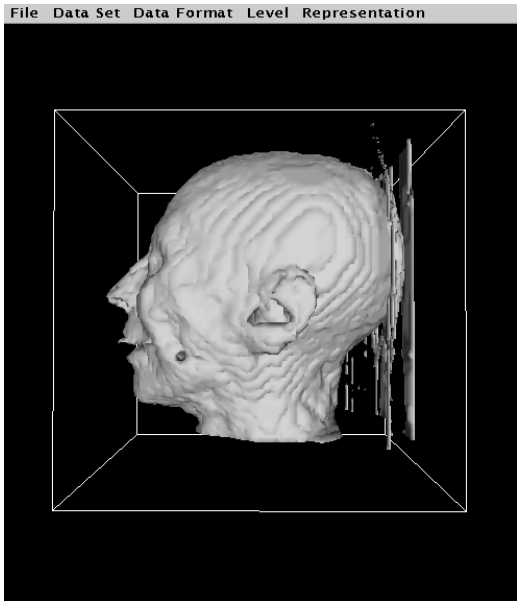

Figure 7.8: This is an AR isosurface with resolution 64^3, isovalue 0.185, and $\delta = 10\%$.

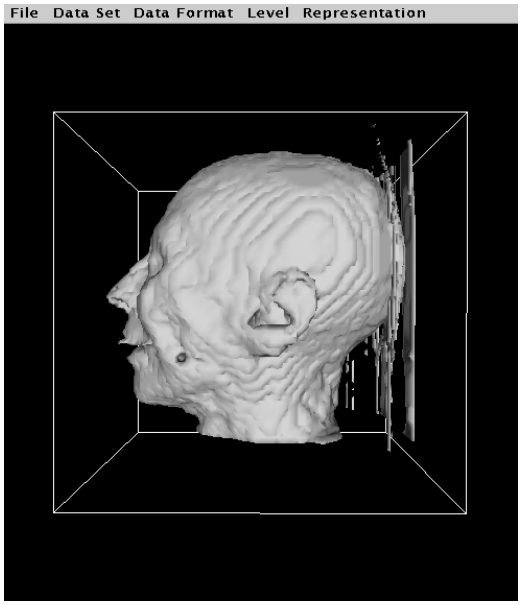

Figure 7.9: This is an AR isosurface with resolution 64^3, isovalue 0.185, and $\delta = 15\%$.

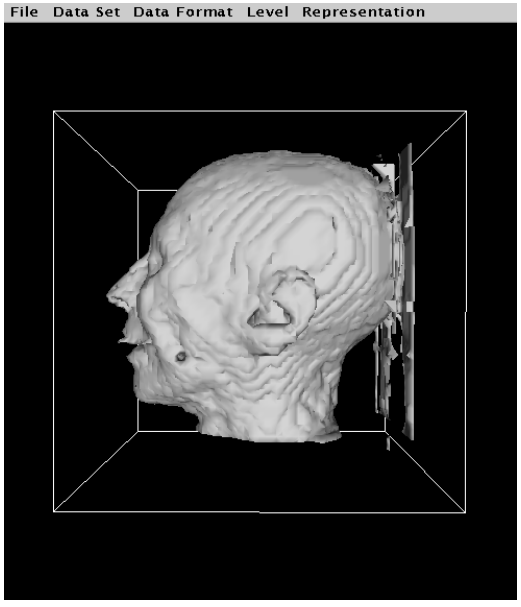

Figure 7.10: This is an AR isosurface with resolution 64^3, isovalue 0.185, and $\delta = 20\%$.

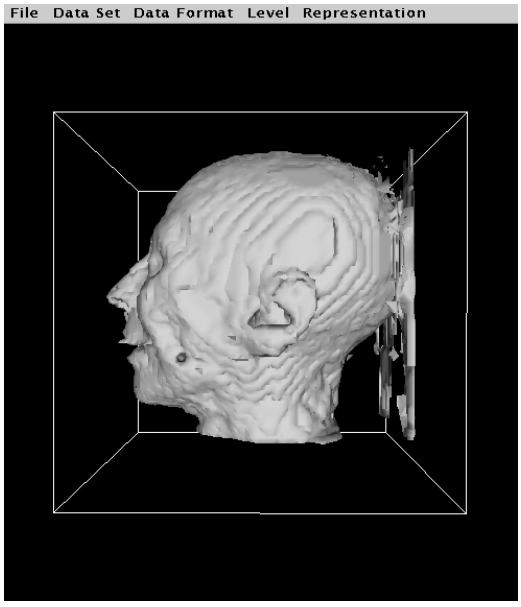

Figure 7.11: This is an AR isosurface with resolution 64^3, isovalue 0.185, and $\delta = 25\%$.

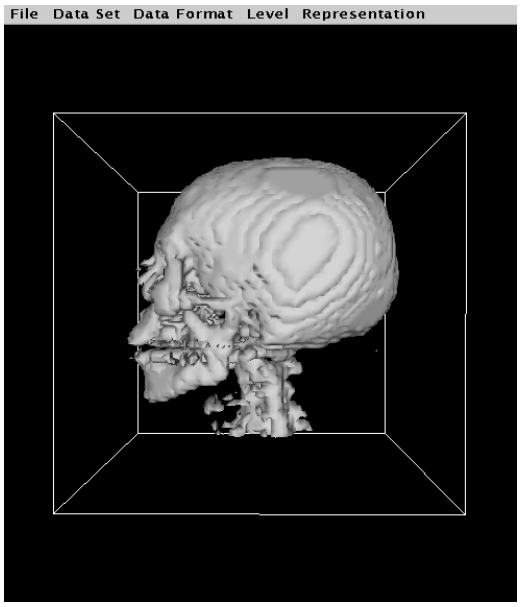

Figure 7.12: This is an AR isosurface with resolution 64^3 and isovalue 0.378.

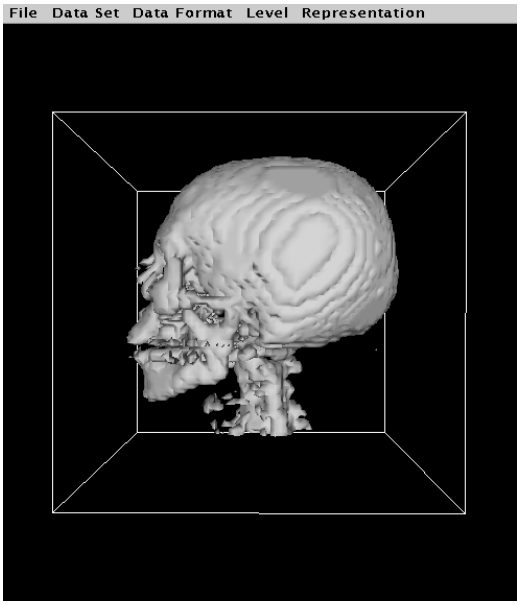

Figure 7.13: This is an AR isosurface with resolution 64^3, isovalue 0.378, and $\delta = 5\%$.

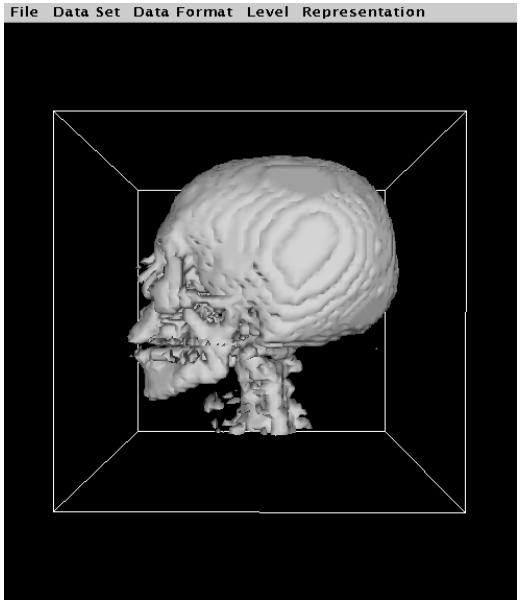

Figure 7.14: This is an AR isosurface with resolution 64^3, isovalue 0.378, and $\delta = 10\%$.

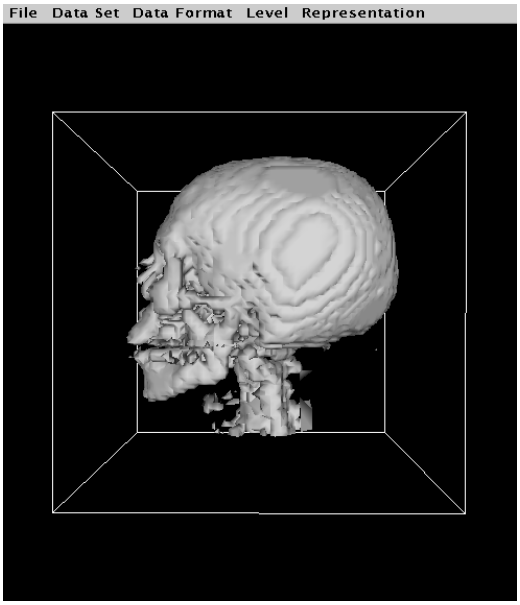

Figure 7.15: This is an AR isosurface with resolution 64^3, isovalue 0.378, and $\delta = 15\%$.

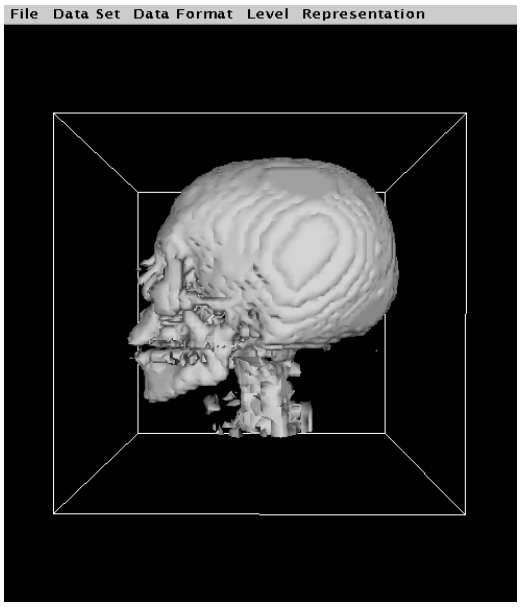

Figure 7.16: This is an AR isosurface with resolution 64^3, isovalue 0.378, and $\delta = 20\%$.

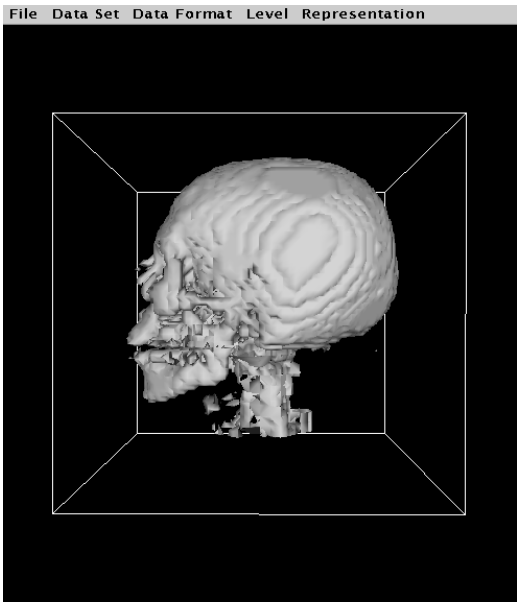

Figure 7.17: This is an AR isosurface with resolution 64^3, isovalue 0.378, and $\delta = 25\%$.

δ	isoval	cubes	polys	unique	dup
64^3 MR					
	0.185	22,240	44,856	21,451	113,117
	0.378	23,332	49,160	23,184	124,296
	0.408	22,368	47,652	22,486	120,470
AR					
1%	0.185	same	same	same	same
	0.378	same	same	same	same
	0.408	same	same	same	same
2%	0.185	same	same	same	same
	0.378	same	same	same	same
	0.408	same	same	same	same
5%	0.185	same	same	same	same
	0.378	same	49,149	23,255	124,192
	0.408	same	same	same	same
10%	0.185	same	44,859	21,523	113,054
	0.378	23,315	49,114	23,648	123,694
	0.408	22,340	47,611	22,604	120,229
15%	0.185	22,009	44,517	21,769	
	0.378	23,340	49,157	24,151	123,320
	0.408	22,279	47,505	22,913	119,602
20%	0.185	20,281	41,436	20,580	103,728
	0.378	23,422	49,345	24,941	123,094
	0.408	22,203	47,275	23,499	118,326
25%	0.185	19,287	39,450	20,103	98,247
	0.378	23,210	48,808	25,271	121,153
	0.408	21,929	46,550	23,761	115,889
128^3 AR					
10%	0.185	105,945	214,768	98,897	545,407
	0.378	105,101	213,845	99,691	541,844
	0.408	101,718	208,258	94,746	530,028
15%	0.185	100,757	203,980	96,500	515,440
	0.378	104,823	213,152	102,407	537,049
	0.408	100,817	206,020	96,437	521,623

Table 7.5: This table compares for each isosurface: (1) the number of cubes surrounding the surface, (2) the total number of triangles making up the isosurface, (3) the number of unique triangle vertices, and (4) the number of duplicate vertices in the isosurface for the cadaver head data set. (*same* refers to the MR data set)

δ	cubes	polys	unique	dup
64^3 MR	9,710	20,476	9,628	51,800
AR				
1%	same	same	same	same
2%	same	same	same	same
5%	same	same	same	same
10%	9,443	20,147	9,615	50,826
15%	9,438	20,143	9,750	50,679
20%	9,405	20,123	9,982	50,477
25%	9,418	20,165	10,004	50,491

Table 7.6: This table compares for an isosurface of value 0.051: (1) the number of cubes surrounding the surface, (2) the total number of triangles making up the isosurface, (3) the number of unique triangle vertices, and (4) the number of duplicate vertices in the isosurface, for the lobster data set.

