Interactive Visualization of Unstructured Grids Using Hierarchical 3D Textures

Joshua Leven Jason Corso
Jonathan Cohen Subodh Kumar

The Johns Hopkins University

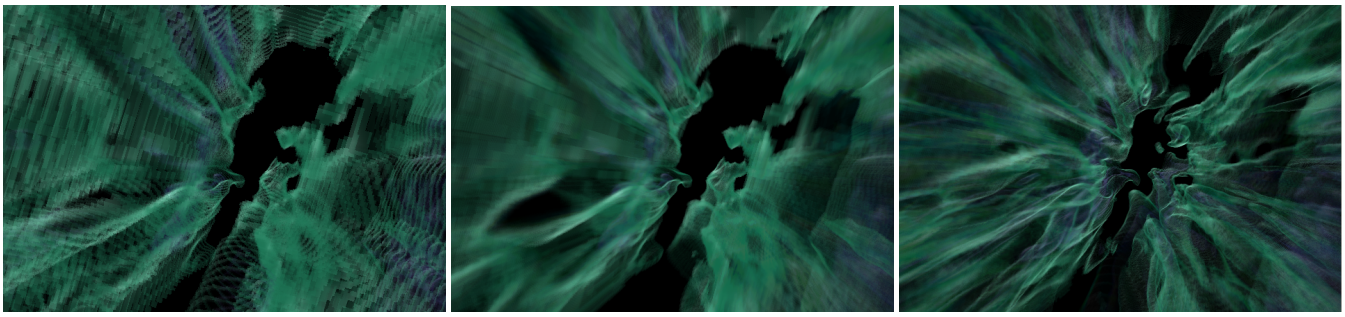

Figure 1: Radmri data set (1 million tetrahedra). Left: 9 texture bricks with standard slicing at 10 frames per second. Middle: 9 textures with extra planes rendered. Right: Refined to 98 textures with extra planes.

Abstract

We present a system for interactively rendering large, unstructured grids. Our approach is to voxelize the grid into a 3D voxel octree, and then to render the data using hierarchical, 3D texture mapping. This approach leverages the current 3D texture mapping PC hardware for the problem of unstructured grid rendering. We specialize the 3D texture octree to the task of rendering unstructured grids through a novel *pad and stencil* algorithm, which distinguishes between data and non-data voxels. Both the voxelization and rendering processes efficiently manage large, out-of-core datasets. The system manages cache usage in main memory and texture memory, as well as bandwidths among disk, main memory, and texture memory. It also manages rendering load to achieve interactivity at all times. It maximizes a quality metric for a desired level of interactivity. It has been applied to a number of large data and produces high quality images at interactive, user-selectable frame rates using standard PC hardware.

1 INTRODUCTION

Direct volume rendering of 3D scalar and vector fields plays an important role in the fields of medicine and scientific visualization. In these fields, 3D data is measured or computed at a large number of points in 3D space and then rendered to produce informative images.

Interaction is an effective technique for guiding the production of these images, allowing a doctor or scientist to navigate to various regions of interest and adjust the transfer function used to map the data into colors. Unfortunately, due to the size of these 3D datasets and the processing required to render them, interaction has only been possible with relatively small data sets and not of the most general organization.

In particular, regular rectilinear grids of volume elements, or *voxels*, are well-suited to acceleration by modern graphics hardware. This hardware can take the form of a dedicated volume processor [Pfister et al. 1999] or a more generic 3D graphics accelerator [Cabral et al. 1994]. For data of the right form and size, these hardware make interaction possible.

However, volume data comes in many sizes and forms. For example, data may be gathered along a warped grid, known as a *structured grid*, or it may be gathered at arbitrary points and connected topologically as an *unstructured grid* (also *irregular grid* or *tetrahedral mesh*). In addition, the data may be larger than the cache on a hardware accelerator or larger than the random access memory of the computer itself.

We present an interactive system to display large 3D unstructured grids that builds upon recent advances in 3D-texture based volume rendering to devise a pipeline that scales well with increasing data size. Because a structured grid is a special case of an unstructured grid, our algorithms also apply to this class of 3D scalar fields, as well as to regular voxel data. In essence, our system brings rendering problems from the domain of unstructured grids into the domain of voxel grids, allowing for both faster rendering of these data than previous approaches and reasonably high quality images. The classic use of uniform resampling of such data typically results in an explosion of both data size and error. To mitigate these problems, we employ a 3D texture octree and develop specialized, out-of-core algorithms for resampling, filtering, and rendering the data. We specialize the octree to account for the distinction between data and non-data voxels, avoiding rendering artifacts that would otherwise occur. In addition, we develop a time-critical (frame-rate driven) algorithm for adapting our octree rendering to the current view.

Given a set of non-overlapping tetrahedra with per-vertex data, the system adaptively creates an octree of voxel cells, with deeper leaf cells in regions of smaller tetrahedra (structured grids are first decomposed into tetrahedra). Each leaf node is resampled from the tetrahedron into a uniform voxel grid. We use a conservative metric to decide when a node may be uniformly sampled and hence produce satisfactory visualizations. Once the leaf cells have been determined, they are filtered to generate the lower resolution octree cells (given a large voxel grid as input, just this filtering is performed).

The interactive visualization system is used to visualize this texture octree data, regardless of its original form. It maintains two levels of LRU cache of the octree cells, one in texture memory on the graphics hardware, and one in main memory, as well as the full data set on disk. Not only are the maximum cache sizes respected, but special care is paid to the bandwidth required to fill the cache by managing the number of data loads per frame. A user-specified frame rate is maintained by using performance feedback from previous frames and adjusting the workload for the current frame accordingly. Thus the user can visualize the data at rates of over 10 frames per second.

Our system has a number of desirable properties:

- **Out-of-core operation:** Both the octree creation and rendering algorithms are designed to operate on data much larger than core memory.
- **General class of 3D input:** The algorithm handles general structured and unstructured grids as well as voxel data. Data and non-data are distinguished at sampling and rendering times using a novel *pad and stencil* algorithm.
- **Interactive performance:** The system maintains a user-specified frame rate in excess of 10 frames per second as well as allowing higher-quality refinement (as seen in Figure 1b-c).
- **Widely deployable:** The system has been implemented on a relatively inexpensive Windows PC using an NVIDIA Geforce4 accelerator.