Perhaps a more important factor in determining accuracy, is measured by comparing the original data with interpolated data values. In effect, this measure of accuracy only measures how linear the original data set is, not how well our algorithm performs. Data error comparison can be done using a standard error measure such as root mean squared error (RMSE). The root mean squared error is the square root of the average squared error. The RMSE can be described as the average size of errors.

This process evaluates how well a trilinear approximation fits a given data set. If the RMSE is computed and saved for each internal node, the user could define an error threshold which could be used to dynamically determine whether to interpolate or to use the original data.

7.3.4 Evaluation of Image Complexity

Evaluation of image complexity is a tertiary priority for us since our primary focus is on the innovation of the AR algorithm and not necessarily peak performance. However, we attempt to evaluate complexity on a preliminary level keeping in mind that performance optimizations may be considered in more detail in future work. To evaluate complexity, we keep statistics about direct ambiguity detection:

- How often does an occurrence of inconsistent interpolation happen?
- How often are there missing vertices?
- How often do facial intersections occur?

Table 7.7 shows one instance of these statistics. These numbers give us an indication of how often discontinuities occur for the isosurfaces shown in section 7.3.3. We gain

isovalue	2^{nd} pass	sub-divides	edge	miss-ing	face
res 64^3					
$\delta = 1\%$					
0.185	0	0	0	0	0
0.378	0	0	0	0	0
0.408	0	0	0	0	0
$\delta = 2\%$					
0.185	0	0	0	0	0
0.378	0	0	0	0	0
0.408	0	0	0	0	0
$\delta = 5\%$					
0.185	0	0	0	0	0
0.378	31	8	81	6	16
0.408	0	0	0	0	0
$\delta = 10\%$					
0.185	26	15	104	25	89
0.378	158	80	468	50	181
0.408	57	23	143	12	52
$\delta = 15\%$					
0.185	219	187	778	127	492
0.378	278	180	957	123	249
0.408	167	97	530	64	249
$\delta = 20\%$					
0.185	371	264	1,237	129	621
0.378	433	323	1,737	233	1,009
0.408	330	232	1,295	187	738
$\delta = 25\%$					
0.185	494	350	1,724	183	847
0.378	542	443	2,283	321	1,447
0.408	447	351	1,897	300	1,140
res 128^3					
$\delta = 10\%$					
0.185	213	182	816	106	538
0.378	1,492	937	4,901	406	2,406
0.408	683	327	2,102	192	929
$\delta = 15\%$					
0.185	1,352	1,145	5,111	822	3,118
0.378	2,430	1,792	8,458	710	4,989
0.408	1,735	1,152	6,060	519	3,236

Table 7.7: This table records (for the isosurface shown in Figure 7.8): (1) the δ threshold for each AR representation (2) the isosurface value (3) the total number of cubes that required a second pass (4) the total number of coarser cubes that were subdivided (5) the total number of edge vertices found on 2^{nd} pass cubes (6) the number of missing vertex cases (7) the total number of facial vertices found on 2^{nd} pass cubes

δ	2^{nd} pass	sub-divides	edge	miss-ing	face
res 64^3					
1%	0	0	0	0	0
2%	0	0	0	0	0
5%	0	0	0	0	0
10%	62	36	142	25	76
15%	100	61	280	44	142
20%	137	80	142	52	197
25%	162	101	480	63	271

Table 7.8: This table records for the lobster data set at isovalue = 0.051 (1) the δ threshold for each AR representation (2) the total number of cubes that required a second pass (3) the total number of coarser cubes that were subdivided (4) the total number of edge vertices found on 2^{nd} pass cubes (5) the number of missing vertex cases (6) the total number of facial vertices found on 2^{nd} pass cubes

considerable savings in time and space from an AR data representation of our test data set with no instances of discontinuities caused by AR data for $\delta = 1\%$ and $\delta = 2\%$ (for the given isosurfaces). Discontinuities don't start until δ reaches around 5%. We can intuit this by reasoning that we are very unlikely to choose an isosurface that passes through a cube with only 1% or 2% difference in its sample points. Furthermore, we *cannot* find an isosurface in cubes with 8 equal scalar vertex values (a $\delta = 0\%$).

Occurrences of discontinuities is clearly a function of both δ and the isosurface value chosen. The coarser the data representation, the more likely we are to encounter AR data for a given isosurface. Also, for a constant δ the number of discontinuities varies with isovalue. It is interesting to note that there is a significant reduction in the number of discontinuities for an isovalue of 0.408 when compared to 0.378 even thought the surfaces are similar in appearance (see Appendix G).

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

Our research has shown that direct isosurface rendering of an AR volume data set can lead to more efficient rendering with minimal loss to image quality. We can extend this research in several directions including:

- improvements to the rendering algorithm
- handling a wider range of MR data representations
- rendering larger data sets

8.1 Algorithm Improvement

Improvements to the algorithm can be made in the following areas:

- more special case handling as an alternative to subdivision
- limiting cell subdivision
- error visualization
- alternative subdivision policies

8.1.1 Special Case Handling

Currently we only handle shared triangle edge vertices before subdividing. It is possible to identify other cases in which we can infer feasible surfaces within a coarse cube without subdividing.

Figure 4.7 on page 16 shows that the coarser and finer resolution cubes have a face in common. We may be able to develop a scheme to classify isosurface discontinuities based on the shared cube vertices that make up this face. We can see from looking at Figure 8.1 since there are 5 cube vertices on the common face of neighboring MR voxels there are 2^5 different possible combinations of shared vertex values. This reduces to the 16 cases as shown by symmetry. We may be able to classify these cases by number(s) of cube edge intersections and number(s) of cube facial intersections in order to help us come up with another partial solution to the surface discontinuity problem. This approach could lead to a more efficient alternative to subdividing the volume in special

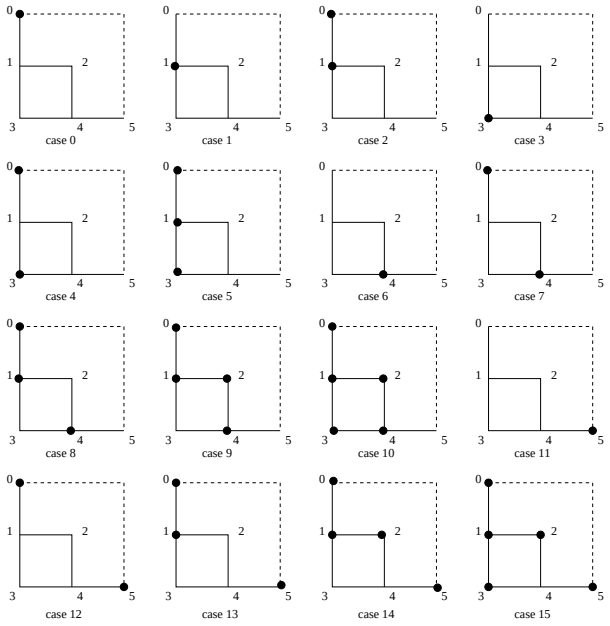

Figure 8.1: There are 16 different combinations of shared vertices on the common face of neighboring MR cubes.

cases, especially for incomplete AR volumes. We also may look into the research describe in section 3.2 Adaptive Marching Cubes (page 9) for inspiration on how to classify discontinuities. R. Shu, C. Zhou, and M. Kankanhalli [22] classified "cracks" into 22 basic categories. There may be a correspondence between the cracks described in their research and the discontinuities described in section 4.4 Isosurface Discontinuities on page 15.

8.1.2 Limiting Cell Subdivision

Another performance improvement might also be gained by reducing the *ripple* effect of cube subdivision. The ripple effect occurs when a second cube requires subdivision as a consequence of subdividing another cube. Table 8.1 shows instances where we may encounter such a ripple effect. The 3^{rd} pass column identifies how many nodes were added to the 2^{nd} pass list *while* processing the $2nd^{nd}$ pass list. In other words, when we are processing cubes on the 2^{nd} pass list, we are subdividing a subset of the cubes. And this may result in a neighboring cube needing another pass in order to examine its facial triangle vertex or edge triangle vertex lists.

We can see the first instance of where the number of nodes needing processing increases *as a result* of processing the 2^{nd} pass list at $\delta = 10\%$. Although we certainly cannot conclude that this is, in fact, going to start a chain reaction, our goal is to shorten the length of the 2^{nd} pass list, not lengthen it.

It would be useful to explore heuristics for limiting the *ripple* effect of subdivision based

resolution	δ	isovalue	2^{nd} pass	3^{rd} pass
64^3	1%	0.185	0	0
		0.378	0	0
		0.408	0	0
	2%	0.185	0	0
		0.378	0	0
		0.408	0	0
	5%	0.185	0	0
		0.378	31	0
		0.408	0	0
	10%	0.185	16	125
		0.378	158	99
		0.408	57	18
	15%	0.185	219	812
		0.378	278	450
		0.408	167	170
	20%	0.185	371	1,934
		0.378	433	1,562
		0.408	330	907
	25%	0.185	494	2,357
		0.378	542	2,584
		0.408	447	1,828
res 128^3	10%	0.185	213	1,201
		0.378	1,492	3,488
		0.408	683	792
	15%	0.185	1,352	7,056
		0.378	2,430	9,668
		0.408	1,735	5,374

Table 8.1: This table records: (1) the δ threshold for each AR representation of the cadaver head (2) the isosurface value (3) the total number of cubes that required a second pass (4) the total number of cubes that required a processing as a result of processing the second pass list

resolution	δ	2^{nd} pass	3^{rd} pass
64^3	1%	0	0
	2%	0	0
	5%	0	0
	10%	62	40
	15%	106	95
	20%	137	154
	25%	162	282

Table 8.2: This table records: (1) the δ threshold for each AR representation of the lobster (2) the total number of cubes that required a second pass (3) the total number of cubes that required a processing as a result of processing the second pass list all at an isovalue of 0.51.

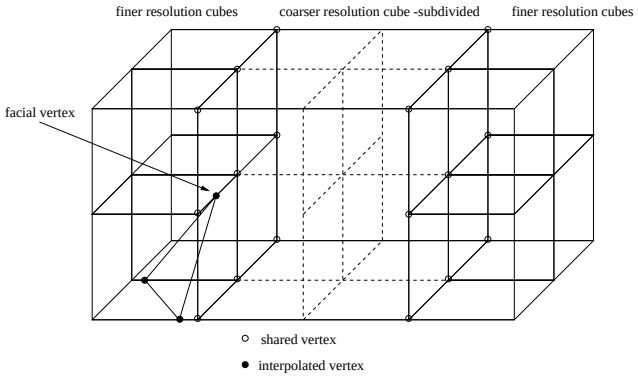

Figure 8.2: The coarser cube in the middle, when subdivided, updates its shared vertices with its finer resolution neighboring vertices' scalar values. However, this may present a problem for the shared vertices abutting the right side of the volume.

on a measure of likely error. As shown in Figure 4.8 on page 16, we subdivide a coarser resolution cube whenever a finer neighboring cube interpolates a vertex on the face of the coarser resolution cube. This situation is shown in Figure 8.2. An interpolated vertex was generated on the left face of the coarser resolution cube. Therefore we subdivide the coarser volume and in doing so, we update all nine of the shared vertices on the left side with the scalar values of its finer resolution neighbor. However, this may present a problem for the shared vertices abutting any other side of the volume if they are at different a resolution. In this example, this occurs on the right side. To correct this problem, we must also update any other shared vertex values if they are shared with finer resolution vertices. The advantage of this solution is an increase in accuracy. The drawback is the added computation.

This is related to section 8.1.1. The less frequent the subdivision (or more frequent the special case handling) the less likely we are to encounter a chain subdivision reaction.

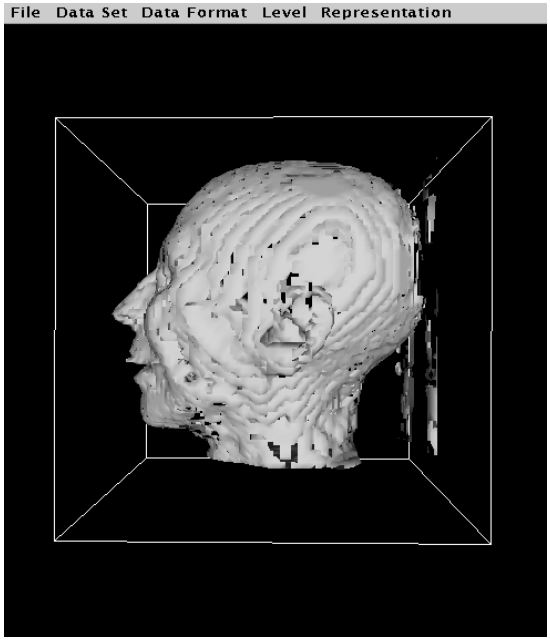

Figure 8.3: This is an AR isosurface with resolution 64^3 with holes indicating areas with high error.

8.1.3 Error Visualization

One prospective feature that could be added to the display algorithm is error visualization. Instead of generating coarser resolution triangles with no distinction with respect to the amount of error associated with each vertex, we could visually highlight a high error. In other words, areas in a surface with an error measure higher than a threshold defined by the user would be left out of the display. We could do this using a coloring policy as a function of error. The result may look something like that of Figure 8.3 in which we can see the appearance of holes.

8.1.4 Alternate Cell Subdivision Policies

Cell subdivision processing enhancements could be explored. For generating our higher resolution volume data at run time we have the following options, each of with we can evaluate on the basis of time:

- using interpolation to approximate higher resolution data and caching the newly generated blocks
- retrieving the necessary original data from secondary storage
- retrieving the necessary original data from over a network

Caching newly generated blocks may be a faster alternative to subdividing the cube more than once as in the case of generating more than one isosurface with the same

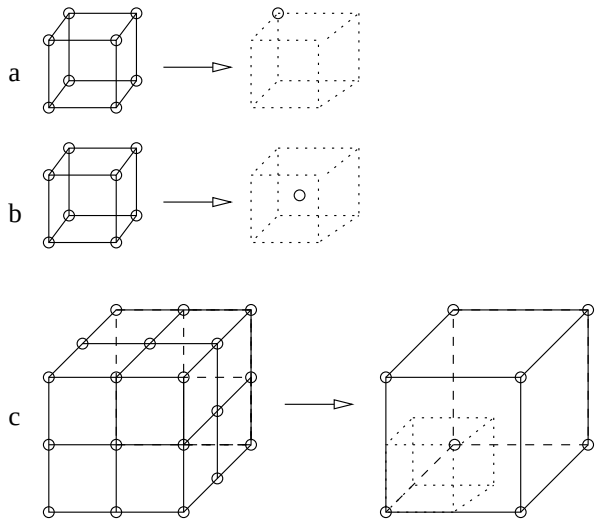

Figure 8.4: Three different ways of generating the next coarser level of data in an MR representation: a) selective removal of vertices b) a weighted average c) selective removal of vertices with a weighted average

AR data set. Retrieving the necessary original data from secondary storage or over a network would serve as a useful comparison of the accuracy versus processing speed trade-off.

8.2 Alternative MR Data Generation

We are interested in minimizing the amount of error caused by generating coarser resolution representations of volume data sets. Computing a coarser resolution representation of a data set using a wavelet transformation minimizes error, however, the topological relationship between levels is not obvious.

Given one level of resolution in an MR hierarchy, there is more than one way to generate the next coarser level. The different computations are shown in Figure 8.4. First, we can simply remove vertices selectively, perhaps in an alternating fashion as shown in 8.4a. This would result in a regular data set analogous to the one shown in Figure 8.6. Second, we can compute a weighted average both geometrically and topologically. A new vertex's position and scalar value are a weighted average of the surrounding eight vertices defining a cube as in 8.4b. However, this method presents complexities from a geometric standpoint. The analogous 2D complexity is illustrated in Figure 8.5. It is not clear how levels computed this way are related topologically. Coarser level vertices do not share the same geometric position as their corresponding higher resolution vertices.

However, a third way of averaging uses a weighted average for only scalar values and preserves the geometric position of coarser level vertices. Conceptually, this is selective

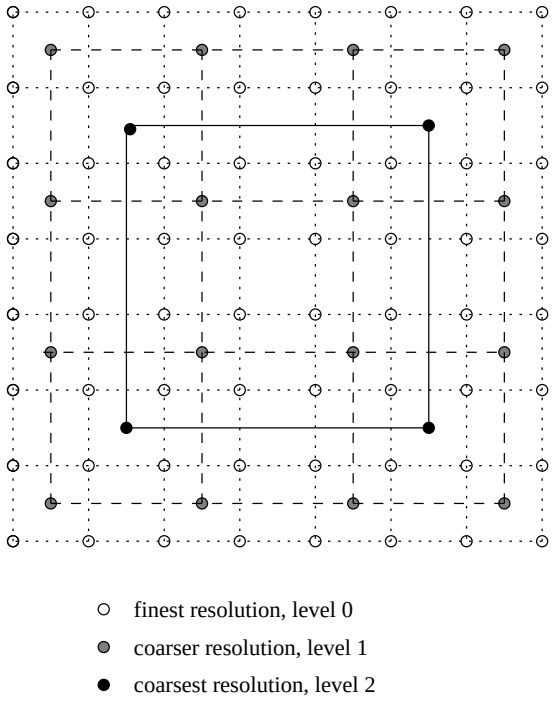

Figure 8.5: Three different levels of an MR hierarchy computed using a geometric averaging create topological complexity

removal of vertices as in Figure 8.4d. We can eliminate all vertices in a cube's sub-volume and leave only the remaining corners. This has the advantage that the corner vertices of all cubes sharing the same corner vertex have the same geometric position, but modifies the value at the corner to a weighted average scalar value. Our current implementation uses the first approach to generating MR data so that a vertex has the same values at all resolutions and vertex positions are preserved at coarser resolutions. Relaxing *either* of these constraints can allow the creation of more accurate MR representations but complicates the processing of the resulting AR representation.

8.2.1 Rendering By Octant

If we try a straightforward rendering of a full octree of size 128^3 we get near the end of octant number 3 before the program starts thrashing and grinds to a halt. Therefore, we store each of the eight octants in separate files. When a 128^3 volume is read in from eight separate files, each file containing an octant worth of volume data, we get the exact same result. The application thrashes to a virtual halt near the end of octant number 3. Therefore, our next approach to rendering an entire octree at 128^3 resolution is to process, from start to finish, one octant at a time. That means:

- reading one octant
- storing in the octree
- performing MC

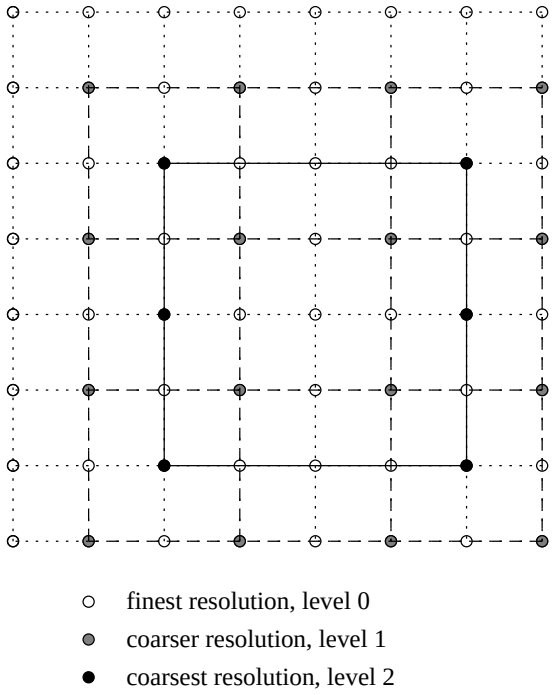

Figure 8.6: Three different levels of an MR hierarchy computed using a weighted averaging for only scalar values and preserving geometric position

- outputting the partial isosurface
- freeing that octant from memory and
- processing the next octant

8.3 Conclusion

In this research we set out to design and implement an algorithm for isosurface rendering of AR data representation. We identified different types of discontinuities caused by AR data including:

1. inconsistent interpolation (section 4.2)
2. missing vertices (section 4.3)
3. discontinuities caused by facial vertices (section 4.4).

We used several different data structures to implement our algorithm including:

1. an AR octree (see section 2.2.5)
2. a 2^{nd} pass list (see section 4.5)
3. a shared vertex list (see Figure 6.2)

We evaluated our algorithm based on factors such as:

1. rendering time (see section 7.3.1)
2. storage space (see section 7.3.2)
3. image quality (see section 7.3.3)
4. image complexity (see section 7.3.4)

Our goals in implementing the algorithm were to improve processing time, save storage space, and preserve image quality. Although the algorithm realizes those goals, it is clear that careful consideration and testing must be performed to choose a reasonable AR δ value and that results also vary according to the data set and isovalue.

BIBLIOGRAPHY

[1] M. Cox and D. Ellsworth. Managing big data for scientific visualization. *ACM Siggraph '97*, 21, August 1997. Course #4 Exploring Gigabyte Datasets in Real-Time: Algorithms, Data Management, and Time-Critical Design.

[2] M. Cox and E. Ellsworth. Application-controlled demend paging for out-of-core visualization. Proc. Of Visualization '97. IEEE Computer Society Press, October 1997.

[3] Klaus Engel, Rudiger Westermann, and Thomas Earl. Isosurface extraction techniques for web-based volume visualization. In *Volume Visualization*, Visualization 99, California, October 1999.

[4] Jean M Favre. Towards efficient visualization support for single-block and multi-block datasets. Proceedings of Visualization 1997, pages 423–428, Los Alamitos, CA, October 1997. IEEE Computer Society, IEEE Computer Society Press.

[5] James D Foley, Andries van Dam, Steven K Feiner, John F Hughes, and Richard L Phillips. *Introduction to Computer Graphics.* Addison-Wesley Publishing Company, 1990.

[6] Jeff Goldsmith and Allan S. Jacobson. Marching cubes in cylindrical and spherical coordinates. *Journal of Graphics Tools*, 1(1):21–31, 1996. http://www.acm.org/jgt/papers/GoldsmithJacobson96/.

[7] R.B. Haber, B. Lucas, and N. Collins. A data model for scientific visualization with provisions for regular and irregluar grids. Visualization 92, pages 298–305. IEEE, 1991.

[8] Philip D. Heermann. Production visualization for the acsi one teraflops machine. Proceedings of Visualization 1998, pages 459–462, Los Alamitos, CA, October 1998. IEEE Computer Society, IEEE Computer Society Press.

[9] Bill Hibbard. The visad java class library developers guide. The World Wide Web, November 1999. http://www.ssec.wisc.edu/billh/visad.html.

[10] Cay S Horstmann and Gary Cornell. *Core Java, Advanced Features*, volume 2. The Sunsoft Press Java Series, 901 San Antonio Road, Palo Alto, CA 94303, 1998. http://www.phptr.com.

[11] Johnston. The harrison group small molecule visualization web page. The World Wide Web, 1999. http://www.cem.msu.edu/johnston.

[12] Arie E. Kaufman, William E. Lorensen, Roni Yagel, Lisa Sovierajski, and Rick Avila. Extracting surfaces from medical volumes. In *Volume Visualization, Algorithms and Applications*, Visualization 95, Atlanta Airport Hilton and Towers, Atlanta, Georgia, October 1995. IEEE Computer Society Technical Committee on Computer Graphics. Tutorial 1.

[13] Marco Lanzagorta, Milo V Kral, J Edward Swan II, George Spanos, Rob Rosenberg, and Eddy Kuo. Three-dimensional visualization of microstructures. Proceedings of Visualization 1998, pages 487–490, Los Alamitos, CA, October 1998. IEEE Computer Society, IEEE Computer Society Press.

[14] C Charles Law, Kenneth M Martin, William J Schroeder, and Joshua Temkin. A multi-threaded streaming pipeline architecture for large structured data sets. In *Volume Visualization*, Volume Visualization 99, California, October 1999.

[15] WE Lorensen and HE Cline. Marching cubes: a high resolution 3d surface construction algorithm. *Comput Graph*, 21:163–169, 1987.

[16] William E Lorensen. Marching through the visible man. Proceedings of Visualization 1995, pages 368–373, Los Alamitos, CA, October 1995. IEEE Computer Society, IEEE Computer Society Press.

[17] Colin R F Monks, Patricia J Crossno, George Davidson, Constantine Pavlakos, Abraham Kupfer, Claudio Silva, and Brian Wylie. Three-dimensional visulaization of proteins in cellular interactions. Proceedings of Visualization 1996, pages 363–366, Los Alamitos, CA, October 1996. IEEE Computer Society, IEEE Computer Society Press.

[18] G Neilson and B Hamann. The asymptodic decider: Removing the ambiguity in marching cubes. Visualization '91, pages 83–91, 1991.

[19] Hanan Samet. *Applications of Spatial Data Structures.* Addison - Wesley Publishing Company, Inc, Reading, Massachusetts, 1990.

[20] Hanan Samet. *The Design and Analysis of Spatial Data Structures.* Addison - Wesley Publishing Company, Inc, Reading, Massachusetts, 1990.

[21] Will Schroeder, Ken Martin, and Bill Lorensen. *The Visualization Toolkit.* Prentice-Hall, Inc, Upper Saddle River, New Jersey 07458, 1996.

[22] Renben Shu, Chen Zhou, and Mohan S Kankanhalli. Adaptive marching cubes. *The Visual Computer*, 11:202–217, 1995.