We have used our system to examine a number of scalar fields, ranging from 1,700 to 6,000,000 tetrahedra, generating from 7 MB to 9 GB of voxel data. All these data sets may be explored at interactive rates of greater than 10 frames per second on a standard PC with OpenGL hardware supporting 3D texturing. This is significantly better than previous approaches to unstructured grid rendering in speed as well as in quality. We believe that the simplicity of our approach is its biggest asset.

2 PREVIOUS WORK

The field of volume rendering has been in existence for well over a decade [Drebin et al. 1988 , Levoy 1988]. The specific areas most closely related to the goals of this work are unstructured grid rendering and 3D texture level of detail.

2.1 Unstructured Grid Rendering

A number of approaches to direct volume rendering of unstructured grids have been applied, including ray casting [Garrity 1990], large-scale parallel rendering [Kniss et al. 2001], and hardware-accelerated cell projection [Shirley and Tuchman 1990]. Cell projection techniques also require fast algorithms for sorting the tetrahedral elements from front-to-back for proper rendering and opacity accumulation [Farias et al. 2000]. The works most closely related to ours are those that put a premium on interactivity, using time-critical raycasting [Farias et al. 2000] or fast hierarchical splatting approaches [Meredith and Ma 2001].

Compared to these recent works, our results exhibit faster frame rates and higher image quality during interaction due to the use of fast modern texture mapping hardware. Our main contribution in the area of unstructured grids is the algorithm for resampling the grid into an adaptive 3D texture octree. This out-of-core algorithm distinguishes between data voxels (those covered by the original grid) and non-data voxels (those outside the grid but within some octree cell) using a pad and stencil algorithm, padding the data boundary and marking the data voxels in a separate 3D stencil texture to allow for proper rendering.

2.2 3D Texture Level of Detail Rendering

3D texturing techniques, whereby a 3D texture is rendered on a stack of translucent polygons slicing through a volume, have recently become popular due to their effective use of texture mapping hardware and due to the abundance and cost-effectiveness of this hardware [Cabral et al. 1994, Westermann and Ertl 1998, Rezk-Salama et al. 2000]. Recent works have built level-of-detail hierarchies from high resolution 3D textures [LaMar et al. 1999, Boada et al. 2001], elaborating on proper methods for correcting texture opacities to account for differing slice distances, *bricking* the textures to avoid artifacts on same-level boundaries, and avoiding artifacts on inter-level boundaries [Weiler et al. 2000].

We build upon the texture octree data structure developed in [LaMar et al. 1999], specializing it to the task of storing resampled unstructured grids. Because our octree is intentionally an incomplete, adaptive octree, we develop a modified filtering algorithm to maintain continuity of adjacent nodes at the same octree level. In addition, we include a padding algorithm to enable accurate rendering at the data/non-data boundary. Finally, our adaptation algorithm for choosing the octree nodes to be rendered for a given frame is geared to maximizing quality for a target frame rate, rather than maximizing the frame rate for a given quality. We respect bandwidth limitations, treating texture memory and main memory as a two-level cache for the full data set on disk. Our focus is on frame rate management, and we emphasize the creation of non-uniform-depth octrees from unstructured grids as an important class of input data. Our frame rate management techniques are inspired in part by those of Iris Performer [Rohlf and Helman 1994] and the Berkeley Soda Hall walkthrough [Funkhouser and Sequin 1993].

3 OVERVIEW

Our approach to interactive volume rendering of unstructured grids involves an adaptive voxelization component, which occurs as a preprocess, and a volume level-of-detail component, which renders the voxel octree produced by the preprocess to provide visualization at interactive frame rates (see Figure 2).

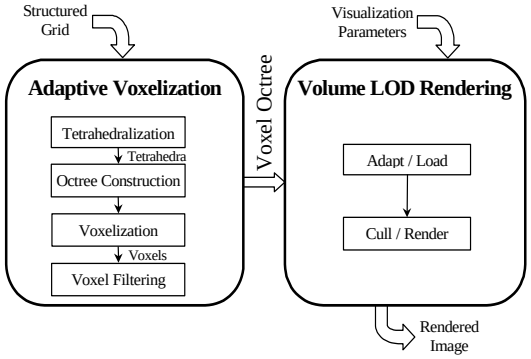

Figure 2: Components of interactive visualization system. Adaptive voxelization occurs as a preprocess, whereas interactive visualization occurs at real-time rates.

3.1 Voxel Octree Structure

The adaptive voxelization preprocess converts either a structured grid, unstructured grid (tetrahedral mesh), or regular voxel grid into a hierarchical 3D texture. For good scalability, we use a hierarchical arrangement of constant-resolution texture *bricks*. Our construction closely follows that of [LaMar et al. 1999, Boada et al. 2001]. (Note that this is somewhat different from that of [Weiler et al. 2000], which uses a uniform grid arrangement of multi-resolution bricks.) As shown in Figure 3, each node of the octree contains a texture brick of fixed resolution (4x4 for the 2D

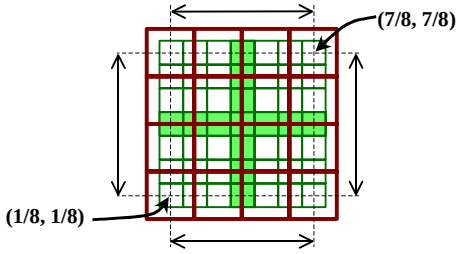

Figure 3: 2-level texture quadtree with 4x4 texture resolution in each node.

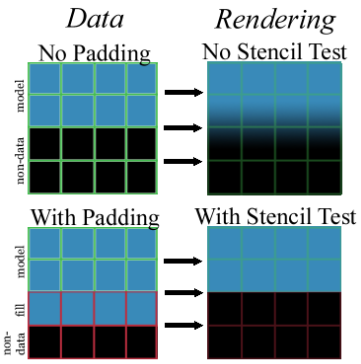

Figure 4: Padding and stenciling to remove non-data voxels.

figure). For a brick resolution *r*, the usable portion of the texture domain is from 1/(2*r*) to (2*r*-1)/(2*r*). Neighboring bricks have 1 texel of overlap, shown as shaded regions in the figure. This provides for continuity of data from brick to brick (at the same level, at least).