[23] Philip Sutton and Charles D Hansen. Isosurface extraction in time-varying fields using a temporal branch-on-need tree (t-bon). In *Volume Visualization*, Volume Visualization 99, California, October 1999.

[24] WC Thibault and BF Naylor. Set operations on polyhedra using binary space partioning trees. *SIGGRAPH*, pages 153–162, 1987.

[25] Ulf Tiede, Thomas Schiemann, and Karl Heinz Hohne. High quality rendering of attributed volume data. Proceedings of Visualization 1998, pages 255–262, Los Alamitos, CA, October 1998. IEEE Computer Society, IEEE Computer Society Press.

[26] Jane Wilhelms and Allen Van Gelder. Topological ambiguities in isosurface generation. Technical report, University of California, Santa Cruz, California, December 1990. Exended abstract in ACM Computer Graphics. 2, 5

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

APPENDICES

APPENDIX A

PREPATORY RESEARCH TASKS SUMMARY

The following is a list of tasks involved in preparing this research:

1. We acquired a source of volume data from the University of North Carolina at Chapel Hill. The Chapel Hill Volume Rendering test data set is located at SoftLab Software Systems Laboratory at the University of North Carolina department of Computer Science, Chapel Hill, NC 27599-3175. The Chapel Hill Volume Rendering test data set, Volume I is a collection of the following files:

   - head data -A 109-slice MRI data set of a human head
   - knee data -A 127-slice MRI data set of a human knee.
   - HIPIP data -The result of a quantum mechanical calculation of a SOD data of a one-electron orbital of HIPIP (high potential iron protein), an iron protein. Provided courtesy of Louis Noodleman and David Case, Scripps Clinic, La Jolla, CA.
   - SOD data - An electron density map of the active site of SOD (superoxide dismutase). Provided courtesy of Duncan McRee, Scripps Clinic, La Jolla, CA.

   The Computer Science Department, University of North Carolina distributes these files by anonymous FTP. The data sets are provided courtesy of Siemens Medical Systems, Inc., Iselin, NJ.

2. We implemented an MR generation program as described in section 2.3. The program is written in *C*. It takes the original volume data set as input and produces a multiresolution representation of the data set.

3. We implemented an AR generation program as described in section 4.1. This program is also written in *C*. It takes the MR data set as input and produces an AR representation of the same data set. It is the output of this program that we use as input to our AR rendering program.

4. We implemented the marching cubes algorithm. This is written in *Java 1.1.7*. As part of the project we ported this to *Java 1.2/2.0*.

5. We wrote an initial display algorithm, the binary space partitioning (BSP) algorithm. This is also written in *Java 1.1.7*. As part of the research, we replaced this display algorithm with *VisAD* (which uses *Java 1.2/2.0* and *Java 3D*).

APPENDIX B

OCTREE DIAGRAM

See section 2.2.5 on page 6 for a description of the octree.

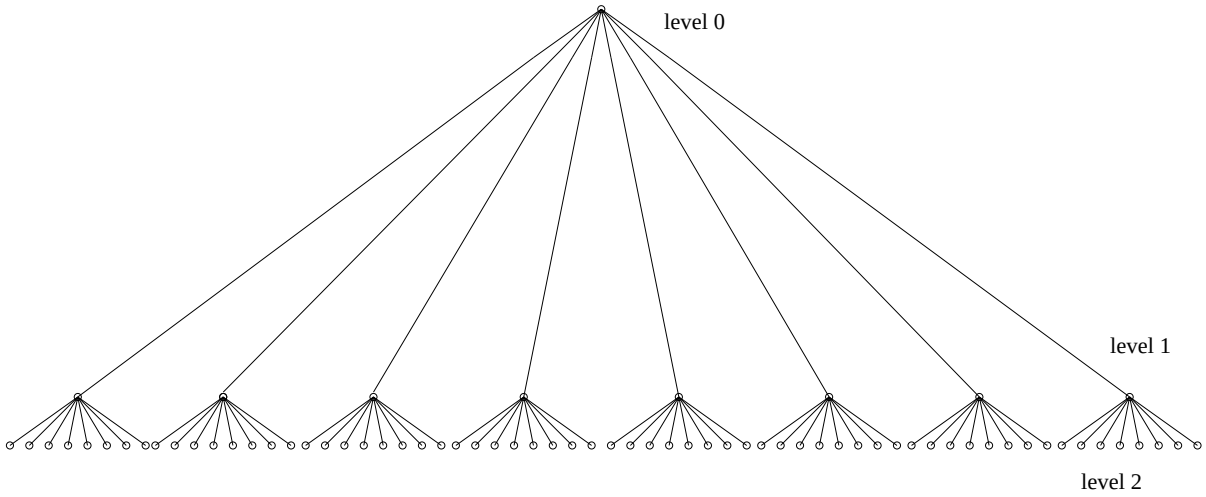

Figure B.1: The octree representation of the volume data in Figure 1.1 on page 1

APPENDIX C

APPLICATION OVERVIEW & DESIGN

Figure C.1 on page 63 shows the application software design in the form of a collaboration graph. The *AR renderer* is the director of the application. It contains the main method responsible for most of the initialization processes and contains the main interfaces for access to the *Marching Cubes* engine and the *Geometric renderer*.

C.1 Collaboration

The application starts off with the AR renderer requesting the *Marching Cubes Reader* to parse the input data. The input file stores adaptive resolution data representing 3D volume data (in both ASCII format for debugging and binary format for performance). It is this parsing of the input data that is the longest process in the application. The Marching Cubes Reader, in addition to parsing the cube data, collaborates with the *Cube* and *Vertex* (a member class of Cube) objects in order that they may be instantiated and added to the *Octree*.

The Octree is our main storage object. It contains the *OctreeNodes* for storage of the cubes. The *Octree* and *Octree Node* classes are where we put much of the innovation we need to implement procedures such as finding the common ancestor of more than one cube and finding a cube's neighbor. See section 2.2.5 on page 6 for a more detailed description of the Octree. Octree Nodes are either internal or leaf nodes. Internal nodes contain a 3 pointers, one to child 0, one to its parent node, and one to its sibling node. Internal nodes also have 2 values: the minimum and maximum values of all its descendants (children). Leaf nodes have 8 data values and a minimum and maximum of those data values. It is the leaf nodes that are rendered. The *Marching Cubes* and *Marching Cubes Cases* objects hold the responsibilities necessary in order to carrying out the algorithm described in section 2.2.

The *Triangle Vertex List* object is a data structure of vertices, some of which may need to be completed or updated. Each triangle vertex is part of a *Triangle* object. When traversing the octree, a cell compares its own resolution with that of its neighbor. We do not have specific x, y, z values for triangle endpoints, but a position object whose x, y, z values may be updated. When the corresponding finer resolution voxels that share this point are traversed, they update this list.

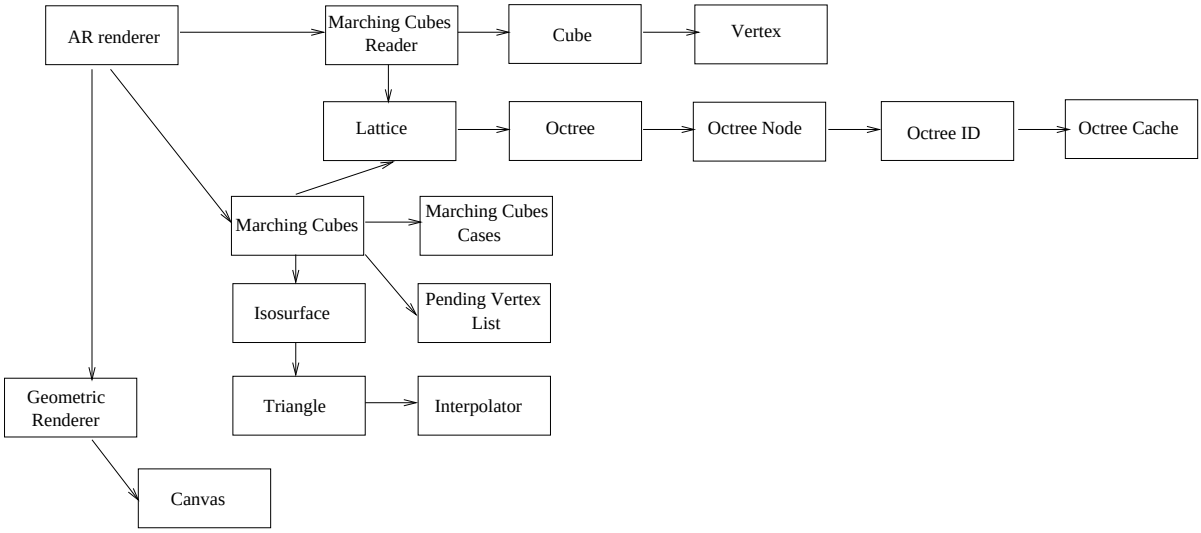

Figure C.1: A collaboration graph showing application design. See section C on page 62 for a complete description

The *Interpolator* class handles all of the interpolation calculations. When an isosurface value falls between two vertex values in a cube, a linear interpolation is computed to find out where the intersection occurs between the two vertices.

An *IsoSurface* object is the output from the rendering algorithm. The IsoSurface is a set of *Triangle* objects. If we are generating more than one isosurface, there may be some significant advantages of being able to pass a set of isosurface specifications, a set of isosurface objects, and go through the data only once, to generate more than one isosurface.

The *Geometric Renderer* object encapsulates Java specific (or Java dependent) rendering responsibilities. The former rendering subsystem that used the Binary Space Partitioning (BSP) algorithm was replaced with *VisAD* which uses Java 3D. See section 4.6 -VisAD for more on the VisAD system.

APPENDIX D

TESTING OPTIONS

We implemented a few testing options for our algorithm. These testing features allow us to verify the proper functionality of old algorithms such as the conventional Marching Cubes as well as new algorithms such as our AR Marching Cubes.

D.1 The Cube Step Function

One testing function we added to the Marching Cubes algorithm was the option of rendering one cube at a time with its associated polygons. Normally, the algorithm works by processing all of the volume data in one sweep. Then the triangles defining the resulting isosurface are rendered. We can slow the process down to one cube at a time. This is a useful tool for examining individual cubes and polygons as they are rendered. We also have the option of displaying the cubes along with the isosurface. Rendering one unit of volume cube at a time is useful if we are interested in identifying which marching cubes case a set of polygons belongs to. It is also useful in identifying which polygons belong to which subvolumes and as a tool to test the classic Marching Cubes algorithm itself with. See Figure D.1 on page 65 for a picture of this utility.

D.2 Rendering Cubes By Direction and Resolution

Another function we added to the Marching Cubes algorithm that makes use of the octree is the ability to choose a direction in which to render an isosurface. The user may choose at any point to pick direction, either left, right, backward, forward, down, or up and see what the isosuface looks like in the subvolume adjacent to the most recently rendered subvolume. Again, the user has the option of turning on or off the rendering of the cubes themselves. This feature is useful for testing our octree neighbor finding methods. The user at any time may also choose to increase or decrease the level of resolution of the isosurface rendering. For example the user may choose to render the subvolume to the right of the current subvolume at the next finest level of resolution. These features are useful to test the complete traversal of the octree.

D.3 MR Cube Utility

One of the goals of the project is the implementation of a tool used to examine neighboring AR data on a case-by-case basis. This tool is useful because we may examine

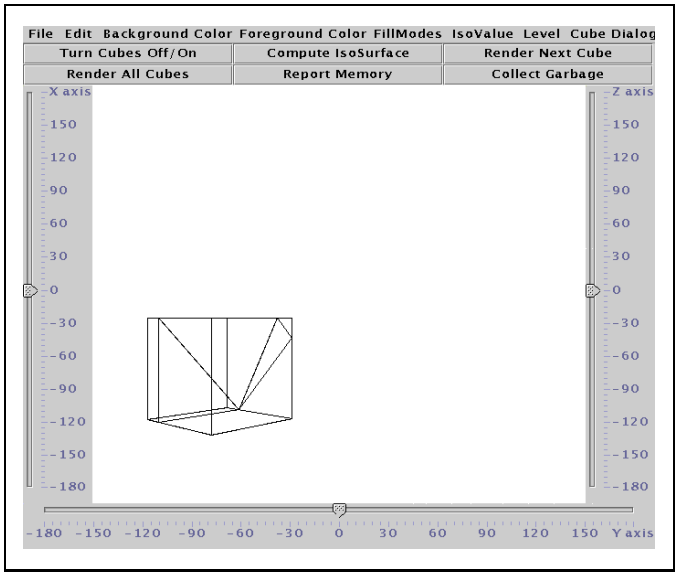

Figure D.1: This is the utility we implemented to examine one cube at a time.

any two AR facial neighbors and examine the discontinuity in the isosurface between them. For example, we may decide to look up close at the discontinuity between marching cubes case 3 and case 6 (Figure 2.2 on page 5) when they are face neighbors at different resolutions. This is useful if we want to examine the possibility of special case handling of isosurface discontinuities. Special case solutions of discontinuities may involve assigning a predetermined solution to a specific instances of neighboring AR cube data. See Figure D.2 on page 66 to see the AR utility in action.

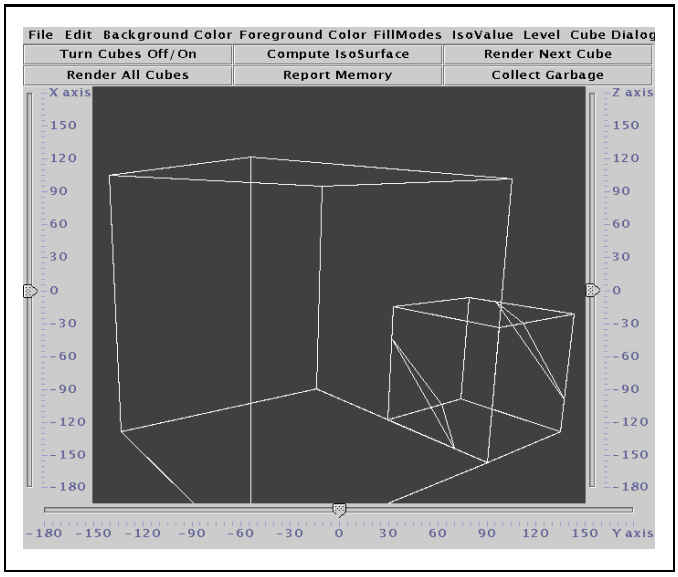

Figure D.2: This is the utility we implemented to examine neighboring AR data.

APPENDIX E

INCONSISTENT INTERPOLATION FIGURE AND CASE TABLES

See Figures E.1 on page 68, E.2 on page 69 and E.3 on page 71 and Table E.1 on page 70 for more details about inconsistent interpolation.

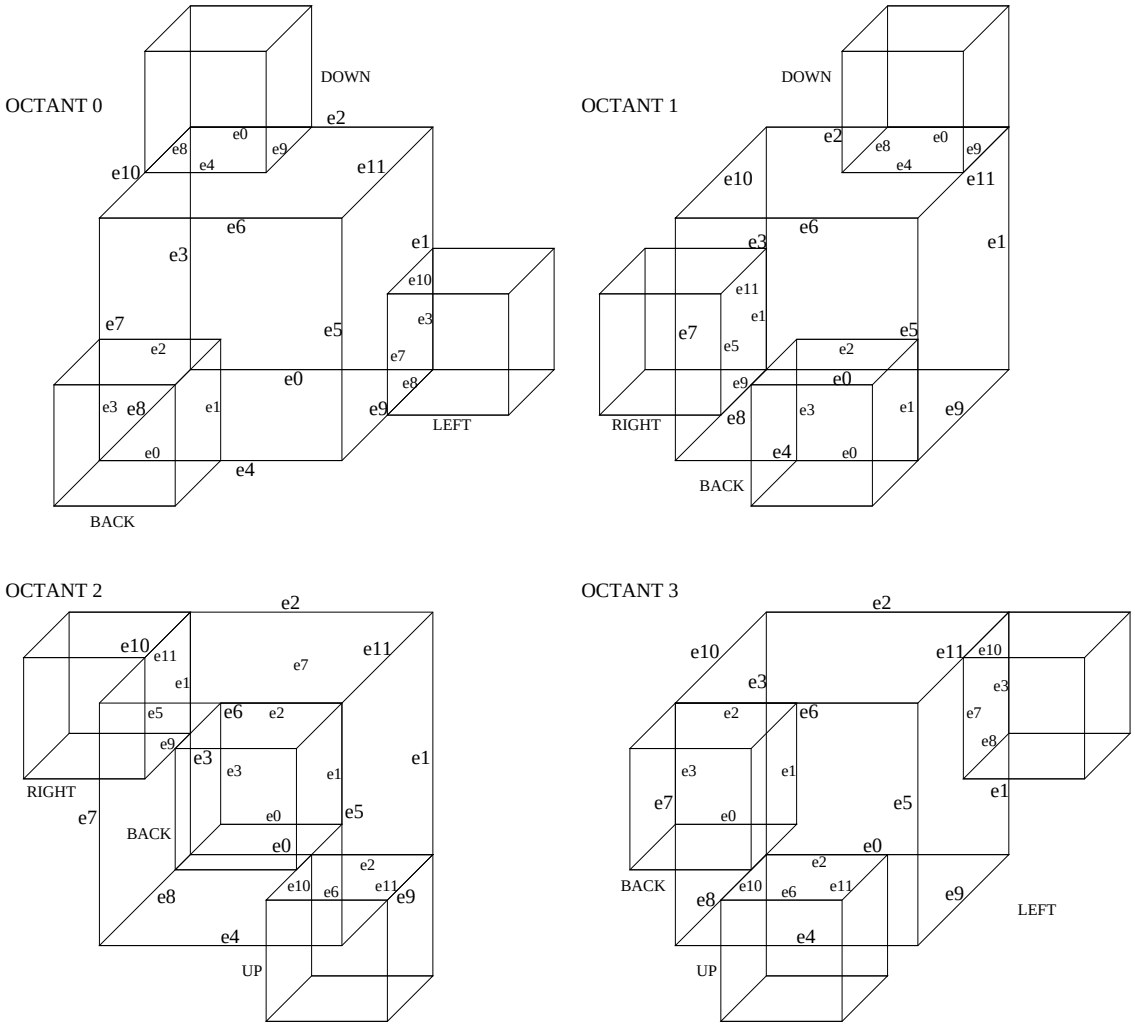

Figure E.1: This figure shows the shared and facial edges between a finer and coarser resolution cube for coarser octants 0 – 3. For example (top, left) a finer resolution cube shares edges on its left, down, and back faces abutting a coarser resolution cube. This is used to correct inconsistent interpolation and identify when to subdivide a cube.

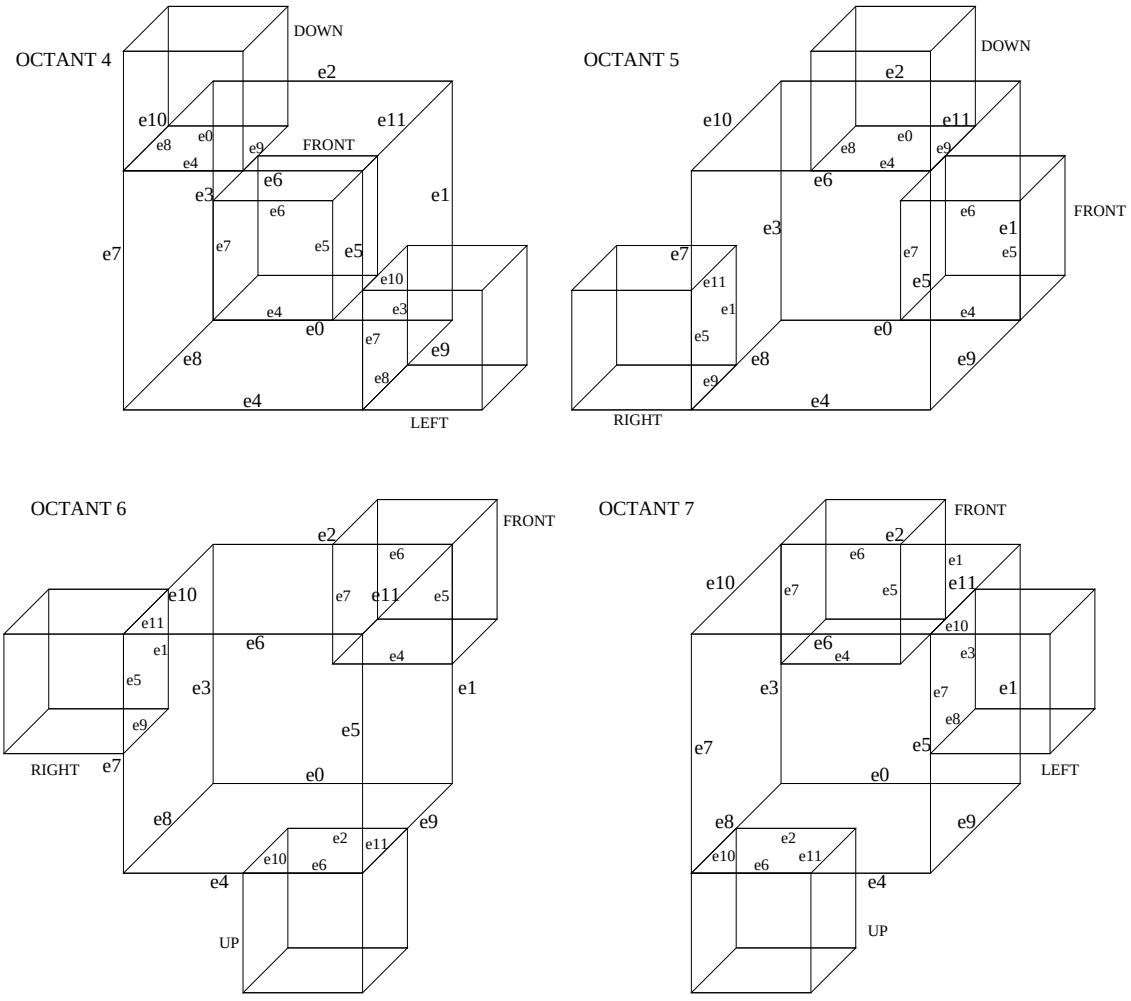

Figure E.2: This figure shows the shared and facial edges between a finer and coarser resolution cube for coarser resolution octants 4 – 7.

OCTANT	DIRECTION	FINE EDGE	COARSE EDGE	FACIAL EDGE
0	LEFT	3	1	10
		8	9	7
	DOWN	0	2	9
		8	10	4
	BACK	0	4	1
		3	7	2
1	RIGHT	1	3	11
		9	8	5
	DOWN	0	2	8
		9	11	4
	BACK	0	4	3
		2	5	2
2	RIGHT	1	3	9
		11	10	5
	UP	2	0	10
		11	9	6
	BACK	1	5	0
		2	6	3
3	LEFT	3	1	8
		10	11	7
	UP	2	0	11
		10	8	6
	BACK	2	6	1
		3	7	0
4	LEFT	7	5	10
		8	9	3
	DOWN	4	6	9
		8	10	0
	FRONT	4	0	5
		7	3	6
5	RIGHT	5	7	11
		9	8	1
	DOWN	4	6	8
		9	11	0
	FRONT	4	0	5
		5	1	6
6	RIGHT	5	7	9
		11	10	1
	UP	6	4	10
		11	9	2
	FRONT	5	1	4
		6	2	7
7	LEFT	7	5	8
		10	11	3
	UP	6	4	11
		10	8	2
	FRONT	6	2	1
		7	3	4

Table E.1: The lookup table used by a finer resolution node to identify shared edge intersections and facial intersections when examining a coarser resolution neighbor. We construct this table by looking at figure of edge intersections.

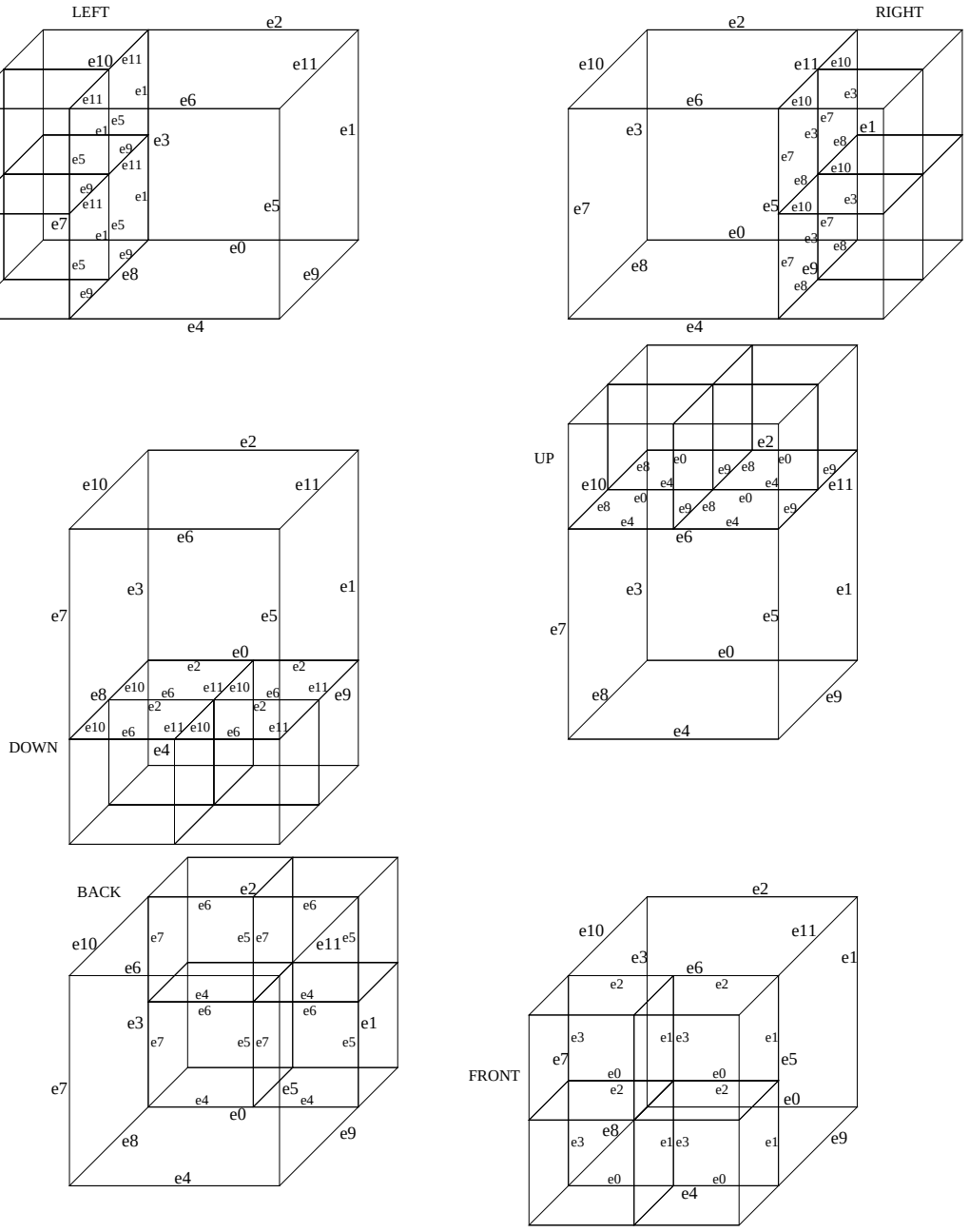

Figure E.3: This figure shows the shared edges between a coarser cube and its four finer resolution neighboring cubes for each of the coarser cubes six faces. This is used when a coarser resolution cube searches finer resolution neighbors for inconsistent interpolation.