We specialize this octree structure of [LaMar et al. 1999] to our application domain by distinguishing between data and non-data voxels. Given an unstructured grid, the data must typically be filled out with some artificial filler to create a valid, rectangular 3D texture. A simple solution is to fill this space with zero-scalar data. However, with this naïve approach, there exists a discontinuity between the real data and the artificial. When using linear interpolation, this discontinuity displays itself as a striping along the boundary of the model. We remove this artifact through a two-step *pad and stencil* algorithm. First, the discontinuity is extended outward from the real data one texel during resampling by replicating the boundary data as described in Section 4.5. Second, during rendering we use a 3D stencil test to clip the replicated data. Thus, the replicated data is never rendered, but it *is* used when linearly interpolating pixels near the original boundaries. This is illustrated in Figure 4. An extreme example of this striping artifact and its correction on the Blunt fin model are shown in Figure 5.

The final data associated with a brick includes a scalar field / gradient texture and a stencil texture.

3.2 Processing Components

The first stage of this preprocess converts a structured grid into a set of tetrahedra (this stage can be skipped if the original input is already in tetrahedral form). The second stage uses a top-down approach to construct an octree to partition the model space. Each octree node is subdivided until an error threshold is reached for the subsequent resampling process. In the third stage, the tetrahedra within each leaf node's spatial cell are resampled into a regular grid of voxels. In the final stage, the leaf node voxels are filtered to produce the voxel grids for the all the interior tree nodes. This process results in a number of voxelized cells, using higher resolution where it is most needed to preserve the original data.

The volume level-of-detail rendering process allows the user to interactively choose a desired frame rate and other visualization parameters, such as the transfer function (mapping data to RGBA color), view position, etc. Given these user-specified parameters and the system parameters, such as RAM, texture memory size, bandwidths, polygon and fill performance, etc., this visualization process maximizes a measure of image quality within the user and system constraints. The adapt/load stage modifies the set of rendered cells from last frame to adapt it to this frame's new viewing parameters, splitting some cells while merging others to maximize quality. Some cells require reloading from RAM to

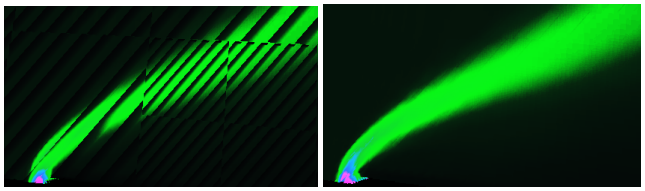

Figure 5: Blunt fin model without then with padding and stenciling.

texture memory, and others may first require loading from disk to RAM. The cull/render phase then traverses the octree from the root, performing a hierarchical cull and rendering the flagged nodes to the screen using 3D texture mapping and blending in back-to-front order.

We next discuss each of these components in more detail before examining the results of our system applied to a number of models.

4 ADAPTIVE VOXELIZATION

Voxelization algorithms have been applied to unstructured grids to produce a single-resolution voxel grid [Kaufman and Shimony 1986, Prakash and Manohar 1995]. Such a uniform grid typically undersamples as well as oversamples large portions of the volume. Instead, the voxel data organization in our work is an adaptive octree, similar to [Boada et al. 2001], employing the alignment strategy of [LaMar et al. 1999]. Our algorithm creates these multi-resolution voxels *directly* from the grid, however. It works on data much larger than could fit the main memory and uses less than 5 MB of RAM for the data sets we have tried so far. This octree partitions the space of the model, sampling local subspaces at appropriate resolutions. The adaptive voxelization algorithm is composed of several steps (see Figure 2) described next.

4.1 Tetrahedralization

If the input to our system is a structured grid, we begin by tetrahedralizing it. Structured grids with convex, linear elements are easily tetrahedralized by slicing off tetrahedra from the corners until only one tetrahedron remains (aka "ear slicing"). For the hexahedral grids of many of our test models, we apply the same tetrahedral decomposition template to each cell, breaking each cell into 5 tetrahedra. (In our current implementation, we limit the input to convex grid-cells. Algorithms to tessellate concave cells into tetrahedra exist [Ruppert and Seidel 1992], as well.)

In order to facilitate linear file I/O, the tetrahedra are written to a flat, unindexed tetrahedron file. In the octree construction that follows, such files are created for each octree node, specifying the dimensions of the octree node and an appropriate subset of the tetrahedra. Each tetrahedron comprises four vertices, each with three 32-bit floating point coordinates and one 16-bit scalar field value, for a total of 56 bytes per tetrahedron.

4.2 Octree Construction

The voxel octree nodes are built top-down, recursively. The procedure is shown in Figure 6. At the termination of each recursion, it samples the current leaf node into a voxel grid. The termination criteria are scalar-field homogeneity and tetrahedra size for the current node. The processing occurs in an out-of-core fashion: each tetrahedron is inspected independently and is discarded after it has been processed. Our current implementation imposes a constant voxel resolution per octree node.

In the current system, each tetrahedron vertex whose scalar field neighborhood varies more than a user-specified threshold lands in its own voxel and is thus sampled at least once. The scalar field error threshold provides a bound on the maximum

```
Process( Node N, Tetrahedra File F_N )
    read N's spatial extent
    create 8 children, C
    foreach child C_i
        create file F_Ci
        write C_i's spatial extent to F_Ci
    N.Recursion ← FALSE
    foreach tetrahedron T in F_N
        foreach child C_i
        if Intersect(C_i, T) = TRUE
            write T to F_Ci
        if T.NeededResolution > Resolution
            N.Recursion ← TRUE
    if N.Recursion = FALSE
        delete 8 C_is and F_Cis
        Voxel grid V_N ← Sample(N, F_N)
    else
        delete F_N
        foreach child C_i
            V_Ci ← Process(C_i, F_Ci)
    return V_N
```

Figure 6: Pseudo-code for the adaptive octree construction procedure.

resampling error during voxelization. In the extreme, a threshold of zero-error may be specified ensuring that no two tetrahedron vertices lie in the same voxel. To make this geometrical constraint feasible in practice, we exclude degenerate (nearly flat) tetrahedra from this computation.

An important component of this phase is the tetrahedron-cell overlap test. We extend the separating axis theorem [Gottschalk et al. 1996] to tetrahedra for this. For our case of intersecting a box (cell) with a tetrahedron, testing 25 particular axes are sufficient to prove whether or not the volumes overlap. The 25 axes consist of 3 axes orthogonal to the faces of the cell, 4 axes orthogonal to the faces of the tetrahedron, and the cross product axes of both objects' edges (3 for the cell times 6 for the tetrahedron yields 18 more axes). We can exit early from this set of tests, making them quite fast in practice.

4.3 Node Sampling

Once a node terminates the recursion and chooses to be a leaf node, it samples its tetrahedra at a uniform resolution. The sampling algorithm used is a voxel-order algorithm. This ordering implies that all the tetrahedra for this node should be loaded into memory at once. In practice, this is not a problem as there are generally many more voxels than tetrahedron vertices at the leaf nodes.

A voxel is sampled by first finding which tetrahedron contains it and then interpolating the tetrahedron's scalar field values according to the voxel location. Our approach does not need any topological connectivity information among the tetrahedra. The tetrahedra are bucketed into a coarse uniform grid (using the intersection test described in Section 4.2). We maintain a one-tetrahedron cache of the most recently used tetrahedron to take advantage of voxel-to-voxel coherence. When this tetrahedron fails a voxel test, the uniform grid is used to accelerate the selection of the next tetrahedron. The combination of coherence and spatial locality makes the testing fast overall.

The point-in-tetrahedron test is performed using the barycentric coordinates of the voxel's sample point with respect to the tetrahedron. If the coordinates sum to less than 1, the point is contained. Conveniently, these barycentric coordinates are then used to perform the interpolation. The computation of the coordinates is accelerated by caching the basis conversion matrix (from Euclidean to barycentric coordinates) for each tetrahedron in the node once before node sampling. Although we have chosen to implement a CPU based algorithm for its simplicity and precision, we should note that efficient hardware-accelerated voxelization algorithms exist [Westermann 2000, Weiler and Ertl 2001]. For applications where voxelization speed is an important concern, such faster voxelization algorithms may be employed at this stage of our preprocessing pipeline.

```
Filter( Node N , FilterStage S)
  create data buffers, B and O
  if ( S = MERGE_CHILDREN )
  read N's 8 children into B
     foreach child C_i
        read Ci's adjacent nodes into B
  create box filter kernel, K
  else // S = COMPUTE_GRADIENT
     read N and N's neighbors into B
     create gradient filter kernel, K
  convolve B by K into O
  perform normalization on O
  write O into N->output_filename
  free temporary data, B and O
```

Figure 7: Pseudo-code for the node filtering with overlapping borders.

4.4 Node Filtering

The next stage of building the voxel octree is voxel filtering to create parent nodes. Two separate filtering stages are used. First, child nodes are merged into parent nodes. A 3x3 box filter is employed for this. (Other filters, such as a Gaussian filter, may also be used during this process.) Second, gradients are computed at all nodes. The central difference method is used to compute gradients and is replaced by a forward or backward difference at the volume boundaries. Both filtering stages execute in a bottom-up level by level fashion (see pseudo-code in Figure 7).

To avoid discontinuities between neighboring cells and to maintain overlapping borders, we include the neighboring nodes' data for each node during processing. During the merging stage, we load the 7 nodes adjacent to each child node into memory and then perform a merge along the common border. This is depicted in Figure 8a, where the three nodes adjacent to *child a* are loaded and the common border is drawn in red (a quadtree is used to facilitate presentation). Likewise, during the gradient filtering stage, the cell's 26 neighbors are loaded into memory to accurately calculate the gradient along the border. These filtering procedures correspond to the largest amount of in-core memory used during the voxelization (128K * 27 nodes). During this filtering, we are only concerned with neighbors that exist on the same level in the octree; discontinuities between neighboring cells of different levels must be corrected during rendering (see Section 5.4). If at any time, a neighbor does not exist on the same level in the octree, a special node containing *null* data is loaded.

Finally, special care is required because of our adaptive, sparse octree with overlapping borders. During the child-merging stage, it is possible that only a portion of a node's octants subdivides thus creating discontinuity between node levels. Since neighbor information is only maintained on a per-level basis, this could introduce discontinuities at these internal borders. In Figure 8b, we show an example of such a case (in 2D). When this node's child *d* merges its children, its top and left borders will be incorrect because *d's* children d_a, d_b, and d_c do not have neighbors at their depth. If we naively merge *a,b,c* and *d*, this incorrect data would be averaged into the resultant parent causing discontinuities at its internal child boundaries. Thus, when *d* merges its

children, we test for this special case and fix the discontinuity by copying the border from *b* and *c* into *d*.

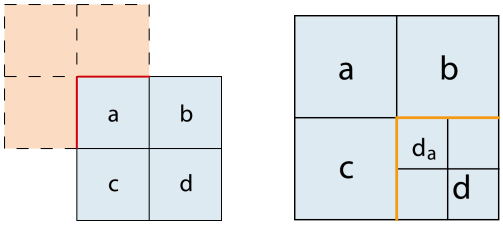

Figure 8: (a) Ensuring border continuity during the child-merge stage. (b) Internal border discontinuity fixing between cells of different levels during the child merge stage. (2D is shown for simplicity.)

4.5 Padding and Stencil Texture

To prepare the octree cells for fixing the striping artifact as discussed in Section 3.1, two further processing stages are included in the voxelization pipeline. The first stage creates a stencil, or binary, texture for each cell. This texture stores one-bit-per-voxel that is 0 for *null* data and 1 for any other data. The next stage pads the embedded volume by one voxel. Along the boundary of the volume in the octree between data and *null* data, we grow the texture by averaging neighboring *non-null* voxels.