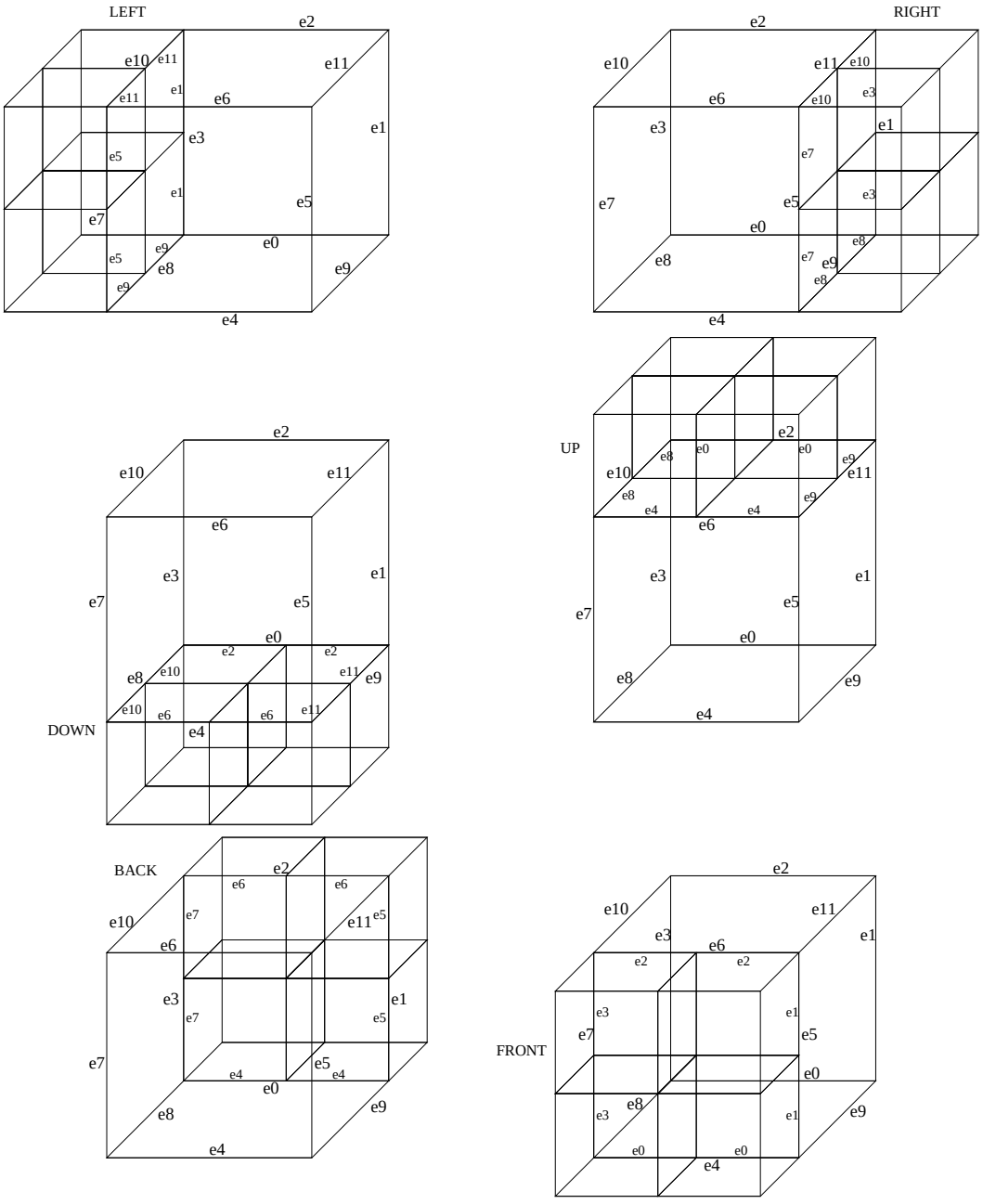

Figure E.4: This figure shows the shared edges between coarser and finer resolution cubes for each of the six faces of a coarser cube. This is used when searching for missing vertices.

APPENDIX F

SUBDIVISION AND CASE TABLE(S)

See figure F.1 for further details about subdivision.

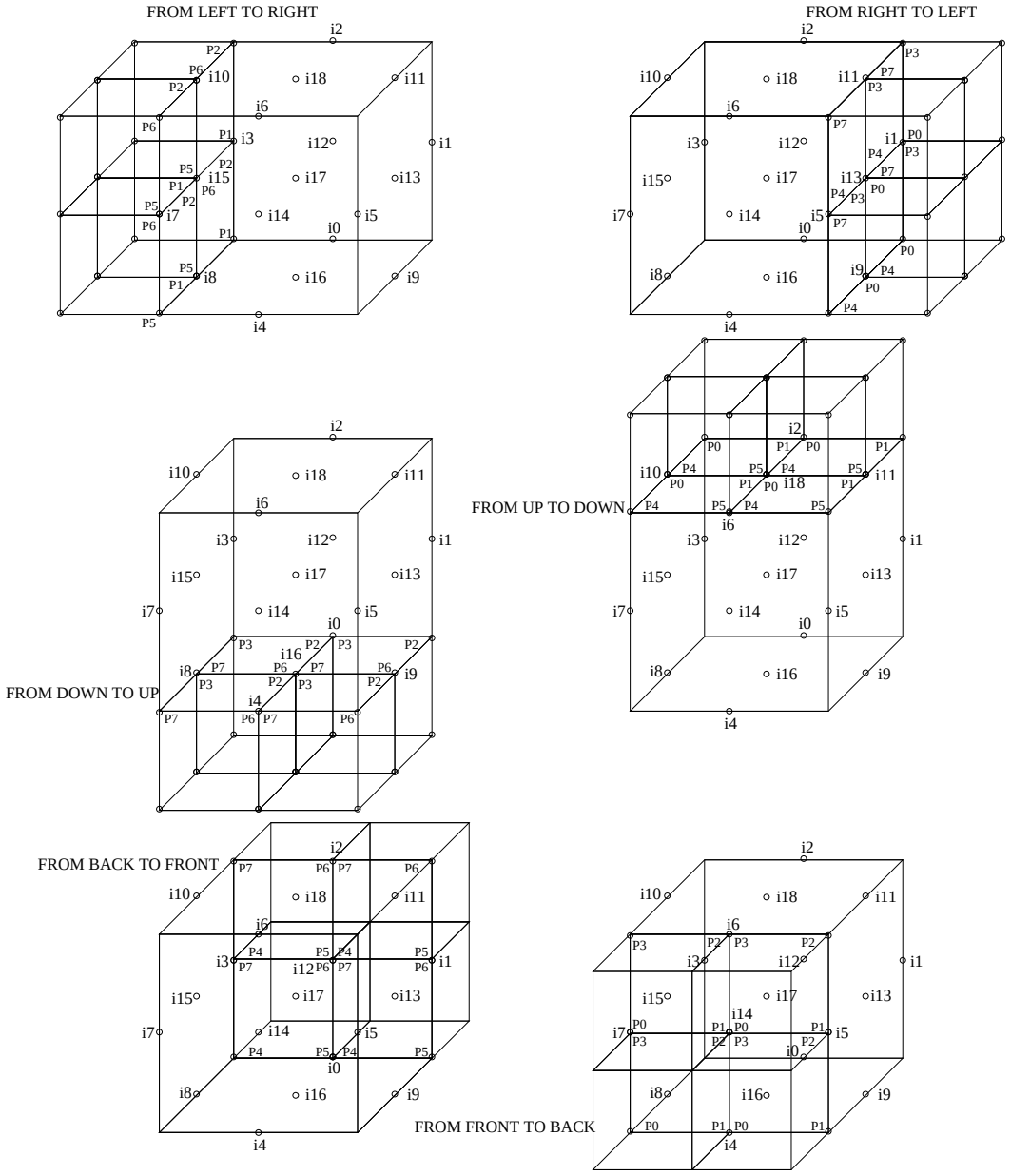

Figure F.1: This figure shows the shared vertices for each cube face when subdividing. The shared vertices are shown for each of the six faces of a coarser resolution cube. (i indicates an interpolated value, P indicates a vertex value.)

APPENDIX G

IMAGES

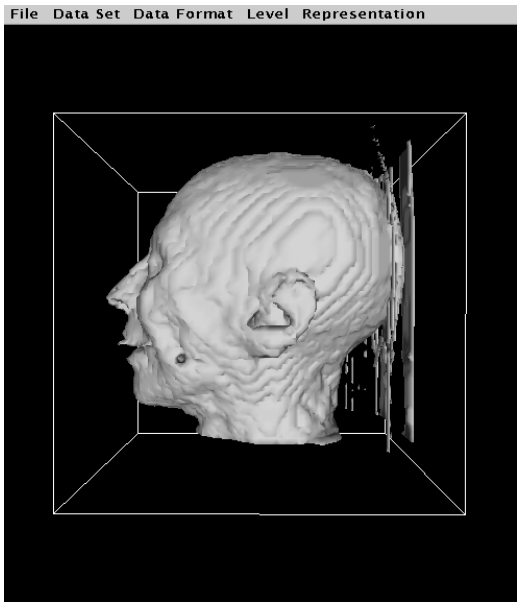

Figure G.1: This is an MR isosurface with resolution 64^3 and isovalue 0.185.

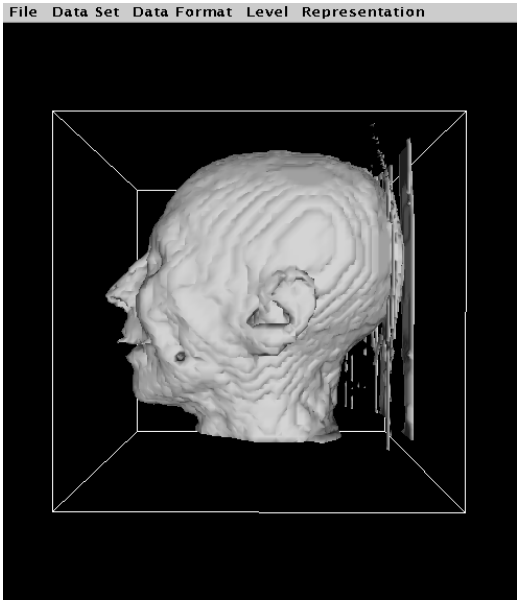

Figure G.2: This is an AR isosurface with resolution 64^3, isovalue 0.185, and $\delta = 1\%$.

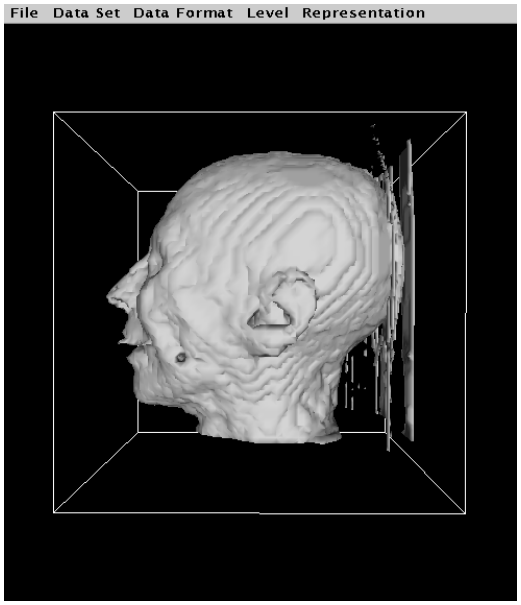

Figure G.3: This is an AR isosurface with resolution 64^3, isovalue 0.185, and $\delta = 2\%$.