5 VOLUME LOD RENDERING

Using the level-of-detail hierarchy described by the voxel octree, our visualization system is designed to maximize quality for a user-specified frame rate. In addition, it can be quickly re-tasked to produce the best image possible at a reasonable frame rate (as defined by rendering all the textures that will fit in memory). This gives the user considerable freedom to explore the data set and examine high-quality images of localized portions of the data. For example, Figure 1 starts with a highly interactive visualization (a) which can be quickly refined by adding additional slices (b). Given more time, subdivision can occur to its fullest extent, allowing the third rendering (c).

We next describe the two phases of the level-of-detail rendering system – the adapt/load phase and the cull/render phase.

5.1 Adapt / Load

The adapt/load phase is responsible for determining which set of octree cells should be rendered for the current frame and for loading any non-resident textures to texture memory either from local RAM or from external memory (hard disk, in our case).

A list of nodes known as the *active cut*, as shown in Figure 9, is maintained from frame to frame. The active cut is a set of nodes whose cells cover the model space in its entirety exactly once. This notion of cut is very similar to those used in view-dependent, continuous level of detail for polygonal models [Xia and Varshney 1996, Duchaineau et al. 1997, Hoppe 1997, Luebke and Erikson 1997]. The cut is maintained by performing a series of merges and splits each frame to produce the best possible quality according to some error metric without going over the budgeted frame time. Note that the number of nodes rendered from the active cut (due to fill rate), and the number of merges and splits (which may require texture loads from RAM or disk) all influence the frame time. Hence, not only must we manage the number of nodes on the cut but also restrict the amount of cut-adaptation that may be performed in a single frame.

In addition to flagging nodes to indicate if they are on the cut, our visualization system constructs two priority queues each frame – the merge queue and the split queue. The merge queue contains the parents of all nodes on the active cut, sorted in order of increasing error. The split queue contains all non-leaf nodes on the active cut, sorted in order of decreasing error.

To decrease the overall error, the first split from the split queue is performed by: placing the splitting node on the merge queue and marking it inactive, then placing its children on the split queue and flagging them as active.

To increase the overall error (and thus the frame rate): the first node is removed from the merge queue, and placed on the split queue, marking it active; then the node's parent is placed on the merge queue and its children are removed from the split queue and marked inactive.

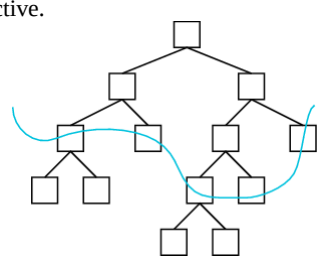

Figure 9: A cut through the tree defines 5 active nodes (only a binary tree is shown for simplicity).

The number of splits and merges performed in a given frame are bounded according to the machine's bandwidths from disk to RAM and from RAM to texture memory. Texture caches are maintained in both main memory and texture memory according to a least recently used policy, and thus splits or merges which cause cache misses are more expensive than those which are already cached.

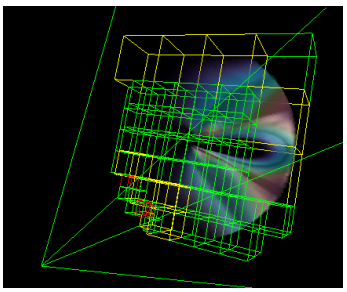

Figure 10: Oxygen post model (513,375 tetrahedra) showing view-dependent refinement

In addition to being limited by bandwidth restrictions, splits are restricted according to the total texture memory size (because they increase texture memory usage) and the desired frame rate (because they increase the number of rendered nodes). If one of these limits is reached, then a split may only be performed after forcing another node to merge. This is only allowed when the merge followed by the split would reduce the overall error (i.e. the merge node's error is less than the split node's error). In practice, a split operation may be performed over multiple frames because it can involve loading several texture nodes from RAM or disk.

Since a split may not complete in a single frame, it is possible that the node in the process of being split is no longer the best candidate for splitting. If a node other than the currently splitting node is more appropriate to split (because of a higher error), progress on the current node is stalled and the new node begins splitting immediately.

The error metric used to determine priorities of merges and splits is the maximum screen space size of a node's texels in pixel units. Ideally, every texel rendered would fill a single pixel – all renderable data would be rasterized, and no extra texture data would be used. As the pixels per texel within a node increases for a non-leaf node, visual quality quickly decreases, deviating from what it could be at a deeper node. The adapt / load phase is

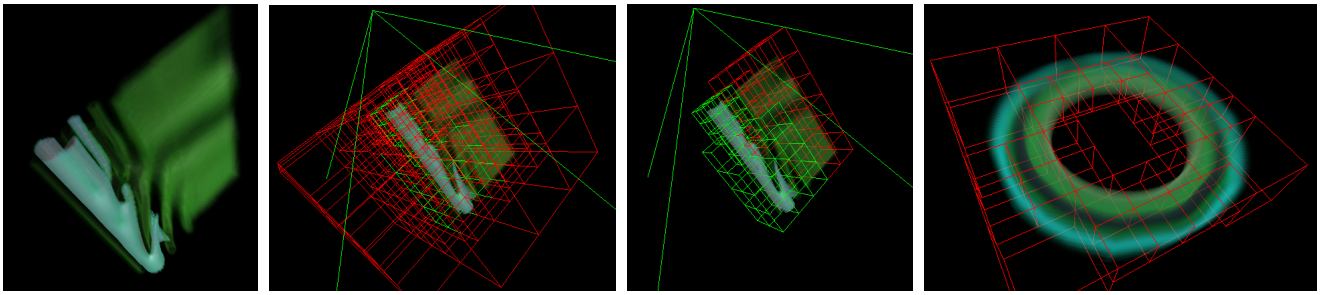

Figure 11: (a) Cylinder data set (615,000 tetrahedra) (b) all nodes in the view frustum (235 nodes) (c) nodes remaining after transparency culling (80 nodes) (d) DC plate with transparency culling (note culling at the center)

responsible for maintaining all texel sizes at as close to a single pixel as possible, while not allowing the frame rate to fall below the current target frame rate.

Figure 10 shows the Oxygen Post model with the active cut created for the depicted viewing frustum. The red cells are leaf nodes and thus at sufficient resolution to represent the original data. The green nodes are non-leaf nodes with projected texel sizes of less than a pixel, and yellow nodes are non-leaf nodes with larger projected texel sizes. Yellow nodes will refine if the budget allows. Note that deeper refinement occurs closest to the viewpoint.

5.2 Cull / Render

The cull/render phase is performed hierarchically using an octree traversal, starting from the root node. When a node is visited, it is culled according to the viewing frustum and the transfer function. If the node is entirely outside the viewing frustum or if its range of scalar field values lies entirely in a range of the transfer function with zero opacity, the node is culled. Culled nodes are set to zero error and placed on the merge queue so their children may be merged upwards next frame. Even if a culled node is not rendered in a given frame, its texture is not swapped out until it is actually merged into its parent. In other words, the cut indeed covers all of the model space. This ensures that in any given frame, we are able to quickly render each portion of the model at some resolution and with any transfer function, even if that portion was culled in the previous frame. If the current node of this octree traversal is flagged as being on the cut and not culled, it is rendered. If the node is not on the cut, then its children are visited in back-to-front order.

Our system renders each node's voxel data using 3D texture mapping with viewport-aligned planes [Westermann and Ertl 1998]. Although it is possible to choose a different normal for each node's stack of planes, coarsely approximating the use of spherical shells [LaMar et al. 1999], we have found that using the same plane alignment for all cells works reasonably well for moderate fields of view. We choose the planes such that the inter-plane distance is equal to the distance between axis-aligned slices. Using this approach, we only need to compute a single set of geometry and texture coordinates for the planes used to render a node for a particular frame. We can use exactly the same primitives for all nodes just by changing the model transformation matrix.

We apply a user-specified transfer function dynamically. The transfer function is stored as a dependent texture, which is a lookup table stored on the graphics accelerator. Because the function is applied for every texture lookup, the user can interactively change the transfer function by simply updating the lookup table.

The use of different plane spacings at different octree levels can cause opacity imbalance artifacts. This imbalance is corrected by applying a correction factor to the opacity which depends on the inter-plane distance, as described in [LaMar et al. 1999, Weiler et al. 2000]. We store several versions of the transfer function lookup table on the graphics accelerator for use with different inter-plane distances (distances which are powers of two multiples of some smallest distance are supported). Each node then uses the appropriate lookup table depending on its plane spacing for the current frame.

Figure 11(a-c) shows the Cylinder model both without, and with, transparency culling. The number of nodes displayed is greatly reduced, increasing rendering speed, and allowing for further refinement of the model. Figure 11d shows the DC Plate. Several nodes at the center of the model are culled within the viewing frustum due to transparency culling.

5.3 Fragment Program

We shade the rendered slice polygons using a custom fragment program on an NVIDIA GeForce4 graphics accelerator. The fragment program applies the current transfer function to the scalar field data, performs two-sided diffuse and specular illumination of the data according to its gradient, and applies a 3D stencil test to clip non-data voxels from the resulting image.

Our fragment program uses three textures for each primitive:

Texture0	3D	Stencil Texture
Texture1	3D	Gradient <x, y, z>, Scalar Value
Texture2	2D	Transfer Function (dependent texture)

The stencil texture uses nearest neighbor interpolation, whereas the others use linear interpolation. We use two fragment registers to store our constants. The RGB of constant color 0 and 1 are the scaled-and-biased light and view directions respectively. The alpha values are used for two illumination coefficients. The first is the ambient coefficient; the second is the diffuse (as well as being one minus the specular) coefficient. The lighting equation is:

$$Final_{RGB} = (I_{ds} * C_d) + ((1-I_{ds}) * I_s^{32}) + I_a$$
$$Final_{Alpha} = (\text{stencil} == \text{valid}) \ ? \ \text{Alpha} : 0$$

where,

I_{ds} = diffuse_and_specular_coefficient
C_d = ($transfer_fn_{rgb}$ x max (V • L, -V • L))
I_s = max (V • H, -V • H)^{32}
I_a = (ambient_coefficient)
Alpha = $transfer_fn_{alpha}$

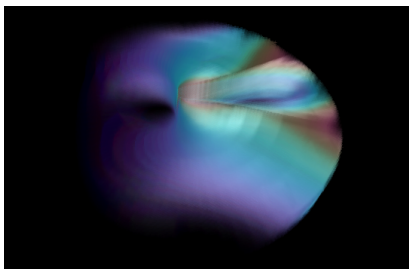

Figure 12: Lit view of the oxygen post.

5.4 Other Quality Issues

Several types of rendering artifacts are possible in a hierarchical 3D texture-based volume rendering system. These include interslice shearing gaps, texture discontinuities, and mathematical imprecision.

Interslice shearing gaps can occur due to the distance between the rendered planes and the non-optimal orientation (i.e. true concentric spheres would not suffer from this artifact). These shearing gaps may be reduced to a pixel or less in size by increasing the number of planes (thus reducing the interplane distance) as a node approaches the viewer. Unfortunately, this can severely degrade performance by dramatically increasing the number of textured fragments rendered. When the target frame rate mode is disabled (either manually or due to a pause in navigation), we reduce the shearing gap sizes by rendering more planes. This effect can also be reduced by pre-integrating the transfer function and rendering the data as thick slabs rather than thin slices [Engel et al. 2001]. This can be incorporated into our fragment program on hardware with 5 or more texture units.

Texture discontinuities occur when adjacent rendered cells are at different octree levels. Several techniques have been proposed to reduce or eliminate these discontinuities. For example, one can load pre-filtered transition textures into the borders of the 3D textures on an as-needed basis depending on which neighbors appear in a given frame [LaMar et al. 1999].

Finally, all 3D texture-based volume rendering that we are aware of is subject to some precision problems, which generally grow worse with increasing numbers of planes (and thus increasing compositing operations) per pixel. This problem should be solved by the next generation of hardware, which promises 32-bit floating point precision throughout the graphics pipeline.