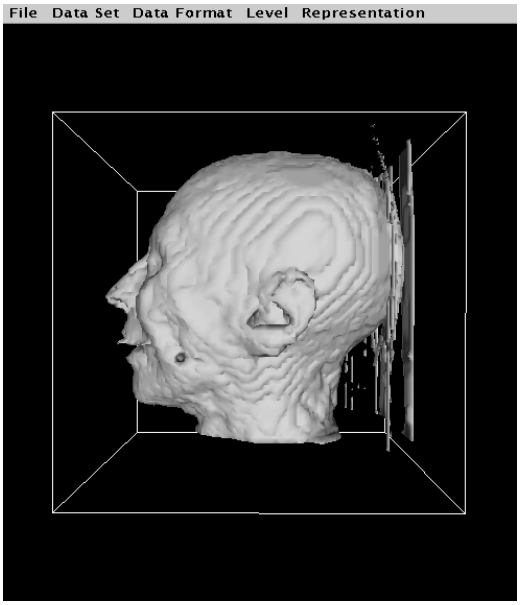

Figure G.4: This is an AR isosurface with resolution 64^3, isovalue 0.185, and $\delta = 5\%$.

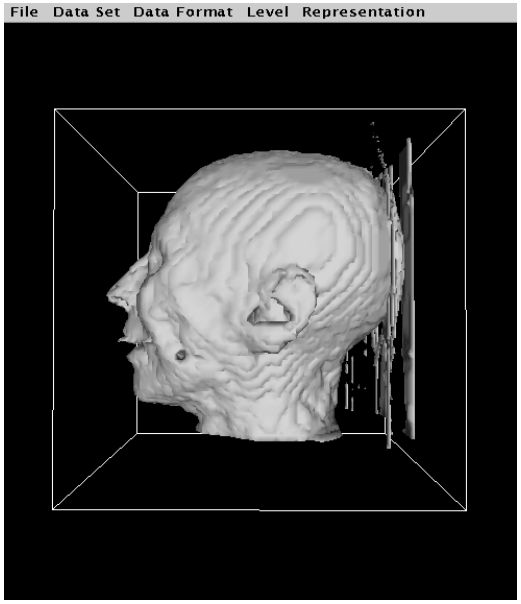

Figure G.5: This is an AR isosurface with resolution 64^3, isovalue 0.185, and $\delta = 10\%$.

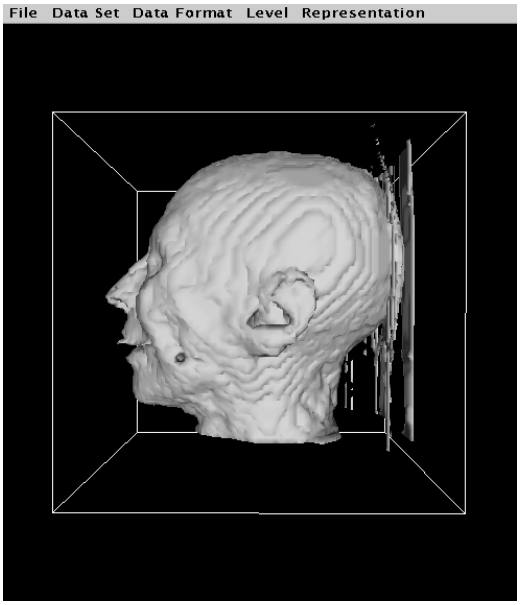

Figure G.6: This is an AR isosurface with resolution 64^3, isovalue 0.185, and $\delta = 15\%$.

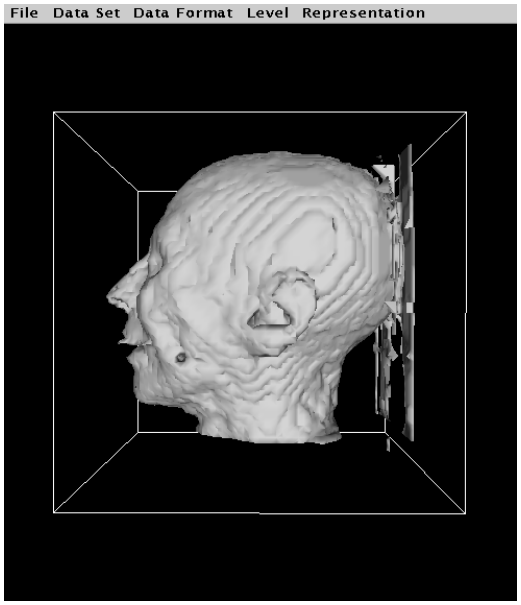

Figure G.7: This is an AR isosurface with resolution 64^3, isovalue 0.185, and $\delta = 20\%$.

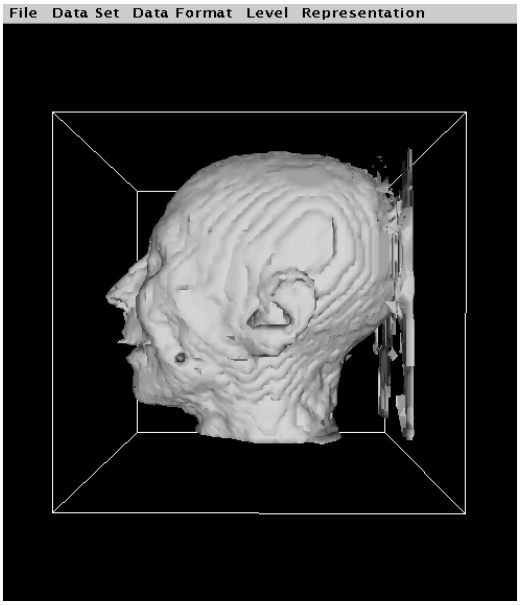

Figure G.8: This is an AR isosurface with resolution 64^3, isovalue 0.185, and $\delta = 25\%$.

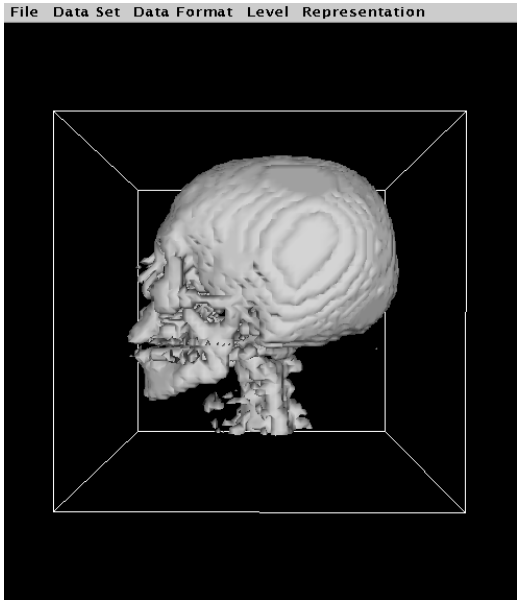

Figure G.9: This is an AR isosurface with resolution 64^3 and isovalue 0.378.

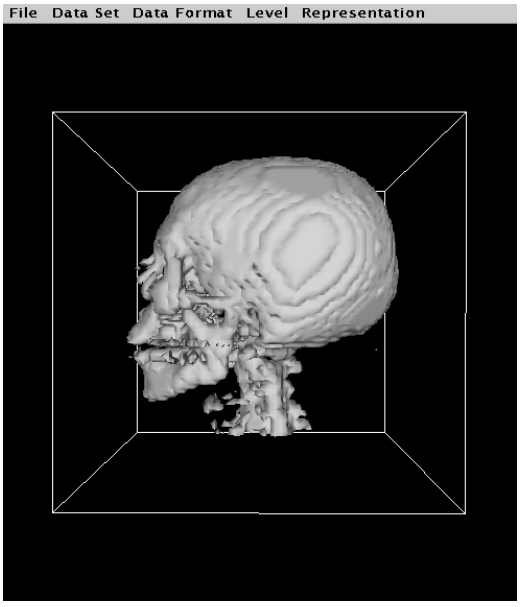

Figure G.10: This is an AR isosurface with resolution 64^3, isovalue 0.378, and $\delta = 1\%$.

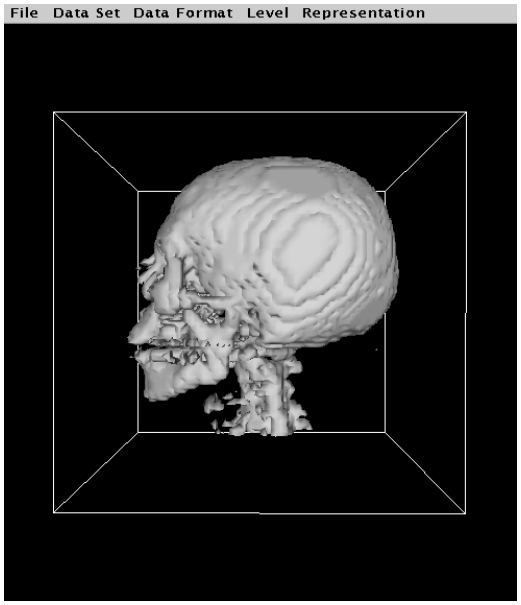

Figure G.11: This is an AR isosurface with resolution 64^3, isovalue 0.378, and $\delta = 2\%$.

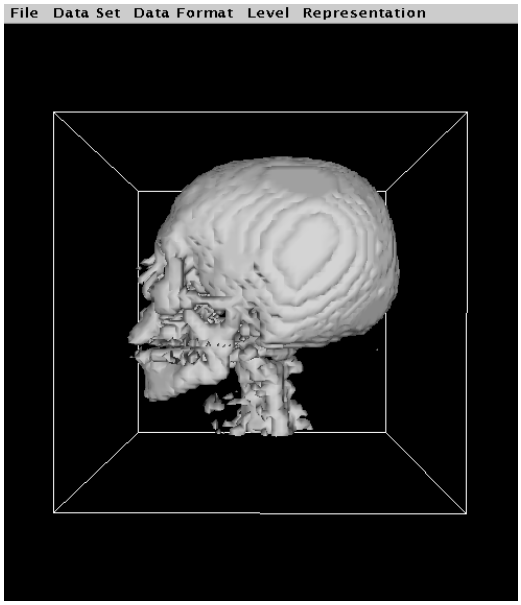

Figure G.12: This is an AR isosurface with resolution 64^3, isovalue 0.378, and $\delta = 5\%$.

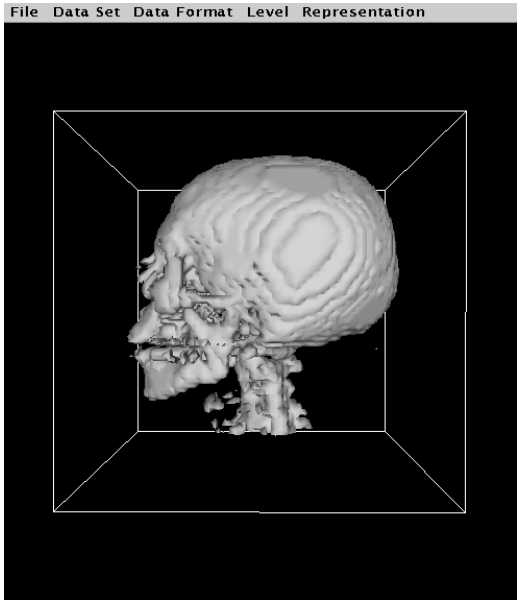

Figure G.13: This is an AR isosurface with resolution 64^3, isovalue 0.378, and $\delta = 10\%$.

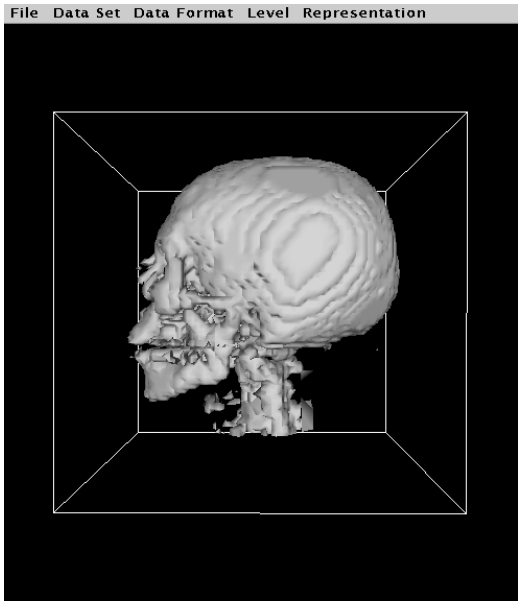

Figure G.14: This is an AR isosurface with resolution 64^3, isovalue 0.378, and $\delta = 15\%$.

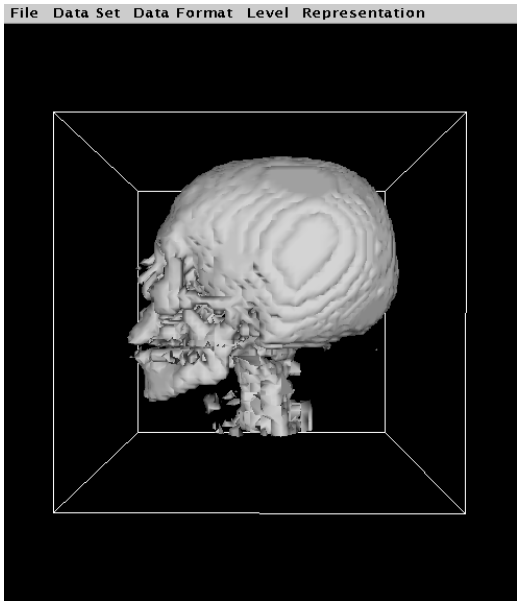

Figure G.15: This is an AR isosurface with resolution 64^3, isovalue 0.378, and $\delta = 20\%$.

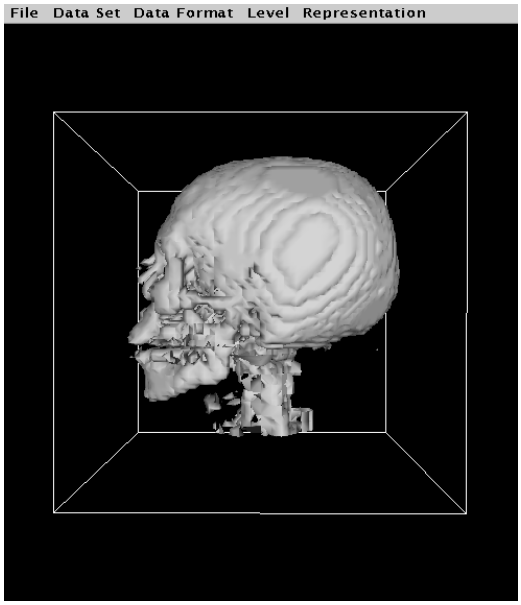

Figure G.16: This is an AR isosurface with resolution 64^3, isovalue 0.378, and $\delta = 25\%$.

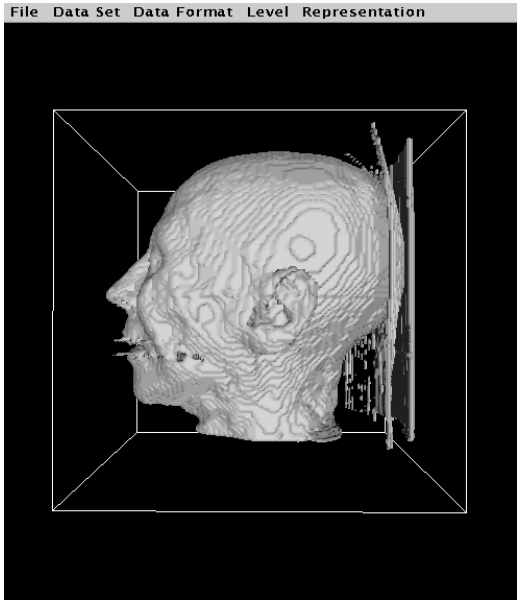

Figure G.17: This is an AR isosurface with resolution 128^3, isovalue 0.185, and $\delta = 10\%$.

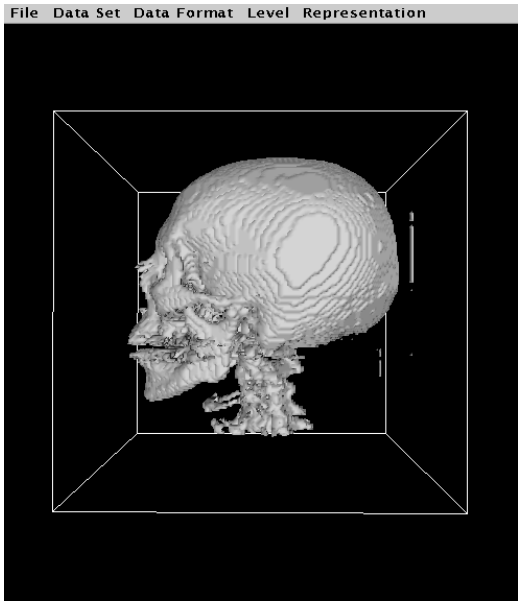

Figure G.18: This is an AR isosurface with resolution 128^3, isovalue 0.378, and $\delta = 10\%$.

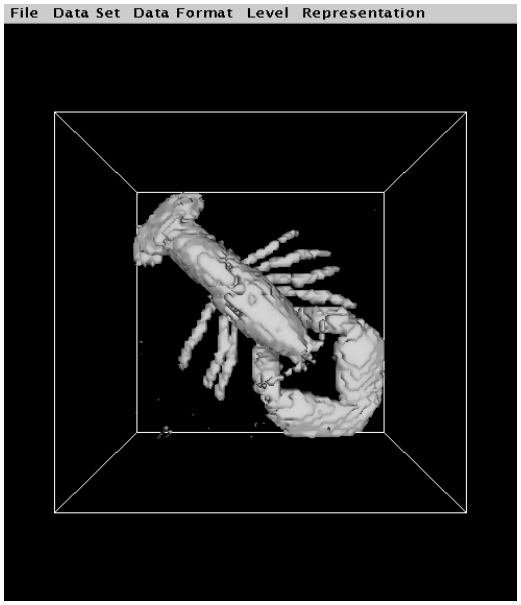

Figure G.19: This is an MR isosurface of a lobster with resolution 64^3 and isovalue 0.051.

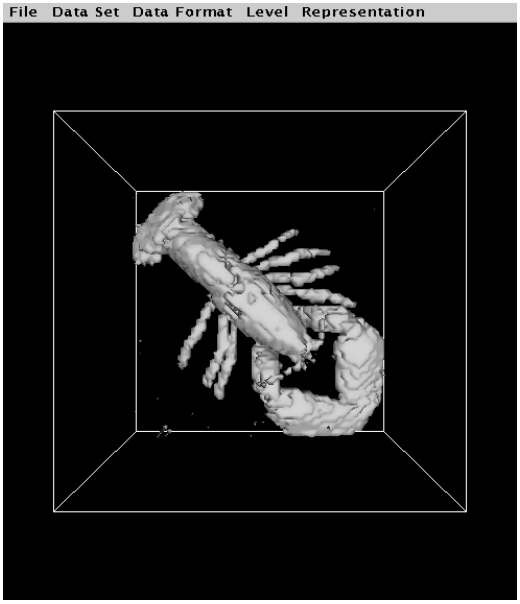

Figure G.20: This is an AR isosurface with resolution 64^3, isovalue 0.051, and $\delta = 1\%$.

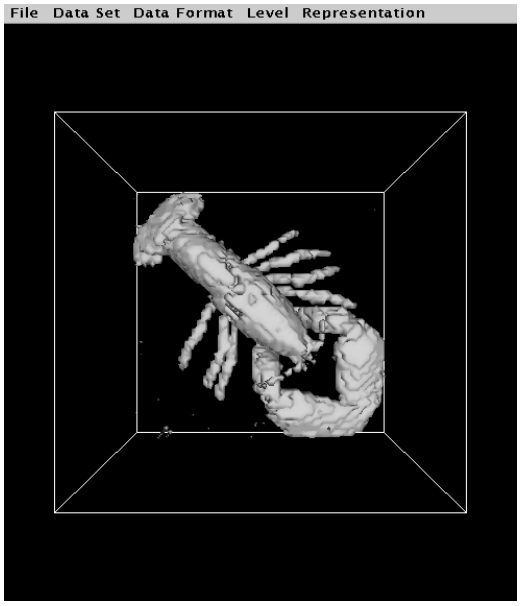

Figure G.21: This is an AR isosurface of a lobster with resolution 64^3, isovalue = 0.051, and $\delta = 2\%$.

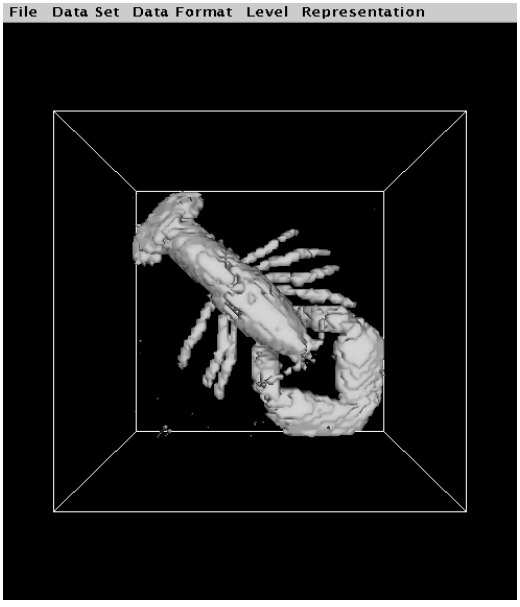

Figure G.22: This is an AR isosurface with resolution 64^3, isovalue = 0.051, and $\delta = 5\%$.

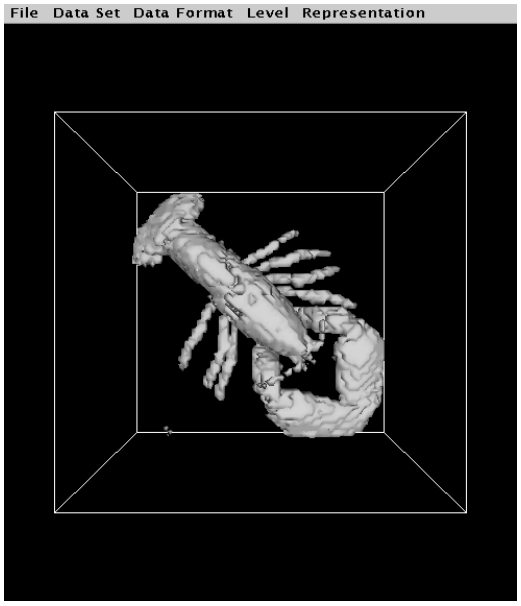

Figure G.23: This is an AR isosurface of a lobster with resolution 64^3, isovalue = 0.051, and $\delta = 10\%$.

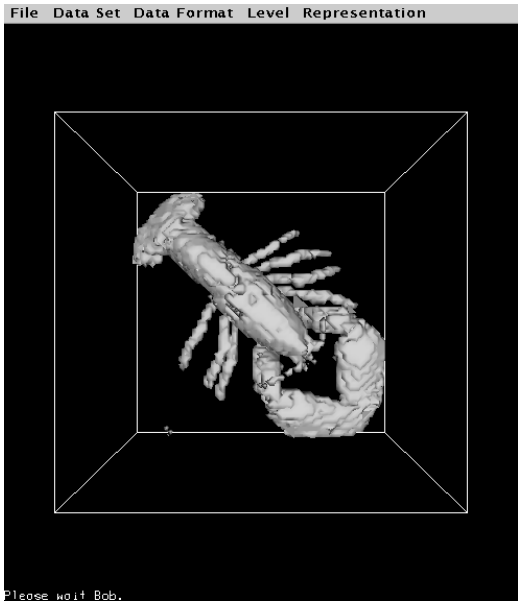

Figure G.24: This is an AR isosurface with resolution 64^3, isovalue = 0.051, and $\delta = 15\%$.

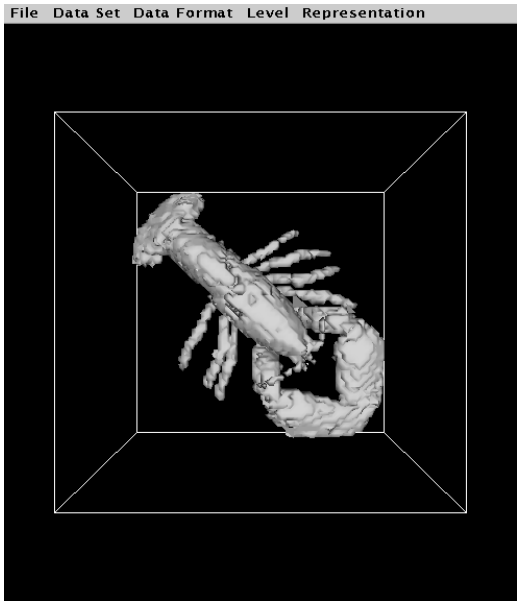

Figure G.25: This is an AR isosurface of a lobster with resolution 64^3, isovalue = 0.051, and $\delta = 20\%$.

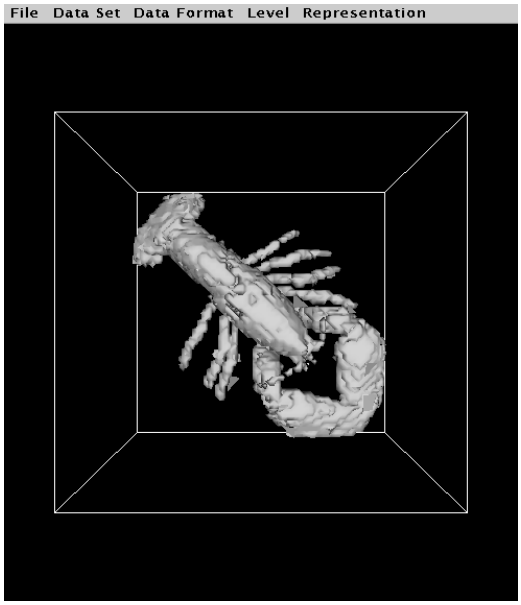

Figure G.26: This is an AR isosurface with resolution 64^3, isovalue = 0.051, and $\delta = 25\%$.