6 RESULTS

We have implemented our system on a Windows 2000 PC with a 1.2 GHz AMD Athlon processor and NVIDIA GeForce4 graphics accelerator with 128 MB of Video RAM. In practice, we can use as much as 120 MB of the VRAM as texture memory. To make loading texture data from disk to texture memory feasible within a small portion of a frame time, we choose to sample all our cells at 32^3 resolution, thus we can fit a maximum of 930 cells in the texture memory cache at once.

Some statistics about the adaptive voxelization of our test models are shown in Figure 13. Most of these models originated as structured grids. and were tetrahedralized using 5 tetrahedra per grid cell.

In Figure 14, we show a graph that characterizes the relation of the scalar field homogeneity and geometric complexity for each of the large models. If we see an early fall-off, like we do for the post model, then it can be inferred that the underlying grid used to simulate the data had many locations where it over-sampled the data. Likewise, if we see a late fall-off, like for the delta model, we can infer that the grid used better sampled the high density portion of the data.

Model	# Tets	Avg/Max Depth	# Nodes	Size (MB)
DC Plate	1,745	2.8/3	51	6.9
Blunt Fin	187,223	8.22/10	22,208	3,002
Oxygen Post	513,375	9.03/11	65,324	8,830
Cylinder	615,195	7.94/9	10,608	1,434
Delta Wing	664,668	9.26/14	13,703	1,852
Radmri	6,011,200	4/4	489	66

Figure 13: Test model statistics after adaptive voxelization. Results shown are for 0% scalar field error threshold.

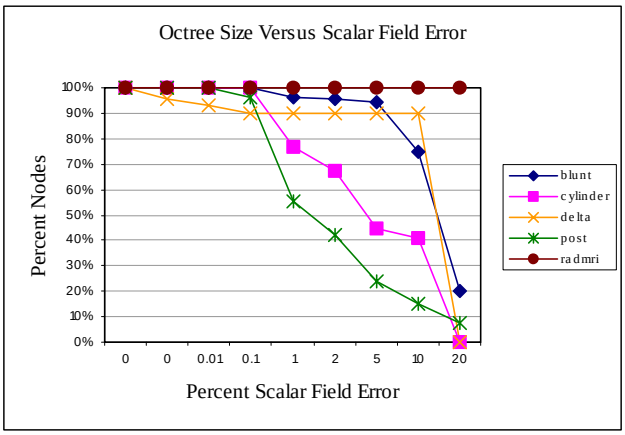

Figure 14: Graph showing the relative reduction in output octree size with varying scalar field error thresholds.

Figure 15 shows data recorded along a path exploring the Blunt Fin model, visualized in a 640x480 resolution window. This was run without register combiners on a GeForce3. In the first graph, we see the path with the target frame rate at 10 frames per second. Notice that for some periods the frame rate is well above that budget. The reason for this is clear on examining the second graph, which shows that the maximum error is usually less than one pixel per texel during those periods, thus no refinement is necessary. Notice as well that as the frame rate increases around frame 380, the system responds by adding more cells until the quality improves to less than one pixel per texel.

7 CONCLUSIONS AND FUTURE WORK

We have demonstrated an interactive volume rendering system capable of exploring scalar fields in the form of unstructured grids as well as structured grids and uniform voxel grids. Our system maintains a user-specified frame rate while maximizing quality (as well as transitioning quickly to a higher refinement mode). It improves the state of the art in unstructured grid rendering in terms of quality and speed on a low cost PC platform. Although we have not yet implemented a quantitative image quality metric, the accompanying video allows us to make a visual claim of quality.

As future work, we would like to employ more system timing parameters such as fill rates to schedule the frames more proactively, reducing our reliance on a feedback mechanism. Also, we would like to explore the use of 3D texture mapping in the domain of time-varying data, which may require much more accurate management to deal with the rapidly changing information.

8 ACKNOWLEDGMENTS

We would like to thank Xueying Cao, Edmund Dorsey, and Siddharth Sonrexa for their programming contributions to this work. Datasets have been graciously provided by NASA and the Space Telescope Science Institute. Funding has been provided by NSF CISE-RI Grant 9703080, NSF CAREER Award CCR9733827 , NSF Engineering Research Center EEC9731478, and a Joseph Pistritto Fellowship.

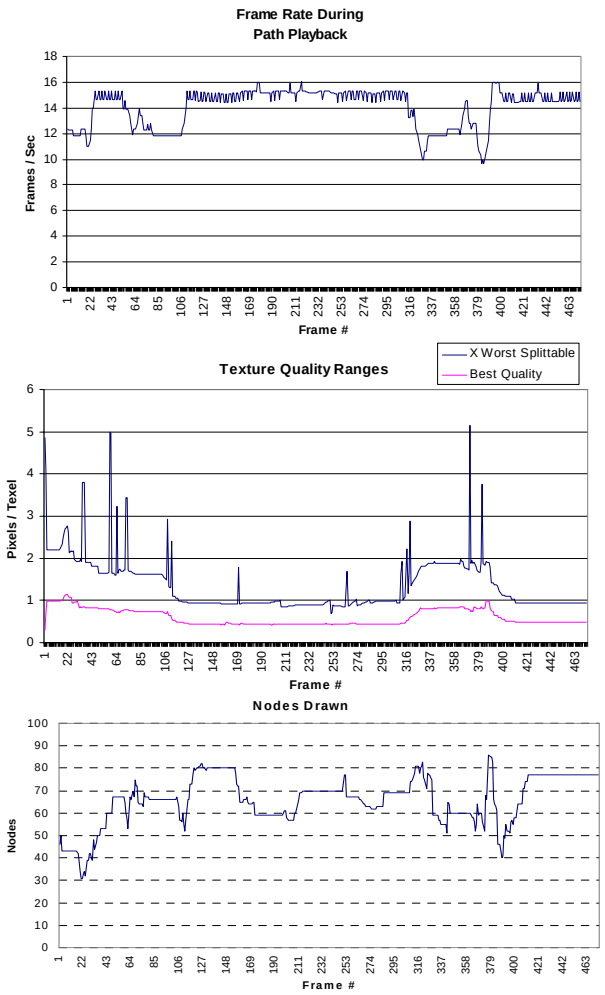

Figure 15: Data from an exploration path around the Blunt Fin.

References

Boada, Imma, Isabel Navazo, and Roberto Scopigno. Multiresolution Volume Visualization with a Texture-Based Octree. *The Visual Computer*. vol. 17. 2001. pp. 185-197.

Cabral, B, N Cam, and J Foran. Accelerated Volume Rendering and Tomographics Reconstruction Using Texture Mapping Hardware. *Proceedings of 1994 ACM Symmposium on Volume Visualization.* pp. 91-98.

Drebin, Robert A., Loren Carpenter, and Pat Hanrahan. Volume Rendering in J. Dill, ed *Computer Graphics (Proceedings of SIGGRAPH 88)* Atlanta, Georgia 1988 pp. 65-74

Duchaineau, M, M Wolinsky, D E Sigeti, M C Miller, C Aldrich, and M B Mineev-Weinstein. ROAMing Terrain: Real-time Optimally Adapting Meshes. *Proceedings of Visualization '97*. pp. 81-88.

Engel, Klaus, Martin Kraus, and Thomas Ertl. High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware 2001.* pp. 9-16.

Farias, R, J Mitchell, and C Silva. ZSWEEP: An Efficient and Exact Projection Algorithm for Unstructure Volume Rendering. *Proceedings of 2000 ACM/IEEE Volume Visualization and Graphics Symposium.* pp. 91-99.

Farias, Ricardo, Joseph S B Mitchell, Claudio T Silva, and Brian Wylie. Time-Critical Rendering of Irregular Grids. *Proceedings of SIBGRAPI 2000.* pp. 243-250.

Funkhouser, T. A. and C. H. Sequin. Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments. *Proceedings of SIGGRAPH 93.* pp. 247-254.

Garrity, Michael P. Raytracing Irregular Volume Data. *Computer Graphics (San Diego Workshop on Volume Visualization).* vol. 24(5). 1990. pp. 35-40.

Gottschalk, Stefan, Ming C. Lin, and Dinesh Manocha. OBB-Tree: A Hierarchical Structure for Rapid Interference Detection. *Proceedings of SIGGRAPH 96.* pp. 171-180.

Hoppe, Hugues. View-Dependent Refinement of Progressive Meshes. *Proceedings of SIGGRAPH 97.* pp. 189-198.

Kaufman, Arie and Eyal Shimony. 3D Scan-conversion Algorithms for Voxel-based Graphics. *Proceedings of 1986 Workshop in Interactive 3D Graphics.*

Kniss, Joe, Patrick McCormick, Allen McPherson, James Ahrens, Jamie Painter, Alan Keahey, and Charles Hansen. Interactive Texture-Based Volume Rendering for Large Data Sets. *IEEE Computer Graphics and Applications.* vol. 21(4). 2001. pp. 52-61.

LaMar, Eric C., Bernd Hamann, and Kenneth I. Joy. Multiresolution Techniques for Interactive Texture-based Volume Visualization. *Proceedings of IEEE Visualization '99.* pp. 355-362.

Levoy, Marc. Display of Surfaces From Volume Data *IEEE Computer Graphics & Applications* vol. 8 (3). 1988 pp. 29-37

Luebke, David and Carl Erikson. View-Dependent Simplification of Arbitrary Polygonal Environments. *Proceedings of SIGGRAPH 97.* pp. 199-208.

Meredith, Jeremy and Kwan-Liu Ma. Multiresolution View-Dependent Splat Based Volume Rendering of Large Irregular Data. *Proceedings of IEEE 2001 Symposium on Parallel and Large Data Visualization and Graphics.*

Pfister, Hanspeter, Jan Hardenbergh, Jim Knittel, Hugh Lauer, and Larry Seiler. The VolumePro Real-Time Ray-casting System. *Proceedings of SIGGRAPH 99.* pp. 251-260.

Prakash, C E and S Manohar. Volume Rendering of Unstructured Grid - A Voxelization Approach. *Computers & Graphics.* vol. 19(5). 1995. pp. 711-726.

Rezk-Salama, C, K Engel, M Bauer, G Greiner, and T Ertl. Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures and Multi-Stage-Rasterization. *Proceedings of Eurographics / SIGGRAPH Workshop on Graphics Hardware 2000.* pp. 109-118 and 147.

Rohlf, John and James Helman. IRIS Performer: A High Performance Multiprocessing Toolkit for Real--Time 3D Graphics. *Proceedings of SIGGRAPH 94.* July 24-29. pp. 381-395.

Ruppert, J and R Seidel. On the Difficulty of Triangulating Three-Dimensional Non-convex Polyhedra. *Discrete Computational Geometry.* vol. 7. 1992. pp. 227-253.

Shirley, P and A Tuchman. A Polygonal Approximation to Direct Scalar Volume Rendering. *Computer Graphics (San Diego Workshop on Volume Visualization).* vol. 24(5). 1990. pp. 63-70.

Weiler, M, R Westermann, C Hansen, K Zimmerman, and T Ertl. Level-Of-Detail Volume Rendering via 3D Textures. *Proceedings of Volume Visualization and Graphics Symposium 2000.* pp. 7-13.

Weiler, Manfred and Thomas Ertl. Hardware-Software-Balanced Resampling for the Interactive Visualization of Unstructured Grids. *Proceedings of IEEE Visualization 2001.* pp. 199-206.

Westermann, Rüdiger. The Rendering of Unstructured Grids Revisited. *Proceedings of 2001 Eurographics / IEEE TCVG Symposium on Visualization.* pp. 65-74.

Westermann, Rüdiger and Thomas Ertl. Efficiently Using Graphics Hardware in Volume Rendering Applications. *Proceedings of SIGGRAPH 98.* pp. 169-177.

Xia, Julie C. and Amitabh Varshney. Dynamic View-Dependent Simplification for Polygonal Models. *Proceedings of IEEE Visualization '96.* pp. 327-334.